Numéro d'ordre : 308

THESE

présentée à

l'U.F.R. des sciences et techniques de
l'Université de Franche-Comté

pour obtenir

Le Diplôme de Docteur de l'Université de Franche-Comté
en AUTOMATIQUE et INFORMATIQUE

par

Laurent PHILIPPE

Titre de la thèse

Contribution à l'étude et à la réalisation d'un système d'exploitation à image unique pour multicalculateur

Soutenue le 21 janvier 1993 devant la commission d'examen :

MM. :	Michel	Trehel	**Président**
	Philippe	Chretienne	**Rapporteurs**
	Jean-Marie	Rifflet	
	Marc	Guillemont	**Examinateurs**
	Pierre	Leca	
	Guy-René	Perrin	

Résumé

Les machines à mémoire distribuée proposent une alternative intéressante pour augmenter la puissance des ordinateurs. Cependant ces “multicalculateurs” sont réservés à une minorité d'utilisateurs avertis. En effet, les environnements de programmation et de développement qu'ils proposent n'offrent pas un confort ni une standardisation suffisants pour permettre une utilisation plus large de ce type d'ordinateurs.

L'architecture CHORUS est conçue pour construire des systèmes d'exploitation répartis : un noyau minimum, intégrant à bas niveau la communication et l'exécution, fournit des services génériques à un ensemble de serveurs qui coopèrent au sein de sous-systèmes pour fournir des interfaces standards. Par exemple, le système CHORUS/MiX offre une interface compatible à UNIX dont il étend les services à la répartition grâce aux propriétés de transparence de la communication. Il offre également de bonne possibilités d'évolution grâce à ses propriétés de modularité et ses interactions, basés sur des échanges de messages transparents à la répartition.

Ce type de système réparti semble indispensable sur les multicalculateurs, pour implanter les services spécifiques attendus sur ce type d'architecture. Son adaptation à ces conditions particulières (couplage des applications, régularité des programmes, équilibrage de la charge des processus, gestion de l'espace mémoire distribué, etc.) est étudié dans cette thèse à travers le portage du systène CHORUS/MiX sur un iPSC/2, multicalculateur commercialisé par Intel.

L'évolution naturelle de ce type de système conduit à offrir une interface de mono-processeur virtuel que nous appelons système à image unique. Ceci pose de nombreux problèmes que nous pouvons regrouper en deux classes : les problèmes liés à la gestion globale des ressources distribuées sur le réseau et les problèmes engendrés par l'aspect centralisé que sous-entend l'interface UNIX. Nous nous sommes intéressés plus particulièrement à la première classe de problèmes en concevant un service de migration de processus et un gestionnaire de charge répartie.

Mots clés

Multicalculateur, système d'exploitation réparti, migration de processus, équilibrage de charge, système à image unique.

Remerciements

Le travail présenté dans cette thèse consitue le reflet d'un investissement personnel qui a été facilité grâce au soutien apporté par plusieurs personnes. Je tiens à les en remercier et plus particulièrement :

Guy-René Perrin de m'avoir encadré pendant ce travail et accueilli au Laboratoire d'informatique de Besançon pour la rédaction du manuscrit. Il a activement participé à l'achèvement de ce document,

Messieurs Philippe Chrétienne et Jean Marie Rifflet, rapporteurs de cette thèse, qui ont relu et commenté le manuscrit,

Monsieur Michel Tréhel qui a présidé, avec bonne humeur, le jury de soutenance et qui y a représenté le Laboratoire d'Informatique de Besançon,

Marc Guillemont pour avoir représenté Chorus Systèmes au jury de soutenance et toute l'équipe de Chorus Systèmes pour la chaleur de son acceuil, pour l'aide et le soutien qu'elle m'a témoigné. Je lui souhaite d'atteindre ses objectifs tout en conservant une aussi bonne ambiance de travail. Parmi ses membres je tiens à remercier tout particulièrement Michel Gien de m'avoir donné l'occasion de faire cette thèse à Chorus Systèmes, Marc Rozier pour ses conseils pendant le portage du micro-noyau, Pierre Lebé pour son aide pendant ce portage et Frédéric Herrmann pour le soutien et l'encadrement constant qu'il m'a apporté tout au long cette thèse,

Pierre Leca pour avoir représenté l'ONERA au jury de soutenance mais également pour le soutien et la motivation qu'il m'a apporté durant tout le projet. Avec lui je remercie toute l'équipe de Calcul Parallèle de l'ONERA pour la sympathie qu'elle nous a témoigné pendant la durée de cette thèse,

Toute l'équipe du Lib de m'avoir accueilli lorsque je suis revenu à Besançon rédiger ce manuscrit,

Et, bien sûr, Bénédicte, mes parents et mes amis qui ont été mon soutien moral pendant ce travail.

Table des Matières

Table des Figures

1 introduction

Dans la plupart des domaines scientifiques et techniques l'ordinateur est devenu un outil indispensable car les modèles mathématiques utilisés pour la résolution des problèmes, sont, s'ils veulent être précis, beaucoup trop complexes pour être résolus par l'homme. Les super-calculateurs, utilisés pour résoudre ces problèmes dans les anneées 80, étaient basés sur une architecture multiprocesseur car les progrès technologiques réalisés sur un seul processeur n'étaient plus suffisant pour satisfaire les besoins croissants de puissance de calcul. Ces processeurs partageant une mémoire commune, la mise en place d'une politique globale d'allocation des ressources simple et efficace était possible grâce aux données communes. Le partage de mémoire pose cependant un grave problème de concurrence d'accès aux données. Les multiprocesseurs commercialisés sont ainsi limités à quelques dizaines de processeurs et les perspectives - pour augmenter de façon significative leur puissance - sont limitées.

Les ordinateurs à mémoire distribuée proposent une alternative intéressante pour augmenter la puissance des ordinateurs. En effet, la bande passante de leur réseau d'interconnexion augmente lorsque le nombre de noeuds du réseau croît. Cette propriété permet donc de résoudre le problème de concurrence d'accès aux données posé par les multiprocesseurs à mémoire partagée. Cependant ces "multicalculateurs" sont actuellement réservés à une minorité d'utilisateurs avertis car les environnements de programmation et de développement n'offrent pas un confort ni une standardisation suffisants pour permettre une utilisation plus large de ce type d'ordinateur. En effet, la distribution de la mémoire et du contrôle remettent en cause les savoir faire algorithmiques et rendent indispensable la conception de nouveaux environnements.

L'analogie avec les systèmes répartis sur un réseau de sites est évidente, quoique les conditions d'exploitation soient fort différentes : couplage des processus, régularité des programmes et des cheminements de données, équilibrage de la charge, etc. Cependant, moyennant l'étude de conditions spécifiques, le portage d'un système d'exploitation réparti général permet de fournir, à moindre coût, une interface équivalente à celles qui sont habituellement utilisées sur les stations de travail.

L'architecture CHORUS est conçue pour construire des systèmes d'exploitation répartis : un noyau minimum, intégrant à bas niveau la communication et l'exécution, fournit des services génériques à un ensemble de serveurs qui coopèrent au sein de sous-systèmes pour fournir des interfaces standards. Par exemple, le système CHORUS/MiX offre une interface compatible à UNIX dont il étend les services à la répartition grâce aux propriétés de transparence de la communication. Ainsi il permet un accès à la plupart des ressources telles que les fichiers et les périphériques, accès qui est indépendant de la localisation de ces ressources. Le système CHORUS/MiX offre de bonne possibilités d'évolution grâce à ses propriétés de modularité et ses interactions, basées sur des échanges de messages transparents à la répartition.

Description

Cette thèse présente une étude sur l'adaptation du système CHORUS/MiX sur un iPSC/2, multi-

calculateur commercialisé par la société Intel. L'évolution naturelle de ce type de système conduit à offrir de nouvelles fonctionnalités destinées à faciliter la mise en oeuvre du parallélisme d'une application et à faire évoluer le système d'exploitation vers ce que nous appelons un système à image unique.

Plan du manuscrit

La première partie de ce manuscrit est divisée en trois chapitres qui ont pour but de poser le problème qui y est abordé : les besoins des multicalculateurs en terme de services du système d'exploitation. Nous donnons donc, dans le premier chapitre, un aperçu de différentes architectures parallèles où nous différencions les architectures à mémoire partagée des architectures à mémoire distribuée pour mettre en valeur l'intérêt des secondes. Nous décrivons ensuite plus en détail l'iPSC/2 sur lequel nous avons travaillé. Au chapitre II, nous montrons que le support logiciel habituellement fourni sur les multicalculateurs est insuffisant. L'inconfort qui en résulte fait que ces machines sont réservées à une minorité d'utilisateurs initiés. Par contre, d'autres applications pourraient bénéficier du rapport coût/performances intéressant de ces ordinateurs si le système d'exploitation était plus standard et facilitait la mise en oeuvre du parallélisme. Dans ce but, nous étudions différents systèmes d'exploitation répartis et leur potentialité à satisfaire les utilisateurs. Parmi ces systèmes, CHORUS/MiX permet de résoudre quelques problèmes posés par les système natifs, tels que l'interface standard et le partage des noeuds entre utilisateurs. Nous relatons le portage de CHORUS/MiX au chapitre III. En utilisant ce système sur l'iPSC/2 nous analysons les lacunes des systèmes d'exploitation répartis puis nous cernons les besoins des utilisateurs.

Dans la deuxième partie, nous spécifions les besoins d'un système d'exploitation à image unique. Au chapitre IV, nous décrivons l'implantation d'un service de migration de processus. Ce service étant destiné à supporter une gestion de charge répartie, nous l'avons conçu sans dépendance résiduelle mais avec un coût minimal d'un point de vue global. Enfin le chapitre V traite de la conception d'un gestionnaire de charge répartie, le LM, dans le sous-système UNIX de CHORUS/MiX. A travers une étude bibliographique, nous montrons que l'architecture des multicalculateurs implique des contraintes spécifiques pour la mise en place d'un tel service. En particulier l'algorithme utilisé doit être extensible pour pouvoir être utilisé sur les machines à parallélisme massif comme sur les machines dont le nombre de noeuds est réduit. Il doit également assurer une bonne dispersion de la charge pour permettre un bon équilibrage de charge.

Contexte

Cette thèse a été réalisée dans le cadre d'une convention CIFRE avec la société CHORUS Systèmes. Les travaux qui y sont présentés ont été réalisés, en partie, pour le projet esprit PUMA.

Première partie

Chapitre 1

Les Architectures Distribuées

Introduction

Dans la plupart des domaines scientifiques et techniques l'ordinateur est devenu un outil indispensable car les modèles mathématiques utilisés pour la résolution des problèmes sont, s'ils veulent être précis, beaucoup trop complexes pour être résolus par l'homme. Les vitesses actuellement atteintes par les meilleurs super-calculateurs sont de l'ordre d'une dizaine de Gigaflops, c'est-à-dire d'une dizaine de milliards d'opérations sur des nombres flottants par seconde. Cette puissance de calcul permet de résoudre certains problèmes physiques tels que l'écoulement des fluides autour d'une aile d'avion, mais elle ne permet pas de considérer l'écoulement pour la totalité de l'avion.

D'autres classes de problèmes pour lesquelles les modèles mathématiques existent ne sont donc pas encore résolues faute de puissance de calcul. Ces classes regroupent "les grands défis" de la recherche et de la technique de cette fin de siècle tels que les prévisions météorologiques, la circulation des océans ou encore la modélisation des supra-conducteurs. Les besoins de ces classes de problèmes sont d'un ordre de grandeur supérieur puisqu'ils nécessitent des puissances de calcul avoisinant le Téraflops. L'objectif de nombreux constructeurs de super-calculateurs est d'atteindre cette puissance dans les cinq années à venir. Parmi les projets de recherche visant à atteindre cet objectif nous pouvons citer différents projets ESPRIT européens et le projet américain *Touchstone* mené conjoitement par Intel Scientific Supercomputer Division et la DARPA.

Ces exemples marquent bien les nouvelles orientations, en terme d'architecture, pour les super-calculateurs des années 90. En effet il s'agit dans tous les cas, de machines parallèles à mémoire distribuée, basées sur des processeurs standard, alors que les super-calculateurs des années 80 étaient principalement des multiprocesseurs à mémoire partagée, basés sur des processeurs vectoriels ou super scalaires (ex : Cray YMP[Cra91]). Ce changement dans l'architecture du super-calculateur était nécessaire pour deux raisons. D'une part, comme nous le verrons dans ce chapitre, les solutions envisagées pour augmenter significativement le nombre de processeurs dans une machine à mémoire partagée sont très coûteuses. D'autre part, les constructeurs de processeurs commencent à atteindre les limites physiques et thermodynamiques pour la réalisation de processeurs. Les difficultés à résoudre pour améliorer de faon significative les performances des

processeurs croissent alors très vite. La conception des micro-processeurs repose sur des technologies plus récentes (VSLI). Certaines études (c.f. les études *Technology 2000* du Massachusetts Institute of Technology) prévoient que les performances obtenues avec les micro-processeurs vont doubler tous les deux ans d'ici la fin du siècle. D'autre part les réseaux d'interconnexion des multiprocesseurs à mémoire distribuée permettent d'augmenter, à moindre coût, le nombre des micro-processeurs d'un réseau. Ainsi les calculateurs massivement parallèles, basés sur ces micro-processeurs, semblent être une alternative intéressante pour atteindre ce nouvel ordre de performances. En effet ces calculateurs ne sont, virtuellement, pas limités en terme de performances.

Nous nous proposons, dans ce chapitre, de classer et de décrire brièvement quelques machines parallèles existantes afin de pouvoir mieux situer la machine sur laquelle nous avons travaillé. En 2, nous donnons une description détaillée de l'iPSC/2.

1 Les architectures de machines parallèles

Il existe actuellement un grand nombre d'architectures parallèles. La description de la totalité de ces architectures dépasse largement le cadre de cette thèse. Nous proposons donc un aperçu des différents types marquants parmi les architectures parallèles [Kra91]. On distingue habituellement les architectures de machines parallèles selon une classification proposée dans [Fly74] qui repose sur l'examen séparé du parallélisme dans le flot de contrôle (instructions exécutées par les processeurs) et le flot de données (instructions lues ou écrites dans la mémoire). Les machines monoprocesseurs traditionnelles sont SISD (Single Instruction stream, Single Data stream). Les architectures de machines parallèles se divisent en SIMD (Single Instruction stream, Multiple Data stream) et en MIMD (Multiple Instruction stream, Multiple Data stream), la classe MISD n'étant pas représentée (certains auteurs y incluent les machines à processeur pipe-line, où le parallélisme s'étend à l'exécution des différentes phases d'une instruction unique).

Selon l'organisation de la mémoire, on distingue deux classes de machines SIMD. Dans les machines vectorielles, telles que le Cray XMP, certaines unités de traitement sont partitionnées en unités fonctionnelles. Chacune de ces unités exécute une opération sur un élément d'un vecteur - emplacement en mémoire ou dans un registre. Les flots de données circulent de façon rythmée à travers la partition d'unités fonctionnelles. Dans les machines SIMD parallèles, un ensemble de processeurs est commandé par un séquenceur unique pour exécuter la même instruction, chaque processeur possèdant sa zone de mémoire privée. Les machines SIMD sont principalement utilisées pour des applications de calcul scientifique telles que la résolution d'équations aux dérivées partielles ou encore le traitement d'image.

Un exemple intéressant de machine SIMD est la *Connection Machine* (CM2)[Hil88]. Elle a été conçue au MIT dans les années 80 pour des applications d'intelligence artificielle. La configuration maximale comporte 64K processeurs qui exécutent le même flux d'instructions. L'architecture de la CM2 repose sur une connexion en hypercube (section 1.2.2) de 2 puissance 12 noeuds. Un noeud est composé de 16 processeurs 1-bit, organisés en tore de 4x4, qui possèdent chacun un registre de 4K bits. La communication entre les noeuds est assurée par un routeur spécialisé.

Les machines MIMD diffèrent selon le mode de partage de mémoire entre les processeurs et

selon les structures d'interconnexion entre les éléments du système : processeur et mémoire. On distingue schématiquement les multiprocesseurs à mémoire partagée et les multiprocesseurs à mémoire distribuée, mais beaucoup d'architectures sont mixtes, des mémoires locales à chaque processeur pouvant coexister avec une mémoire commune.

Nous nous intéresserons plus particulièrement aux machines MIMD dans la suite de notre exposé.

1.1 Les multiprocesseurs à mémoire partagée

Les multiprocesseurs à mémoire partagée (ou commune) représentent l'architecture parallèle MIMD la plus fortement couplée. La mémoire commune permet en effet de représenter un état global immédiatement accessible à l'ensemble du système d'exploitation ou d'une application. Ainsi la connaissance d'un état global du système permet la mise en oeuvre d'une politique simple et efficace de l'allocation des ressources. Néanmoins la concurrence d'accès à l'information et aux ressources génère des goulets d'étranglement lorsqu'un grand nombre d'accès concurrents est nécessaire. De ce fait, le temps d'accès à une ressource n'est plus forcément borné. L'addition d'antémémoires (caches) ou de mémoires locales, pour assurer un accès rapide, pose alors un problème de cohérence de l'information entre les mémoires privées. Ce problème peut être simple à résoudre tant que le nombre de processeurs concerné est petit - quelques dizaines - mais il devient très vite compliqué et coûteux à résoudre lorsque le nombre de processeurs croît. Notons que ce problème doit être résolu au niveau matériel pour que la mémoire apparaisse partagée au niveau logiciel.

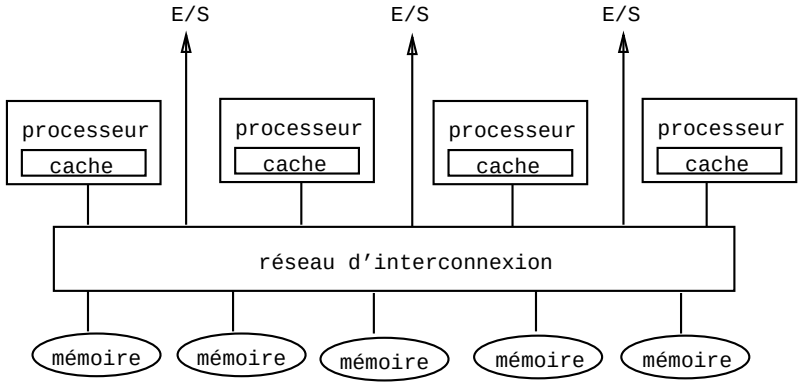

Figure I.1 : Multiprocesseur à mémoire partagée

Ce schéma (figure I.1) connaît de nombreuses variantes, selon :

– la répartition de la mémoire entre mémoire locale et mémoire commune
– la nature du réseau d'interconnexion qui permet aux processeurs de communiquer entre eux et avec la mémoire. Parmi les réseaux d'interconnexion, nous distinguons les bus et les réseaux à base de commutateurs.

1.1.1 Les multiprocesseurs à bus

L'interconnexion par bus est l'un des systèmes les plus répandus pour les multiprocesseurs (figure I.2), en raison de sa simplicité. La principale limitation de ce système provient du fait qu'un bus ne peut traiter qu'une requête à la fois. La probabilité de conflit et la dégradation corrélative des performances augmentent rapidement avec le nombre de processeurs.

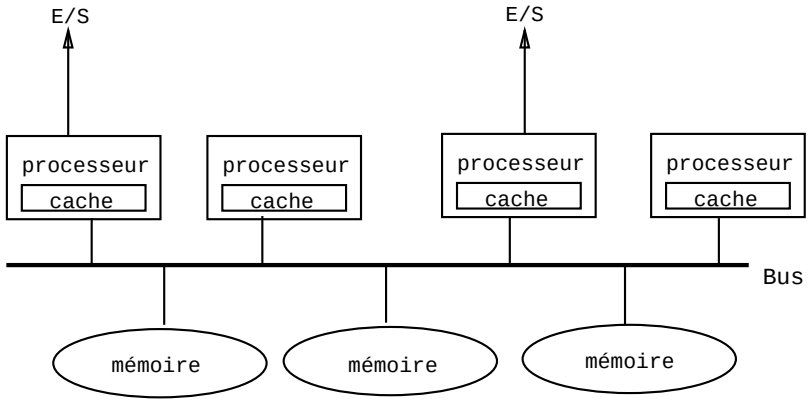

Figure I.2 : Multiprocesseurs à bus

La situation peut être améliorée par l'utilisation de plusieurs bus (figure I.3) et par le traitement des requêtes en "pipeline" (deux adresses différentes peuvent être traitées en même temps). D'autre part, l'utilisation de mémoires locales et de caches permet de réduire le nombre d'accès au bus global et à la mémoire commune, en exploitant la localité des programmes (mais se posent alors des problèmes de cohérence entre les mémoires).

Les multiprocesseurs à bus commercialisés sont limités à quelques dizaines de processeurs, pour des raisons de coût. Au delà, d'autres méthodes d'interconnexion sont utilisées. Des exemples de multiprocesseurs construits autour de bus sont :

– **Alliant FX/80** (figure I.4). Il est composé[Tab90] de deux types de processeurs : l'IP (Interactive Processor), et le CE (Computational Element). La configuration maximale comprend douze IPs et huit CEs. Les IPs sont en général utilisés pour traiter les exécutions interactives alors que les CEs sont dédiés aux calculs intensifs qui peuvent tirer profit d'une vectorisation. Les IPs et les CEs partagent la même mémoire à travers un bus. Tous les accès au bus ont lieu à travers des caches locaux pour réduire le nombre d'accès. De plus les CEs sont connectés entre eux, à travers un bus de contrôle de concurrence, pour leur permettre de se synchroniser pour les accès à la mémoire.
– **Encore Multimax (520)**. Il est organisé[Tab90] autour d'un seul bus synchrone et non-multiplexé, appelé le nano-bus, qui a une capacité de 100 Mo/sec. Sur ce bus sont connectées, au plus, dix cartes processeurs qui ont chacune deux CPUs et un cache de 256 Ko. permettant des accès mémoire très rapides (35 nsec.). Le cache est géré suivant un protocole d'écritures

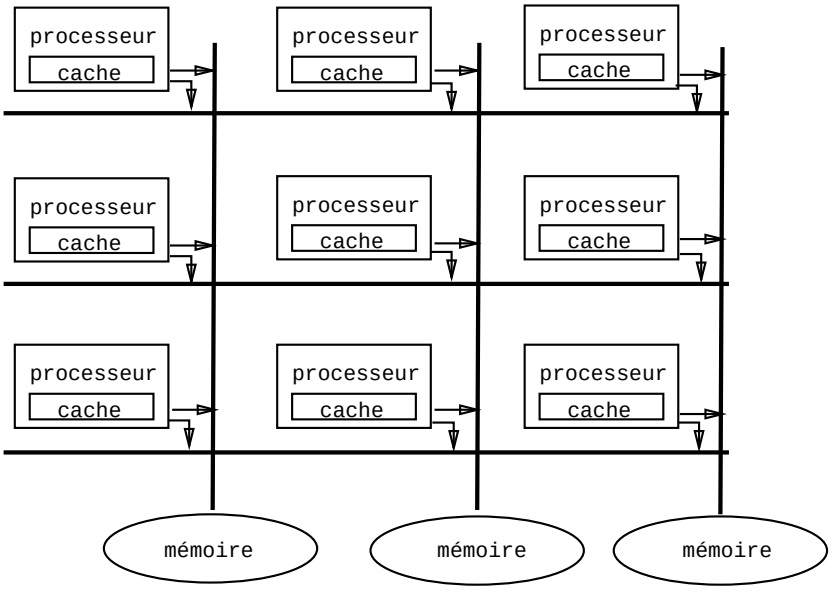

Figure I.3 : Multiprocesseur à plusieurs bus

différées. Ce protocole écrit les données dans le cache plutôt qu'en mémoire centrale. Il évite l'écriture en mémoire centrale aussi longtemps que possible. Les données peuvent alors être lues et écrites plusieurs fois dans le cache avant d'être remplacées et, par conséquent, l'accès à travers le nano-bus disparaît pratiquement. La cohérence entre les caches est maintenue à travers un système de possession où le processeur qui veut écrire dans son cache doit obtenir la possession de l'addresse. Lorsqu'un processeur obtient cette possession, les autres processeurs invalident l'adresse correspondante dans leur propre cache.

– **Sequent Balance**. Il contient[Tab90] de deux à trente processeurs qui sont connectés en même temps que les modules de mémoire et les contrôleurs d'entrée/sortie à un bus. Chaque processeur a 8Ko de mémoire cache qui contient une copie du bloc mémoire lu le plus récemment. Des verrous implantés par la machine sont accessibles à l'utilisateur pour garantir l'exclusion mutuelle des structures de mémoire partagée.

1.1.2 Les multiprocesseurs à réseau de connexion

L'objectif des réseaux d'interconnexion (figure I.5) est de réduire la probabilité de conflit lorsqu'un grand nombre de processeurs émettent des requêtes d'accès à la mémoire. Une première solution consiste à relier processeurs et mémoires par une matrice d'interconnexion construite au moyen de commutateurs programmables.

Ce système a notament été utilisé sur la machine expérimentale C.mmp[Sa78] Chaque processeur possède un cache de 8 Ko, principalement utilisé par le système d'exploitation. Il accède à la

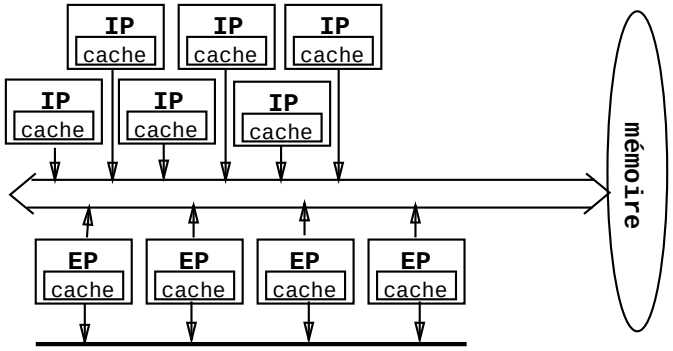

Figure I.4 : Alliant

mémoire à travers un réseau d'interconnexion qui permet à chaque processeur d'accéder à chaque unité mémoire (matrice d'interconnexion). Ainsi la bande passante de ce réseau augmente avec le nombre de processeurs. Son inconvénient majeur est que le nombre de commutateurs, donc la complexité et le coût du réseau, croissent comme le carré du nombre de processeurs. Un bus supplémentaire permet aux processeurs de communiquer entre eux, pour se synchroniser, autrement que par mémoire partagée.

Les réseaux multi-étages tels que le réseau oméga (figure I.6), présenté en 1975 par Lawrie, ou le réseau n-cube évitent cet inconvénient. Un réseau oméga est construit à partir de commutateurs élémentaires à k entrées et k sorties (en général on prend k=2), implantant 4 fonctions de base : tout droit, échange, diffusion haute, diffusion basse. Ces commutateurs sont connectés suivant le modèle du "perfect shuffle". Il peut y avoir conflit entre deux processeurs pour l'accès à un commutateur, ce qui est impossible dans une matrice d'interconnexion.

Dans un réseau n-cube les commutateurs n'ont que deux positions. D'autres structures de réseaux (réseau de Benes) évitent totalement les conflits d'accès au prix d'une structure plus complexe que celle des réseaux oméga.

Un exemple de multiprocesseur utilisant des réseaux multi-étages est :

– **BBN Butterfly**. Le Butterfly[Tab90] est composé d'un nombre variable (jusqu'à 256) de noeuds identiques et indépendants. Ces noeuds sont interconnectés à travers un réseau à grande vitesse (figure I.7). Les noeuds de base disposent d'une unité centrale et d'une unité flottante. Le processeur accède à 4 Mo. de mémoire locale à travers un contrôleur de noeud qui gère cette mémoire locale et l'accès du processeur aux mémoires distantes (à travers le réseau). En fait, toute référence mémoire est traitée par le contrôleur de noeud et c'est lui qui décide d'accéder à une mémoire distante ou locale donc d'utiliser le réseau ou non. Les contrôleurs coopèrent pour l'accès à une mémoire distante.

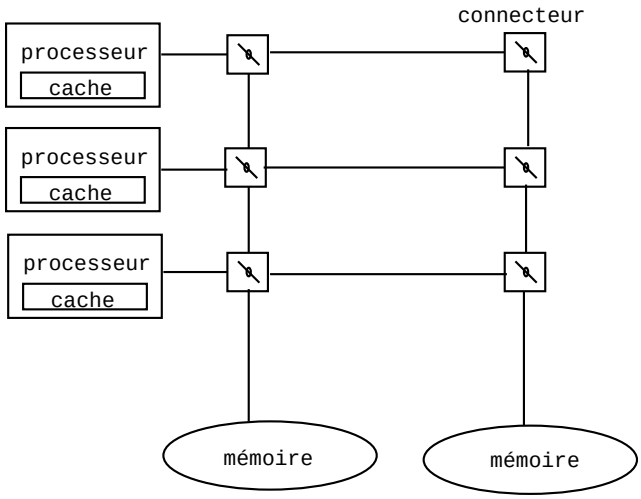

Figure I.5 : Multiprocesseur à réseau de connexion

1.2 Les multiprocesseurs à mémoire distribuée

Avec les multiprocesseurs à mémoire distribuée (ou multicalculateurs), on cherche à éliminer la complexité matérielle (réseau d'interconnexion à étages) ou logicielle (gestion des caches multiples) induites par l'accès de processeurs multiples à une mémoire commune. Les multiprocesseurs à mémoire distribuée sont constitués d'éléments de calcul autonomes, appelés noeuds, réunissant processeur et mémoire locale. Les noeuds sont reliés par un réseau d'interconnexion qui leur permet d'échanger des données. Ce système doit essentiellement permettre la communication par messages entre les élèments et se réduit donc à des liaisons simples. Cette architecture permet à la fois d'éviter le goulet d'étranglement de l'accès concurrent à la mémoire et d'augmenter la capacité du calculateur à échanger des informations en augmentant son nombre de noeuds ou de connexions.

L'interconnexion complète de N éléments, qui nécessite de l'ordre de N puissance deux liens, est impraticable pour un grand nombre de processeurs. On choisit donc des méthodes d'interconnexion indirectes qui permettent de réduire le nombre de liens, au prix d'un temps de communication plus élevé pour les éléments entre lesquels il n'existe pas de lien direct. Pour un multiprocesseur à mémoire distribuée, son diamètre est défini par la distance maximale entre deux de ses noeuds.

Un des avantages très net de cette architecture est la possibilité de faire évoluer le nombre de noeuds à des ordres de grandeur différents sans pour autant altérer le comportement du système. Nous définissons le débit d'un réseau comme étant la quantité de données qu'il peut transmettre en un temps fixe. Alors l'augmentation du nombre de processeurs entraîne une augmentation du débit du calculateur puisqu'elle ajoute des connexions. Cette augmentation du débit permet de satisfaire les besoins des noeuds ajoutés.

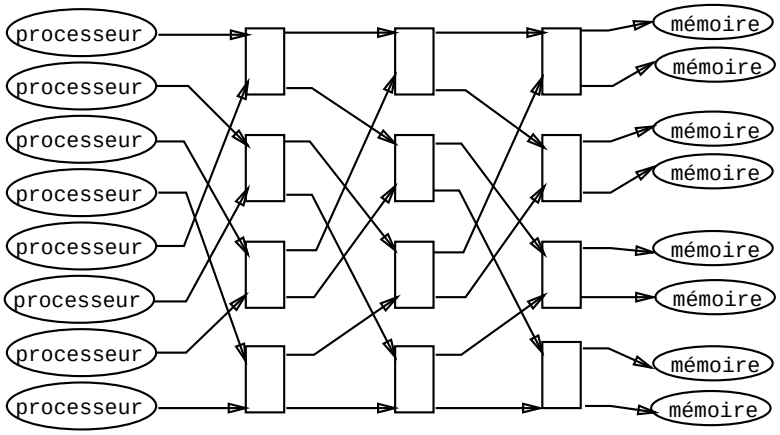

Figure I.6 : Réseau oméga

Pour un multicalculateur nous distinguons deux types de noeuds : les noeuds de base - composés d'un ou plusieurs processeurs, de la mémoire et de leur connexion au réseau - et les noeuds d'entrée/sortie - qui sont des noeuds de base auxquels sont ajoutés des organes d'entrée/sortie. Suivant la fonction attribuée au calculateur la configuration varie pour le type des noeuds d'entrée/sortie et leur nombre. Ainsi un calculateur plus particulièrement dédié aux bases de données aura probablement un ou plusieurs disques par noeud alors qu'un calculateur scientifique ne possèdera que quelques noeuds avec des disques.

Les deux modes d'interconnexion les plus usuels sont la matrice de processeurs et l'hypercube.

1.2.1 Matrice de processeurs

Dans une matrice de processeurs (figure I.8) le nombre de connexions disponibles pour chaque noeud est fixé à quatre, les noeuds frontière de la matrice n'en ayant que deux ou trois. Ce nombre n'évolue pas quelque soit le nombre de noeuds du multiprocesseur. L'avantage de cette architecture est qu'elle peut évoluer vers un nombre de noeuds d'un ordre de grandeur supérieur (de l'ordre de 10^6) sans perturbation importante pour les noeuds. Le routage y est relativement simple, à condition que chaque noeud connaisse sa place dans la matrice. Il peut être statique, c'est-à-dire qu'il utilise toujours le même chemin entre deux noeuds ou dynamique s'il peut évoluer en fonction de la charge ou des pannes du réseau. Le diamètre d'une matrice de processeurs est 2N pour une matrice de NxN.

Un exemple de matrice de processeurs est fourni la machine Paragon[Int87b], commercialisée par Intel. La structure de la matrice est composée par des modules de routage (MRC) qui permettent la communication dans quatre directions. Les noeuds de traitement sont connectés à ces modules de routage. Chaque noeud est désigné, dans la matrice, par deux coordonnées (X et Y) qui sont utilisées dans le routage des messages. Un noeud est composé d'un processeur (i860

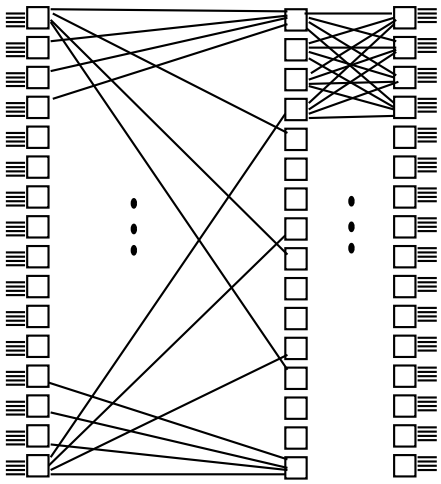

Figure I.7 : Réseau d'interconnexion du Butterfly

XP) qui est chargé de l'exécution, d'une mémoire locale dont la taille peut varier ente 16 et 64 Moctets, d'un processeur (i860 XP) de communication qui décharge le processeur principal du traitement des communications et d'un contrôleur d'interface réseau qui vérifie le transfert des messages. Intel prévoit également de commercialiser des machines Paragon dont les noeuds seront des multiprocesseurs à mémoire partagée.

1.2.2 Hypercubes

La principale limitation des matrices de processeurs est le temps de communication entre deux processeurs non adjacents. Dans les architectures à base d'hypercubes, on parvient à une croissance beaucoup moins rapide du diamètre, qui permet d'augmenter le nombre de processeurs connectés. Cette amélioration est obtenue au prix d'une augmentation du nombre de connexions.

Un hypercube (figure I.9) possède 2 puissance N éléments (processeurs et mémoire), placés aux sommets d'un hypercube à N dimensions. Un hypercube de dimension N est construit à partir de deux hypercubes de dimension N-1, en reliant deux à deux les élèments homologues. Chaque élément est relié à N voisins. Le diamètre d'un hypercube de dimension N est N, ce faible diamètre rend cette architecture très intéressante pour un diamètre peu important (le maximum utilisé actuellement est une quinzaine) car le coût du réseau croît avec le diamètre. L'hypercube est également intéressant pour les facilités de routage, tant statiques que dynamiques, que sa structure implique.

Plusieurs constructeurs se sont tournés vers l'architecture hypercube pour développer des machines pouvant aller jusqu'à 32000 processeurs.

(1) **Intel** : Intel a produit trois générations de machines basées sur ce modèle :

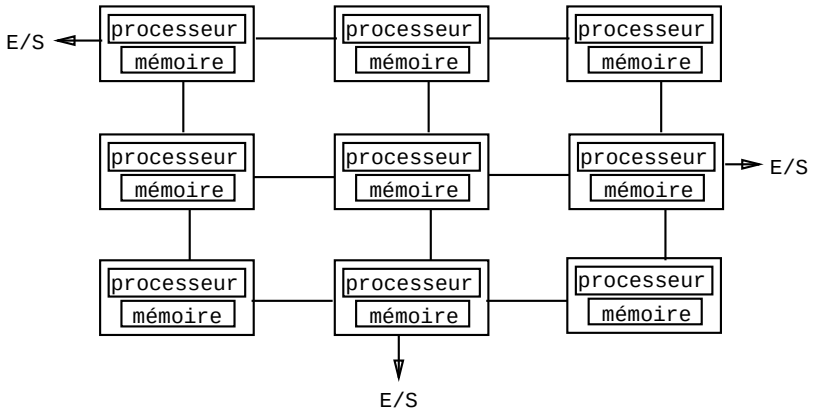

Figure I.8 : Matrice de processeurs

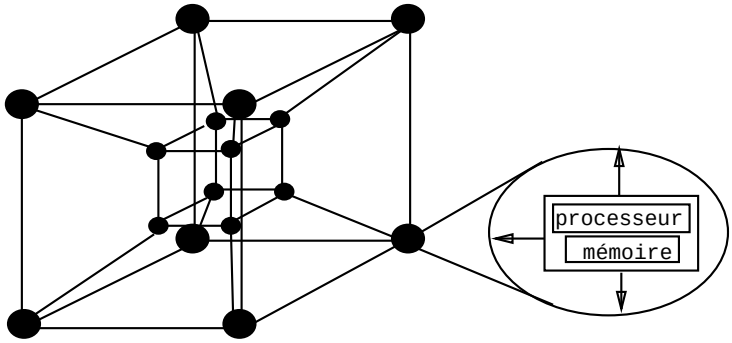

Figure I.9 : Hypercube

- l'iPSC, qui fut l'un des premiers hypercube commercialisé, était basé sur un processeur Intel 286, complété par un coprocesseur Intel 287. Chaque noeud possédait sept connexions vers d'autres noeuds plus une connexion Ethernet qui lui permettait de communiquer avec une station maître. L'échange de messages était asynchrone et se faisait entre voisins. La configuration maximale comprenait 128 noeuds gérés par une station maître (PC/AT).
- l'iPSC/2[Int87a] est basé sur un micro-processeur 80386. Une description détaillée de l'iPSC/2 est donnée au paragraphe 2.
- L'iPSC/860 est très semblable à l'iPSC/2 pour son réseau d'interconnexion puisque les modules de routage sont presque les mêmes. La différence la plus marquante entre les deux machines est le micro-processeur gérant les noeuds qui est, dans ce cas un i860, beaucoup plus performant que le 80386.

(2) **nCUBE** : Les machines développées par la société nCUBE reposent aussi sur une archi-

tecture d'hypercube. Pour le moment deux générations se sont succédées :

- La gamme nCUBE comprennait trois produits, tous basés sur une structure d'hypercube, offrant 4, 128 ou 1024 noeuds. Le processeur de base est un processeur fabriqué par nCUBE.
- La gamme nCUBE 2 [nCU90] offre des machines pouvant atteindre 8192 processeurs, eux aussi développés par nCUBE. Les modules de communication assurent un routage automatique des messages entre les noeuds du nCUBE 2 ; l'échange de messages étant asynchrone. Le débit peut atteindre jusqu'à 2,22 Moctets/secondes sur les canaux de communication. La configuration d'une machine nCUBE 2 est extensible. Des noeuds d'entrée/sortie peuvent être ajoutés aux noeuds de calcul pour permettre la connexion de disques, lecteurs de bande, etc.

(3) **Floating Point Systems** : Les séries T de Floating Point Systems (FPS)[Tab90] offrent des hypercubes pouvant avoir jusqu'à 16364 noeuds. Chaque noeud, appelé carte vectorielle, est composé d'un transputer T414 d'Inmos qui contrôle la mémoire centrale (256K) et d'une unité vectorielle (VPU). Ces noeuds sont regroupés en modules ; chacun des modules comprenant huit cartes vectorielles, une carte système (contrôlée par un T414) et un disque.

La machine de FPS, comme les autres hypercubes, est basée sur le modèle station maître-machine dédiée.

Deux réseaux relient alors les noeuds de la machine FPS : le réseau système et le réseau hypercube. Le réseau hypercube sert à l'utilisateur pour les échanges de messages entre les parties de son application. Pour obtenir un plus grand nombre de connexions que celui offert par le transputer, les canaux sont gérés par un multiplexeur qui offre ainsi 16 canaux bidirectionnels (env. 1 Mo./s. dans chaque sens, quatre peuvent être actifs en même temps) et un échange de messages synchrone. Le réseau système est utilisé par le système d'exploitation. Il connecte la station maître à une carte système pour les communications avec l'extérieur, les cartes systèmes entre elles (anneau) et les huit cartes vectorielles pour les communications entre noyaux.

1.2.3 Bus et anneaux

Ils ne résolvent pas très bien les problèmes de concurrence d'accès car la bande passante n'augmente pas avec le nombre de processeurs, néanmoins cette architecture (figure I.10) est moins onéreuse et elle peut être utilisée par des applications qui ne communiquent pas beaucoup. Cette architecture est, par exemple, celle de la machine Loral Dataflow LDF 100, basée sur un modèle data-flow à large grain. Elle peut comprendre jusqu'à 256 noeuds.

1.2.4 Divers

Comme nous l'avons dit au début de cette partie, notre propos n'est pas de détailler toutes les architectures de calculateurs parallèles existantes. Nous présentons cependant quelques architectures plus particulières afin de bien montrer que l'architecture parallèle est très variable, voir même reconfigurable.

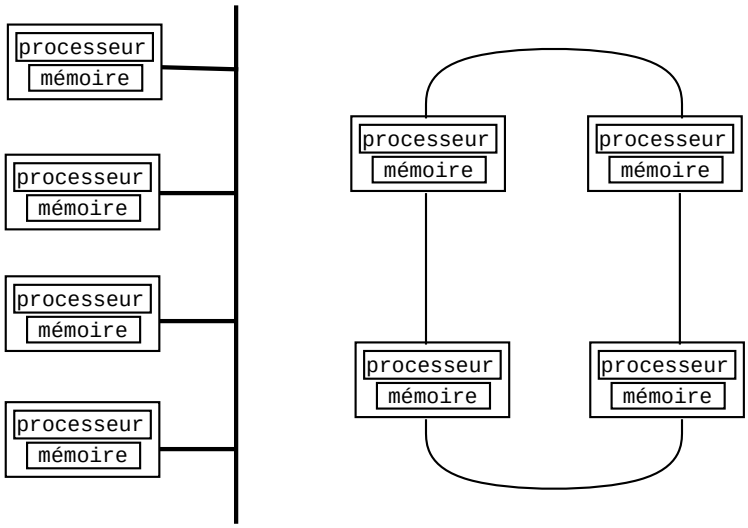

Figure I.10 : Connexion en bus et en anneau

La structure d'arbre : elle est peu répandue, elle peut être réalisée à partir de réseaux reconfigurables du type des transputers. Cette structure (figure I.11) est très centralisée mais elle peut néanmoins être adaptée pour un type particulier d'algorithmes. Ainsi Thinking Machine Corp. commercialise la CM-5[Thi92], basée sur un réseau en arbre.

La connexion en tore : elle est peu différente de la matrice de processeurs, les noeuds frontière de la matrice sont connectés entre eux afin de former un tore.

Les réseaux reconfigurables : ils permettent de définir la structure du réseau en temps d'exécution. Le réseau formé ne correspond alors pas forcément à une structure standard de réseau mais il peut ainsi être configuré pour être adapté au mieux à l'application qu'il supporte.

Un exemple de réseau reconfigurable donné par les réseaux utilisant les modules de routage développés pour des machines telles que le Tnode de Telmat ou le GC de Parsytec[Par91]. Ces deux machines utilisent des transputers pour processeurs de base.

Un Transputer (INMOS) est une unité élémentaire de traitement comportant un processeur, une mémoire locale, et 4 canaux d'entrée/sortie qui permettent à chaque transputer de communiquer avec ses quatre voisins dans une organisation matricielle. Le transputer T800[Inm89] implante une communication par octet en utilisant un protocole simple, ce qui permet un débit moyen de 5Mbit/s. La communication entre processeurs adjacents est fondée sur le principe de rendez-vous, donc synchrone.

Dans le Tnode, tous les canaux des transputers sont connectés à un commutateur. En configurant le commutateur l'utilisateur peut obtenir la configuration qu'il désire. Ces modules de routage

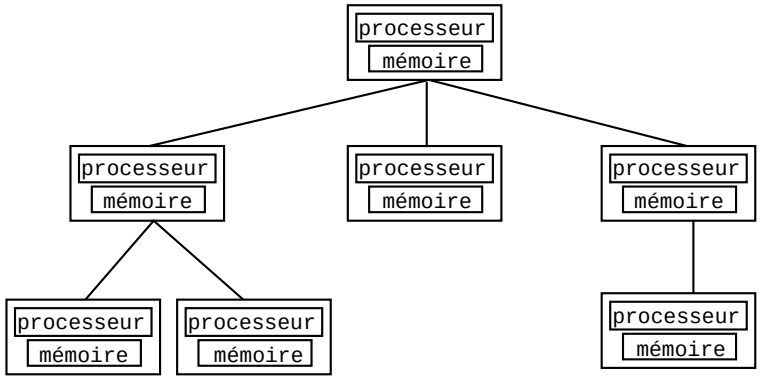

Figure I.11 : Connexion en arbre

permettent la connexion de 16 transputers. La connexion de plusieurs de ces commutateurs permet d'obtenir des machines ayant jusqu'à 1024 noeuds.

La future génération des machines GC de Parsytec utilisera les transputers T9000[Inm91] interconnectés par des modules de routage (C104) qui permettront la connexion de 16000 transputers. De plus, ces transputers offrent la possibilité de définir des canaux virtuels reconfigurables entre processeurs non adjacents. Elle implante ainsi des possibilités de routage automatique entre deux processeurs. L'interconnexion est alors transparente à l'utilisateur.

2 Architecture de l'iPSC/2

2.1 Vue générale

L'iPSC/2[Int87a] est basé sur une architecture hypercube. Il est commercialisé par Intel Scientific Supercomputer Division (ISSD). La configuration de base de l'iPSC/2 sur lequel nous avons travaillé dispose de 16 noeuds mais ISSD propose des iPSC/2 incluant jusqu'à 128 noeuds.

Comme la plupart des multicalculateurs actuels l'iPSC/2 apparaît à l'utilisateur comme le coprocesseur d'une station maître. Dans le cas de l'iPSC/2, cette station maître est un PC/AT appelé SRM (System Resource Manager). Les noeuds communiquent entre eux par un module de communication spécialisé - le Direct Connect Module (DCM) - qui assure automatiquement le routage entre les noeuds. A chaque noeud, formant la structure hypercube de l'iPSC/2, peut être connecté - par une ligne DCM - un noeud gérant un périphérique. Sur l'iPSC/2 ce périphérique peut être, par exemple, un disque ou une carte Ethernet. Le noeud gérant le périphérique est appelé noeud d'entrée/sortie et il est connecté à un noeud de la structure hypercube. Notre configuration dispose d'un noeud d'entrée/sortie, connecté au noeud 2, qui gère deux disques.

2.2 Les noeuds

Chaque noeud[Clo88] de l'iPSC/2 est une carte micro-processeur (figure I.12), numérotée suivant sa place dans la structure hypercube (figure I.13), qui est composée de :

- un processeur 80386, cadencé à 20MHz.
- éventuellement un coprocesseur mathématique qui peut être du type Weitek 1167 ou Intel 80387 SX.
- une mémoire vive (DRAM) de 4 à 16 Mo. accédée à travers un cache de 64 Ko. (SRAM). Notre configuration dispose de 12 Mo. de mémoire vive. Les échanges de mémoire entre le processeur et le module de communication sont gérés par un processeur spécialisé (ADMA).
- un module de communication (DCM) qui assure l'acheminement des messages d'un noeud à l'autre (c.f. 2.3).
- une interface ILBX, compatible SCSI, permettant l'adjonction d'un disque, d'une carte de communication ou encore d'un processeur vectoriel.
- une connexion vers la station maître (USM), qui est gérée par une UART (c.f. 2.4).

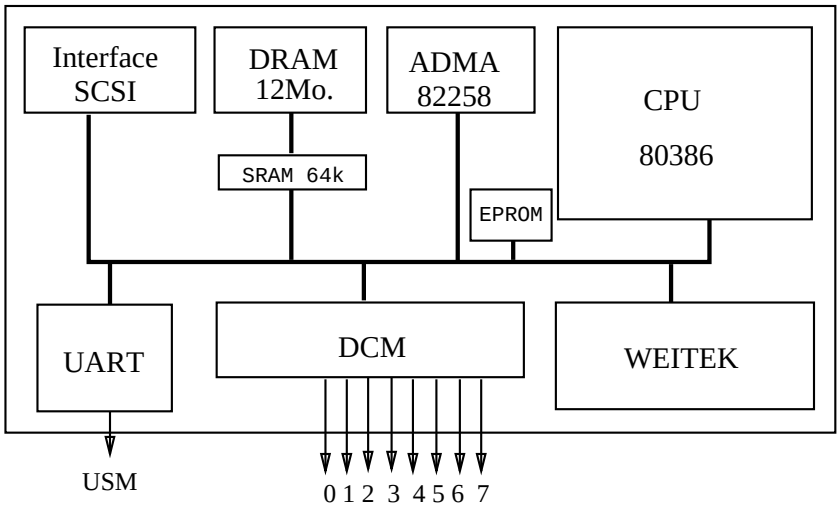

Figure I.12 : Détail d'un module de base

2.3 Le module de communication

Le module de communication (DCM) assure l'acheminement direct des messages d'un noeud à l'autre, sans que l'utilisateur n'ait à gérer le routage. A ce titre le réseau de connexion de l'iPSC/2 apparaît comme complet à l'utilisateur puisque l'envoi de message est direct et que les temps de transmission entre deux noeuds adjacents et deux noeuds éloignés sont quasiment équivalents.

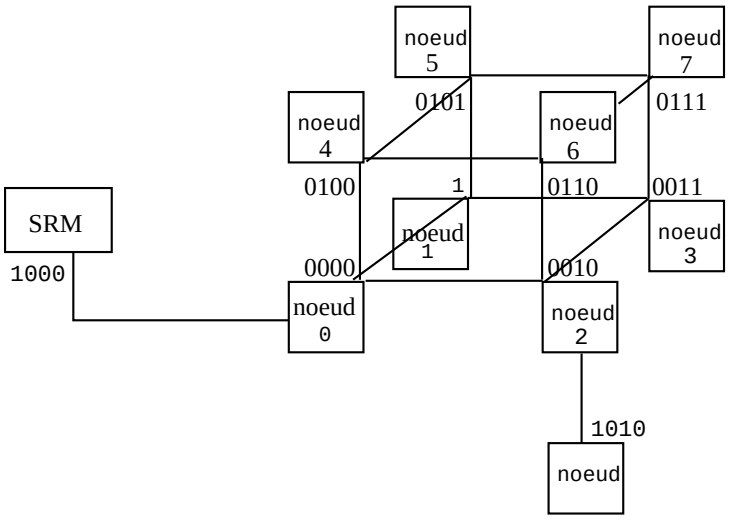

Figure I.13 : Architecture du DCM

Le routage Chaque DCM offre huit canaux qui permettent d'établir la connexion avec les autres noeuds. En fait un canal représente une direction permettant d'atteindre le noeud dont la représentation binaire est égale à un *ou exclusif* (`xor`) entre celle du noeud de départ et celle de la direction. Par exemple, sur la figure I.13 nous pouvons constater que le noeud 2 (010) est connecté au noeud 6 (110) par le troisième canal de direction 100. Au plus bas niveau, le routage est obtenu en calculant un `xor` entre la représentation binaire du noeud source et celle du noeud destinataire. Le résultat obtenu permet de savoir sur quel canal doit être envoyé le message : le premier noeud l'envoie sur le canal correspondant, dans la représentation binaire du résultat, au premier 1 rencontré, le second le transmet sur le canal correspondant au 1 suivant, et ainsi de suite jusqu'au dernier 1 qui permet d'atteindre le noeud destinataire. Tout ceci est directement réalisé par le matériel. Le routage est donc statique et ne peut être adapté si un des canaux utilisé est bloqué dans l'échange d'un autre message. En fait, avant l'échange effectif du message, se déroule une première phase au cours de laquelle le DCM réserve la totalité des canaux qui seront utilisés pour l'échange des données. Une fois le chemin obtenu, le DCM transmet les données. Par exemple pour envoyer un message du noeud 0 au noeud 6, nous calculons *000* `xor` *110* ce qui fait *110*. Le message emprunte alors le troisième canal du noeud 0 qui est relié au noeud 4 puis le deuxième canal du noeud 4 qui est relié au noeud 6.

Parmi les canaux du module de routage seuls les sept premiers sont utilisés pour former la structure hypercube. Le huitième canal étant réservé, par convention, pour connecter un noeud d'entrée/sortie. Ainsi l'accès aux noeud d'entrée/sortie est aussi direct.

Le transfert des données Les lignes reliant les DCM sont des liens bi-directionnels pouvant atteindre un débit de 2,8 Mo/s. Chaque DCM est étroitement couplé à l'ADMA pour la transmis-

sion des données. Ainsi, lors d'un envoi de message, les données à échanger sont directement lues en mémoire centrale par le DCM, qui ne possède pas de mémoire propre. L'utilisateur est prévenu de la fin de l'envoi par une interruption mais aucune garantie n'est donnée sur la réception du message.

Entre les DCMs l'envoi de message est asynchrone. C'est-à-dire que le DCM ne se soucie pas de savoir si le destinataire est prêt à recevoir. Ceci implique que l'utilisateur ait alloué un espace mémoire pour la réception avant que celle ci se produise. Du point de vue de l'utilisateur l'échange de message est géré en allouant une place mémoire au DCM pour qu'il puisse recevoir les messages. Il est ensuite prévenu de la réception d'un message par une interruption générée par le DCM.

2.4 La station maître

La station maître (SRM) est une station de travail Intel du type PC/AT basée sur un microprocesseur 80386 cadencé à 15 MHz. Cette station dispose d'une connexion Ethernet vers un réseau de stations de travail. En effet la puissance de cette station n'est pas suffisante pour pouvoir accueillir tous les utilisateurs de l'hypercube ; elle se révèle d'ailleurs souvent être un goulet d'étranglement pour l'échange de données entre les noeuds de l'hypercube et les stations de travail distantes.

La communication entre les noeuds et le SRM est assurée par deux types de lignes séries :

– **une ligne USM** (Unit Service Module) de type RS422. Il y a une connexion par l'USM (figure I.14) entre chaque noeud et le SRM. En fait, quand la station maître écrit sur l'USM tous les noeuds lisent la même information et il ne peut y avoir qu'un seul noeud à la fois qui écrit sur l'USM.

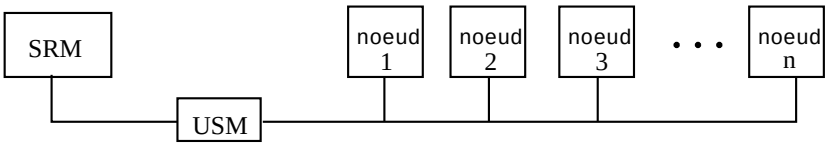

Figure I.14 : Architecture de l'USM

  Une ligne USM peut servir pour communiquer avec, au plus, 32 noeuds. Certaines configurations d'iPSC/2 disposent donc de plusieurs lignes USM.

– **la ligne DCM** qui est directement connectée au noeud 0. Cette ligne établit une connexion point à point entre la station maître et le noeud zéro, de ce fait la station maître peut apparaître comme un noeud de l'hypercube puisqu'elle sera accédée de la même façon. Le SRM est connecté au noeud zéro par le même type de lien (DCM) que celui reliant les autres noeuds. A partir du noeud zéro il est perçu donc comme un noeud d'entrée/sortie.

Le SRM dispose d'une carte (301 interface board) mise sur le fond de panier qui lui permet de dialoguer avec l'USM. Ceci permet au SRM de contrôler les noeuds. En particulier pour la réinitialisation de l'hypercube il utilise cette ligne pour tester les noeuds, les ré-initialiser puis

charger le système d'exploitaion qui les gérera.

Conclusion

Nous avons vu que les multicalculteurs représentent une alternative intéressante pour augmenter la puissance de calcul des super-calculateurs. Cependant avec cette nouvelle classe d'architectures de nouveaux problèmes se posent. Par exemple un trop grand déséquilibre entre les trois valeurs de base qui caractérisent la machine (flux de communication, taille mémoire par noeud et puissance du processeur) induit que l'une des ressources devient un goulet d'étranglement. De même, le nombre et la répartition des périphériques jouent un rôle très important dans l'amélioration des performances. Ces architectures étant modélisables, des études ont été menées pour savoir quel type de routage serait le mieux adapté, quelle capacité mémoire convient à chaque noeud ou encore quelle capacité de stockage doit être allouée [Eck90].

Pour un très grand nombre de processeurs l'architecture matricielle semble éveiller un intérêt chez les constructeurs de super-calculateurs puisque les machines décrites par Intel et OMI comme devant atteindre des performances de l'ordre du TeraFlops sont basées sur cette architecture. Les routages sont alors dynamiques. Dans les nouvelles machines le diamètre devient un faux problème car les temps de communication pour la transmission des données d'un noeud à l'autre chutent par rapport aux temps passés à encapsuler le message pour qu'il réponde aux normes du protocole logiciel utilisé. D'autre part les nouveaux modules de routage (Inmos, Intel) intègrent le routage à bas niveau, donnant à l'utilisateur la vision d'un réseau complet.

Cependant pour faire bénéficier de nombreuses applications du rapport coût/performances intéressant que suppose ces architectures, celles-ci doivent offrir les logiciels qui permettront de les exploiter au mieux. En effet, la répartition des ressources sur le réseau rend l'utilisation de ces machines plus complexe que celle d'un monoprocesseur ou d'un multiprocesseur à mémoire partagée. Le système d'exploitation, qui constitue la base de support de l'ensemble du logiciel, doit alors fournir les services sur lesquels pourront s'appuyer les utilitaires et les applications. Nous abordons dans les chapitres suivants les problèmes des systèmes d'exploitation dédiés aux multicalculateurs.

Chapitre 2

Les systèmes d'exploitation répartis

Introduction

L'apparition de technologies telles que les multicalculateurs donne de nouvelles possibilités aux utilisateurs, principalement au niveau des performances. Cependant ces nouvelles technologies, notamment dans le cas des architectures à mémoire distribuée, impliquent des changements importants dans les savoir faire par rapport aux architectures monoprocesseurs traditionnelles, en particulier au niveau de la programmation et du mode d'utilisation. L'adaptation à ces nouvelles pratiques suppose un effort important pour concevoir des systèmes d'exploitation et des environnements de programmation permettant une utilisation plus proche de celle des monoprocesseurs traditionnels ou des systèmes multiprocesseurs déjà classiques, organisés autour d'une mémoire centrale commune. Dans ce chapitre nous nous intéressons plus particulièrement au cas du système d'exploitation. En particulier, nous étudions différents systèmes d'exploitation répartis dans le but d'évaluer leur potentialité à gérer un multicalculateur et à servir de support au développement de nouveaux services, mieux adaptés aux besoins des utilisateurs.

Dans la partie 1 nous analysons les lacunes des systèmes d'exploitation qui gèrent actuellement les multicalculateurs. Dans la partie 2 nous tentons une classification des propriétés des systèmes d'exploitation en fonction de leur importance dans la répartition et de leur utilité dans le cas d'un multicalculateur. Nous étudions ensuite les systèmes d'exploitation existant actuellement par rapport aux propriétés énoncées en 2. Dans la partie 3 nous décrivons le système CHORUS sur lequel nous avons travaillé. Cette description nous permet de situer quelques uns des systèmes d'exploitation répartis, parmi les plus marquants, par rapport aux fonctionnalités de CHORUS. Nous insistons plus particulièrement, en 4, sur la comparaison des fonctionnalités des trois micronoyaux implantant un système d'exploitation réparti : Amoeba, CHORUS et MACH. Dans la partie 5 nous décrivons d'autres systèmes qui implantent des services intéressants pour les multicalculateurs.

1 Les systèmes natifs

Nous appelons *systèmes natifs* les systèmes d'exploitation actuellement proposés sur les multicalculateurs. La caractéristique principale de tous ces systèmes est que ce sont des systèmes dédiés. Ces systèmes sont adaptés à l'exécution de programmes parallèles écrits par des utilisateurs avertis qui gèrent eux-mêmes les contraintes dues à la répartition de leurs programmes. Le but de cette partie n'est pas de critiquer les systèmes natifs ou leur conception, étant entendu qu'ils ont été développés pour satisfaire des besoins qui sont différents des nôtres. Nous recensons les points qui nous paraissent s'opposer à une utilisation plus répandue des multicalculateurs.

Allocation statique des noeuds Une des restrictions imposées par les systèmes d'exploitation natifs nous semble incompatible avec une utilisation dans un contexte d'utilisation général : le partage de la machine entre plusieurs utilisateurs - sans en fixer le nombre a priori - n'est pas possible, car le partage d'un noeud entre deux utilisateurs n'est pas autorisé. Cette restriction peut être justifiée dans le cas d'applications coûteuses en temps de calcul - dans la mesure où l'obtention de bonnes performances passe par la disponibilité maximum du matériel au logiciel - mais elle est un frein à une utilisation du multicalculateur dans un contexte plus général puisqu'elle n'assure pas, à l'utilisateur, la disponibilité des ressources. Par exemple, le système NX/2[Pie06] qui gère l'iPSC/2 oblige l'utilisateur à réserver un nombre fixe de noeuds avant de lancer une exécution. Le système Helios[Gar87] réserve lui-même les noeuds au début de l'exécution mais il ne permet pas le partage.

On peut assez facilement établir un parallèle entre le début des multicalculateurs et le début des machines monoprocesseur, même si certains raisonnements sont parfois simplistes. Lors de l'apparition des monoprocesseurs, l'utilisateur devait attendre que le processeur soit disponible pour lancer son programme. L'allocation statique des noeuds avant une exécution sur l'iPSC/2 conduit à la même contrainte. La soumission des exécutions en batch apparait avec l'iPSC/860. En ce qui concerne les monoprocesseurs, l'évolution qui a suivi est l'apparition des systèmes multi-utilisateurs. Un tel service, sur les multicalculateurs, suppose une allocation dynamique des noeuds et la possibilité de partager un noeud entre plusieurs utilisateurs. Enfin, la dernière évolution, en terme d'allocation de temps d'exécution, sur les monoprocesseurs est le temps partagé, offert pour les systèmes multitâche. Pour offrir ce niveau d'allocation sur un multicalculateur il faut que chaque processeur soit géré par une politique de temps partagé qui permette de choisir le site d'exécution des différents composants d'une application. Bien sûr ces dernières fonctionnalités posent des problèmes beaucoup plus complexes sur les machines parallèles que sur les monoprocesseurs car la mise en place d'une politique d'allocation des processeurs dans le cas distribué nécessite l'implantation d'un ordonnancement global.

Dépendance vis-à-vis de l'architecture Les systèmes natifs favorisent l'écriture de programmes dépendants de l'architecture, plus particulièrement du nombre de noeuds ou de l'architecture de la machine. Ainsi sur l'iPSC/2, l'utilisateur charge le plus souvent une même instance de programme sur tous les noeuds. Il teste ensuite le numéro du noeud sur lequel il s'exécute pour déterminer la partie de code qui doit être exécutée. Il est évident qu'un tel modèle de program-

mation ne permet pas de tirer partie d'un nombre de noeuds différent et est donc peu extensible. De même, dans le système Helios, l'utilisateur doit fournir, à la compilation de son application, le placement de ses processus sur le réseau de la machine. L'utilisateur tient compte de ce placement dans l'écriture de l'application.

Confort d'utilisation Le manque de confort peut également être compté au nombre des handicaps des systèmes natifs. En effet l'interface de ces systèmes est, en général, très restreinte. Par exemple, même sur des plateformes offrant une base pour la gestion d'une mémoire virtuelle (i386 pour l'iPSC/2) ces systèmes n'offrent que peu de possibilités : pas de mapping, pas de partage, etc. La transparence offerte par ces systèmes est réduite dans la mesure où un envoi de message est fait d'un site à l'autre en précisant le numéro de site. Certains développements complètent l'interface réduite de ces systèmes pour offrir un niveau de fonctionnalités plus proche de celui des systèmes répartis présentés en 2 : nous pouvons citer [BB88]. Dans cet esprit, il est donc indispensable d'apporter une attention toute particulèire à la conception d'un système d'exploitation pour les multicalculateurs.

De plus, les systèmes d'exploitation doivent faciliter l'adaptation des utilisateurs à ces nouvelles architectures de multicalculateurs. En particulier, l'interface du système d'exploitation doit répondre aux standards auxquel l'utilisateur est habitué et l'accès à la machine doit être facilité par le support des fonctionnalités multi-utilisateurs.

Pour offrir une interface de système d'exploitation qui prenne en compte ces contraintes il y a deux solutions : adapter un système natif pour y intégrer les services nécessaires ou porter un système offrant ces services sur les multicalculateurs. La première solution implique d'intégrer dans le système natif des services qui existent déjà dans d'autres systèmes et de modifier les services existants pour qu'il puissent supporter les nouvelles fonctionnalités. Ceci revient a réécrire une grande part du système d'exploitation. Par contre la seconde solution n'implique qu'une modification des couches basses du système et de quelques services. L'utilisateur bénéficierait ainsi - à moindre frais - d'une interface standard et de services répartis tels que des systèmes de fichiers répartis (ex : NFS) et de facilités d'exécution distantes (ex : rsh). Cependant l'interface de ces systèmes offre généralement peu de fonctionnalités, telles que la communication par messages, qui permettent une programmation adaptée aux multicalculateurs. Pour cette raison, les systèmes d'exploitation répartis semblent mieux adaptés à de tels ordinateurs. Nous avons donc, dans cet esprit, étudié différents systèmes d'exploitation répartis et leur potentialité à satifaire, sur un multicalculateur, les besoins de la plupart des utilisateurs.

2 Généralités

Un **site** est un ensemble de ressources physiques fortement couplées : processeurs, mémoire, périphériques. Une station de travail ou un multiprocesseur à mémoire partagée sont des exemples de sites. Un **système réparti** est un ensemble de sites faiblement couplés. Un réseau de stations de travail ou un multiprocesseur à mémoire distribuée constituent un système réparti.

Le but d'un système d'exploitation est de fournir à ses utilisateurs une interface simple pour

accéder aux ressources du matériel qu'il gère. Un système d'exploitation est donc une couche logicielle gèrant les ressources d'un ordinateur ou d'un réseau d'ordinateurs et offrant à l'utilisateur des fonctions d'accès à ces ressources. Pour cette raison il est facile d'établir un parallèle entre l'évolution des systèmes d'exploitation et l'évolution du matériel qu'ils gèrent. Nous nous intéressons ici plus particulièrement aux systèmes d'exploitation répartis. Leur évolution peut être ordonnée comme suit :

- *LES SYSTEMES D'EXPLOITATION CENTRALISES MONO-UTILISATEURS* :
  Les premiers micro-ordinateurs peu puissants et possédant trop peu de mémoire supportent des systèmes d'exploitation centralisés mono-utilisateurs c'est-à-dire que ces systèmes d'exploitation ne gèrent qu'un seul site pour le compte d'un seul utilisateur. Le système MS/DOS [Mer89] est un exemple de système d'exploitation centralisé mono-utilisateur.
- *LES SYSTEMES D'EXPLOITATION CENTRALISES MULTI-UTILISATEURS* :
  L'apparition de processeurs disposant d'une gestion de mémoire virtuelle, d'un plus grand espace d'adressage et aussi d'une plus grande puissance de traitement a permis au système d'exploitation de supporter les requêtes de plusieurs utilisateurs ou plusieurs requêtes du même utilisateur concurremment. Pour cela, a été introduite la notion de **contexte d'exécution concurrent** qui correspond à la sauvegarde de l'état du processeur et des données propres à chaque processus : en restaurant un contexte d'exécution le système d'exploitation permet de retrouver l'environnement dans lequel le processus s'exécute. Dans l'évolution des systèmes d'exploitation vers la répartition nous différencions parmi les systèmes d'exploitation centralisés ceux qui supportent plusieurs utilisateurs de ceux qui n'en supportent qu'un, car l'apparition de la notion de contexte d'exécution concurrent permet d'introduire un parallélisme virtuel et ainsi de se rapprocher des schémas qui seront ceux de la répartition. UNIX [RT74][TR79] est l'un des représentants les plus connus de cette classe de systèmes d'exploitation centralisés multi-utilisateurs.
- *LES SYSTEMES D'EXPLOITATION REPARTIS* :
  Avec l'apparition des réseaux de communication les systèmes d'exploitation ont à nouveau évolué pour intégrer cette nouvelle fonctionnalité. Des protocoles de communication permettent alors à des contextes d'exécution d'échanger des informations entre différents sites. Dans cette évolution nous pouvons discerner trois phases, suivant le niveau auquel le système d'exploitation intègre les protocoles de communication utilisant le réseau.
  - dans un premier temps les protocoles de communication n'ont été utilisés que par des applications particulières, dédiées à la mise en oeuvre de services répartis bien identifiés tels que le transfert de fichiers ou la connexion à distance sur un système. Cette phase n'implique pas de modifications importantes dans le système. Ceci est le cas d'un noyau UNIX auquel sont ajoutés les services de TCP/IP[Com88].
  - les protocoles de communication ont ensuite été intégrés au sein du noyau du système d'exploitation, pour étendre à la répartition certaines des fonctions du noyau. Cette approche, qui permet d'étendre les fonctionnalités de systèmes d'exploitation existants sans les modifier de façon importante, a notamment été appliquée au système UNIX, principalement dans le cadre de l'extension du système de fichiers à la répartition (ex : NFS[Ste86]).
  - certains systèmes d'exploitation intègrent les protocoles de communication au coeur du noyau

du système d'exploitation : tous les objets (processus, fichiers, etc.) sont potentiellement manipulables de façon répartie, quand ils ne sont pas des objets répartis, introduisant ici un premier niveau de transparence à la répartition.

Nous qualifions de système d'exploitation réparti tout système d'exploitation qui offre une possibilité d'échange avec une autre instance de système d'exploitation. Parmi les systèmes répartis nous pouvons également différencier ceux qui sont issus de l'interconnexion de systèmes centralisés de ceux qui ont été conçus en intégrant la répartition. Les seconds offrent généralement un niveau de transparence à la répartition qui est plus élevé.

Parmi ces trois classes, il est évident que seuls les systèmes d'exploitation répartis offrent un intérêt pour les multicalculateurs. Plus particulièrement ceux qui intègrent les protocoles de communication au coeur du noyau du système d'exploitation car ces systèmes ont été conçus pour gérer un environnement réparti.

Un système d'exploitation est caractérisé par les services qu'il offre et par son interface. Nous ne traitons pas ici des différences d'interface mais plutôt des fonctionnalités qui distinguent les systèmes d'exploitation. Le but des deux parties suivantes est de donner un aperçu des différentes abstractions, propriétés et services relatifs à la répartition. Ceux-ci ont été classés en trois parties : les abstractions que nous pensons nécessaires à l'utilisateur du système d'exploitation pour prendre en compte la répartition - nous les appelons concepts de base - les propriétés qui sont liées la répartition sans être indispensables à un système réparti et enfin les nouveaux services qui facilitent à l'utilisateur la prise en compte de la répartition. Nous ne traitons ici que les fonctionnalités liées à la répartition : les systèmes d'exploitation répartis doivent également implanter les fonctionnalités standards d'un système d'exploitation, c'est-à-dire les accès aux ressources locales (processeur, mémoire, disque ou autre périphérique) et la gestion de ces ressources doit également pouvoir être étendue pour prendre en compte leur répartition.

2.1 Concepts de base supportant la répartition

Un **système d'exploitation réparti** est un système d'exploitation qui gère les ressources d'un ensemble de sites faiblement couplés [Kra87] [BB90]. Il offre à l'utilisateur les fonctionnalités fondamentales qui lui permettent de décrire des applications en utilisant les ressources du système et leur répartition. Pour pouvoir décrire une application répartie il a besoin d'exprimer la concurrence d'exécution sur les différents sites et la communication entre les entités d'exécution. Pour chacune de ces fonctionnalités nous essayons de décrire les développements réalisés dans les systèmes d'exploitations répartis actuels.

2.1.1 Les entités d'exécution concurrentes

A la notion d'entité d'exécution est attachée au moins la notion de contexte d'exécution, c'est-à-dire l'état du processeur à un instant donné. La sémantique associée à l'entité d'exécution dépend du système d'exploitation et détermine la taille des données associées à cette entité.

Une des entités d'exécution parmi les plus connues est probablement l'abstraction de **processus** telle qu'elle est implantée par UNIX. Le système UNIX a largement contribué à la reconnaissance

de la notion d'entité d'exécution concurrente en offrant cette abstraction et les fonctions qui la manipulent au niveau de son interface. Cette notion a d'ailleurs facilité l'évolution d'UNIX vers la gestion de systèmes répartis. La notion de processus sert à décrire l'exécution d'un programme séquentiel. Elle associe au contexte d'exécution de ce programme l'espace d'adressage contenant son code, ses données, sa pile ainsi que les données du système d'exploitation associées à cette exécution. Par exemple, dans UNIX la sémantique d'un processus inclut les références sur les fichiers ouverts, les droits d'accès aux ressources, les périphériques associés, etc.

Cependant, si l'abstraction du processus convient bien au cas d'applications d'ordre général, elle est trop lourde et complexe dans certains cas. Par sa structure elle répond bien aux besoins de processeurs ne partageant pas de mémoire, puisque les processus eux mêmes en partagent rarement. Par contre elle s'avère peu intéressante pour implanter des processus temps-réel ou encore s'exécuter sur un multiprocesseur à mémoire partagée. D'autre part cette abstraction n'est pas bien adaptée à l'implantation d'un langage objet. En effet, elle contient beaucoup plus d'informations que n'en a besoin l'objet lui-même. La gestion d'objets, calqués sur des processus, coûte alors trop cher.

Dans les processus légers la notion d'espace d'adressage a été dissociée de celle de contexte d'exécution. Plusieurs processus légers peuvent alors s'exécuter dans un même espace d'adressage. Un processus léger ne mémorise que l'état des registres du processeur plus quelques données du système d'exploitation. Du point de vue des performances cette classe d'entité d'exécution permet d'effectuer des changements de contexte très rapides car peu de données y interviennent. Une autre structure du système contient alors l'espace d'adressage et accueille ces entités d'exécution. Cette structure correspond beaucoup mieux au modèle de multiprocesseurs à mémoire partagée ou aux besoins de concepteurs d'applications temps réel ou orientées objet. Ainsi un processus léger peut s'exécuter au sein d'un processus normal ; les données telles que les fichiers ouverts, les droits d'accès, etc., ainsi que l'espace d'adressage sont alors partagées entre les processus légers. Le processus léger trouve également son utilisation dans les traitements interactifs où une part du service attend les ordres extérieurs et l'autre part effectue les traitements. Les processus légers sont implantés, par exemple, dans les systèmes CHORUS et MACH où ils sont connus sont le nom de **threads**. Il existe d'autres implantations de processus légers pour UNIX faites sous forme de librairies (elles utilisent le même processus et les changements de contexte sont organisés dans le processus).

En plus de ces deux exemples connus il existe toute une gamme d'entités d'exécution, chacune avec ses données associées et sa sémantique. Nous définirons donc une **entité d'exécution** comme étant l'abstraction minimale offerte par le système pour contenir le contexte d'une exécution. Nous employons généralement le terme de processus pour désigner les entités d'exécution lorsqu'elles associent un, ou plusieurs, contextes d'exécution à l'espace d'adressage qui le supporte.

Les besoins des multicalculateurs, en terme d'entité d'exécution, dépendent des caractéristiques des noeuds. Un modèle "à la UNIX" peut convenir à des noeuds monoprocesseurs. Il est souhaitable d'offrir la notion de processus léger pour les noeuds multiprocesseurs.

2.1.2 La communication

Dans un environnement distribué la communication est à la fois la base qui permet au système d'intégrer la répartition et une fonctionnalité primordiale pour le développement des applications réparties. Pour pouvoir être utilisée par les systèmes d'exploitation dans la gestion des ressources la communication a été intégrée dans leur noyau.

La communication consiste en l'échange de données entre entités d'exécution. Cet échange ne pose pas de problèmes particuliers si les entités d'exécution partagent de la mémoire (il peut alors être réalisé en synchronisant les entités d'exécution sur une partie de la mémoire partagée). Si les entités ne partagent pas de mémoire, le système doit leur fournir un support pour cet échange de données. Le modèle de communication varie en fonction du système d'exploitation considéré.

Parmi les problèmes posés par l'implantation de la communication, nous discernons plusieurs classes : la structuration des données échangées, l'adressage d'une entité à l'autre, la sémantique liée à l'échange des données, la réalisation. Nous présentons ici différents choix d'implantation afin de donner un aperçu des solutions existantes aux différentes classes de problèmes.

La structuration Deux processus communicants échangent des flux de données. Ceux-ci ne sont pas toujours structurés par le système d'exploitation. Par exemple, dans le système UNIX deux processus peuvent communiquer par un *tube*. Lorsque ces deux processus échangent des données, le système n'intervient pas dans leur formatage. Ainsi, du point de vue du processus récepteur, le processus émetteur écrit sur le tube un flux non structuré de données. Le lecteur ne peut donc pas discerner la limite entre deux envois de données si aucune convention n'a été fixée entre les protagonistes.

L'abstraction de **message** est reconnue par, pratiquement, tous les systèmes d'exploitation répartis comme étant l'unité de structuration des données échangées entre deux processus. Pour le système d'exploitation, le message forme une entité indissociable. Si l'émetteur envoie un ensemble de données sous forme de messages, le récepteur reçoit ces données avec la même structure et non intercalées avec les données d'autres messages. L'abstraction de message peut elle-même couvrir différents niveaux de structuration. Par exemple, dans le système MACH les messages sont typés. C'est-à-dire que l'utilisateur du service de communication doit donner au système le type des données échangées. Par contre, d'autres systèmes d'exploitation tels que CHORUS gèrent le contenu des messages comme des suites d'octets.

Le degré de structuration des messages induit un certain niveau de confort pour l'utilisateur. Par contre, lorsque la communication est moins structurée son utilisation est plus souple, mais le niveau de service rendu est moins important.

L'adressage Une fois le message formé et structuré, l'émetteur est confronté au problème de l'accès au destinataire. Il y a différentes manières d'adresser le message, nous distinguons :

– les stratégies **statiques** des stratégies **dynamiques**,
– les stratégies **directes** des stratégies **indirectes**.

Les stratégies statiques ne permettent pas à deux entités d'exécution de créer entre elles un lien de communication après leur activation. Un tel lien peut par exemple être défini à la compilation. Dans ce cas l'utilisateur fixe l'adresse - mémoire ou réseau - à laquelle vont être échangées les données. Un lien de ce type peut aussi être défini après la compilation par un ascendant de l'entité d'exécution. Dans ce cas l'entité d'exécution hérite de l'adresse du lien créé et peut communiquer avec d'autres entités possédant également cette adresse. Ces stratégies sont bien adaptées pour des applications dont le placement est défini avant l'exécution ou qui ne nécessitent pas de reconfiguration dynamique.

A l'opposé, les stratégies dynamiques permettent à une entité d'exécution de créer, après son activation, une adresse lui permettant de communiquer avec d'autres entités d'exécution. Ce mode d'adressage engendre souvent un problème de diffusion des adresses de communication aux autres entités d'exécution. En effet l'adresse de communication est créée localement à une entité. Comme ces entités sont indépendantes et qu'elles ne partagent pas de mémoire elles ne peuvent pas échanger leurs adresses facilement. Ces stratégies sont adaptées pour développer des applications dynamiquement reconfigurables.

Nous dirons d'une stratégie qu'elle est **directe** si un émetteur utilise l'identificateur de l'entité destinatrice comme adresse pour envoyer ses données. Dans ce cas il n'y a pas création d'une ressource du système dont la fonction est l'adressage. La mise en oeuvre de cette stratégie est simple et performante car aucune indirection n'est utilisée. Mais l'inconvenient de ce schéma vient du fait qu'il ne permet pas de reconfiguration dynamique transparente. En effet si un processus A utilise une ressource gérée par un serveur B, en cas de panne de B, A doit changer l'adresse d'envoi de ses requêtes s'il veut accéder à un serveur C, remplaçant B.

Les stratégies **indirectes** envoient le flux de données à une structure intermédiaire. Le récepteur lit lui-même les données sur cette structure. La structure joue le rôle d'indirection, d'adresse fixe, à laquelle un émetteur envoie ses données. Si différentes entités d'exécution peuvent lire à une même adresse alors la mise en place d'un service continu ou d'applications dynamiquement reconfigurables est plus aisée puisque le remplacement du récepteur n'entraîne pas de modification chez l'émetteur. Différentes approches existent dans l'implantation de cette stratégie. Par exemple, la ressource d'indirection est allouée, à un instant, à une seule entité d'exécution mais elle peut être déplacée vers une autre entité pour permettre la reconfiguration. La ressource d'indirection peut aussi être une entité indépendante - non liée à une ressource d'exécution - accessible par plusieurs récepteurs.

Nous présentons différents exemples d'implantation de stratégies d'adressage :

– Le système Helios[Gar87], permettant l'exécution de processus sur des réseaux de transputers, utilise un modèle de communication que nous qualifions de statique direct. Le graphe de communication des processus est défini avant leur exécution et ne peut être modifié lorsque les processus ont été activés.
– Un exemple d'adressage statique indirect est offert par les tubes UNIX. Le support de communication est alloué par un processus pour en faire profiter un ou plusieurs de ses fils avant leur création. Dans ce cas l'aspect statique permet à tous les descendants d'un processus d'accéder facilement à une adresse de communication, puisqu'ils en héritent dans leur contexte. Par contre ils ne peuvent communiquer de cette manière qu'avec les processus issus de l'ascendant

ayant créé le tube. D'autre part l'aspect indirect permet à un processus de ne pas connaître l'identification exacte du consommateur des données qu'il écrit sur le tube.
- Le système V[Che88], développé à l'université de Stanford, offre un exemple d'adressage dynamique direct. Les messages sont adressés directement aux processus, la connaissance de l'identificateur du processus destinataire est obtenue par un lien de parenté avec le processus émetteur, comme dans le cas des tubes UNIX. D'autres services du système permettent la diffusion de l'identificateur entre des processus sans lien de parenté.
- La solution généralement utilisée dans les systèmes répartis est une stratégie dynamique indirecte entre les entités communicantes. On peut classer en deux catégories les abstractions utilisées pour implanter cette indirection :
  - Le **canal** définit un lien bidirectionnel entre deux entités communicantes. Un système à base de canaux définit donc un graphe d'exécution à un instant donné. Le graphe peut être connu par le système, ce qui lui permet d'optimiser le placement des entités communicantes ou de prévenir les correspondants d'une entité d'exécution lors de la mort de celle-ci [ACF87].
  - La **porte** définit une adresse à laquelle envoyer et lire un message. C'est aussi une boite aux lettres qui permet de maintenir une file de messages en attente d'être lus. Pour envoyer un message sur une porte il faut en connaître l'adresse. Les manières d'utiliser les portes - droits d'accès, dépendance - varient beaucoup selon les systèmes d'exploitation répartis.

Comme nous l'avons déjà souligné, les différentes stratégies dynamiques, sont confrontées au problème de la diffusion de l'identification d'une ressource de communication. En effet cette ressource est allouée dynamiquement par une entité du système ce qui fait qu'elle n'est pas connue des autres entités. Dans les applications réparties ce problème est souvent résolu en utilisant l'héritage : lorsqu'une entité d'exécution en crée une autre elle lui transmet des données. Parmi les données transmises peuvent se trouver des adresses de communication. Ceci permet la mise en place de communication entre entités issues d'un même ascendant mais ne résoud pas le problème pour des entités ne partageant pas d'ascendants. Le système doit alors mettre en place un mécanisme particulier pour permettre l'accès dynamique aux ressources de communication. D'autre part l'adresse d'une ressource de communication sert aussi dans ce cas de protection puisque seules les entités la connaissant peuvent y accéder : elle ne peut donc être diffusée systématiquement à l'ensemble des entités du système. Les concepteurs du système doivent alors choisir entre une solution qui permet une bonne protection mais qui rend difficile la connaissance d'une adresse de communication ou l'inverse. Les solutions généralement proposées font alors appel à des identificateurs statiques : serveurs de noms, groupes statiques.

Les types de communication Plusieurs types de communication peuvent être différenciés parmi ceux généralement proposés par les systèmes d'exploitation.

Le type de communication **asynchrone** consiste en l'envoi simple d'un message d'un émetteur vers un destinataire sans attente de réponse de la part du destinataire. Le destinataire possède généralement une file d'attente pour les messages qu'il n'a pas encore reçus et il les traite suivant l'ordre d'arrivée. Avec le type de communication asynchrone l'émission et la réception d'un message peuvent être très espacées dans le temps sans pour autant bloquer l'une ou l'autre des entités d'exécution impliquées dans l'échange du message. Suivant la sémantique associée à l'envoi

asynchrone l'entité d'exécution initiatrice de l'envoi peut être bloquée soit, seulement pendant le temps nécessaire au système d'exploitation pour effectuer l'envoi du message sur le réseau, soit jusqu'à ce que le système soit assuré d'avoir déposé le message dans la file d'attente destinatrice.

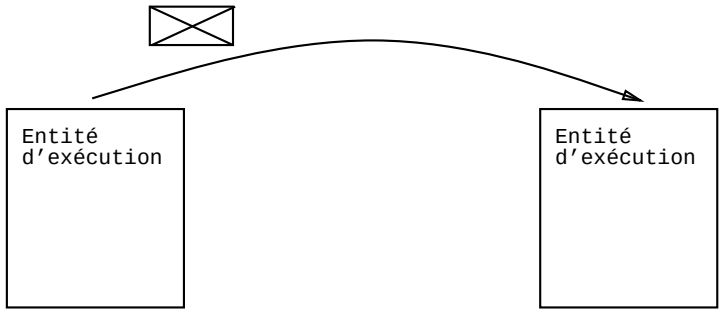

Figure II.1 : Envoi asynchrone

Le type de communication **synchrone** permet à deux entités d'exécution de se synchroniser à travers un échange de message. Un émetteur A envoie un message à un destinataire B et se met en attente de la réponse de B. Ce type de communication est souvent référencé dans la littérature comme un **appel de procédure à distance** (RPC)[Nel81]. En effet ce schéma est équivalent à un appel de procédure (passage synchrone de paramètres) standard. Ce type de communication est utilisé dans les systèmes d'exploitation basés sur le modèle client-serveur pour accéder aux serveurs.

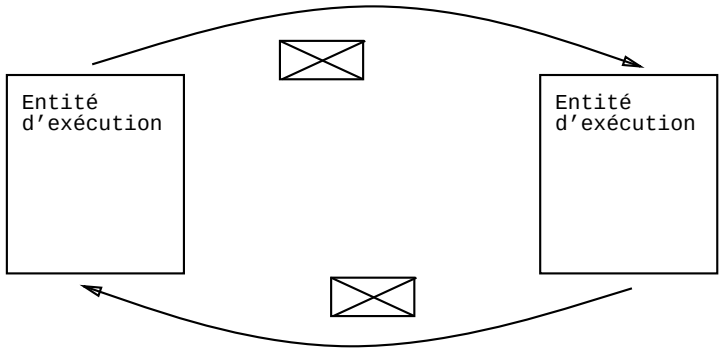

Figure II.2 : Envoi synchrone

Certains systèmes offrent des facilités de **diffusion** d'un message. Dans un système réparti composé de n sites, la diffusion d'un message à un ensemble de sites est l'envoi de ce message par un

site aux n-1 autres sites du système. La diffusion à un ensemble d'entités se fait généralement au travers d'une abstraction de groupe. Le groupe réunit l'ensemble des entités concernées par un même message, en cas de diffusion le message est distribué à l'ensemble des entités du groupe. Certains algorithmes permettent de fiabiliser la diffusion[BSS91b] :

- le protocole ABCAST(Atomic BroadCAST[BSS91b]) résiste aux défaillances des sites et assure un ordre de réception uniforme par ordonnancement global des messages. L'ordre de réception des messages est alors le même sur tous les sites, même s'il n'est pas l'ordre exact d'émission. Cet ordonnancement est réalisé au moyen d'estampilles.
- le protocole CBCAST(Causal BroadCAST[BSS90]) est un protocole de broadcast qui respecte la dépendance causale. C'est-à-dire que si un message A est diffusé avant un message B, tous les destinataires doivent recevoir A avant B. Une première réalisation de ce protocole reposait sur la transmission, avec tout message, d'un tampon "historique" comprenant tous les messages émis ou reçus par le site émetteur. Diverses optimisations permettent de réduire la taille de ce tampon. Une réalisation plus récente utilise les horloges vectorielles.

Les protocoles L'implantation de la communication sur des réseaux nécessite d'établir un protocole entre les sites. La mise en oeuvre du protocole dépend du médium de communication puisqu'il doit palier à d'éventuelles lacunes de celui-ci. Par exemple, le protocole peut mettre en place un service de fiabilité si le médium n'est pas fiable. Les différentes couches décrivant un protocole sont structurées suivant la norme ISO[Zim80]. C'est le protocole qui va supporter une partie de la sémantique liée à la communication. Par exemple, des algorithmes tels que ceux qui sont décrits pour la diffusion et la fiabilité de la communication sont implantés dans un protocole.

Avec l'évolution des systèmes répartis il y a de plus en plus de cas d'hétérogénéité entre des machines connectées par un réseau. Les protocoles qui vont être utilisés entre deux noeuds du multicalculateur et deux stations de travail ne sont probablement pas les mêmes. Le sytème devra donc intégrer plusieurs protocoles et permettre la communication entre ces protocoles hétérogènes. En général le système définit une notion de localité associée avec le protocole : les noeuds du réseau qui constituent les passerelles physiques entre deux réseaux homogènes deviendront des passerelles pour les protocoles.

Exemple de communication : le modèle client-serveur Dans quelques systèmes la communication elle-même devient une des bases du système d'exploitation et non plus seulement un service réservé à l'utilisateur. L'exemple le plus marquant est le recours au modèle client-serveur dans les systèmes d'exploitation répartis. Lorsqu'une ressource est susceptible d'être utilisée par plusieurs applications, y compris le système d'exploitation lui-même, la partie du code qui la gère peut être isolée pour former un serveur indépendant. Les bénéficiaires du service accèdent alors au serveur en lui envoyant des requêtes d'un type prédéfini. Le type des requêtes constitue l'interface d'accès au serveur. Les applications accédant au serveur sont appelées clients. Prenons l'exemple d'un serveur de fichiers. Ce serveur possède un nom d'accès (porte ou canal) connu par ses clients, et une file d'attente pour mémoriser les requêtes. Lorsqu'un client veut demander un service au serveur, il lui envoie un message en mode RPC contenant son adresse, ainsi que le code du service requis. Le client est bloqué en attente de la réponse du serveur.

2.2 Propriétés liées à la répartition

Ces concepts ne sont pas indispensables à un système d'exploitation pour pouvoir le qualifier de réparti. Ils apportent néanmoins à l'utilisateur un supplément de fonctionnalités appréciables, soit en lui simplifiant la tâche, soit en offrant un meilleur niveau de qualité. Ils sont souvent issus de problèmes nouveaux engendrés par la répartition des ressources.

2.2.1 La transparence

Différents niveaux de transparence sont offerts par les systèmes d'exploitation répartis. Le niveau le plus bas peut être illustré par un noyau UNIX incluant le protocole TCP/IP pour communiquer avec l'ensemble des noeuds du réseau. Les utilisateurs peuvent accéder à chacune des ressources du réseau mais ils doivent le faire explicitement en précisant le nom du site sur lequel est gérée la ressource. Le niveau le plus élevé est la possibilité d'accès à toutes les ressources sans en connaître la localisation, comme sur un monoprocesseur. Ces ressources comprennent : les fichiers et autres périphériques, la mémoire propre à chaque processeur et le processeur lui-même. Nous appellerons les systèmes offrant un tel niveau de transparence des **systèmes à image unique**. L'utilisateur a alors une vue des ressources qui est identique à celle d'un ordinateur traditionnel. Ce niveau de transparence est généralement offert sur les multiprocesseurs à mémoire partagée. Dans ce cas une seule instance du système s'occupe de l'allocation des ressources. A l'heure actuelle nous ne connaissons pas de système d'exploitation qui offre réellement ce niveau de transparence sur un multiprocesseur à mémoire répartie.

On peut s'interroger sur la nécessité d'un tel système d'exploitation dans le cas de stations de travail interconnectées par un réseau local, par contre, il est certain qu'un tel système serait très utile pour les multiprocesseurs à mémoire distribuée dont l'utilisateur aurait une vue simplifiée. Nous reviendrons sur cette question au troisième chapitre.

Certains systèmes ont fait un premier pas pour cacher la répartition des ressources aux utilisateurs. Entre autres fonctionnalités nous pouvons citer l'accès transparent aux fichiers dans des gestionnaires de fichiers tels que NFS, RFS[RFHa86], etc. qui permettent de nommer un fichier indépendamment du serveur qui le gère. Ces systèmes de fichiers étant intégrés dans UNIX ils peuvent aussi offrir un accès transparent aux périphériques autres que les disques puisque leur interface d'accès est la même que celle des fichiers.

D'autres systèmes, plus récents, offrent un nommage des abstractions du système indépendant de la distribution. Ainsi les échanges de messages utilisent la même syntaxe en local et en distant, les mécanismes de mémoire virtuelle permettent l'accès à des pages distantes, etc. La fonctionnalité qu'il manque en général dans ces systèmes pour pouvoir les qualifier de systèmes à image unique est l'accès transparent aux processeurs.

2.2.2 La tolérance aux pannes et le service continu

Avec la multiplication du nombre de machines en service sur un même réseau, ou de noeuds dans une machine parallèle, et les risques de surcharge qu'elle entraîne, avec la distribution des

ressources et les problèmes de concurrence d'accès que cela engendre, la probabilité d'apparition de pannes augmente dans un contexte réparti. Pour l'utilisateur il devient donc important d'être sûr de pouvoir exécuter complètement son application même si un des sites impliqués est indisponible. Il devient donc impératif de proposer des services continus, c'est-à-dire des services auxquels les utilisateurs pourront accéder même si l'un des serveurs impliqués dans ce service est en panne.

Il est possible de faire de la tolérance aux pannes sur une architecture centralisée en implantant, par exemple, des mémoires stables[BJMa91]. Pour ce qui est de la continuité d'un service, les possibilités restent, par nature, limitées : si la machine est en panne elle ne peut pas rendre un service. Par contre, l'architecture d'un système réparti offre des possibilités intrinsèques de tolérance aux pannes[Bir91][Bab91] dans la mesure où les mêmes ressources matérielles peuvent être indépendantes et où les données peuvent être dupliquées. Ainsi l'indépendance des sites permet qu'une panne locale à un site n'entraîne pas forcément l'arrêt des autres sites. Dans ce contexte, il devient intéressant de développer des outils - logiciels et matériels - qui permettent, dans un premier temps, de ne pas répercuter une panne sur les autres sites, c'est-à-dire d'arrêter proprement le service (par exemple : sans générer de messages erronés) et, dans un deuxième temps, de ne pas perdre d'information lors d'une panne et permettre de reprendre une exécution.

La duplication des ressources apparaît comme une des fonctionnalités matérielles utilisées pour implanter la tolérance aux pannes. Cela permet de réduire la probabilité de perte d'une information - par exemple avec des disques miroirs - ou d'offrir un service continu - en utilisant un processeur de secours. L'utilisation de ces possibilités matérielles doit ensuite être prise en compte par le système d'exploitation. Les structures matérielles nécessaires étant assez coûteuses, une partie de ces fonctionnalités peuvent être mises en place par le logiciel. Beaucoup de recherches ont déjà été menées dans le domaine de la tolérance logicielle aux pannes. Parmi les implantations de la tolérance aux pannes nous pouvons citer [BBMa87].

2.2.3 L'extensibilité

Nous qualifions un système d'exploitation d'extensible (en anglais *scalable*) s'il est capable d'étendre sa gestion d'un nombre limité de sites à un ordre de grandeur supérieur. Par exemple, s'il est capable de gérer sans dégradation de service du parallélisme et du parallélisme massif. Il est, en effet, évident qu'un système d'exploitation ne gère pas des milliers de processeurs de la même façon que quelques dizaines. Il faut alors que les algorithmes sur lesquels il base sa gestion puisse répondre aux contraintes de l'extensibilité. Par exemple, un algorithme d'implantation d'un temps global sur un multicalculateur qui centralise les informations de temps sur un processeur peut convenir pour une dizaine de noeuds mais pas pour plusieurs milliers : il y aurait rapidement apparition d'un goulet d'étranglement.

Comme nous disposons de peu de machines de cet ordre de grandeur peu de tests ont pu être réalisés et l'extensibilité reste une notion assez mal connue dans les systèmes d'exploitation. Cependant l'extensibilité constitue une propriété indispensable pour un système d'exploitation destiné aux multicalculateurs dans la mesure où le nombre de processeurs peut être très grand.

2.2.4 Hétérogénéité

La mise en réseau d'un nombre de plus en plus grand de sites est intéressante car elle permet, en général, d'accéder un éventail beaucoup plus large de types de ressources tant par leur type que par leur nombre. Pour que l'utilisateur puisse accéder à ces ressources, les systèmes d'exploitation des différents sites doivent être capables de communiquer. Différents obstacles peuvent s'opposer à cette communication. Ils peuvent être dus soit à l'héterogénéité physique des sites, soit à l'hétérogénéité des média de communication reliant les deux sites. Un problème très difficile est la différence de codage de l'information d'un processeur à un autre. Par exemple, les processeurs *Motorola* codent les nombres entiers différemment des processeurs *Intel*. D'autre part deux sites ne sont pas forcément connectés directement. Plusieurs média de communication peuvent alors être utilisés pour acheminer un message d'un site à l'autre. Les protocoles gérant les média peuvent également être différents. Autant de cas d'hétérogénéité qui doivent être résolus par le système d'exploitation.

Notons cependant que les multicalculateurs actuels sont généralement homogènes, cette propriété a donc peu d'incidence pour le choix du système d'exploitation.

2.2.5 Sécurité

Le problème de la sécurité n'est pas uniquement lié à la mise en réseau des ordinateurs. Les systèmes d'exploitation centralisés multi-utilisateurs devaient également empêcher l'intrusion d'un utilisateur chez un autre. Cependant la connexion des ordinateurs entre eux introduit de nouveaux besoins de sécurité[Des91] tels que la sécurité de transmission sur un réseau et l'authentification du propriétaire d'une ressource.

L'écoute d'un réseau permet d'obtenir des informations précieuses telles que le mot de passe d'un utilisateur puisque toutes les données traitées lors d'une connexion à distance passent sur le réseau. Pour protéger les données importantes certains systèmes cryptent leurs messages à l'aide d'une clé connue uniquement de l'émetteur réel et du récepteur.

La connaissance du descripteur d'une ressource joue souvent un rôle de protection dans les systèmes d'exploitation répartis. Ce type de protection est alors très vulnérable à une écoute du réseau. Le système Amoeba[TMvR86] sécurise cette identification au moyen de fonctions non-inversibles qui empêche un éventuel intrus de décrypter le contenu des messages. D'autre part, ces descripteurs contiennent parfois les droits d'accès à une ressource. Le cryptage de ces descripteurs empêche l'utilisateur de modifier ses droits sans autorisation du serveur.

Pour les multicalculateurs, la sécurité n'est pas un critère décisif. En effet, la conception de ces machines fait qu'elles sont moins soumises à des risques d'écoute du réseau. Cependant pour permettre une utilisation en multi-utilisateurs le système doit garantir la protection entre les utilisateurs.

2.3 Services liés à la répartition

2.3.1 Exécution distante et migration

Les systèmes répartis doivent permettre de tirer parti de toutes les ressources distantes. L'exécution distante et la migration de processus permettent de profiter des processeurs distants. Cependant les implantations actuelles de ces services dans les systèmes répartis n'offrent généralement pas de transparence.

La migration des entités d'exécution, dans un contexte réparti peut servir soit à la continuité du service lors de la maintenance d'un site, soit à la gestion globale de la charge. Dans le premier cas, il est intéressant de pouvoir déplacer les processus d'un site lorsque celui-ci risque de s'arrêter ou a besoin d'être réinitialisé. La migration doit assurer alors un transfert des processus de façon transparente pour que le service soit utile. Dans le second cas, lors d'une gestion globale de la charge, la migration de processus peut être utilisée afin d'obtenir un équilibrage fin entre les sites du système.

Différentes techniques de migration des processus ont été developpées dans des systèmes tels que V[TLC85], Accent[Zay87] ou Locus[BP86]. L'idée de base est de stopper le processus à un instant de son exécution. Les données concernant le processus - généralement son contexte système et ses données - sont alors envoyées au site destinataire de la migration. En utilisant ces données, le processus peut être relancé sur le site destinataire. Pour améliorer les performances de la migration, le système Accent envoie uniquement le contexte du processus au site destinataire, la mémoire est ensuite demandée, par le gestionnaire de mémoire du site destinatiare, à chaque défaut de page. Ceci engendre des *dépendances résiduelles*, c'est-à-dire que le site origine conserve une partie du processus. Cette technique ne peut être utilisée pour implanter la tolérance aux pannes. Le système V continue d'exécuter le processus pendant son transfert pour réduire le temps pendant lequel le processus est arreté.

2.3.2 Mémoire virtuelle répartie

Les systèmes répartis posent le problème de dispersion des données. Une application parallèle agit fréquemment sur un ensemble de données communes aux différents processus de l'application (par exemple dans une multiplication de matrices ou une résolution de système) et pose donc le problème de la cohérence de ces données. La gestion de la cohérence dans un environnement réparti peut répondre à différentes contraintes. Nous parlons ainsi de cohérence forte lorsque le lecteur d'une donnée (cohérente) est assuré de lire exactement la dernière valeur de la donnée. La cohérence sera faible si le lecteur n'a pas de garantie quand à la validité temporelle de la donnée. Une description détaillée de différents algorithmes de gestion de la cohérence est donnée dans [LH89].

Actuellement la gestion de la cohérence de cet ensemble de données est laissée à l'utilisateur, c'est-à-dire qu'une partie de son code est dédiée à l'échange des données pour maintenir cette cohérence. Des systèmes tel que UNIX SVR4 donne la possibilité aux utilisateurs de mapper des fichiers directement dans leur espace mémoire : l'extension de ce type de systèmes à un environnement réparti pose alors le problème de la cohérence des fichiers mappés. Pour ces raisons des

systèmes tels que Amoeba[Mul87], CHORUS[Ort92] ou MACH[TRYa87] implantent des services de mémoire virtuelle répartie. Une implantation d'application parallèle utilisant un tel service, sur un multicalculateur, peut être trouvée dans [LP91].

3 CHORUS

La démarche des concepteurs [AGHR89] du système CHORUS a consisté à restructurer le noyau UNIX d'une manière plus modulaire et d'y intégrer les services nécessaires à la prise en compte de la répartition par les utilisateurs. Cette démarche s'appuie sur la nécessité d'appréhender les systèmes répartis d'une manière générale pour qu'ils puissent s'adapter aux diverses configurations d'environnement réparti. Le système CHORUS[RAA$^+$88] est composé d'un noyau de petite taille (appelé micro-noyau), intégrant la communication et l'exécution, qui offre des services génériques utilisés par un ensemble de serveurs coopérants au sein de sous-systèmes. Cette architecture CHORUS permet de construire des systèmes ouverts et répartis.

3.1 Le noyau CHORUS

3.1.1 Description macroscopique

Un système CHORUS est composé d'un **Noyau** de petite taille (micro noyau) et d'un ensemble de **Serveurs Systèmes**. Ces serveurs coopèrent au sein de **Sous-Systèmes** (par exemple le sous-système UNIX), pour fournir un ensemble cohérent de services et d'interfaces aux utilisateurs (Figure II.3).
Le noyau CHORUS (figure II.4) joue un double rôle :

(1) Services locaux :

La gestion des ressources physiques d'un site, est assurée par trois composants distincts :

- *L'Exécutif temps-réel multitâches* gère l'accès au processeur : il fournit les primitives de synchronisation de bas niveau et offre un ordonnancement préemptif basé sur des priorités fixes.
- *Le Gestionnaire de Mémoire (Virtuelle)* gère la mémoire locale,
- *Le Superviseur* permet aux composants des sous-systèmes implantés hors du noyau de contrôler les évènements matériels : interruptions, déroutements, exceptions.

(2) Services globaux :

Le Gestionnaire d'IPC (Inter Process Communication) fournit les services de communication, en assurant la transmission de *messages* de manière uniforme et transparente à la répartition, c'est-à-dire indépendamment de la localisation des correspondants. Il s'appuie sur un serveur externe (Gestionnaire de réseau) pour la réalisation des protocoles utilisés.

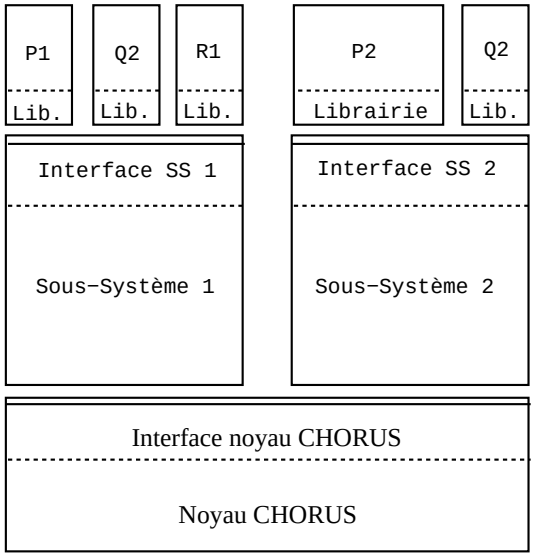

Figure II.3 : Architecture du système CHORUS

3.1.2 Abstractions de base offertes par le noyau CHORUS

Un système CHORUS[RA90] s'exécute sur un ensemble de *sites* (machines ou cartes), interconnectés par un *système de communication* (réseau ou bus). Un **site** est constitué de ressources physiques fortement couplées : un ou plusieurs processeurs, de la mémoire, et éventuellement des périphériques. Il y a un noyau CHORUS par site.

Un **acteur** représente l'unité de répartition et d'allocation de ressources. Un *acteur* définit un espace d'adressage protégé. Une ou plusieurs **activités** (*processus légers*) peuvent s'exécuter au sein d'un même acteur (donc du même espace d'adressage). Un espace d'adressage est composé d'un espace *utilisateur* et d'un espace *système*. Sur un site donné, l'espace système est commun à tous les acteurs de ce site, mais son accès est réservé au mode d'exécution privilégié. Chaque acteur ayant son propre espace d'adressage utilisateur, un acteur définit une machine virtuelle protégée. Un acteur est lié à un site, et ses activités s'exécutent sur ce site. Plusieurs acteurs peuvent s'exécuter simultanément sur un site.

L'activité est l'unité d'exécution dans un système CHORUS. Elle est caractérisée par un *contexte* correspondant à l'état du processeur (registres, compteur ordinal, pointeur de pile, etc). Une activité est liée à un acteur et un seul. Toutes les activités d'un acteur partagent les ressources de cet acteur et de celui-là seulement. Les activités sont ordonnancées par le noyau comme des entités indépendantes. En particulier, les activités d'un acteur peuvent s'exécuter en parallèle sur les différents processeurs d'un site multiprocesseur à mémoire partagée. L'ordonnancement des activités est préemptif, et basé sur leur priorité (fixe).

Les activités d'un même acteur peuvent communiquer en utilisant la mémoire de cet acteur

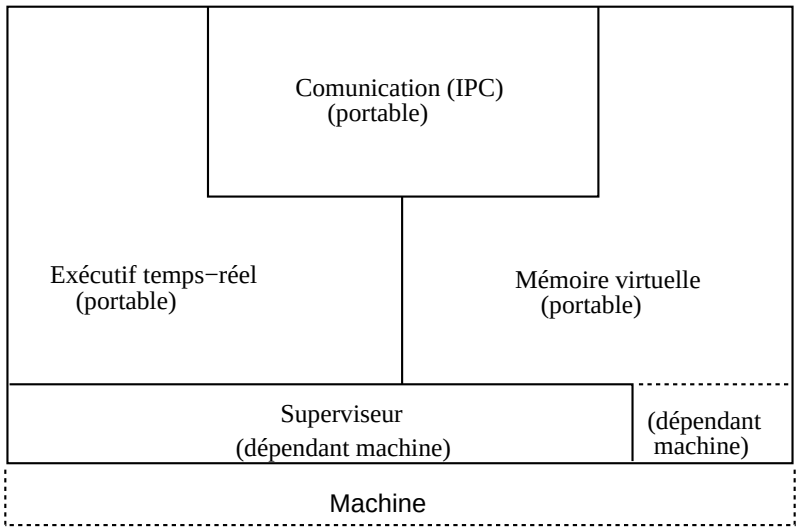

Figure II.4 : Le micro-noyau CHORUS

comme mémoire partagée. CHORUS offre également un mécanisme de communication par message (appelé IPC par la suite) qui permet à toute activité de communiquer et de se synchroniser avec n'importe quelle autre activité s'exécutant sur n'importe quel autre site. Les échanges de messages peuvent être asynchrones ou synchrone (RPC). La caractéristique principale de l'IPC CHORUS est sa transparence vis à vis de la localisation des activités : l'interface de communication est identique que les messages soient échangés entre activités d'un même acteur, entre activités d'acteurs différents du même site ou entre activités d'acteurs différents résidant sur des sites différents.

Un **message** est composé d'un *corps* optionnel et d'une *annexe* également optionnelle. L'annexe et le corps sont tous deux une suite d'octets continue et non typée. Le couplage étroit entre le Gestionnaire de Communication et le Gestionnaire de Mémoire permet d'éviter les recopies physiques d'information lors des échanges de messages.

Les messages ne sont pas adressés directement aux activités, mais à des entités intermédiaires appelées **portes**.

Une **porte** est une adresse logique à laquelle des messages peuvent être envoyés, et une file d'attente sur laquelle ils seront reçus. Une porte est attachée à un acteur, et non à une activité. Une porte est donc une ressource partagée par toutes les activités d'un même acteur. Une porte ne peut être attachée qu'à un seul acteur à un instant donné, mais peut être attachée successivement à plusieurs acteurs. Une porte peut ainsi *migrer* d'un acteur à un autre. Toutes les activités d'un acteur (et elles seules) peuvent consommer les messages reçus sur les portes de cet acteur. Par contre, il suffit à une activité de connaître le nom d'une porte pour pouvoir lui adresser un message.

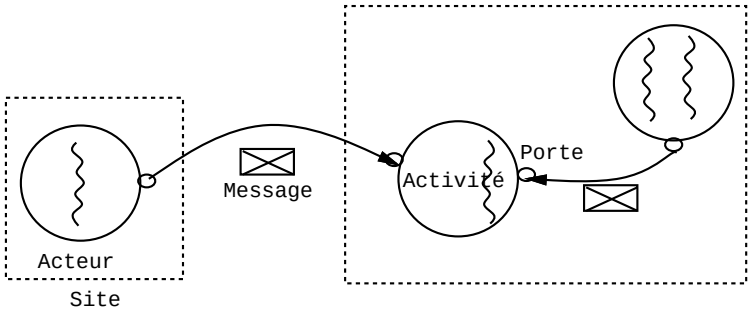

Figure II.5 : Abstractions de base du noyau

La notion de porte fournit une mécanique élémentaire pour la reconfiguration dynamique : une activité "cliente", adressant une requête à un acteur "serveur" au travers d'une porte, n'est pas affectée par la migration de cette porte vers un autre acteur, qui peut ainsi "reprendre" le service d'une manière transparente pour le client.

Les portes peuvent être dynamiquement regroupées en ensembles logiques appelés **groupes** de portes. Les *groupes* enrichissent les mécanismes d'adressage pour l'émission de message. Ils offrent des mécanismes de *diffusion* permettant à une activité de communiquer directement avec un groupe d'activités au sein d'acteurs différents. Ils offrent aussi un mode d'adressage dit *fonctionnel* permettant à une activité de communiquer avec une autre choisie parmi un groupe d'activités (souvent fournissant des services équivalents). Les portes d'un même acteur peuvent être regroupées dans un **ensemble** de portes. Une activité de l'acteur peut alors se mettre en attente de réception les portes appartenant à l'ensemble.

Les portes sont désignées par des identificateurs uniques globaux (appelés UI par la suite). Un **UI** est unique dans un système CHORUS. Le Noyau CHORUS implémente un mécanisme de localisation, permettant ainsi aux activités d'utiliser ces noms sans se soucier de la localisation réelle des entités désignées. Les *UI* peuvent être échangés entre acteurs.

Les noms globaux pour les autres types d'objets sont basés sur les *UI*, mais contiennent d'autres informations, comme par exemple des attributs de protection. Ces noms sont appelés **capacités**. Une *capacité* est composée d'un *UI* et d'une structure additionnelle : la *clé*. Quand les objets sont des objets du noyau (par exemple un acteur), l'*UI* est alors le nom global de l'objet et la clé est seulement une clé de protection. Quand un objet est géré par un serveur externe au noyau CHORUS (par exemple un fichier), l'*UI* est alors le nom global d'une porte du serveur et la signification de la clé est définie par le serveur lui-même. Généralement, la clé identifie l'objet dans le serveur et contient un attribut de protection. Les capacités comme les *UI* peuvent être librement échangées entre acteurs.

3.1.3 Mémoire Virtuelle

Le Gestionnaire de mémoire CHORUS[ARG89] gère des espaces d'adressage séparés (si le matériel le permet) associés aux acteurs, appelés **contextes**. Il offre des services pour le transfert de données entre *contexte* et mémoire secondaire. Les mécanismes sont adaptés à des besoins divers tels que l'IPC, l'accès aux fichiers (par lecture/écriture aussi bien que par "mapping"), le partage de mémoire entre contextes ou la duplication de contexte.

Un *contexte* est composé d'un ensemble de **régions** disjointes, qui composent les parties valides du *contexte*.

Les régions permettent (généralement) d'accéder (par mapping) à des objets stockés en mémoire secondaire appelés **segments**. Les segments sont gérés à l'extérieur du noyau CHORUS par des serveurs appelés **mappeurs**. Les mappeurs gèrent l'implantation des segments ainsi que leur protection et leur désignation.

3.1.4 Le Superviseur

Pour permettre aux acteurs systèmes de gérer les évènements matériels comme les interruptions ou les traps, le noyau CHORUS offre les services suivants :

Les activités s'exécutant en mode système peuvent attacher des "handlers" (procédures situées dans l'espace d'adressage de leur acteur) à des interruptions matérielles. Quand une interruption se produit, ces handlers sont exécutés. Plusieurs handlers peuvent être simultanément connectés à une même interruption, des mécanismes de contrôle permettent d'ordonner ou d'arrêter leur exécution. Ces handlers d'interruption peuvent communiquer avec d'autres activités en utilisant les primitives de synchronisation offertes par le noyau ou l'IPC asynchrone.

Les acteurs sytèmes peuvent aussi connecter des procédures aux déroutements matériels. On peut connecter une procédure ou un tableau de procédures (dans ce cas, la procédure réellement invoquée est spécifiée par un "numéro" de service stocké dans un registre).

Enfin, une porte d'exception ou une procédure d'exception peuvent être associées à un acteur, permettant à un sous-système de réagir aux erreurs survenant dans d'autres acteurs.

3.2 Le Sous-Système UNIX

3.2.1 Structure

Les services UNIX peuvent être logiquement partitionnés en plusieurs classes suivant le type des ressources gérées : processus, fichiers, périphériques, tubes, sockets. L'architecture du sous-système UNIX de CHORUS[HAR+88], basée sur une définition rigoureuse des interactions entre ces différentes classes de services, offre une structure modulaire. Nous décrivons ici l'implantation d'un sous-système UNIX compatible avec la version 3.2 d'UNIX system V. Une implantation de la version SVR4.0 est décrite dans [Arm90]. Nous appellons CHORUS/MiX l'implantation d'UNIX utilisant le noyau CHORUS.

Le sous-système UNIX est composé d'un ensemble de serveurs système qui s'exécutent sur le

Noyau CHORUS. Chaque type de ressource système (processus, fichier...) est géré par un serveur système dédié. Les interactions entre ces serveurs sont basées sur l'IPC CHORUS.

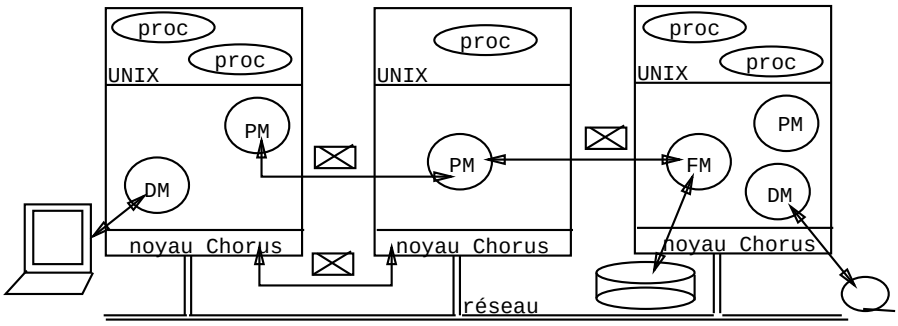

Figure II.6 : UNIX modulaire

(1) Le Gestionnaire de Processus (Process Manager ou PM) :

Le code du sous-système UNIX relatif à la gestion de processus, la gestion des signaux et les interfaces systèmes accessibles depuis un processus sont inclus dans le Gestionnaire de Processus. Le PM gère le contexte système des processus. Quand le PM ne peut pas fournir lui-même un service UNIX (invoqué par un appel système UNIX), il fait appel au(x) serveur(s) approprié(s). Les PM des différents sites d'un réseau CHORUS coopèrent pour fournir des services répartis (tels que des signaux distants, ou des exécutions distantes).

Cet acteur est chargé à l'initialisation du système. Son code et ses données résident dans l'espace système. La présence du PM dans l'espace système lui permet d'implanter l'invocation des appels système par `traps` comme pour un noyau UNIX traditionnel. Il est ainsi possible d'assurer une compatibilité au niveau binaire avec d'autres systèmes UNIX.

(2) Le Gestionnaire de Fichiers (File Manager ou FM) :

Le Gestionnaire de Fichiers offre les services de gestion de fichiers. Il existe deux versions principales qui sont compatibles avec les versions 3.2 et 4 du system V, tant pour les services offerts que pour la structure du disque. Le FM joue aussi le rôle d'un *mappeur externe* CHORUS, participant à la gestion de la mémoire virtuelle répartie, en répondant aux requêtes de chargement/déchargement de pages émises par la gestion de mémoire virtuelle du Noyau CHORUS.

Le Gestionnaire de Fichiers est un acteur système qui possède deux portes sur lesquelles il reçoit des messages de requêtes. L'une d'elles sert exclusivement pour les messages de demande de pagination émis par le Gestionnaire de Mémoire Virtuelle du Noyau CHORUS. L'autre porte reçoit toutes les autres requêtes : services UNIX, messages de coopération entre les PM et les FM, etc.

Une fois l'initialisation terminée, plusieurs activités s'exécutent en parallèle dans le FM. Elles traitent les messages de l'une ou de l'autre porte. Comme le contexte du processus

pour lequel le FM traite une requête ne lui est pas directement accessible, l'information contextuelle nécessaire pour rendre un service est incluse dans le message de requête avec les paramètres de l'appel système. En retour, le serveur inclut dans le message de réponse les informations nécessaires pour la mise à jour du contexte fichier du processus. Les autres serveurs comme les gestionnaires de tubes, de périphériques ou de sockets opèrent suivant un mécanisme similaire.

(3) Le Gestionnaire de Périphériques (Device Manager ou DM)

Le Gestionnaire de Périphériques gère les lignes asynchrones, les écrans bitmap, les pseudo-terminaux (pseudo-ttys), etc. Il implante les "*line disciplines*" UNIX. Plusieurs DM peuvent s'exécuter en parallèle sur un site où sont connectés plusieurs types de périphériques.

(4) Le Gestionnaire de Tubes (Pipe Manager ou PiM)

Le Gestionnaire de Tubes[HP88] implante la gestion et la synchronisation des tubes UNIX. Les requêtes d'ouvertures des tubes nommés, reçues par les Gestionnaires de Fichiers sont transmises aux Gestionnaires de Tubes. Il peut y avoir un Gestionnaire de Tubes par site, cela permet notamment de réduire le trafic réseau quand les tubes sont utilisés sur des stations sans disque, donc sans FM.

(5) Le Gestionnaire de Sockets (Socket Manager ou SM)

Le Gestionnaire de Sockets offre les services sockets compatibles avec BSD 4.3 fournissant l'accès aux protocoles TCP/IP.

Les serveurs systèmes peuvent s'exécuter en *espace utilisateur* ou en *espace système*. Ceux qui doivent attacher leurs procédures aux exceptions (comme le PM) ou exécuter des instructions privilégiées (comme des instructions d'entrée/sortie) s'exécutent en espace système. Charger un serveur en espace système, permet des gains de performance en évitant des changements de contexte mémoire quand le serveur est invoqué.

3.2.2 Extensions Fonctionnelles

Des extensions aux services UNIX traditionnels ont été apportées, essentiellement par extension transparente de ces services à la répartition.

(1) **Extensions du Système de Fichiers**

Les services de désignation offerts par UNIX ont été enrichis pour permettre la désignation de services accessibles par des portes. Des noeuds de type *Porte Symbolique* peuvent être créés dans une arborescence UNIX. Ils associent un nom de noeud à un identificateur unique (UI) de porte. Ce mécanisme est similaire à celui utilisé pour la désignation des périphériques UNIX. Quand le nom d'un noeud de type "Porte" est trouvé au cours de l'analyse d'un chemin d'accès, la requête correspondante est transmise à la porte associée. La requête est préalablement marquée avec l'état courant de l'analyse du chemin d'accès.

Des serveurs utilisateur comme des serveurs système peuvent être désignés par de tels noms de portes symboliques, permettant ainsi d'étendre dynamiquement le système. Ce mécanisme est en particulier utilisé pour interconnecter les systèmes de fichiers d'un réseau

de machines CHORUS et ainsi fournir un espace global de désignation. Par exemple, dans la Figure II.7, "pipo" et "piano" sont des noms de portes symboliques.

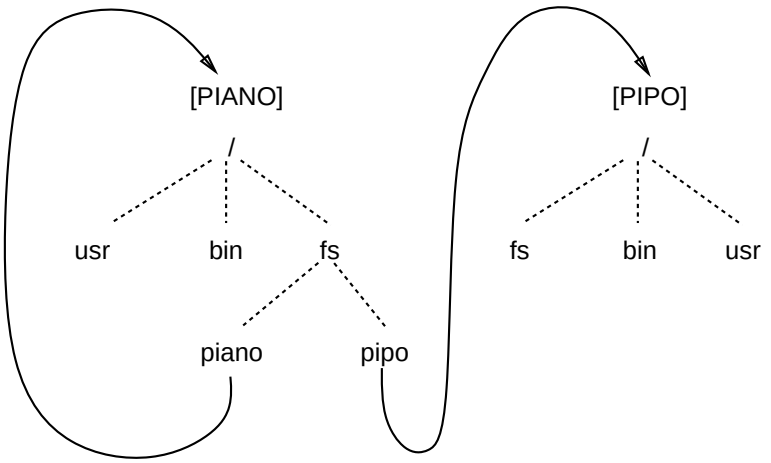

Figure II.7 : Interconnexion d'arborescences de fichiers

(2) **Extensions de la Gestion des Processus**
Un nouvel appel système (`fexec(2)`) permet la création dynamique d'un processus différent du père. Cet appel est une contraction des appels `fork(2)` et `exec(2)`. La création (`fexec(2)`) ou l'exécution (`exec(2)`) de processus peut être invoquée sur des sites distants. Un attribut "site de création" a été ajouté au contexte système des processus UNIX. Cet attribut est hérité lors des appels système `fexec(2)` et `exec(2)`. Il peut être positionné au moyen d'un nouvel appel système : `csite (SiteId)`. Lors d'un `fexec(2)`, le processus fils sera créé sur le site désigné par `SiteId`. Lors d'un `exec(2)`, le processus commencera l'exécution du nouveau programme sur le site désigné par `SiteId`.

(3) **Autres Extensions**
Les services offerts aux processus UNIX sont complétés par quelques uns des services offerts par le Noyau CHORUS comme l'IPC, la Mémoire Virtuelle et les Activités. Ces services ne sont pas fournis directement par le noyau CHORUS, mais plutôt par le Gestionnaire de Processus UNIX (PM), de manière à rendre homogène l'ensemble des services fournis et à éliminer les incohérences potentielles avec la "sémantique" UNIX. Par exemple, si un processus UNIX pouvait créer une *activité* en invoquant directement le Noyau CHORUS en "contournant" le Gestionnaire de Processus, cette activité ne pourrait pas invoquer d'appels systèmes UNIX correctement. En conséquence, quelques services offerts par le Noyau CHORUS ne sont pas accessibles aux processus UNIX (par exemple les services de création ou de destruction d'acteurs), et quelques restrictions et contrôles sont effectués : par exemple, un processus ne peut pas créer d'activités dans un autre processus ; dans l'appel système `portMigrate` l'identificateur de processus (pid) doit être utilisé et non la capacité d'un

acteur CHORUS.
Pour distinguer clairement les deux niveaux d'interface, les primitives UNIX permettant d'accéder aux services CHORUS ont été préfixées par "u_" (par exemple : `u_portCreate` au lieu de `portCreate`).

- *Inter Process Communication (IPC)*
  Les processus UNIX peuvent créer des portes, insérer des portes dans des groupes et émettre et recevoir des messages. Ils peuvent faire migrer des portes d'un processus à un autre. Les mécanismes de l'IPC CHORUS leur permettent de communiquer de manière transparente à travers le réseau. Les applications peuvent ainsi être testées sur une seule machine puis réparties sur le réseau sans qu'aucune modification soit nécessaire pour les adapter à cette nouvelle configuration. La possibilité de faire migrer des portes ou d'utiliser les groupes de portes peut servir de base pour développer des applications qui se reconfigurent dynamiquement et/ou résistent aux pannes.
- *Processus UNIX Multi-Activités*
  Il est possible d'écrire des processus UNIX multi-activités, ou multi-**u_threads**. Une *u_thread* peut être considérée comme un processus léger s'exécutant dans un processus UNIX classique. Elle partage toutes les ressources du processus et en particulier son espace d'adressage et sa table de fichiers ouverts. Chaque *u_thread* représente une entité d'exécution différente.
  Quand un processus est créé par `fork(2)`, il commence son exécution avec une seule activité (u_thread), de même dans le cas d'un recouvrement par `exec(2)` ; quand un processus arrête son exécution par `exit(2)`, toutes les u_threads de ce processus se terminent avec lui.
  Les traitements sont associés aux signaux sur la base des u_thread et non seulement du processus : chaque u_thread possède son propre contexte de traitement des signaux. Un signal émis suite à une exception (division par zéro) est délivré à l'activité ayant provoqué l'erreur ; de même les signaux d'alarme sont délivrés à l'activité ayant positionné l'alarme. Tous les autres signaux sont diffusés à l'ensemble des u_threads du processus. Les handlers de signaux sont exécutés dans la pile de l'activité l'ayant positionné. Ainsi, la cohérence est assurée avec les handlers de signaux existants. Les u_threads peuvent invoquer des appels système UNIXet, pour des raisons de simplicité (assurer efficacement la cohérence du contexte système d'un processus), les appels systèmes sont sérialisés à l'exception des appels bloquants comme `read(2)`, `write(2)`, `pause(2)`, `wait(2)`, `u_ipcReceive(2)` et `u_ipcCall(2)` (c.a.d. ceux interruptibles par signaux).
- *Services liés à la Mémoire Virtuelle*
  Les processus UNIX peuvent utiliser les services du Noyau CHORUS pour créer des régions, "mapper" des segments dans des régions, partager des régions, etc. Ils peuvent ainsi avoir accès à la mémoire physique (par exemple : accès à la mémoire d'un écran graphique bitmap).

3.2.3 Implantation

(1) Structure d'un processus UNIX :

Un processus UNIX classique peut être perçu comme une entité d'exécution unique dans un espace d'adressage. En conséquence, chaque processus UNIX est implanté comme un acteur CHORUS déroulant une seule activité (u_thread). Son contexte système UNIX est géré par le Gestionnaire de Processus. L'espace d'adressage d'un processus est structuré en régions de mémoire pour le code, les données et la pile.

De plus, le Gestionnaire de Processus crée une porte de contrôle et une activité de contrôle à chaque acteur implantant un processus UNIX. Cette porte et cette activité de contrôle ne sont pas visibles du programmeur du processus. L'activité de contrôle qui s'exécute dans le contexte du processus partage son espace d'adressage et peut facilement accéder et modifier son image mémoire (modifications de la pile sur réception de signaux, accès au code et aux données pour le débogueur, etc). Elle gère aussi les évènements asynchrones reçus par le processus (principalement des signaux). Ces évènements sont implantés comme des messages CHORUS reçus sur la porte de contrôle du processus.

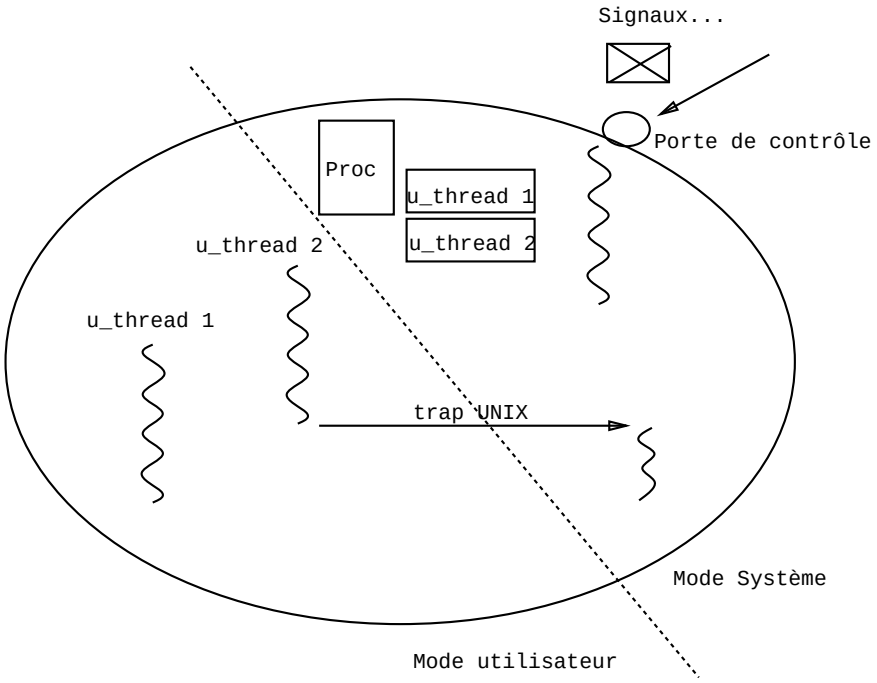

Figure II.8 : Processus UNIX

Comme un processus peut être multi-activités, le contexte système d'un processus a été séparé en deux : un contexte associé au processus (`Proc`) et un contexte d'activité (`u_thread`).

La plupart des services implantés hors du Gestionnaire de Processus sont relatifs aux fichiers. Cependant, le "contexte fichier" d'un processus (c.a.d. catalogues courant et racine, table

des fichiers ouverts, attributs `umask` et `ulimit`) sont stockés dans la structure `Proc` gérée par le Gestionnaire de Processus. Pour ce faire, un protocole spécifique entre le Gestionnaire de Processus et les autres serveurs a été conçu. Les deux contextes système `Proc` et `u_thread` sont gérés par le gestionnaire de Processus du site courant d'exécution du processus. Ces contextes ne sont accédés ni par le Noyau CHORUS, ni par les autres serveurs système. Par ailleurs, le sous-système UNIX ne peut accéder les structures internes du Noyau associées aux acteurs et aux activités. Le seul moyen de les accéder est d'utiliser les services offerts par le Noyau CHORUS (ceci est essentiel, pour autoriser plusieurs sous-systèmes à cohabiter au dessus d'un même Noyau CHORUS).

(2) Environnement d'un processus identifié par un ensemble de portes :

Les propriétés des portes CHORUS (noms globaux uniques, adressage indépendant de la localisation) les rend intéressantes pour désigner des entités système. Leur intérêt principal réside dans l'indirection qu'elles induisent entre un processus et son environnement, et dans leur robustesse vis à vis des évolutions de configuration. Les noms de portes stockés dans le contexte d'un processus restent valides si le processus migre vers un autre site (c.a.d. `exec` sur un site distant) ou si une entité ainsi désignée migre.

Les portes constituent l'essentiel du contexte d'un processus, qu'elles soient utilisées directement ou incluses dans une *capacité*. Incluses dans des capacités, les portes sont utilisées pour désigner les ressources du processus : fichiers ouverts, segments mappés dans l'espace d'adressage du processus (code, données). Mais, elles peuvent aussi être utilisées pour adresser directement des processus (processus père).

(3) Ressources et Capacités :

Chaque ressource (gérée par un serveur) utilisée par un processus est désignée de manière interne par une capacité : fichier ouvert, tube ouvert, périphérique ouvert, répertoires courant et racine, segments de codes et de données, etc. De telles capacités peuvent être utilisées pour créer des régions de mémoire virtuelle, leur structure est donc celle exportée par le Noyau CHORUS.

Par exemple, l'ouverture d'un fichier associe la capacité retournée par le serveur approprié à un descripteur de fichier ouvert. Cette capacité se compose de la porte du serveur qui gère le fichier d'une part et de la référence du fichier ouvert dans le serveur d'autre part. Par la suite, toutes les requêtes sur le fichier ouvert (par exemple : `lseek(2)`, `read(2)`) sont transcrites et envoyées directement au serveur concerné sous forme de message.

Les serveurs étant désignés par des portes, localisées par l'IPC CHORUS, le sous-système UNIX n'a pas besoin de localiser lui-même ses serveurs.

Les capacités sont construites et retournées par les serveurs. Un serveur peut ainsi déléguer la réalisation d'un service à un autre serveur, sans que les clients sachent quel serveur répond réellement à leurs requêtes.

3.3 Etat

CHORUS-V3 est la version courante du système développée par Chorus systèmes. Des versions précédentes ont été développées dans le cadre du projet CHORUS à l'INRIA entre 1979 et 1986.

CHORUS-V3 est écrit en C++ (et en C). Il est disponible actuellement sur différentes plateformes (machines à base de 680x0, de 88000, de 80386, de transputers, etc.) tant sur des réseaux de stations que sur des configurations multiprocesseurs. Le sous-système UNIX que nous avons présenté offre une interface compatible avec la version 3.2 d'UNIX system V. D'autres sous-systèmes UNIX existent : l'un offre une interface compatible avec la version 4.0 du système SVR4 et un sous-système UNIX compatible BSD est en cours de développement.

Nous présentons dans les parties suivantes d'autres exemples de systèmes d'exploitation répartis - principalement des projets de recherche. Nous avons choisi ces systèmes soit parce qu'ils ont inspiré les concepteurs de CHORUS (V kernel), soit parce qu'ils sont des concurrents de CHORUS par leur similitude (Amoeba, MACH) ou encore parce que certains de leurs aspects pouraient apporter des améliorations à CHORUS (Mosix).

4 Les systèmes à micro-noyau

Notre étude étant centrée sur le système d'exploitation CHORUS/MiX il semble intéressant de le situer plus clairement vis-à-vis d'autres systèmes à base de micro-noyau. Nous avons choisi d'approfondir cette comparaison dans le cas des deux systèmes qui lui sont le plus proches : Amoeba et MACH. Ainsi les descriptions qui en sont données référencient souvent les caractéristiques de CHORUS. Cette comparaison est effectuée à différents niveaux : les fonctionnalités offertes par les systèmes, leur architecture, ou encore leur implantation. Nous analysons également les répercussions engendrées par les différences existantes.

Peu d'utilisateurs ont actuellement pu tester plusieurs systèmes de type micro-noyau afin de les comparer. Il serait en effet intéressant de connaître les implications relatives aux différences fonctionnelles concernant les services de base offerts par ces noyaux. Nous pouvons néanmoins citer [nMKS91].

4.1 Amoeba

Le système d'exploitation Amoeba[Mul87] a été conçu et implanté à l'université de Vrije à Amsterdam. Le projet, commencé en 1980, est développé depuis 1984 conjointement avec le Centrum voor Wiskunde en Informatica (CWI), Amsterdam. Le but de ce projet est de développer un système d'exploitation réparti qui offre des fonctionnalités et des performances équivalentes à un système d'exploitation traditionnel tel qu'UNIX. Ses points forts sont la sécurité et ses performances.

4.1.1 L'architecture du système

Le projet Amoeba vise une architecture d'ordinateurs connectés par un réseau. Parmi ces ordinateurs on identifie quatre composants principaux :

– les stations de travail : ce sont des stations sans disque qui sont utilisées pour supporter un gestionnaire de fenêtres et non pas pour exécuter des programmes complexes.

– un ensemble de processeurs qui peuvent être alloués dynamiquement à la demande pour l'exécution de programmes complexes.
– des serveurs spécialisés comme par exemple un serveur de fichiers ou un serveur de base de données.
– les passerelles : utilisées pour connecter entre eux différents réseaux gérés par Amoeba. Le système global est vu comme formant un ensemble uniforme. Ces connexions ont été testées dans le cas de passerelles donnant accès à plusieurs pays.

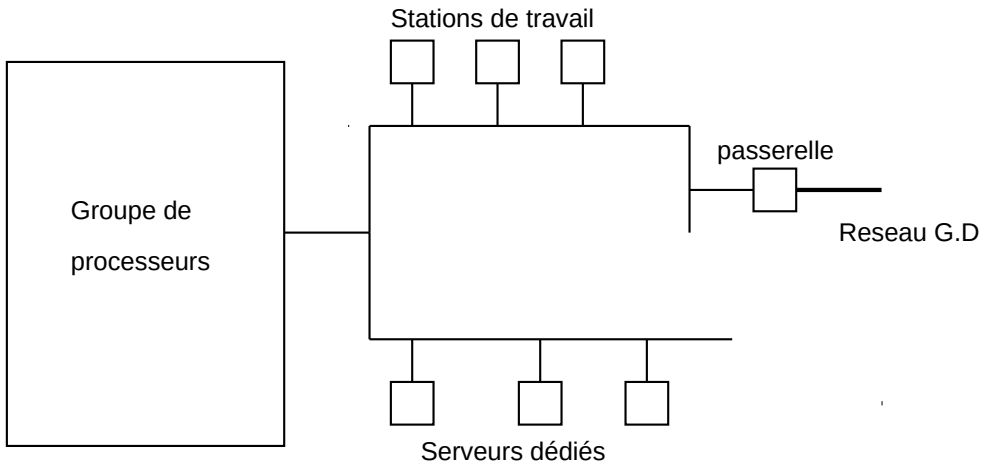

Figure II.9 : Le système Amoeba

Dans un système Amoeba, comme dans un domaine CHORUS, toutes les machines exécutent le même noyau. L'approche architecturale du système est équivalente à celle de CHORUS. L'idée de base est de conserver un noyau de petite taille, les services système étant exportés le plus possible dans des processus utilisateurs. Par contre la réalisation diffère dans la mesure où le noyau Amoeba gère certaines entrées/sorties en plus des processus et des services de communication.

Amoeba est un système basé sur les objets. Il peut être vu comme un ensemble d'objets, chacun contenant les opérations qui le concernent. Par exemple, un objet fichier contient des opérations d'ouverture, de fermeture, de lecture ou d'écriture. Les objets sont implantés par des processus serveurs qui les gèrent. Les clients qui veulent utiliser des opérations sur un objet envoient une requête au serveur par un appel de procédure à distance, chaque objet étant identifié et protégé par une capacité. Ainsi l'interface d'accès aux services est différente entre les systèmes Amoeba et CHORUS. L'utilisateur du système CHORUS accède aux ressources par un *trap*, comme dans le système UNIX tandis qu'Amoeba offre une interface d'accès basée sur des bibliothèques de fonctions - appelées `stub`. Cette implantation ne permet pas, à la version actuelle du système Amoeba, de fournir une interface compatible au niveau binaire avec UNIX.

4.1.2 Le micro-noyau

Les services génériques offerts par le micro-noyau sont basés sur un ensemble d'abstractions. A quelques détails près, ces abstractions sont équivalentes à celles qui sont fournies par le noyau CHORUS.

Les processus et les activités Le modèle d'exécution proposé par Amoeba est multitâches et multi-activités. Un *processus* correspond à un acteur CHORUS et les entités d'exécution sont aussi reconnues comme étant des *activités* ou processus légers. Ces activités partagent également les ressources du processus ce qui leur permet de communiquer - les informations sont dans la mémoire commune - ou de se synchroniser. Par contre l'ordonnancement des activités dans un processus Amoeba est différent de l'ordonnancement des activités dans un acteur CHORUS. Ici, le noyau ne gère que l'ordonnancement entre les processus. La répartition du temps de calcul à l'intérieur d'un processus est effectué par du code résidant dans le processus. Il n'y a pas de priorité associée aux activités et pas de temps partagé entre les activités d'un même processus. De cette façon une activité ne peut pas être interrompue dans son exécution par une autre activité du processus.

Cet ordonnancement a été conçu pour régler les problèmes de concurrence d'accès aux données partagées. Il n'empêche pas la mise en oeuvre d'un ordonnancement de type UNIX au niveau des processus et devrait faciliter le choix de différentes politiques à l'intérieur d'un processus, implantées par l'utilisateur. Par contre, il limite le parallélisme à l'intérieur d'un processus puisqu'une activité ne peut s'exécuter que lorsqu'une autre activité du processus est en attente. Ce type d'ordonnancement ne peut pas, non plus, convenir à l'implantation d'applications temps-réel car celle-ci nécessite une politique globale de l'allocation du processeur.

Comme dans CHORUS, les processus multi-activités sont principalement utilisés pour la programmation de serveurs.

La communication Comme pour les entités d'exécution, les abstractions de base supportant la communication sont très similaires à celles de CHORUS. Les principales abstractions sont les *portes* et les *messages*. De même la structure des messages, même si elle diffère, reste assez proche puisqu'un messages Amoeba est composé d'une entête de 32 octets et d'un corps qui peut aller jusqu'à 30Ko. Néanmoins, la sémantique qui y est attachée n'est pas la même et cela entraîne des fonctionnalités différentes.

Le système Amoeba offre une sémantique de communication qui est uniquement basée sur des échanges de messages synchrones. De plus il n'y a pas de file d'attente associée à une porte. Ces rèstrictions permettent de traiter les messages d'une manière simple ce qui induit de bonnes performances de la communication en général. Par contre le manque de souplesse est sensible lorsqu'il faut introduire une notion d'asynchronisme entre deux entités d'exécution. La communication sert alors principalement dans un modèle client-serveur où une requête est appelée une *opération distante*. Dans Amoeba les appels distants sont effectués dans des routines offertes aux utilisateurs de telle sorte que l'interface apparait exactement comme un appel de procédure. Notons également qu'une entité d'exécution ne peut traiter qu'un seul message à la fois - contrai-

rement à CHORUS - ceci permettant probablement d'améliorer les performances en réduisant la complexité. Cependant ceci entraine qu'un serveur ne peut pas accéder, par envoi de message, à une ressource gérée par un serveur distant, lors du traitement d'une requête. Le système de communication garantit que la requête est délivrée au moins une fois et qu'elle ne l'est qu'aux processus qui sont autorisés à la recevoir.

Les messages du système Amoeba sont typés alors que les messages CHORUS ne le sont pas. Ce typage permet une meilleure protection quant aux droits d'accès à un service pour le modèle client-serveur dans la mesure où le serveur n'accepte que des messages correctement typés. Réciproquement, l'utilisation de messages CHORUS est plus simple et la protection peut être mise en oeuvre par le serveur qui le désire. Par contre, un serveur ne voulant pas subir les contraintes liées à la protection n'y est pas obligé.

Dans Amoeba les portes font partie d'un espace global : toutes les entités d'exécution du système peuvent lire sur une porte. Les requêtes sont envoyées sur des portes dont chacune représente un type de service. Plusieurs serveurs peuvent lire sur une même porte. Cette implantation pose des problèmes de protection, c'est pourquoi dans Amoeba, la protection des ressources accédées à travers une porte a été élaborée très sérieusement[MT84]. Par contre, dans le cas d'un tel schéma, la notion de groupe telle qu'elle implantée dans CHORUS n'est plus complètement nécessaire puisque l'abstraction de porte recouvre, partiellement, cette notion. De même la reconfiguration dynamique est facilitée : il n'y a plus besoin de migrer la porte pour continuer le service. Par contre, Amoeba présente une perte de fonctionnalité,par rapport à CHORUS, puisqu'il n'est pas possible de diffuser un message à un ensemble d'entités d'exécution.

La désignation et la protection La sécurité du système Amoeba repose principalement sur les capacités. Les capacités Amoeba sont équivalentes aux capacités CHORUS. Par contre leur usage a été généralisé à tous les objets du système. L'ensemble des opérations que peut effectuer un utilisateur sur un objet est codé dans la capacité possédée par cet utilisateur. Ainsi deux utilisateurs peuvent avoir des capacités différentes pour décrire le même objet si leurs droits d'accès sont différents. C'est la personne qui conçoit et réalise l'objet qui définit et modifie les droits qui sont attribués aux utilisateurs.

Amoeba utilise également des identificateurs uniques pour décrire les objets - ceci permet au système d'exploitation de trouver facilement le processus serveur d'un objet. L'identificateur unique d'un objet est inclu dans la capacité qui donne l'accès à l'objet. Il contient également l'identificateur du serveur qui le gère.

La protection dans Amoeba repose sur ce codage des capacités. Pour éviter que l'utilisateur modifie les droits qui lui sont attribués, ces droits sont cryptés dans la capacité. Ce cryptage utilise des fonctions non inversibles dont la propriété est la suivante : étant donné la capacité N et les droits non encryptés R il est facile de calculer C = f(N xor R) mais, connaissant C on ne peut pratiquement pas trouver un argument de la fonction f qui permette de générer C. Le serveur possède alors les deux arguments N et C mais le client ne possède que C. Comme il doit fournir la preuve des droits qui lui sont attribués lorsqu'il s'adresse au serveur et qu'il ne peut générer une autre valeur de C il ne peut acquérir des droits plus étendus du fait des propriétés de la fonction f.

La gestion mémoire Amoeba supporte également la notion de segments pouvant être mappés dans l'espace d'adressage d'un processus. Ces segments sont également implantés par des *mappeurs*. Cependant, en s'appuyant sur la chute du coût de la mémoire centrale d'un ordinateur, Amoeba simplifie grandement la gestion de la mémoire virtuelle : la pagination à la demande n'est pas implantée, les segments de mémoire doivent être contigus en mémoire. Dans ce cas, la perte de souplesse et de confort est alors compensée par les gains de performance.

4.1.3 Les serveurs

L'implantation multiserveur a été réalisée pour Amoeba comme pour CHORUS/MiX. Cependant nous ne trouvons pas exactement les mêmes serveurs dans les deux systèmes. Ainsi les principaux serveurs d'Amoeba sont : les serveurs des réseaux (Wide Area Network Server et TCP/IP server), le serveur de processus et de mémoire et les serveurs gérant les fichiers (Directory Server et Bullet Server). Le serveur TCP/IP peut être comparé au serveur de réseau de CHORUS, le serveur de processus et de mémoire fournit les ressources du PM et de la partie mappeur du FM. Enfin les serveurs de fichiers fournissent la partie gestion des fichiers du FM. Le système Amoeba, ne cherchant pas à être compatible au niveau binaire avec UNIX, les codes tels que la gestion des fichiers ont été réécrits entièrement. Par contre dans CHORUS le découpage a été effectué suivant les abstractions UNIX pour obtenir plus facilement la compatibilité. Ceci a également permis de reprendre, dans l'implantation du serveur de fichiers, une partie du code de gestion de fichiers UNIX.

– le serveur de processus et de mémoire : Amoeba n'utilise pas la même stratégie qu'UNIX pour dupliquer les processus (appel système `fork(2)`). L'argumentation repose sur le fait que les processus font généralement appel à la primitive `exec(2)` juste après la duplication due au `fork`. Cette méthode est acceptable pour un système d'exploitation centralisé puisque la duplication n'est pas effectuée réellement. Mais elle ne convient pas à un système d'exploitation réparti dans la mesure où elle représente beaucoup de travail inutile car la copie est envoyée sur un site distant avant d'être détruite. Pour permettre la duplication des processus, Amoeba utilise les concepts de descripteurs de processus et de segments.
  Un segment est un morceau de mémoire contigue" pouvant contenir du code et des données. A chaque segment est associée une capacité. Un segment est en quelque sorte un fichier en mémoire avec des propriétés similaires. Un descripteur de processus est une structure de données qui contient toutes les informations nécessaires à l'exécution du processus : type de processeur utilisé, minimum de mémoire nécessaire, description des segments de mémoire, etc. Pour exécuter un processus sur un site distant il suffit alors d'envoyer cette structure au serveur de processus sur ce site. La duplication est donc effectuée par l'envoi du descripteur de processus à exécuter. Le serveur de processus est également chargé de placer les processus à leur création. Il choisit pour cela un processeur peu chargé dans l'ensemble de processeurs dont il dispose. Le descripteur de processus contient les informations nécessaires à l'identification du support matériel dont le processus à besoin. Le serveur est aussi chargé de gérer l'héritage de ce processus (par exemple : les fichiers ouverts par son père).
– le serveur de fichiers : c'est un processus utilisateur. Plusieurs serveurs de fichiers ont déjà été

écrits dans Amoeba. Le serveur actuel ou "*bullet server*" a été conçu dans le but d'avoir de bonnes performances. L'idée de stocker les fichiers comme des blocs disque de taille fixe (comme dans UNIX) a été abandonnée. Dans ce serveur tous les fichiers sont stokés de façon contigue" à la fois sur le disque et en mémoire. Lorsque le *bullet serveur* s'initialise, il charge entièrement la table des "i-nodes" en mémoire et n'accède pas au disque par la suite pour consulter cette table. Ainsi l'accès à un fichier qui n'est pas en mémoire, nécessite une seule lecture sur le disque pour le charger. Ce serveur de fichier ne permet pas la modification des fichiers ; ceux-ci ne pouvant être que créés ou détruits. En considérant que ces dernières années le coût des disques et des mémoires a diminué considérablement, le gain de performances compense le surplus d'occupation des disques et de la mémoire (environ 20%).

– le serveur de répertoires : le "*bullet serveur*" n'offre aucun service de nommage, les fichiers ne sont décrits que par des capacités. Le serveur de répertoires gère les noms et les capacités. Par exemple, lorsqu'un processus veut accéder un fichier, il envoie au serveur de répertoires une capacité représentant le répertoire (cette capacité contient les droits d'accès) ainsi que le nom du fichier. Le serveur lui retourne la capacité correspondante.

4.1.4 Les réseaux à grande distance

L'idée de base d'Amoeba est qu'un ensemble de machines connectées par un réseau local doit être capable de communiquer à travers un réseau à grande distance avec un ensemble de machines similaires. Dans ce cas le but d'Amoeba est de fournir une communication transparente tout en conservant des performances satisfaisantes.

Les systèmes Amoeba sont divisés en domaines, chaque domaine étant composé d'une interconnexion de réseaux locaux. Deux domaines peuvent être connectés par un lien. Les machines réalisant la connexion des deux domaines sont appelées des passerelles. Sur chacune d'elles est exécuté un processus lien chargé de la gestion de cette liaison. Si une machine du domaine diffuse un message, celui-ci sera reçu par toutes les machines de ce domaine et ne sera pas reçu par une machine extérieure à ce domaine.

Il n'est pas nécessaire que tous les services soient connus de toutes les machines du système. Pour permettre ceci la notion de service "public" a été introduite. Si un service veut être connu à l'extérieur de son propre domaine, il contacte le processus chargé de la liaison à grande distance ou SWAN (Service for Wide Area Network) de son domaine et lui demande que sa porte soit publique dans un ensemble de domaines. Pour réaliser ceci le SWAN effectue des RPC aux SWAN des domaines concernés pour leur communiquer la porte du service. Quand un service est rendu public dans un domaine, un processus "agent serveur" correspondant à ce service est créé. Ce processus s'exécute sur la machine passerelle et se met en attente de requêtes sur la porte reçue.

Lorsqu'un processus veut accéder au serveur distant dont la porte a été publiée, le noyau effectue une diffusion à laquelle l'agent serveur correspondant répond. L'agent serveur construit un message et l'envoie au processus lien local. Ce processus transmet le message à travers le réseau grande distance à son homologue distant. Lorsque le message arrive sur la passerelle, un processus agent client est créé. Cet agent client effectue un RPC ordinaire au serveur concerné.

De cette manière la communication entre un client et un serveur distant - c'est-à-dire résidant

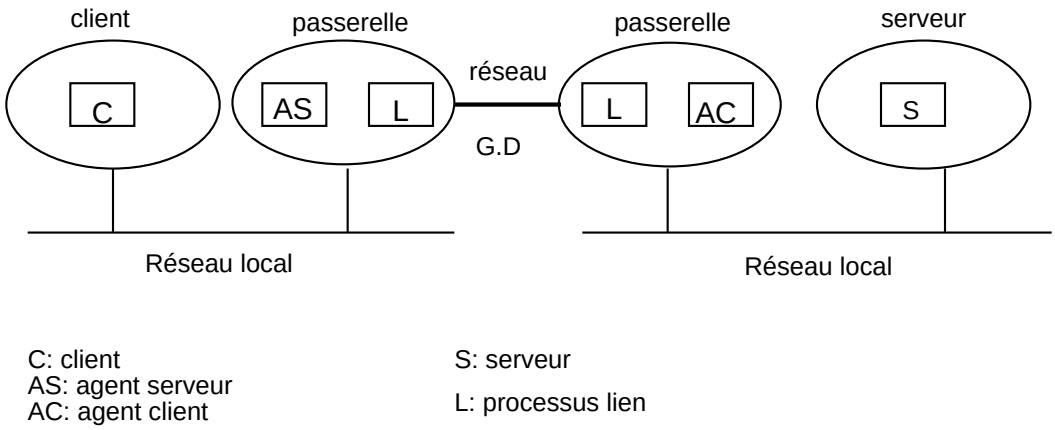

Figure II.10 : Les réseaux dans Amoeba

sur un autre réseau local - est totalement transparente, le client effectue un RPC normal. Seul le processus lien résidant sur la passerelle doit savoir gérer le protocole d'accès au réseau grande distance. Les autres processus effectuent des RPC sans se soucier de problèmes techniques. La communication à l'intérieur d'un même réseau local conserve donc de bonnes performances, seule la communication à travers un réseau à grande distance est plus lente. Ce schéma permet également de vérifier que les processus qui veulent accéder à un service extérieur au réseau local en ont bien le droit.

4.1.5 Implantation

Le système Amoeba a obtenu de très bonnes performances. Malheureusement il semble que cela soit au détriment de fonctionnalités étendues. Il n'y a pas de mémoire virtuelle paginée dans Amoeba et il n'y en aura pas dans la prochaine version du système. Les concepteurs du système justifient ce choix par la différence de coût élevé de l'implantation d'une gestion de mémoire virtuelle paginée par rapport au faible coût des extensions mémoire.

Le mode de communication RPC ne permet que la communication point à point : on ne peut pas effectuer de diffusion, ni de diffusion restreinte. Dans la prochaine version cela sera possible. De même cette version devrait permettre un meilleur ordonnancement entre les activités d'un même processus.

Amoeba ne propose pas une interface standard, il offre cependant une émulation d'UNIX. Mais l'implantation de cette émulation ne permet pas une compatibilité binaire, les utilitaires standards ont donc du être réécrits pour être portés. De ce fait le système Amoeba semble peu convenir aux besoins des multicalculateurs.

4.2 MACH

MACH est dévelopé par l'université Carnegie-Mellon, USA[ABBa86]. Ses concepts sont inspirés des résultats de la conception et expérimentation des systèmes RIG[Ras88] et Accent[RR81]. Contrairement à CHORUS ou Amoeba, le noyau MACH n'a pas été developpé en réécrivant la totalité du système. Afin de faciliter une grande diffusion de MACH, ses concepteurs ont préféré intégrer MACH au sein d'un noyau Unix BSD 4.3. Une version modulaire de MACH a ensuite été développée en implantant hors du noyau les services UNIX. L'architecture qui en résulte est un micro-noyau supportant l'exécution d'un sous-système UNIX monoserveur. Une version multiserveurs est en cours de développement.

L'intérêt de cette implantation monoserveur est évident : en s'appuyant sur un système existant le système MACH a obtenu la compatibilité binaire avec le système UNIX de référence à moindre frais. Il a suffi ensuite de séparer l'interface basse du système pour construire le micro-noyau mais cela n'a que peu changé le niveau haut de l'interface. Nous pouvons justifier le fait que l'implantation monoserveur des sous-systèmes permet de mieux réutiliser le code existant dans la mesure où l'environnement de ce code est conservé. Par contre, dans CHORUS/MiX, pour pouvoir isoler le code UNIX de la gestion des fichiers dans un serveur il a fallu simuler l'environnement UNIX dans ce serveur. Ce type d'opération n'a pas été nécessaire dans MACH.

Ce type d'architecture comporte cependant des désavantages. Par exemple, les interactions entre les parties de code composant le sous-système ne sont pas modifiées par rapport au système UNIX, et donc pas clarifiées. D'autre part, cette définition claire entre les services, basée sur les échanges de messages, permet à CHORUS/MiX de bénéficier de la transparence de l'IPC. Le manque de modularité est un autre désavantage. En effet, le sous-système monoserveur est composé de la totalité du code du système afin de pouvoir offrir les mêmes fonctionnalités que le système initial, le tout encapsulé dans un seul serveur. Il n'est alors pas possible d'adapter le système à une configuration particulière de machine : par exemple, sur un noeud sans disque on ne peut pas supprimer le code de gestion des fichiers. De plus les services UNIX ne sont plus rendus en utilisant l'échange de message et ne bénéficient pas ainsi de la transparence.

4.2.1 Le micro-noyau

Comme les noyaux Amoeba et CHORUS le noyau MACH implante une interface basée sur un ensemble d'abstractions - équivalentes à celles des deux autres micro-noyaux. La similitude des services offerts par trois micro-noyaux peut être considérée comme la preuve de l'adéquation des services offerts. Notons que, comme pour le noyau Amoeba, les fonctionnalités associées aux abstractions de MACH diffèrent de celles définies dans CHORUS.

Les tâches et les activités Le noyau MACH offre également un modèle de programmation multitâches et multi-activités. Les *tâches* de MACH sont similaires aux acteurs de CHORUS et les entités d'exécution sont aussi reconnues comme étant des *activités*. Toutes les activités d'une même tâche ont, de la même façon, accès aux ressources contenues dans la tâche. Par contre le noyau n'offre pas de service explicite de synchronisation - tel que les sémaphores - aux activités

partageant les mêmes ressources. Cela doit être géré par l'utilisateur, par exemple en utilisant les services de communication.

Les tâches et les activités sont gérées par l'utilisateur à partir de fonctions, comme dans CHORUS. Les activités sont ordonnancées de façon indépendante par le noyau mais cet ordonnancement ne satisfait pas les contraintes temps-réel. Pour compléter le service d'ordonnancement offert par le noyau, une interface permet à l'utilisateur de gérer lui-même l'exécution de son programme. Cette possibilité d'intégrer facilement de nouvelles politiques d'allocations des processeurs a permis de faciliter le portage de MACH sur les plateformes multiprocesseurs à mémoire partagée. Sur un tel multiprocesseur les activités d'une même tâche peuvent être exécutées concurremment sur des processeurs différents. Elles sont alors ordonnancées en groupe (en anglais : gang-scheduling[Bla90]). Les activités peuvent tout de même exécuter les primitives du noyau ou les fonctions d'interface en parallèle.

MACH offre une interface de traitement des exceptions qui permet à un processus utilisateur de gérer ses exceptions ou celles d'un autre programme. CHORUS réserve cette fonctionnalité aux acteurs privilégiés et Amoeba ne fournit pas de service équivalent. Le noyau MACH traduit les exceptions, comme par exemple un accès à une adresse invalide ou protégée, en messages qui sont délivrés à un processus désigné.

La communication Les abstractions représentant la communication inter-activités dans MACH sont les mêmes que dans CHORUS et Amoeba : les *messages* et les *portes*. L'interface de communication permet des échanges synchrones et asynchrones, comme dans CHORUS, mais les services diffèrent.

Un message est composé d'une entête de taille fixe et d'un corps de taille variable. Comme dans Amoeba, les messages sont typés, c'est-à-dire que dans le corps du message, chaque objet est précédé d'un descripteur de type (ex : entier, flottant, caractère), de sa taille et du nombre d'objets de ce type contenu dans le message à la suite de celui-ci. Le noyau définit un certain nombre de type de données, en particulier ceux qui sont utilisés par l'interface du noyau. Les utilisateurs sont autorisés à définir des types supplémentaires quand cela est nécessaire.

Comme dans CHORUS, l'envoi de messages est couplé à la gestion de la mémoire afin de faciliter l'envoi de messages longs. Le noyau MACH implante la communication au-dessus du protocole TCP/IP.

Une porte est un canal de communication sur lequel les messages peuvent être reçus ou envoyés. De manière logique, une porte est une queue de messages de longueur finie, comme dans CHORUS. Une porte est protégée par des droits d'accès, ainsi plusieurs tâches peuvent envoyer un message sur une porte déterminée mais seulement une tâche aura les droits d'accès pour recevoir ce message sur cette porte. Une porte est désignée par un entier. Cet entier comprend les droits qui sont attribués à l'utilisateur.

Dans ses première implantations, MACH offrait uniquement la possibilité d'envoi de messages asynchrones. Le RPC a été introduit plus tard, construit à partir des primitives d'émission et de réception, à la différence du RPC de CHORUS qui a été conçu comme une fonctionnalité de base au même titre que l'échange de messages asynchrone. Pour permettre l'implantation du RPC

et en particulier l'implantation du modèle client serveur, le noyau MACH offre un droit d'accès unique à une porte. Cette différence de conception induit des différences d'interface. Par exemple, il est nécessaire de préciser la porte de l'initiateur du RPC pour répondre à une requête, car la réponse à un RPC est dérivée de la primitive d'envoi, dans laquelle il faut préciser le destinataire. Ceci oblige un serveur à connaître la porte de son client. Dans CHORUS, la réponse d'un RPC est automatiquement retournée à la porte d'où a été émise la requête.

Les messages peuvent contenir des droits d'accès à des portes dans leur corps. Lorsqu'il s'agit des droits d'envoi sur une porte, les droits sont dupliqués : ainsi l'expéditeur et le récepteur auront les droits d'envoi. Lorsque ce sont les droits de réception sur une porte qui sont envoyés, l'expéditeur perd ses droits de réception au profit du destinataire du message. Grâce à cette fonctionnalité MACH n'a pas besoin d'implanter la migration d'une porte puisque le transfert des droits d'accès à cette porte aboutit au même résultat. D'autre part, une tâche peut également désallouer ses droits d'accès à une porte. Quand les droits de réception sur une porte sont désalloués, la porte est détruite et les tâches qui possèdent des droits d'envoi en sont averties. Quand tous les droits d'envoi sont désalloués, le récepteur peut choisir d'être prévenu soit de façon asynchrone (au moment où les derniers droits sont désalloués) soit la prochaine fois qu'il essaie de recevoir sur la porte.

Chaque entête de message contient l'identification d'une porte destinatrice et d'une porte de réponse. Ces deux identificateurs font implicitement partie du message. C'est-à-dire que le récepteur du message possède automatiquement les droits d'envoi sur la porte de réponse. Ceci permet également de transférer un message reçu sur une porte à une autre porte, sans que l'initiateur de la requête détecte le renvoi. Par exemple, le gestionnaire du réseau utilise cette technique. Quand un message est envoyé à une porte distante il est, en réalité, envoyé à la porte locale du gestionnaire de réseau qui transfert ce message au gestionnaire de réseau sur le site distant. Le message est ensuite mis en queue derrière la porte distante. Cette indirection permet de mettre en place une communication transparente à la répartition. C'est-à-dire que la sémantique d'accès à une porte est la même quelque soit la localisation de la porte destinatrice sur le réseau.

MACH définit une notion d'**ensemble de portes** équivalente à celle proposée par CHORUS mais il en autorise plusieurs par acteur. Un ensemble de portes permet à un serveur de se mettre en attente de message sur plusieurs portes. Une porte pour laquelle une tâche possède les droits de réception peut être insérée dans un ensemble de portes. Lorsqu'un serveur effectue une réception, il peut préciser un ensemble de portes à la place d'une porte ; il reçoit alors les messages d'une porte quelconque de l'ensemble. Une porte ne peut pas être membre de plus d'un ensemble à la fois. Lorsqu'une porte est insérée dans un ensemble de portes, les messages qui arrivent sur cette porte ne peuvent plus être lus que par l'ensemble de portes. C'est-à-dire qu'une requête de lecture sur cette porte échouera. De même que les portes, les ensembles de portes sont représentés par des entiers, par contre, ils ne véhiculent pas de droits d'accès. De plus les portes et les ensembles de portes partagent le même espace de nommage dans une tâche. Les ensembles de portes n'introduisent pas de notions supplémentaires d'ordonnancement à la réception. C'est-à-dire qu'il n'y a pas de priorités associées aux portes dans l'ensemble.

Le noyau MACH n'offre qu'une seule garantie sur l'ordre des messages : si une tâche envoie plusieurs messages sur une porte, il seront reçus selon leur ordre d'émission. Il n'y pas de garantie

d'ordre de réception dans le cas où l'émission de messages est effectuée par des tâches différentes sur une même porte, ou pour des messages issus d'une même tâche envoyés sur plusieurs portes. Il n'y a pas, dans MACH, de notion équivalente à celle des groupes de portes CHORUS.

La gestion de mémoire virtuelle La gestion de mémoire virtuelle implantée par MACH est proche de celle de CHORUS : elle est paginée à la demande et elle gère des régions de mémoire qui représentent des espaces d'adressage. Le système complète aussi ses fonctionnalités en offrant la possibilité de mapper des parties de mémoire secondaire : les *objets mémoire* correspondent aux segments de CHORUS. Dans les deux cas, le noyau fait appel à un serveur pour gérer les segments. L'interface de mémoire virtuelle offre les fonctions de base pour gérer le contenu de l'espace d'adressage virtuel : allocation, libération et protection d'adresse mémoire. Elle offre de plus la possibilité d'établir des relations de mémoire partagée. La réalisation de l'héritage de régions mémoire - par exemple dans le cas d'un `fork(2)` est différente de celle de CHORUS. Ces différences sont décrites dans [ARS89].

Une implantation de mémoire partagée répartie a été réalisé, comme pour CHORUS, dans un mappeur.

L'addressage Les portes sont aussi utilisées pour représenter les objets, comme dans Amoeba. Un message, envoyé sur une porte qui représente un objet, peut être interprété par le gestionnaire de l'objet comme une requête de traitement sur l'objet, suivant le modèle client-serveur. En ce sens, le noyau MACH peut être vu comme un serveur de tâches et d'activités puisque l'accès à celles-ci se fait au travers de portes.

Dans MACH les objets sont référencés par des identificateurs locaux. Le *netmsg server* est chargé de transformer ces identificateurs locaux lorsqu'ils sont échangés d'un site à un autre. Les messages étant typés il est facile d'y reconnaître ces descripteurs pour les transformer. Ces descripteurs doivent alors être uniquement transmis au travers de messages pour rester valables.

Une activité s'exécutant peut acquérir un droit sur les portes que le noyau utilise pour représenter cette activité et sa tâche. Une tâche peut s'allouer des portes pour représenter ses propres objets ou pour exécuter des communications. Une tâche peut aussi désallouer ses droits sur une porte.

4.2.2 Le sous-système UNIX

Contrairement au sous-système UNIX de CHORUS, le sous-système UNIX de MACH ne permet pas l'accès transparent aux ressources. En effet, ce sous-système complète l'interface UNIX en ajoutant des services, tels que les messages et les activités, mais n'étend pas les fonctionnalités standards d'UNIX à la répartition. Ainsi la gestion des ressources telles que les fichiers est la même dans UNIX et dans MACH.

La version 2.5 du noyau MACH, intègre ces concepts directement dans le noyau UNIX BSD 4.3 pour obtenir la compatibilité à moindre frais. Dans ce cas il n'y a pas réellement de sous-système UNIX. La seule remarque que nous pouvons faire est que, dans cette version de MACH, les services du système UNIX sont complétés pour permettre à l'utilisateur de mieux prendre en compte la

répartition de ses ressources.

La version 3.0 du noyau MACH est composée d'un micro-noyau qui supporte un sous-système UNIX monoserveur. Ce serveur est compris dans une seule tâche multi-activités. Il émule les appels système UNIX en s'appuyant sur le micro-noyau et il sert de pageur externe au micro noyau. Chaque requête arrivant à ce serveur est traitée par une activité désignée parmi un ensemble d'activités en attente.

Sur la base de ce micro-noyau, l'Open Software Foundation (OSF) à l'intention de développer un autre sous-système UNIX basé sur plusieurs serveurs[Fou90], semblable au sous-système UNIX de CHORUS.

4.2.3 Implantation

Le noyau MACH a été porté sur un grand nombre d'architectures. Principalement des architectures centralisées ou multiprocesseurs symétriques. Par contre, cette architecture monolithique ne permet pas d'envisager de bonnes facultés d'adaptation sur des multiprocesseurs à mémoire distribuée, c'est cette raison qui a motivée le développement d'un sous-système UNIX multiserveur.

Le noyau MACH est actuellement utilisé par l'OSF pour générer son système OSF/1 [Fou91]. Ce système doit être porté sur des machines multiprocesseurs à mémoire distribuée qui seront commercialisées par ISSD. Pour effectuer ce portage l'OSF éclate le sous-système UNIX de MACH en plusieurs serveurs.

4.3 Plan 9

Plan 9 [PPTa90] [PPTa91] est un système qui a été récemment développé aux Bell Labs. Ses concepteurs le définisse comme un environnement logiciel physiquement réparti entre plusieurs machines. L'originalité du système consiste en un accès uniforme aux ressources en les représentant toutes sous forme de fichiers. L'approche Plan 9 est intéressante mais semble très dépendante de l'architecture cible choisie (figure II.11). Nous pouvons remarquer que cette architecture cible est proche de celle visée par Amoeba. Le système Plan 9 n'est donc pas facilement adaptable aux multicalculateurs.

4.3.1 Généralités

En prenant exemple sur le modèle qui a rendu UNIX populaire, les concepteurs de Plan 9 affirment qu'un petit nombre d'abstractions est suffisant pour offrir la plupart des fonctions d'un système général. Plan 9 offre les mêmes fonctions qu'UNIX.

L'architecture cible choisie est la suivante :

– des serveurs d'exécution sans disque concentrent la puissance de calcul dans de gros multiprocesseurs,
– des serveurs de fichiers offrent de grandes possibilités de stockage,

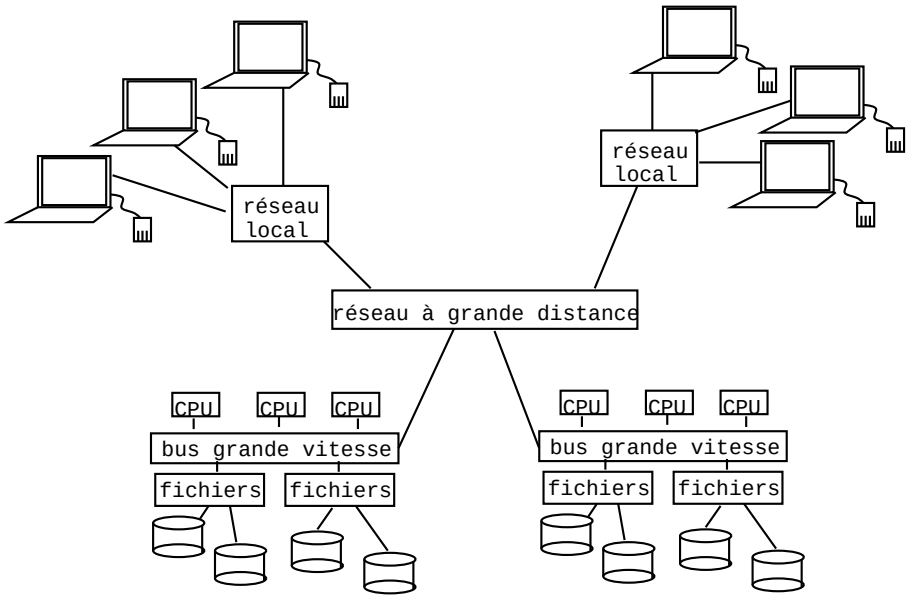

Figure II.11 : Architecture visée par Plan9

– chaque utilisateur du système possède son propre terminal, avec un écran sur lequel il dispose d'un système de fenêtrage.

Cette vue centralise l'administration et donc la simplifie. Dans cette vue, les utilisateurs exécutent les programmes interactifs localement et utilisent le serveur pour les exécutions demandant plus de capacité de traitement.

4.3.2 Minimalisme

Toutes les ressources accédées par un processus, à l'exception de la mémoire du programme, sont décrites par un seul espace de nommage et sont accédées de façon uniforme. Pratiquement chaque type de ressource est implanté sous forme d'un système de fichiers, il est donc appelé système de fichiers. Un système de fichiers peut être d'un type traditionnel, représentant un stockage permanent sur un disque. Il peut aussi représenter des périphériques tels que des terminaux ou encore des abstractions plus complexes que sont les processus. Un système de fichiers peut être implanté par un pilote du noyau, un processus utilisateur ou un serveur distant. Par exemple, un système de fichiers représentant une ligne RS232 sera un répertoire contenant les fichiers *data* et *ctl*. Le fichier *data* contient une suite d'octets transmis/émis sur la ligne. Le fichier *ctl* est un canal de contrôle utilisé pour changer les paramètres de la ligne comme par exemple la vitesse. D'autres systèmes de fichiers représentent des composants logiciels : les processus sont des répertoires contenant des fichiers séparés qui représentent la mémoire, le texte et le contrôle. Beaucoup d'appels système sont ainsi remplacés par des accès à des fichiers. Par exemple, pour

obtenir l'identificateur d'un processus il suffit de le lire dans un fichier. Chaque système de fichiers, donc chaque ressource, est alors géré par un pilote, comme le sont les périphériques sous UNIX. Lorsque la ressource est locale le traitement est assuré par le site d'exécution. Si la ressource est distante le pilote transforme la requête en un message sur un canal de communication. Cette manière de gérer les ressources a permis la réduction de la taille du code du noyau puisque elles sont toutes traitées de la même manière.

4.3.3 L'espace de nommage virtuel

L'espace de nommage se présente sous forme d'une arborescence. Quand un utilisateur se connecte, un nouveau groupe de processus est créé pour son processus. Ce groupe de processus s'initialise avec un espace de nommage contenant au moins une racine (/), quelques binaires pour le processeur sur lequel il s'exécute (/bin/*) et quelques périphériques (/dev/*). Les processus du groupe peuvent ensuite soit augmenter soit réarranger leur espace de nommage en utilisant deux appels système : *mount* et *bind*. Ces appels système sont utilisés pour attacher de nouveaux systèmes de fichiers, c'est-à-dire des accès à d'autres ressources, à un point de l'espace de nommage initial. Un certain niveau de transparence est obtenu grâce à la description identique des ressources, qu'elles soient distantes ou locales.

4.3.4 La commande du CPU

Les serveurs d'exécution sont considérés comme des accélérateurs pour les terminaux, les commandes s'y exécutent tout en conservant leur environnement initial. Il est important qu'il y ait le minimum de changement possible lorsque l'on s'exécute sur le serveur d'exécution : ceci est possible grâce à l'espace de nommage virtuel. Une commande, *cpu*, appelle un serveur d'exécution à travers le réseau. Sur le serveur, un processus crée un nouveau groupe de processus pour le client, initialise un espace de nommage et lance un processus *shell* dans le nouveau groupe de processus. Le nouvel espace d'adressage, créé par le serveur, est analogue à celui du processus client. Seuls les exécutables sont modifiés pour devenir ceux du serveur d'exécution et ainsi gagner en temps d'accès.

4.4 V kernel

Le V kernel [Che88] est un projet relativement ancien par rapport aux projets présentés dans cette section. Il mérite néanmoins de figurer dans cette liste car il a beaucoup influencé la conception des autres systèmes d'exploitation répartis, en particulier en ce qui concerne la communication. Il provient lui-même de l'évolution des deux systèmes d'exploitation Thoth et Verex.

Le noyau V a été développé à l'université Stanford. Son but était de connecter des stations de travail et de permettre des accès et une localisation transparents tout en conservant un niveau acceptable de performances. Un petit noyau offre des primitives de communication efficaces. Il est utilisé par des serveurs qui complètent ses fonctionnalités pour offrir un niveau de service équivalent à celui de la plupart des systèmes d'exploitation.

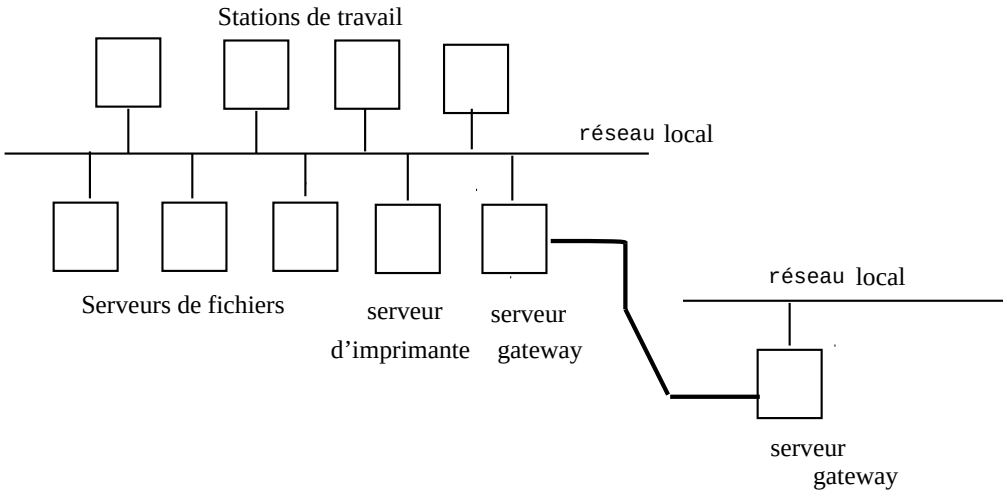

Figure II.12 : Le système V kernel

4.4.1 Le noyau

Le noyau V est composé de trois parties principales : la **communication inter-processus** ou **IPC**, le **serveur noyau** et le **serveur de périphériques**. La connexion entre les différents composants du système repose sur la communication permettant ainsi de bénéficier de la transparence du service d'IPC. Le serveur noyau est un pseudo-processus qui implante la gestion des processus et la gestion de la mémoire virtuelle. Ces services sont accessibles à travers l'IPC. Le serveur de périphériques est similaire au serveur noyau et offre l'accès aux périphériques à travers une interface basée sur l'échange de messages, appelé "V I/O protocole".

La communication inter-processus L'interface de communication par messages de V kernel peut être comparé à celle de CHORUS. Les messages ont une longueur fixe de 32 octets - ils sont équivalents aux annexes de message CHORUS - et les processus peuvent donner, en paramètre d'un message, l'accès à un segment de mémoire pour le copier dans l'espace d'adressage du destinataire - ce qui équivaut à l'envoi d'un corps de message. Les noyaux utilisent un protocole spécifique au V kernel pour communiquer. La communication est principalement dédiée au mode client/serveur puisque les primitives de communication sont uniquement synchrones.

Les services de communication de V implantent la notion de groupe de communication. Un processus peut donc envoyer un message à un groupe de processus. Il est alors en attente tant qu'au moins un des membres de ce groupe ne lui a pas répondu. Les appels de base concernant

l'utilisation des groupes sont équivalents à ceux définis dans CHORUS. Les groupes sont également utilisés pour connaître dynamiquement l'identificateur d'un serveur. En effet pour accéder à un serveur, un processus doit connaître son identificateur, ce qui n'est généralement pas le cas lors d'un premier accès (sauf si le serveur est son père).

Les objets dans V sont désignés par des noms globaux et uniques. Ces noms sont obtenus en ajoutant à chaque nom d'objet le nom de son serveur. Si un nom n'est pas connu, un message est diffusé à tous les serveurs possibles pour l'acquérir.

4.4.2 Le serveur noyau

Le serveur noyau est un serveur implanté directement par le noyau. Il offre une interface, accessible par messages, implantant la gestion des processus et de la mémoire. En fait, ce n'est pas réellement un processus mais un ensemble de procédures du noyau. En utilisant l'IPC, les processus peuvent être contrôlés à distance par émission de requêtes au serveur noyau distant. L'entité d'exécution décrite dans V se rapproche de celle d'un processus léger. Ces processus peuvent partager des espaces d'adressage, ils sont alors regroupés en *équipe*. La notion d'équipe est équivalente à celle d'acteur CHORUS ou de tâche MACH. A chaque processus est associée une priorité d'exécution.

Le serveur noyau du V kernel offre la possibilité de migrer les processus [TLC85] soit pour permettre la tolérance aux pannes soit pour implanter un ordonnancement global sur le réseau de stations de travail. Pour obtenir de bonnes performances les processus ne sont pas migrés dès que la commande de migration est donnée. Leur contexte et leur espace d'adressage est d'abord envoyé au site destinataire qui alloue les structures nécessaires à la réception du processus. Pendant ce temps le processus continue de s'exécuter sur le site d'origine. Lorsque le site destinataire a effectué l'installation du processus il reçoit une mise à jour des données modifiées, entre temps, par le processus. Si la taille des données modifiées n'est pas trop importante, le processus est stoppé, la mise à jour est faite puis le processus est redémarré sur le site destinataire. Cet algorithme permet de réduire le temps de transfert d'un processus, une analyse en est donnée au chapitre IV.

Le service de migration de processus a été utilisé pour mettre en oeuvre un équilibrage de charge sur le réseau de machines. Cette implantation utilise largement la notion de groupe de processus supportée par le V kernel.

4.4.3 Traitement des exceptions

Le traitement des exceptions et interruptions n'est pas intégré dans le noyau. Lorsque l'exception ne le concerne pas le noyau oblige le processus générateur de l'exception à envoyer un message à un serveur d'exceptions. Ce serveur est externe au noyau. Un processus, tel qu'un deboggeur, peut se déclarer responsable des exceptions d'un autre processus. Ainsi, lorsque le serveur reçoit le message décrivant l'exception il la transmet au processus deboggeur. Ceci permet d'avoir un traitement minimum pour les exceptions de la part du noyau. Ce mécanisme a aussi été utilisé pour résoudre les défauts de page d'un processus. Il permet à un processus d'accèder à un segment de mémoire distant.

5 D'autres systèmes

5.1 Locus

Le système LOCUS est une version distribuée d'UNIX qui rend le réseau entièrement transparent à l'utilisateur. C'est un **système à image unique** - c'est-à-dire un système qui offre à l'utilisateur la vue d'une machine monoprocesseur virtuelle à partir d'un réseau d'ordinateurs - qui gère des réseaux de stations de travail. La transparence est supportée même dans un environnement hétérogène. Le système LOCUS a été développé à UCLA (Los Angeles) à partir d'un noyau UNIX qui a été modifié par étapes afin d'y intégrer la répartition. Comme pour le noyau MACH cette démarche permet d'obtenir une compatibilité au niveau binaire à moindre frais. Le système LOCUS offre des services de tolérance aux pannes. Nous nous sommes principalement intéressés aux aspects système à image unique.

5.1.1 La transparence

La transparence est l'idée de base et l'originalité du système LOCUS. Les applications existantes (sous UNIX) peuvent s'exécuter sans aucune modification sous LOCUS tout en tirant tout de même des bénéfices de la répartition. En effet, la transparence assurée par le système permet de distribuer toutes les ressources utilisées par cette application sans impliquer de problèmes d'accès aux ressources. Par exemple, pour une application standard, il suffit de positionner quelques paramètres de configuration pour qu'une partie de l'application s'exécute sur une machine et l'autre partie sur une machine différente. Ceci permet d'offrir sans modification un grand nombre d'outils même dans un environnement réparti.

Le système LOCUS permet également de contrôler la répartition des données sur le réseau et les accès aux données, aux périphériques, aux processeurs et aux ressources logiques. Ainsi, tout en offrant un environnement transparent aux utilisateurs, LOCUS n'empêche pas d'optimiser les applications en fixant le placement des ressources et des processus. Pour un utilisateur qui ne veut pas prendre en compte l'éventuelle distribution de ses programmes le système LOCUS offre une interface semblable à celle d'une machine monoprocesseur. Ainsi les processus opèrent et interagissent avec les fichiers et les autres processus à travers le réseau de la même façon et avec le même effet que localement. En adhérant à cette philosophie, le développement d'applications réparties est considérablement simplifiée.

L'hétérogénéité des processeurs a été prise en compte par LOCUS. Ainsi, lorsqu'une tâche doit s'exécuter sur un processeur, c'est l'instance du binaire adapté à ce processeur qui est exécutée automatiquement (à condition qu'elle existe).

5.1.2 La gestion globale de tâches

La gestion globale de tâches mise en place s'appuie à la fois sur les appels système standards d'UNIX, `fork(2)` et `exec(2)`, et de nouvelles fonctions offertes par le système LOCUS. Ainsi le système implante deux nouveaux appels système : l'appel `setxsites()` agit, à la manière de

`csite(2)` dans CHORUS, sur le site d'exécution du fils du processus et l'appel `migrate()` permet de changer de site un processus pendant son exécution.

La répartition de la charge sur le réseau est alors implantée par un processus démon qui observe la charge du système et, si besoin, migre les processus vers des sites moins chargés. Une forme simple de répartition de charge a été obtenue en exécutant à distance les commandes données à l'interpréteur de commande par l'utilisateur.

De même certaines applications possédant un parallélisme inhérant ont été parallélisées pour tirer un meilleur parti de la répartition des ressources. Par exemple, la commande `make` peut facilement être exécutée en parallèle puisqu'elle est composée de compilations intermédiaires qui n'interagissent qu'à la fin de leur exécution à travers des fichiers.

5.2 Mosix

Le système MOSIX[BW89] a été conçu pour gérer des ensembles d'ordinateurs composés aussi bien de multicalculateurs que de monoprocesseurs ou de stations de travail interconnectées par un réseau local. Le but du projet est de fournir un système unique, offrant une interface compatible UNIX, sur ce type de machines. La base de départ du projet est un noyau UNIX restructuré afin de séparer les parties dépendantes de la machine de celles qui ne l'étaient pas. Ainsi l'interface obtenue est indépendante du processeur géré par le système. Il ne s'agit cependant pas d'un micro-noyau.

Une propriété importante de MOSIX est la possibilité pour le système de gérer un grand nombre de processeurs. Cette propriété est obtenue, d'une part, en s'assurant que les interactions entre noyaux n'impliquent jamais plus de deux processeurs en même temps et d'autre part, en limitant les accès au réseau pour chaque noeud. Ainsi, chaque processeur a une surcharge fixe quelle que soit la taille du réseau.

5.2.1 Architecture du noyau

Le noyau MOSIX est composé de trois couches :

- La *partie basse* comprend les procédures qui accèdent aux ressources locales telles que les pilotes de disques ou les structures dépendantes du processeur. Cette couche est étroitement liée au processeur local et elle ne peut pas faire la différence entre une requête locale et une requête provenant d'un autre processeur. Cette couche connait parfaitement tous les objets locaux au site, comme les fichiers.
- La *partie haute* est indépendante du processeur et ne connaît pas l'identificateur du processeur sur lequel elle s'exécute. Son interface offre les appels système UNIX standards. Cette couche connaît la localisation de tous les objets qu'elle manipule. Par exemple, pour l'ouverture d'un fichier c'est elle qui analyse le chemin d'accès et détermine quel est le site qui gère ce fichier. Ensuite l'appel est transmis à la couche de communication avec, en paramètre, l'identificateur du site sur lequel la requête doit être exécutée.
- Le *lieur* reçoit les appels de la partie haute et détermine si l'appel est local ou distant. Il transmet ensuite la requête à la partie basse correspondante.

Le protocole de communication entre les sites a été conçu de manière à être sûr. En effet, les informations transmises à ce niveau sont très sensibles. Les messages sont ordonnés pour chaque paire de noeuds.

Pour prendre en compte la distribution des fichiers, le système de fichiers permet d'établir des liens entre les machines. Les fichiers sont toujours gérés de façon locale par rapport au disque sur lequel ils résident. Un algorithme de ramasse-miette est utilisé pour supprimer les références à un fichier en cas de panne d'une machine.

5.2.2 Implantation

La version originale de MOSIX, appelée MOS, était compatible avec UNIX version 7. Elle gérait des PDP-11 interconnectés par un réseau local ProNet-10. Les versions suivantes de MOSIX ont été développées pour des Vax de Digital et des ordinateurs basés sur des processeurs du type NS 32000. La version la plus récente de MOSIX gére également des stations de travail multiprocesseurs.

Conclusion

Les systèmes d'exploitation actuellement proposés sur les multicalculateurs sont en général des systèmes minimaux. C'est-à-dire qu'ils offrent un niveau de fonctionnalités réduit, donc un confort minimum, aux utilisateurs de ce type de machines. Ces systèmes d'exploitation implantent principalement des primitives de communication et une gestion minimum des noeuds. La gestion des périphériques et des interactions avec les utilisateurs est assurée par le système d'exploitation de la station maître. Le multicalculateur est alors utilisé comme un co-processeur puissant connecté à une station de travail. Les utilisateurs actuels de ce type de machines se contentent de cette interface car ils ont besoin de fortes puissances de calcul. Cependant d'autres domaines d'application pourraient bénéficier des gains de performances que permettent les multicalculateurs. Pour étendre son utilisation, ce type de machines doit offrir à l'utilisateur un environnement de programmation et de mise au point dont le confort et la simplicité sont équivalents à un environnement de station de travail. En particulier cet environnement doit s'appuyer sur des standards pour faciliter le passage d'un type de machines à un autre et l'intégration dans un réseau local.

Dans ce chapitre, nous avons décrit l'évolution des fonctionnalités des systèmes d'exploitation. Nous avons également comparé différents systèmes d'exploitations répartis. Cette étude nous montre que CHORUS/MiX apparaît mieux adapté à la gestion d'un multicalculateurs que les autres systèmes que nous avons décrits. Les raisons principales sont sa compatibilité binaire avec le système UNIX standard, sa modularité qui permet de l'adapter à une configuration particulière et la transparence de sa communication qui étend l'interface UNIX à la répartition.

Nous pouvons remarquer que les évolutions technologiques du matériel induisent souvent des changements d'interface dans les systèmes d'exploitation, même si certaines innovations s'avèrent également intéressantes dans d'autres contextes. Nous pouvons citer l'apparition des messages dans les systèmes et leur importance grandissante qui va de paire avec le développement des

réseaux locaux. L'utilisation des processus légers pour les multiprocesseurs à mémoire partagée est aussi révélatrice de ce type d'interaction. Si nous essayons de projeter ce raisonnement sur les évolutions récentes du matériel nous pouvons nous interroger sur les nouveautés qu'apporteront les calculateurs massivement parallèles. En effet ces architectures ont un rapport prix/performances qui nous permet d'imaginer qu'elles serviront bientôt à une gamme plus étendue d'utilisateurs. De quelle manière évolueront les systèmes d'exploitation pour répondre à ces nouvelles technologies ?

Chapitre 3

CHORUS/MiX sur multicalculateur

Introduction

Nous avons constaté que les systèmes natifs des multicalculateurs proposent un niveau de service insuffisant pour permettre une utilisation générale de ce type d'ordinateur. Il faut développer de nouvelles fonctionnalités systèmes et une interface standard pour faciliter l'acceptation de ces multicalculateurs par des utilisateurs non-initiés. Le système CHORUS/MiX a été conçu pour gérer différents types d'ordinateurs : son architecture, qui se compose d'un micro-noyau et d'un ensemble de serveurs, lui permet de s'adapter à plusieurs configurations matérielles et le service de communication transparente du micro-noyau CHORUS, qui supporte l'implantation des services standards UNIX le destine plus particulièrement aux environnements répartis. Afin d'offrir les services manquants sur les multicalculateurs, nous avons donc choisi de porter le système CHORUS/MiX sur un iPSC/2[HP91], dans le but d'évaluer l'adéquation des systèmes répartis aux architectures à mémoire distribuée.

Dans la première et la deuxième partie nous décrivons les principaux problèmes qui doivent être traités pour adapter un système réparti, tel que CHORUS/MiX, sur un multicalculateur. Nous analysons, dans une troisième partie, les bénéfices et les lacunes mises en évidence par ce portage, ce qui nous conduit à spécifier les propriétés souhaitables pour qu'un système d'exploitation satisfasse les besoins des utilisateurs.

1 Portage du micro-noyau CHORUS et de la communication

Comme nous l'avons vu au chapitre précédent le noyau CHORUS/MiX est composé d'une couche de base (le *micro-noyau*) et d'un ensemble de serveurs systèmes qui implantent une interface UNIX. Le *micro-noyau* est, en partie, dépendant du processeur sur lequel il s'exécute puisqu'il gère le processeur, la mémoire et le réseau. Les serveurs utilisent les services implantés par le *micro-noyau* pour mettre en oeuvre les services plus élaborés d'une interface de système d'exploitation. Néanmoins ces serveurs peuvent également être partiellement dépendants de la machine dans la mesure où ils peuvent accéder à des périphériques, au contexte système d'une exécution, etc. Le

portage et l'adaptation d'un noyau CHORUS/MiX se fait donc en deux étapes :
– portage des couches dépendantes de la machine du *micro-noyau* : gestion des interruptions, gestion mémoire, accès réseau, etc.
– portage du sous-système UNIX sur le *micro-noyau*.

1.1 Le micro-noyau

Dans notre cas, le travail sur le *micro-noyau* s'est quasiment limité au chargement du système sur l'ensemble des noeuds puisque les processeurs de base d'un iPSC/2 sont des processeurs Intel 80386, déjà supportés par le noyau CHORUS. Seules quelques modifications ont été nécessaires pour intégrer les particularités de l'IPSC/2. L'annexe 1 explique les modifications qui ont été réalisées pour porter le *micro-noyau* sur les noeuds de l'iPSC/2.

Le chargement du système d'exploitation pose un problème aux multicalculateurs actuels. En effet ceux-ci ne disposent pas systématiquement d'une unité de stockage secondaire. Le chargement de leur système d'exploitation est donc réalisé à partir d'une station maître : un protocole est établi entre la station maître et les sites pour ce chargement. Le chargement de CHORUS peut généralement être réalisé au moyen de cet utilitaire. Contrairement aux systèmes d'exploitation monolithiques, le système CHORUS/MiX est constitué de plusieurs binaires : le programme d'initialisation, le *micro-noyau* et les différents acteurs des CHORUS/MiX. Pour utiliser l'utilitaire de chargement l'ensemble des binaires doit alors être regroupé dans une archive de chargement. La même archive étant chargée sur tous les sites nous sommes alors obligés de charger tous les serveurs sur tous les sites. Pour permettre le chargement d'archives différentes sur les noeuds d'un multicalculateur il faudrait que ceux-ci soient indépendants du système hôte. Une description plus complète du travail réalisé pour constituer et charger l'archive binaire sur les noeuds peut être trouvée en annexe 1.

1.2 La communication

Le travail d'adaptation de la communication est important pour un multicalculateur du fait de la forte dépendance des noeuds entre eux : chaque noeud ne dispose pas forcément de toutes les ressources (disques ou autres périphériques) nécessaires aux exécutions. De ce fait il devra donc accéder fréquemment au réseau pour satisfaire les requêtes de ses clients. Dans ce cas la communication n'est plus considérée comme une facilité donnée à l'utilisateur mais plutôt comme une ressource de base du système. Son développement est donc très important.

D'autre part, pour faciliter les échanges entre un réseau externe (tel qu'un réseau local) et le réseau interne du multicalculateur nous devons permettre la gestion de différents média de communication.

Pour pouvoir faire communiquer les noyaux nous avons du développer un pilote pour le DCM.

1.2.1 La couche matérielle

Lorsque la carte DCM est correctement initialisée, elle permet l'envoi d'un message d'un noeud à un autre de façon transparente (le routage est déterminé statiquement) : il suffit de préciser le numéro du noeud destinataire à la carte pour qu'elle sache envoyer le message. L'envoi de message à partir du DCM est asynchrone, c'est-à-dire que l'exécution du code peut reprendre avant que le message soit reçu par le destinataire. L'émetteur reçoit seulement une interruption dès que la totalité du message a été envoyée ; il n'y a pas de protocole de fiabilité qui assure que le message est bien reçu par le destinataire. Notons cependant que ce médium est généralement fiable.

La transmission d'un message est indépendante de l'état des différents processeurs, c'est-à-dire qu'un message peut transiter par un noeud dont le processeur est stoppé. Même si un processeur est surchargé cela n'induit donc rien sur la transmission du message : celle-ci est uniquement gérée par les cartes de communication.

A bas niveau, le DCM de l'émetteur agit de façon synchrone pour réserver un chemin au message à envoyer : il envoie un message de réservation qui, lorsqu'il lui revient lui assure que le chemin est disponible et que le DCM distant est prêt à recevoir. Lorsque ce chemin est établi, le DCM envoie le message accompagné d'une trame de terminaison qui libère le chemin. L'utilisateur n'est, pour ainsi dire, pas limité par la taille du message puisque la taille maximum acceptée par le DCM est 16 Mo., ce qui est la taille mémoire maximum d'un noeud.

Le routage choisi pour aller d'un noeud à l'autre est statique, c'est à dire que pour se rendre d'un noeud X à un noeud Y, le message utilisera toujours le même chemin. Le routage statique, dans le cas d'un hypercube, est simple à réaliser puisqu'il suffit d'appliquer la fonction *XOR* entre le numéro de noeud de l'émetteur et celui du destinataire pour obtenir les directions successives à emprunter par le message (figure III.1).

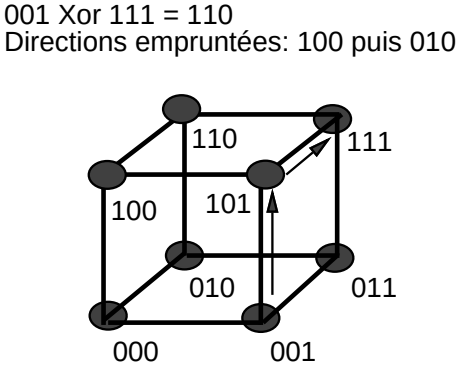

Figure III.1 : Routage entre les noeuds 001 et 111

Dans le cas de l'iPSC/2, le système fournit le numéro du noeud destinataire ainsi que le résultat de la fonction *XOR* au DCM. Le DCM, à partir de ce résultat, choisit les canaux à emprunter dans l'ordre prédéfini. Les contraintes à respecter lors de l'émission d'un message portent princi-

palement sur l'initialisation des structures décrivant le message. En effet, le système doit fournir deux types de structures pour envoyer un message.

– L'une contient le destinataire du message, ainsi que les directions à utiliser pour l'atteindre. Elle est destinée au DCM. Ces informations lui servent à établir le chemin entre lui et le DCM du destinataire.
– L'autre structure décrit les données à envoyer. Elle est destinée au ADMA (contrôleur de mémoire élaboré) qui gère la recopie des données entre la mémoire centrale et la mémoire du DCM. Cet ADMA est capable de transmettre aux tampons du DCM un message, même si ce message n'est pas contigu en mémoire centrale. Dans ce cas, les différentes parties du message sont décrites dans la structure passée au ADMA. L'utilisateur du DCM doit toujours lui fournir des adresses de mémoire physique.

La réception d'un message est, elle aussi, asynchrone. Pour cette raison, le système doit allouer un tampon - de taille suffisante pour contenir tout le message - à la carte DCM avant la réception d'un message. Le système est prévenu de la réception d'un message par une interruption.

1.2.2 La communication dans CHORUS

L'architecture du service de communication a été conçue pour gérer plusieurs types de média et de protocoles. Le but recherché étant de pouvoir communiquer entre des sites qui font partie de réseaux différents, connectés entre eux par des passerelles. CHORUS divise les réseaux en deux : les **domaines** et les **sous-réseaux**. Un sous-réseau représente un groupe de sites qui sont connectés par un même médium. Dans la figure III.2, le multicalculateur représente un sous-réseau. Les domaines sont des groupes de sous-réseaux ou de sites connectés par au moins un médium. Par exemple la figure III.2 représente deux domaines : le réseau local de stations de travail et le réseau à grande distance.

L'implantation de ce service de communication suppose que le routage dans les sous-réseaux est unique - la communication utilise toujours le même chemin d'un site à l'autre - et qu'un domaine supporte des facilités de diffusion - au moins au niveau logiciel. Sur une telle architecture CHORUS permet l'échange de messages entre sites distants quelque soit le réseau sur lequel ils se trouvent. Au niveau de l'utilisateur cet échange de message est transparent. Il repose sur une désignation unique des sites : un identificateur de site est composé d'un numéro de site, d'un identificateur de sous-réseau et d'un identificateur de domaine. Néanmoins le service communication ne gère pas actuellement l'hétérogéneité entre les sites, celle-ci devant être implantée par les serveurs qui en ont besoin.

Le service de communication est composé de plusieurs couches indépendantes. Chacune de ces couches fournit une interface à la couche inférieure pour la réception de message et une interface à la couche supérieure pour l'envoi de message. Pour optimiser les performances de la communication ces interfaces sont des adresses de fonctions. Toutes ces couches étant dans le même espace d'adressage, une activité du noyau peut exécuter la totalité d'un envoi de messages en appelant successivement toutes les fonctions, évitant ainsi plusieurs changements de contexte.

L'architecture de la communication CHORUS prévoit que le noyau n'implante pas de protocoles. La partie implantée par le noyau reste ainsi indépendante du médium : aucun a priori n'est fait

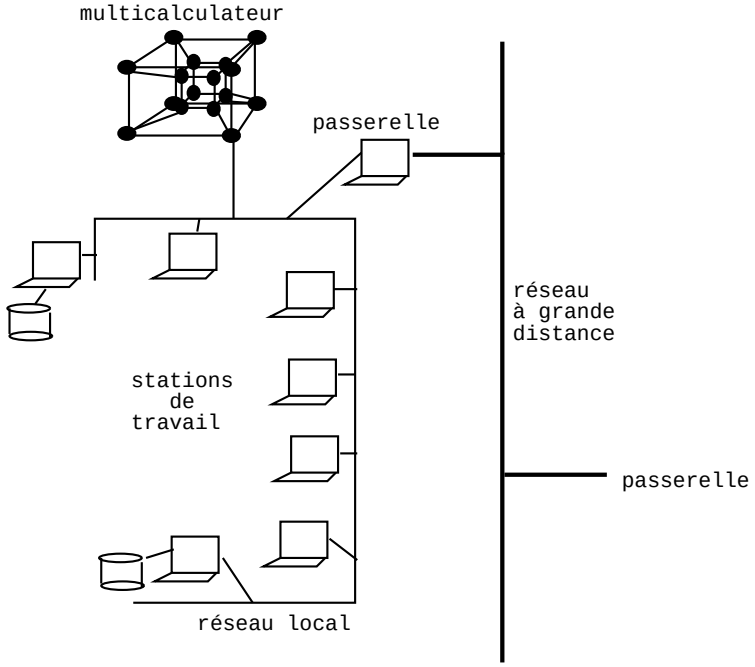

Figure III.2 : Exemple de réseau

sur la mise en place et l'architecture du protocole et de la gestion du matériel. Dans la figure III.3, cette partie est composée des couches : IPC CHORUS, unité de fragmentation et gestionnaire de sites distants. La partie propre au matériel utilisé est adaptable pour permettre d'implanter le protocole de communication en fonction du niveau de service offert par le matériel. L'implantation actuelle de la communication CHORUS est réalisée sur un réseau Ethernet. Dans ce cas, la partie dépendante du matériel, représentée dans la figure III.3, comprend trois parties : un protocole (CNP ou CHORUS Network Protocol), un gestionnaire de pilotes de réseau (NDM ou Network Device Manager) et le pilote de réseau.

L'IPC CHORUS L'IPC CHORUS implante les communications distantes synchrones (RPC) et asynchrones. Les échanges de messages sont gérés sous forme de transactions, c'est-à-dire qu'un échange de message est considéré comme une unité indivisible : soit il aboutit soit il échoue. A chaque envoi de message le service de communication acquiert une transaction, qu'il libère seulement lorsqu'il a atteint le niveau de service associé à la sémantique, c'est-à-dire :
– dès que le message est délivré au pilote, lorsqu'il s'agit d'IPC asynchrones, car le système ne donne aucune garantie sur l'envoi d'un message asynchrone,
– après avoir déposé la réponse au message sur la porte émettrice, s'il s'agit d'un RPC, car dans ce cas le système garantit la fiabilité de l'échange.
L'IPC CHORUS est également responsable de la localisation des portes. C'est lui qui implante la sémantique liée à la transparence de l'échange de messages. Il gère un cache dans lequel il

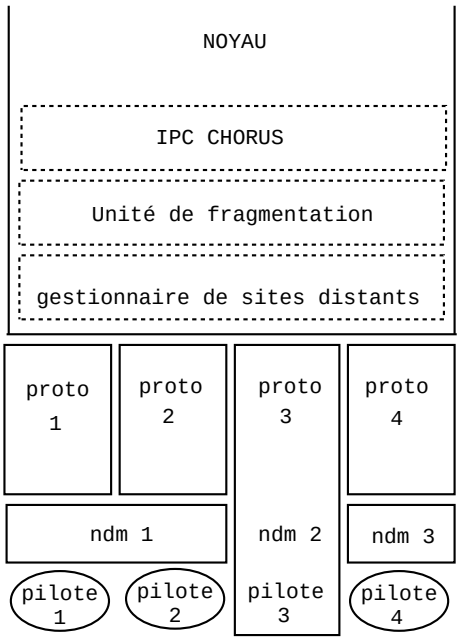

Figure III.3 : Structure de la communication distante dans CHORUS

mémorise la localisation des portes qu'il connait. Ces enregistrements ne sont que des indications dans la mesure où les portes peuvent migrer. Lors d'une émission distante l'IPC envoie d'abord à la porte contenue dans le cache. Si l'envoi n'aboutit pas ou si la porte n'est pas dans le cache une demande d'informations est diffusée sur le réseau. Les portes localisées sont mémorisées dans le cache.

Lorsque le message est prêt à être envoyé l'IPC appelle la fonction d'envoi de l'unité de fragmentation.

L'unité de fragmentation Les données contenues dans un message CHORUS ont une taille inférieure à 64 Ko. L'unité de fragmentation est chargée de couper les messages en trames à l'émission et de les reformer à la réception. La taille de page gérée par le processeur local sert généralement d'unité de fragmentation ce qui permet d'améliorer les performances en utilisant les services offerts par la mémoire virtuelle.

Pour l'envoi de chaque trame l'unité de fragmentation fait appel aux services du gestionnaire de sites distants. Pour cela elle utilise deux modes : régulé et non-régulé permettant de faire ou de supprimer le contrôle du flux d'émission des données.

A la réception, le gestionnaire de site distant possède les données. Si il a besoin du tampon dans lequel se trouvent les données il peut alors demander à l'unité de fragmentation de recopier le message chez elle. Sinon elle est chargée de reformer le message à partir des trames qu'elle reçoit.

Le gestionnaire de sites distants Le gestionnaire de sites distants (RSM) est chargé du routage des trames et du choix du protocole auquel les trames sont délivrées. Il est composé :

- d'un routeur qui doit déterminer le protocole à utiliser en fonction de la porte destinatrice et l'adresse réseau qui doit être utilisée par le protocole. Le routeur possède une table dans laquelle il mémorise les points d'entrée des protocoles. C'est le routeur qui invoque les autres composants du RSM puis qui délivre le message au protocole.
- d'une unité de résolution d'adresse qui détermine les protocoles à utiliser. La diffusion d'un message utilise parfois plusieurs protocoles en particulier sur un site qui forme la passerelle entre deux réseaux. L'unité de résolution maintient un cache qui mémorise la correspondance entre une adresse et un protocole. Si l'adresse accédée n'est pas dans le cache, l'unité de résolution d'adresse la demande à tous les protocoles auquel elle appartient.
- d'une unité de routage qui détermine, à partir de l'adresse CHORUS, l'adresse réseau du prochain destinataire. Par exemple il détermine l'adresse Ethernet à partir du numéro de site ou l'adresse de la passerelle si le destinataire est connecté à un autre réseau. Si le site est accessible localement l'adresse obtenue sera celle du site destinataire sinon ce sera celle de la passerelle vers l'autre réseau.
- une unité d'administration du réseau qui est chargée d'initialiser la configuration des protocoles et des pilotes qui doivent être utilisés. Elle fixe également le numéro CHORUS du site local.

Le protocole Un protocole est dépendant du médium, il est attaché à un ou plusieurs pilotes de réseau à travers le gestionnaire de pilote du réseau. C'est le protocole qui est chargé d'implanter la diffusion sur le médium si celui-ci ne possède pas d'adresse physique de diffusion. Certains protocoles devront mettre en place un redécoupage en trame de plus petites taille, gérer la diffusion, etc. Dans ce cas le découpage en un protocole, un gestionnaire de pilote du réseau et un pilote se justifie pleinement pour bien fixer les différentes couches du service de communication. Par contre certains réseaux de communication offrent déjà des fonctionnalités de haut niveau, dans ce cas le protocole est quasiment vide et il peut être regroupé dans un même serveur avec le pilote (figure III.3).

CHORUS implante son propre protocole (CNP ou CHORUS Network Protocole) pour les réseaux du type Ethernet. Ceci lui permet d'être mieux adapté aux besoin du noyau. Cette architecture permet l'intégration de protocoles standard tels que IP[Com88] ou encore de protocoles implantant des fonctionnalités supplémentaires telles que celle du X-kernel[PR91] ou d'Isis[BSS91b].

Le gestionaire des pilotes du réseau Le rôle du gestionnaire des pilotes du réseau est de fournir une interface générique et fixée à partir du pilote de périphérique. Un NDM peut gérer plusieurs pilotes à la fois dans la mesure où ces pilotes accèdent au même type de médium. Par exemple, s'il existe plusieurs cartes gérant des cables Ethernet, un NDM peut accéder à deux pilotes gérant ces cables.

L'interface des différentes couches de communication Pour des raisons de performance c'est, en général, une activité du noyau qui exécute la totalité de l'envoi du message. Par contre,

si les structures nécessaires à l'envoi ne sont pas disponibles le noyau met le message en attente et c'est une activité du RSM qui sera chargée d'envoyer le message. De cette façon, on évite un changement de contexte à chaque changement de serveur. Le noyau accède au code de la communication tout en restant totalement indépendant du code et des données des serveurs de communication - c'est-à-dire qu'il n'y a pas d'édition de liens entre le code du noyau et le code de la communication. Au moment de l'initialisation des serveurs de communication chaque couche échange, avec la couche supérieure, l'adresse des fonctions qui peuvent être utilisées. Puisque tous ces serveurs partagent le même espace d'adressage système, ils peuvent utiliser les fonctions offertes par un autre serveur. Ainsi, une activité du noyau désirant envoyer un message appellera successivement toutes les fonctions des différents serveurs jusqu'aux traitements les plus bas. Réciproquement, la réception d'un message oblige une activité du gestionnaire de réseau à exécuter les fonctions des couches supérieures.

Dans le cadre de cette spécification, l'architecture des couches de communication est adaptable aux besoins de la machine sur laquelle elle est implantée.

1.2.3 La gestion de la communication sur l'iPSC/2

D'une manière générale, la communication dans les machines parallèles à mémoire distribuée subit des contraintes spécifiques, par rapport aux stations de travail interconnectées, dans la mesure où les media utilisés n'ont pas les mêmes caractéristiques. Ainsi les possibilités de blocage dans un réseau de type hypercube ou grille doivent être supprimées pour assurer un bon fonctionnement. Par contre elles sont quasiment inexistantes sur les réseaux de type bus ou anneau qui caractérisent les réseaux locaux. D'autre part, les protocoles de communication doivent prévenir la famine et garantir l'équité de l'accès au réseau par tous les noeuds. Dans le cas de l'iPSC/2 nous devons ajouter les raisons suivantes :

- Chaque site ne disposant pas d'un disque et d'un terminal, il doit accèder aux sites distants lorsqu'il a besoin d'un de ces services. Pour cette raison les performances de la partie communication du système vont avoir une incidence sur le comportement de tout le système.
- les performances sont également cruciales pour l'utilisateur qui écrit a priori des applications parallèles adaptées à l'architecture de cette machine. Lors de la conception de ses programmes, il doit pouvoir compter sur une mise en place d'une communication fiable et performante.
- le médium de communication est différent : il offre une grande fiabilité ainsi qu'une bande passante supérieure à celle d'un réseau local. Dans le cas de l'iPSC/2, le routage est pris en charge par la couche matérielle. Le non blocage de la communication est garanti par le routage.

Il est donc très important d'avoir de bonnes performances en communication. L'intérêt de l'architecture standard de communication CHORUS est qu'elle laisse au concepteur de la communication suffisamment de liberté pour réaliser une implantation adaptée à la spécificité de l'architecture concernée. Dans le cas de l'iPSC/2, il se trouve que nous n'avons pratiquement pas besoin de protocole ni de gestionnaire de réseau dans la mesure où le médium est considéré comme fiable et que le code de gestion du réseau est simple. Le RSM n'est pas non plus nécessaire car nous nous limitons à un seul protocole, celui de la communication inter-noeuds. Pour optimiser les communications sur l'hypercube nous avons donc réuni en un seul serveur - appelé gestionnaire

de communication - l'ensemble des composants implantant la communication.

L'interface avec le noyau Notre serveur de communication (figure III.4) est accédé par le noyau, pour envoyer un message au travers de la fonction *sendData()*. A la réception d'un message le serveur utilise la fonction *recvData()* du noyau. Comme dans le cas général, le gestionnaire de communication de l'iPSC/2 déclare, au moment de l'initialisation du système, l'adresse à laquelle le noyau peut l'appeler, c'est-à-dire celle de la fonction *sendData()* et reçoit, en retour celle à laquelle il peut appeler le noyau, c'est-à-dire celle de la fonction *recvData()*. De cette façon, c'est directement l'activité du noyau qui envoie le message, en utilisant les ressources du serveur de communication, ce qui évite de changer d'activité à chaque envoi de message. Ceci est possible car le serveur de communication, comme le micro-noyau, se trouve dans l'espace d'adressage système. Nous obtenons ainsi de meilleures performances en évitant les changements de contexte entre deux activités. De même pour la réception c'est le serveur de communication qui exécute les traitements du noyau pour délivrer les messages.

La fonction *sendData()* du gestionnaire de communication utilise trois paramètres : l'adresse des données à envoyer, leur taille et un indicateur. Cet indicateur permet au gestionnaire de communication de savoir si les données à transmettre doivent être copiées dans un tampon intermédiaire avant d'être envoyées sur le réseau ou si ces données peuvent être prises directement dans l'espace d'adressage de l'émetteur.

La fonction *recvData()* est implantée par le noyau. Le gestionnaire de communication, lors de l'appel de *recvData()*, passe en paramètre au noyau l'adresse à laquelle se trouvent les données ainsi que leur taille. Le noyau garantit que le tampon du gestionnaire de communication utilisé pour mémoriser les données du message reçu peut être libéré au retour de l'appel de la fonction *recvData*.

Structure et ressources du gestionnaire de communication Le gestionnaire de communication de l'iPSC/2 est un acteur indépendant du noyau. Comme tout acteur CHORUS il est créé avec un espace d'adressage, une activité et une porte.

Ce serveur possède plusieurs activités pour permettre le traitement simultané de plusieurs messages : lorsque l'activité courante est mise en attente d'une ressource, une autre activité traite les messages en attente.

Le serveur gère un ensemble de tampons alloués dans une région de son espace d'adressage. Tous ces tampons sont de la même taille. Pour faciliter leur gestion, le serveur alloue pour chacun d'eux une structure qui les décrit (adresse virtuelle et adresse physique). Cette structure peut être chainée dans une liste des tampons libres afin de faciliter l'allocation au moment de la réception d'un message. Ces tampons sont utilisés par le DCM pour la réception des messages, puisque celui-ci doit, avant la réception, connaître l'espace mémoire dans lequel il écrira le message reçu. Ces tampons servent aussi à recopier les messages lors de l'émission lorsque cela est nécessaire.

Le serveur de communication utilise une mini-porte et un ensemble de mini-messages pour mettre en place la réception des messages, en particulier pour permettre de différer le traitement des interruptions.

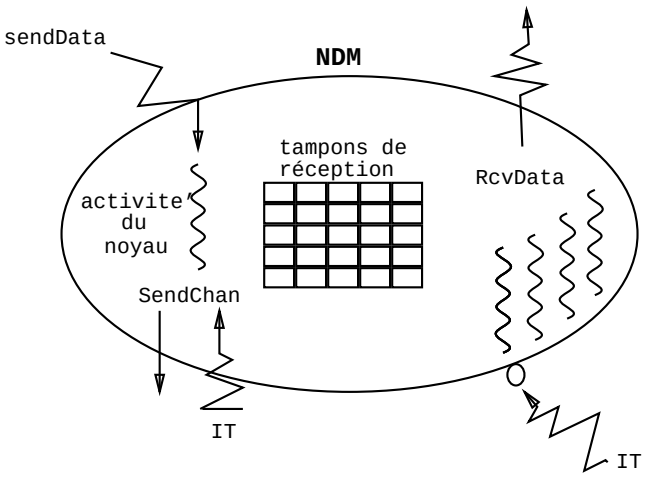

Figure III.4 : Structure du gestionnaire de communication

Pour se faire connaître aux autres serveurs et plus particulièrement du noyau, le serveur utilise un groupe de portes dont le nom est fixé statiquement.

Traitements effectués par le serveur de communication Il n'y a pas, à proprement parler, de protocole permettant de communiquer entre deux noeuds. Le serveur de communication se contente d'ajouter une entête minimum au message que le noyau lui transmet. Cette entête ne contient que la taille et le type du message. Le serveur de communication prévoit l'envoi de messages d'une taille maximum équivalente à la taille d'une page de mémoire physique : cette limitation arbitraire doit permettre de ne pas réserver la bande passante du réseau à un seul utilisateur. En effet un DCM se connecte au noeud destinataire avant d'envoyer un message : il réserve les liens qui le relie à ce noeud pour l'envoi du message. Ces liens, utilisés lors de l'échange, sont indisponibles pour les autres utilisateurs. Puisque le routage est statique cela signifie que les autres messages désirant utiliser ce lien sont bloqués tant que le message n'est pas transmis. Le flux des messages est régulé en découpant les messages en trames, permettant un accès plus équitable au réseau.

Le serveur de communication implante trois traitements : l'initialisation du serveur, la réception de messages et l'émission de messages.

(1) L'initialisation :
L'initialisation du serveur consiste en :
- la connexion des fonctions de traitement des interruptions du DCM en utilisant les primitives CHORUS telles que *svItConnect*. En fait, le serveur de communication ne gère que deux interruptions : l'interruption d'émission qui prévient que le message a été envoyé sur le médium et l'interruption de réception qui annonce la réception d'un message.
- l'allocation de la région de mémoire dans laquelle seront alloués les tampons. L'activité procédant à l'initialisation coupe la région mémoire en tampons de taille fixe. Pour chaque

tampon elle calcule son adresse physique, génère la structure qui le décrit et la place dans la liste des tampons libres.
- l'initialisation des activités chargées de la réception.
- la création de la mini-porte et l'initialisation des mini-messages.
- l'insertion de la porte du serveur dans le groupe statique prévu à cet effet.

(2) Une fois l'initialisation terminée, l'activité communique avec le noyau pour lui donner :
- le numéro de site,
- l'adresse de la fonction *sendData* qui permettra au noyau d'envoyer des messages.

Lorsqu'elle a terminé ce traitement, l'activité d'initialisation se détruit.

(3) L'émission de messages :

Lors de l'émission d'un message (figure III.5), deux cas se présentent. L'activité du noyau exécutant l'envoi de message étant en mode protégé, si le message provient d'un autre serveur, il n'y a pas de problème car il se trouve en espace système : l'activité du noyau a donc le droit d'écriture sur cet espace. Par contre, si le message est situé dans l'espace utilisateur, l'activité du noyau ne pourra pas l'accéder. Dans ce cas, le message devra être recopié dans la mémoire du serveur de communication. C'est dans l'indicateur passé en paramètre de la fonction *sendData* que le noyau précise si le message doit être recopié dans la mémoire du serveur de communication ou s'il peut être directement utilisé.

```
Si ( mode == protégé )
        vérifier que le message est contigu
        Si ( contigu )
                former le descripteur en une partie
        Sinon
                former le descripteur en deux parties
        fSi
Sinon
        copie du message dans un tampon
        utiliser le descripteur du tampon
fSi
envoyer le message
attendre interruption de fin d'envoi
```

Figure III.5 : Algorithme d'envoi d'un message

Puisque le DCM travaille avec des adresses physiques le système doit s'assurer, avant d'envoyer un message, qu'il est contigu en mémoire physique. Lorsque ce message est recopié dans la mémoire du serveur il l'est dans un tampon alloué de telle sorte qu'il soit contigu en mémoire physique. Dans ce cas, il n'y a pas de problème : l'adresse du tampon est transmise au DCM. Dans les autres cas, le message ne pouvant dépasser la taille d'une page de mémoire, il sera au plus en deux parties contigues en mémoire physique. La structure passée au ADMA contiendra alors leurs deux adresses.

Une fois que la structure décrivant le message est initialisée, elle est transmise au DCM pour qu'il effectue l'envoi. Pendant ce temps l'activité du noyau se met en attente de la réception de l'interruption d'émission. Lorsque cette interruption a eu lieu l'activité retourne à ses

traitements dans le noyau.

(4) La réception de messages :

Pour la réception (figure III.6) toutes les activités du serveur de communication sont en attente derrière une mini-porte. Lorsque le DCM reçoit un message il envoie une interruption au processeur. La fonction connectée à cette interuption lors de l'initialisation du serveur de communication alloue immédiatement un tampon pour que la réception d'un nouveau message puisse être effectuée le plus tôt possible. Elle envoie ensuite un mini-message contenant les références du message reçu sur la mini-porte du serveur prévue à cet effet, puis rend immédiatement la main au système.

Traitement de l'interruption de réception:

```
former le mini−message
allouer un nouveau tampon de réception
```

Traitement de la réception:

```
tant que (1)
    attendre un message sur la mini−porte
    lire l'entête du message
    donner le message au noyau
fin tantque
```

Figure III.6 : Algorithme de réception d'un message

Ce mécanisme permet au système d'effectuer un minimum de traitements lors de l'interruption, réduisant ainsi le temps pendant lequel les autres interruptions sont masquées. Lorsque le mini-message arrive sur la mini-porte du serveur, une de ses activité est réveillée. Elle appelle la fonction *recvData* - en lui passant en paramètres les informations contenues dans le mini message - du noyau qui va délivrer le message à son destinataire.

Performances de la communication Nous avons comparé les performances de débit obtenus par notre gestionnaire de communication à celles obtenues sur le système NX. La comparaison et l'analyse de ces performances sont données en annexe B. Cette analyse nous permet d'observer une dégradation des performances dans notre implantation de la communication par rapport au système NX/2. Nous donnons deux explications à cette dégradation : en premier le niveau de service et de confort offert par CHORUS est supérieur à celui de NX/2 (NX/2 n'offre aucune transparence ni possibilité de localisation) et en deuxième notre implantation est une expérimentation qui n'a pas été optimisée.

Cette première étape de portage du système d'exploitation CHORUS/MiX sur l'iPSC/2 nous montre que le micro-noyau ne pose pas de problème particulier pour être porté sur une plateforme du type multicalculateur. Au contraire, le peu de modifications réalisées dans le micro-noyau pour l'adapter à l'iPSC/2 nous font supposer que celui-ci serait bien adapté à ces architectures.

1.2.4 La communication entre la station maître et les noeuds

Pour mettre en place la gestion des fichiers et des périphériques, nous avons développé la communication entre la station maître et les noeuds du multicalculateur.

1. Le protocole NX

Nous décrivons rapidement le protocole utilisé pour échanger des messages dans le système NX :

- si la taille du message est inférieure à 100 octets le système envoie le message directement au destinataire sans accord préalable. A cet effet chaque noeud possède un ensemble de tampons de 128 octets - cela correspond à la taille maximum du message et de son entête. Les tampons sont attribués par noeuds, chaque noeud en possède un nombre fixe : ces tampons déterminent le nombre de messages en provenance des autres noeuds de l'hypercube autorisés à être reçus par le noeud concerné. Pour gérer ces tampons chaque noeud possède le compte des tampons libres qui lui sont réservés sur les autres noeuds et le compte des tampons qu'il réserve aux autres noeuds. Ces compteurs sont mis à jour à chaque échange de messages entre deux noeuds : le système NX inclut dans l'entête du message le nombre de tampons qu'il a libéré depuis le dernier envoi en faveur du destinataire. Le système utilise aussi des messages spéciaux prioritaires permettant de signaler la libération des tampons lorsque l'échange de messages se fait principalement dans un sens. La réception de ces messages spéciaux ne nécessite pas de tampons car ces messages sont immédiatement interprétés par le système.
- si la taille du message est supérieure à 100 octets le système NX envoie un message spécial demandant au destinataire de réserver la place mémoire correspondant au message à envoyer. Lorsque le noeud récepteur est prêt, il envoie un message de réponse pour donner son accord. A la réception de ce message le noeud initiateur de la communication envoie le contenu du message.

Lorsque l'utilisateur veut envoyer un message à un noeud (y compris la station maître) il donne le numéro du noeud auquel il fait l'envoi ainsi que le *PID* du processus destinataire. Puisque les processus ne peuvent pas se déplacer ils sont identifiés à l'aide de ces deux identificateurs. Contrairement au processus UNIX, les processus NX fixent eux-mêmes leur *PID*. Puisqu'un noeud est toujours alloué à un seul utilisateur il n'y a pas de risque de collision.

Les messages NX sont typés pour différencier les messages spéciaux des messages classiques. L'entête d'un message NX contient : le noeud destinataire, le *PID* du processus destinataire, le noeud émetteur, le *PID* du processus émetteur, le type du message, la taille des données, etc.

2. L'implantation de la communication avec la station maître

Le matériel (DCM) permet l'accès direct de tous les noeuds à la station maître (SRM). La communication avec le SRM nécessite l'utilisation d'un protocole différent de celui qui est utilisé entre les noyaux CHORUS. Il est donc possible de charger sur chaque noeud le code de simulation du protocole pour permettre l'envoi de messages au SRM. Cependant le SRM est principalement utilisé pour la gestion des périphériques. L'accès simultané de plusieurs

noeuds aux structures du SRM impliquerait de gérer la concurrence d'accès. D'autre part le débit de la ligne DCM qui accède au SRM n'est pas très élevé, celle-ci formant rapidement un goulet d'étranglement. Nous ne pouvons donc pas attendre de meilleures performances d'un accès multiple au SRM. Nous avons choisi un accès depuis un seul noeud.

- **Les structures du serveur de communication** :
  La communication avec la station maître n'a pas été implantée de la même manière que la communication inter-noeuds. Comme les données peuvent être envoyées directement - car le site destinataire est fixé - nous n'avons pas besoin de faire appel aux services de localisation ou de fragmentation du noyau. En effet, les utilisateurs voulant communiquer avec le frontal ne s'adressent pas au noyau pour effectuer leurs échanges de messages mais directement à une porte spécifique du serveur de communication. Les messages qui arrivent sur cette porte sont destinés à la station maître. Le traitement de ces messages est réalisé par plusieurs activités dédiées du serveur de communication.
  Les activités du serveur de communication gérant les échanges avec le frontal partagent leurs données avec celles implantant la communication avec les noeuds puisqu'elles sont dans le même serveur. Elles reçoivent les messages dans les mêmes tampons. Puisque le tampon doit être alloué avant la réception il n'est pas possible de savoir si le message qui sera reçu provient des noeuds ou du frontal. Une nouvelle mini-porte est donc allouée au serveur de communication pour gérer la réception des messages en provenance du frontal. Pour simuler le protocole NX, le serveur possède les compteurs nécessaires à la gestion des tampons du frontal. Par contre le serveur ne gère pas lui-même de tampons au sens protocole NX, il ne fait que simuler en tenant à jour les compteurs concernant la station maître et le noeud 0 uniquement.
- **Les traitements** :
  Au moment de son initialisation le serveur de communication connait le numéro du site sur lequel il s'exécute. Si il est sur le site 0, il initialise une activité supplémentaire. Cette activité va allouer les structures nécessaires à la communication avec la station maître puis se mettre en attente sur la porte qu'elle vient de créer. Contrairement au traitement effectué pour la communication inter noeuds, cette activité ne crée pas immédiatement les autres activités destinées à la communication avec la station maître. Le protocole utilisé est le suivant : un acteur qui veut utiliser la communication avec la station maître doit d'abord demander au serveur de communication de lui créer une nouvelle activité pour gérer cette communication et de lui donner une porte sur laquelle il pourra envoyer ses requêtes.
  Pour générer l'entête du message à envoyer à la station maître le serveur de communication a besoin du *PID* du processus auquel il envoie le message. Ce *PID* doit être précisé par l'acteur client du service de communication avec la station maître. Ceci permet à l'acteur de fixer lui-même le processus auquel il veut accéder, et donc le type de service dont il veut profiter. C'est aussi la raison pour laquelle chaque client doit utiliser une activité séparée : il est ainsi assuré de ne pas mélanger ses données avec celles d'un autre client. De plus, cette communication est entièrement synchrone pour simplifier les problèmes de gestion des tampons, numéro de messages dans l'entête, etc.

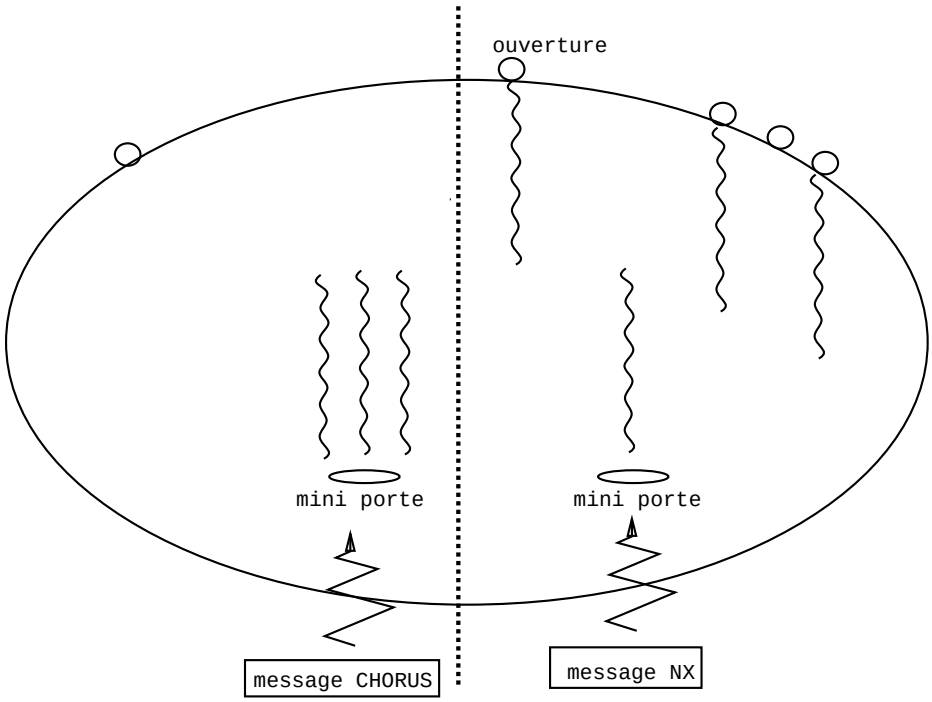

Figure III.7 : Structure de la communication sur le noeud 0

Pour l'émission et la réception de messages le serveur procède de la même manière que pour la communication avec les noeuds en utilisant une mini-porte. C'est dans le traitement de l'interruption de réception que le serveur fait la différence entre les messages en provenance des noeuds et ceux émis par la station maître : l'entête du message possède un champ donnant le numéro du site émetteur. Lorsque le message reçu est une demande d'autorisation d'envoi d'un long message, le serveur envoie immédiatement son accord à la station maître.

3. La gestion de la communication sur la station maître

Ce sont des processus UNIX, s'exécutant sur la station maître qui reçoivent les messages envoyés depuis les noeuds, les traitent et renvoient la réponse à la requête reçue. Les processus de la station maître utilisent les primitives NX de communication pour l'émission et la réception de leurs messages. L'utilisation de plusieurs processus sur la station maître permet de simuler un comportement multi-utilisateur et donc de valider nos hypothèses.

La simulation du protocole NX a été assez délicate, de ce fait l'implantation que nous en avons fait n'est pas complète. Par exemple nous ne sommes limités à l'envoi de messages courts (inférieurs à 100 octets) à la station maître. Par contre les fonctionnalités que nous utilisons sont fiables.

2 Portage de CHORUS/MiX

Le micro-noyau seul n'offre pas une interface de système d'exploitation complète dans la mesure où il ne gère que le processeur, la mémoire et le réseau. Pour offrir une interface de plus haut niveau nous avons dû porter les serveurs du sous-système UNIX au-dessus du micro-noyau. En fait, il n'y a pas de problème particulier concernant le portage des serveurs eux-mêmes dans la mesure où ils utilisent principalement la base offerte par le noyau pour mettre en place leurs services. Les développements imposés par le portage du sous-système proviennent principalement des différences d'architecture entre un multicalculateur et un réseau de stations de travail.

2.1 Choix des périphériques

Comme beaucoup de multicalculateurs l'iPSC/2 ne possède pas un disque et un terminal par noeud. Ceci tendrait à empêcher l'implantation d'un système UNIX traditionnel qui nécessite l'utilisation d'une console par site. Dans ce cas, un système réparti multiserveurs tel que CHORUS/MiX est plus à même de supporter un tel environnement puisqu'il peut être configuré en fonction des caractéristiques d'un site. Par exemple, l'architecture de l'hypercube impose de configurer le système de manière à éviter de charger sur les noeuds du code inutile tel que celui de la gestion du disque. Par contre l'implantation d'une interface de système d'exploitation qui soit générale implique d'offrir à l'utilisateur des facilités d'accès à des fichiers et à, au moins, un terminal. Par exemple, un processus UNIX dépend toujours d'un terminal de contrôle et de l'exécutable dont il est issu. Ces problèmes sont traités ici pour l'iPSC/2 mais la similitude de configuration avec nombre de multicalculateurs nous permet de supposer que les réalisations doivent être applicables à d'autres configurations comme d'autres types de multicalculateurs.

Pour implanter la gestion de fichiers et des terminaux nous avons les possibilités suivantes :

– utiliser un disque et un terminal directement connecté à un noeud du multicalculateur. Cette solution n'est pas toujours envisageable puisque certains multicalculateurs ne disposent pas de disque ou de console autres que ceux de la station maître.
– utiliser le disque et la console de la station maître. En profitant des services de communication entre un noeud et la station maître.

Nous avons utilisé la seconde solution du fait de la configuration de notre iPSC/2. Néanmoins, la volonté des constructeurs de réaliser des machines sans station maître - mais avec les périphériques nécessaires - permettra l'implantation plus fréquente de la première solution qui serait mieux adaptée à notre objectif.

2.2 La configuration de CHORUS/MiX sur l'iPSC/2

Pour les multicalculateurs, l'intérêt majeur du système CHORUS/MiX est la possibilité de configurer le système d'exploitation en fonction des ressources disponibles sur le site. Cette propriété est due à l'implantation multiserveurs du sous-système UNIX et au fait que les interactions entre serveurs sont basées sur l'échange de message CHORUS (IPC). Puisque l'IPC permet d'accéder à un destinataire distant de façon transparente, il permet également aux serveurs qui l'utilisent

d'accéder aux ressources de façon transparente. En particulier le gestionnaire de processus (PM), qui traite les appels système des processus (figure III.9), peut accéder de façon transparente aux ressources utilisées par le processus. Ainsi, sur un noeud qui ne possède pas de disque il est possible de ne pas charger le code du système qui correspond à la gestion du disque. La place mémoire du noeud, disponible pour les applications, est ainsi préservée. Notons en effet que l'espace réservé aux applications est précieux dans la mesure où une application qui manque de place va entraîner le déchargement sur un disque distant (donc coûteux d'accès) d'autres applications puis leur rechargement lorsqu'elles devront être exécutées. Le gain de place mémoire permet donc également un gain de temps d'exécution. La possibilité de configurer le système en fonction des ressources est donc un bénéfice apporté par CHORUS/MiX.

Le micro-noyau et la communication Le système peut être configuré de façon à offrir différents niveaux de service. Le niveau minimal est constitué du micro-noyau et du serveur de communication. En effet, afin de supporter l'exécution des applications ou des serveurs et la communication entre sites nous avons besoin d'un micro-noyau par site. Par contre, il n'est pas nécessaire de charger des serveurs sur les sites devant exécuter des applications utilisant uniquement les appels système CHORUS(c'est par exemple le cas des applications temps réel) Notons que ces applications peuvent tout de même accéder aux serveurs du sous-système UNIX par échange de message plutôt que par appels système.

Le sous-sytème UNIX Chaque noeud offrant une interface UNIX doit charger un gestionnaire de processus (PM). En effet, dans le sous-système UNIX, c'est le PM qui implante les appels système : lorsqu'un processus exécute un déroutement pour accéder aux ressources du système, le traitement est effectué par le PM. La requête peut alors être redirigée vers le serveur concerné, en fonction de son type.

La configuration matérielle désigne le noeud 0 comme une passerelle vers le réseau local sur lequel est connectée la station maître (SRM). Il est donc logique d'implanter le protocole (NX) de communication avec le SRM uniquement sur le noeud 0. Suivant les caractéristiques de la communication mise en place entre une station maître et le multicalculateur, le protocole de communication entre les deux machines sera placé sur le noeud site directement connecté à la station maître.

Lorsqu'un seul noeud implante la communication avec la station maître et que cette connexion est utilisée pour mettre en place la gestion des périphériques il est raisonnable, pour des raisons d'efficacité, que le gestionnaire de fichiers (FM) se trouve également sur ce noeud qui peut alors être considéré comme un noeud d'entrée/sortie. De cette manière, la charge d'exécution d'autres applications n'influence pas sur la gestion des fichiers.

Cette configuration du système d'exploitation sur le multicalculateur montre que les instances du système ne peuvent être considérées comme plusieurs entités s'exécutant sur des sites différents mais bien comme une seule entité dont les parties coopèrent pour implanter l'unicité du système. Le système d'exploitation adapté aux multicalculateurs est donc différent de celui des stations de travail qui doivent, pour remplir leur rôle, être indépendantes les unes des autres, même si elles autorisent l'accès aux ressources qu'elles gèrent.

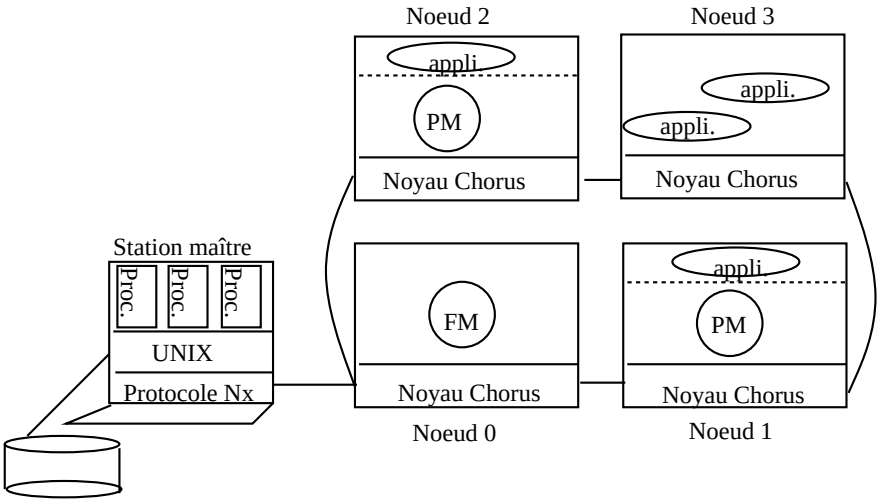

Figure III.8 : Configuration des serveurs sur l'iPSC/2

L'interface utilisateur Sur une station de travail le gestionnaire de processus exécute, à l'initialisation, un interprèteur de commande, appelé *shell*, qui permet d'obtenir l'interactivité avec l'utilisateur du système d'exploitation. Sur un multicalculateur il n'est pas intéressant d'exécuter un *shell* par noeud à l'initialisation. D'une part car les *shells* utilisent des terminaux de contrôle, qui sont différents sur un réseau de stations de travail où chaque site possède son propre terminal, mais qui est unique sur un multicalculteur. Ce terminal est utilisé pour lire les instructions de l'utilisateur ce qui peut créer des problèmes dans la mesure où chaque *shell* lit les mêmes instructions. D'autre part, commander indépendamment chaque noeud au moyen de son *shell* est assez fastidieux pour l'utilisateur dès que le nombre de noeuds dépasse la dizaine. Ces constations nous ont conduit à ne créer, à l'initialisation, qu'un seul *shell* pour le multicalculateur. L'utilisateur peut ensuite, à sa convenance, en créer de nouveaux qui utilisent des terminaux virtuels. Dans notre implantation un *shell* est lancé sur le noeud 1 au moment de l'initialisation. Les gestionnaires de processus des autres sites se mettent simplement en attente d'exécution.

2.3 Le gestionnaire de processus

Nous parlons peu du portage du gestionnaire de processus car il est très réduit. La seule modification que nous ayons fait est d'empêcher les gestionnaires de processus qui se trouvent sur des sites autres que le noeud 0 de faire appel au programme */etc/init* après leur initialisation. De cette manière, ces gestionnaires de processus n'exécutent pas le *shell* mais se mettent en attente d'une commande d'exécution émise par un noeud distant.

Le gestionnaire de processus ne s'exécute pas sur le noeud 0.

2.4 Le gestionnaire de fichiers

Nous n'avons pas porté le DM sur l'hypercube, aussi le gestionnaire de console est encapsulé dans le gestionnaire de fichiers. Chaque fois qu'un périphérique est ouvert (disque ou tty) le système crée, dans le serveur de communication, une nouvelle activité qui va prendre en charge la communication avec ce périphérique.

2.4.1 La gestion des fichiers

A partir de la communication entre la station maître et le multicalculateur, il existe deux possibilités pour implanter une gestion des fichiers compatible UNIX :

(1) la première solution consiste à faire tout le traitement relatif aux fichiers sur la station maître. Le gestionnaire de processus peut envoyer ses requêtes à la station maître plutôt qu'à un serveur de fichiers. C'est ce qui se passerait effectivement si la station maître était gérée par CHORUS/MiX. Dans notre cas, il faut développer un processus qui s'exécute sur la station maître et qui est capable de répondre aux requêtes envoyées par le serveur en utilisant les primitives UNIX de gestion des fichiers. Notre but étant de montrer la potentialité de CHORUS/MiX à développer un système UNIX dédié aux multicalculateurs il nous semble plus intéressant de porter également le gestionnaire de fichiers sur les noeuds puisque nous voulons tester les capacités du système à offrir une interface d'ordre général sur le multicalculateur.

(2) la seconde solution consiste à porter le gestionnaire de fichiers sur un noeud. Par rapport à un gestionnaire de fichiers traditionnel nous devons extraire le code du pilote de disque tout en maintenant le maximum de code de gestion des fichiers sur le noeud [Arm91]. Cette solution consiste à encapsuler les requêtes du gestionnaire de fichiers au pilote de disque dans des messages. Ces messages sont envoyés à la station maître où le processus, qui tient lieu de pilote de disque, est chargé de les traiter.

Le pilote de disque doit être capable de satisfaire trois requêtes principales : l'initialisation du pilote, l'ouverture d'un disque, la lecture et l'écriture sur un disque. Pour l'initialisation, le gestionnaire de fichiers se contente d'envoyer, au serveur de communication une requête de communication avec la station maître (ce qui se traduit par la création d'une activité dédiée à cette communication). Il obtient ainsi la porte sur laquelle il peut envoyer les messages destinés au processus dédié à la gestion des disques sur la station maître. Le code d'encapsulation des requêtes est assez simple puisqu'il se contente de générer un message structuré à partir de l'appel de fonction et de l'envoyer à la porte du serveur de communication gérant les échanges avec la station maître. Sur la station maître le traitement est simplifié car la requête est reçue par un processus UNIX. Celui ci a donc accès à toutes les primitives UNIX et en particulier aux fonctions de lecture et d'écriture sur le disque.

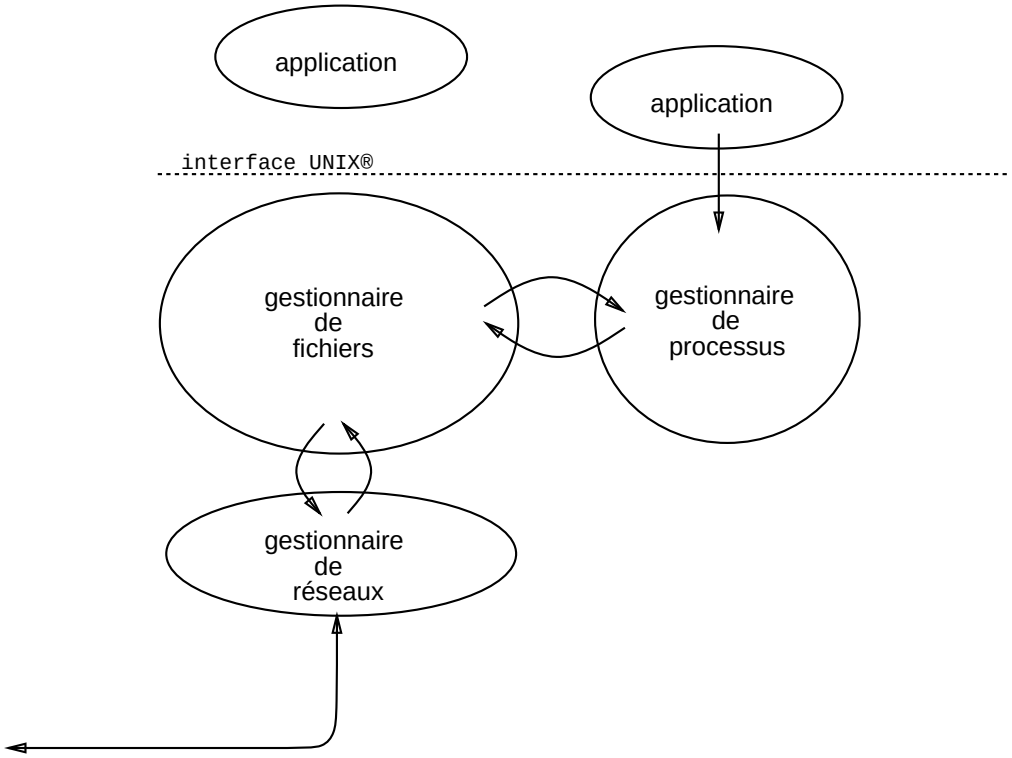

Figure III.9 : Interactions lors de l'accès aux fichiers

2.4.2 La gestion des terminaux

La communication avec la station maître permet également de développer sur les noeuds les fonctionnalités liées aux terminaux. A titre experimental, nous gérons les fonctions de base d'ouverture, de lecture et d'écriture sur un terminal, sans implanter les fonctions du gestionnaire de périphérique (DM) telles que les "*lines disciplines*".

Le pilote de terminaux a été intégré dans le gestionnaire de fichiers. Il est très semblable au pilote de disques car l'interface UNIX d'accès aux terminaux est semblable à celle des fichiers. Par contre la sémantique de la lecture sur un terminal est différente en ce sens qu'elle est boquante : lorsqu'un processus désire lire sur un terminal il se met en attente des caractères à lire. Le processus n'est réveillé que lorsque la totalité des caractères ont été lus. Cela implique qu'une requête de lecture peut rester bloquée un temps non borné. Comme la communication entre la station maître et le noeud est synchrone et que l'activité qui envoie un message à la station maître se met en attente de sa réponse, nous ne pouvons pas gérer tous les terminaux avec une même activité du serveur de communication. A chaque ouverture de terminal, une nouvelle activité est donc allouée dans le serveur de communication.

De même sur la station maître nous avons un processus par terminal virtuel. Ceci nous permet d'utiliser le multifenêtrage d'une station de travail pour avoir plusieurs fenêtres sur les noeuds

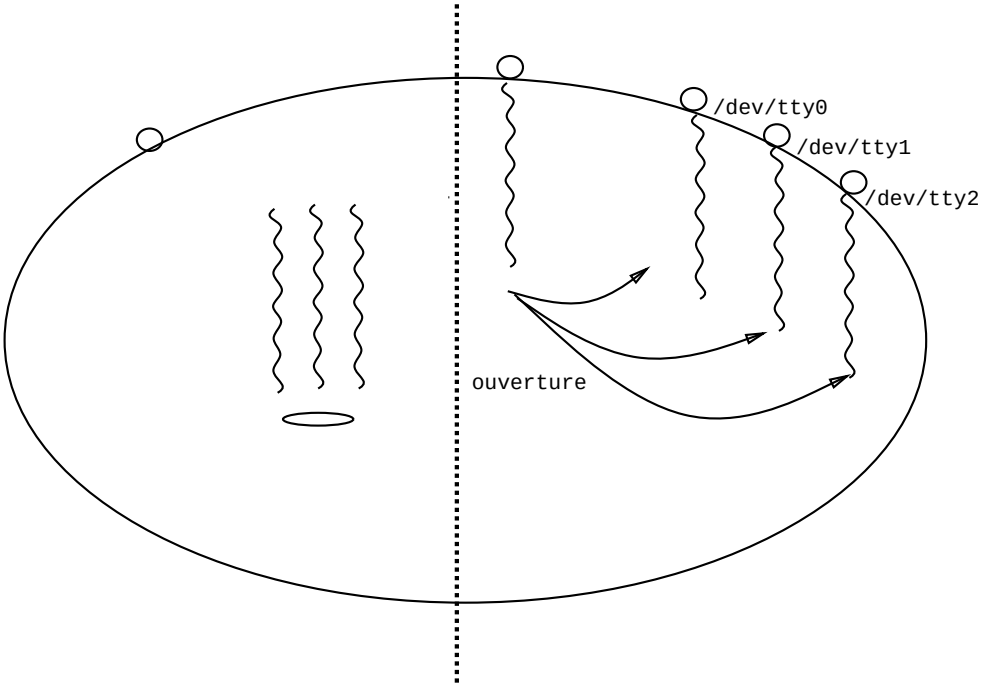

Figure III.10 : Initialisation de la communication NX

de l'hypercube. Nous utilisons l'identificateur des terminaux pour les différencier entre eux. En particulier, ici l'identificateur correspond au *PID* du processus qui gère le terminal. Puisque les messages NX sont adressés grâce aux *PID*, le message envoyé pour un terminal arrive directement au processus qui gère ce terminal. Le processus utilise alors les fonctions standard d'UNIX pour satisfaire les requêtes.

3 Evaluation de l'implantation

3.1 Utilisation du système

Le portage réalisé valide un premier but : offrir sur un multicalculateurs une interface de système d'exploitation standard. En effet nous pouvons utiliser tous les appels système UNIX et les commandes du *shell*. Cette première étape correspond au portage de CHORUS/MiX sur l'iPSC/2. Dans les faits, elle apporte plus que l'interface UNIX standard car les utilisateurs peuvent profiter des extensions CHORUS/MiX exposées au chapitre II. Pour tester notre réalisation et son intérêt nous avons produit une démonstration qui utilise les possibilités de l'iPSC/2 et de CHORUS/MiX.

La première partie de cette démonstration (figure III.11) a surtout pour but de montrer que nous bénéficions d'une interface UNIX complète sur chaque noeud. Nous avons mis sur un pseudo-disque les commandes standard du *shell* et nous les avons chargées sur l'iPSC/2. Ainsi nous

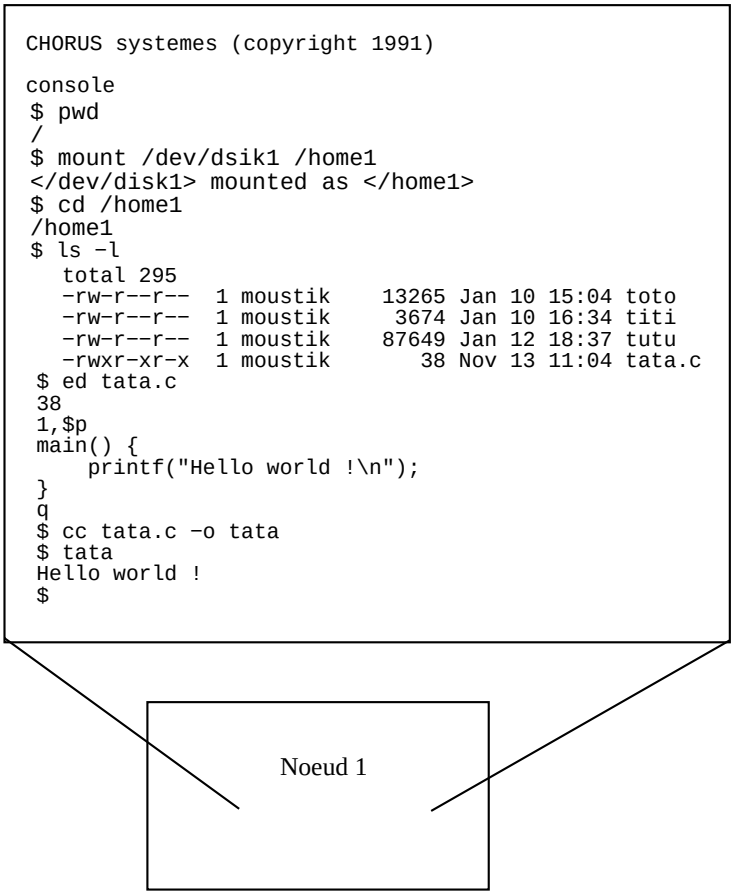

Figure III.11 : Exemple de démonstration

avons pu, sur un noeud : éditer des fichiers (`ed`) et les compiler `cc` ; lire (`ls`, `pwd`), nous déplacer (`cd`) et modifier l'arborescence de nos fichiers (`cp`, `mv`, `rm`) ; afficher et rediriger des traces (`cat`, `echo`) ; contrôler des processus (`ps`, `kill`) ; gérer des disques (`sync`, `mount`). Les exécutables utilisés pour cette démonstration ont été simplement recopiés parmi les binaires utilisés par la station maître (SRM). Grâce à la compatibilité binaire de CHORUS/MiX avec le système UNIX du SRM nous avons pu les utiliser tels quels sur les noeuds. Nous pouvons ainsi remarquer un des premiers avantages d'une interface standard : elle nous permet de bénéficier, à moindre frais, de tous les utilitaires et environnements disponibles sur d'autres machines. En particulier dans le cas des multicalculateurs, qui manquent généralement de support logiciel, nous héritons de l'environnement complet et connu qu'est UNIX. Nous disposons d'un niveau de confort équivalent à celui offert sur les stations de travail.

Le problème alors est de savoir si cet environnement est adapté aux multicalculateurs. En effet le système UNIX et son interface ont été conçus pour des stations de travail. Pour adapter l'interface UNIX à un environnement réparti le système CHORUS/MiX offre des services supplémentaires :

3.1.1 Au niveau du shell

Pour permettre d'exécuter un processus à distance nous utilisons la commande : *remex(2)*. Cette commande prend un numéro de site et le nom d'un exécutable en paramètres. Elle exécute le code correspondant sur le site nommé. Elle permet donc d'accéder explicitement à un site distant depuis un *shell*. En particulier elle permet d'exécuter plusieurs processus en parallèle sur des sites différents en les lançant simultanément (remarque : les processus doivent être lancés en arrière plan pour permettre à l'utilisateur de reprendre la main et lancer d'autres processus). Elle permet également d'exécuter plusieurs *shell* en parallèle, sur le même site ou sur des sites différents. Nous pouvons ainsi simuler une session multi-utilisateurs : en redirigeant le terminal de contrôle d'un *shell* sur une autre fenêtre nous avons plusieurs *shell* qui s'exécutent indépendamment à partir de terminaux différents.

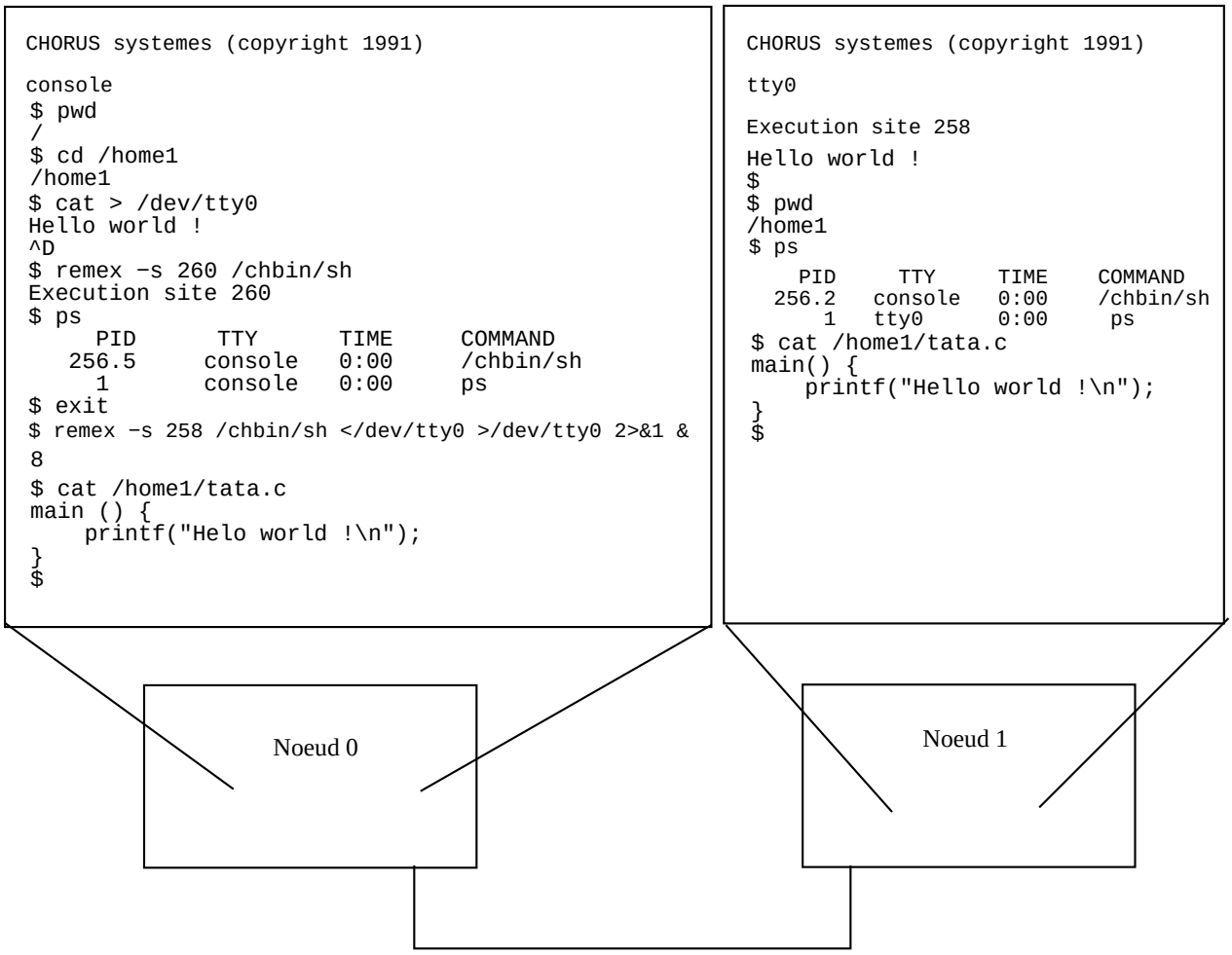

Figure III.12 : Exemple de session en parallèle

La figure III.12 montre une session où l'utilisateur exécute des *shells* sur différents sites en uti-

lisant la commnade `remex`. En particulier, dans la fenêtre de gauche, qui est la fenêtre *console*, l'utilisateur exécute un premier *shell* sur le site 2 au moyen de la commande :

```
remex -s 2 /chbin/sh
```

Les traces générées par ce *shell* n'étant pas redirigées sur une autre fenêtre la banière qui apparaît est celle du *shell* distant. Ainsi lorsque la commande `ps` est exécutée ce sont les processus qui s'exécutent sur le site 2 qui apparaissent. Notons que l'identificateur attribué au *shell* (`/chbin/sh`) est 1.5 pour marquer qu'il est issu du site 1. Une autre exécution à distance d'un *shell* est réalisée par la commande :

```
remex -s 2 /chbin/sh </dev/tty0 >/dev/tty0 2>&1 &
```

Dans ce cas les traces du *shell* sont affichées dans une fenêtre distincte (`tty0`). Nous vérifions que l'environnement d'exécution du *shell* est bien le même que celui du *shell* qui a lancé son exécution : son répertoire courant est `/home1`. De même la commande `ps` permet de voir le site d'exécution.

Les extensions de CHORUS/MiX à la distribution permettent également le contrôle de processus distants. En particulier la commande `kill` accepte un identificateur de processus composé du numéro de site ayant créé le processus et d'un identificateur local. Par exemple dans la figure III.13 l'utilisateur du noeud 2 contrôle l'existence et détruit des processus sur le site 3. Il est donc possible de contrôler l'exécution d'une application parallèle depuis un seul noeud.

3.1.2 Au niveau de la programmation

Toutes ces remarques sont valables pour le *shell* c'est-à-dire pour les traitements interactifs. Pour le moment le multicalculateur n'est pas destiné aux traitements interactifs. Il est principalement fait pour permettre à des grosses applications de s'exécuter rapidement. Nous avons donc vérifié que le système CHORUS/MiX pouvait aussi convenir pour exécuter des applications parallèles. Comme les systèmes natifs des multicalculateurs le système CHORUS/MiX permet l'échange explicite de messages ainsi que l'expression de la concurrence entre les processus. Toutes les fonctionnalités offertes par ces systèmes sont disponibles dans CHORUS/MiX. La structure générale d'une application parallèle sera la suivante :

- un processus maître est lancé depuis le *shell* comme une application standard. Ce processus, après avoir éventuellement initialisé quelques structures, exécute l'appel système *fexec(2)* pour lancer les processus de l'application parallèle. En positionnant la variable *csite* avant chaque appel à *fexec(2)* le processus maître peut définir le placement de ses processus fils sur les noeuds de l'hypercube. Le processus maître peut aussi être utile dans l'initialisation des communications. A la fin de l'initialisation le processus maître peut, soit devenir un processus de calcul comme ses fils, soit se mettre en attente des résultats produits par ses fils.
- un ensemble de processus fils qui sont les processus de calcul. Ces processus sont répartis sur le réseau, ils communiquent par échange de messages entre portes. La connaissance de la porte d'un autre processus de l'application peut être obtenue :

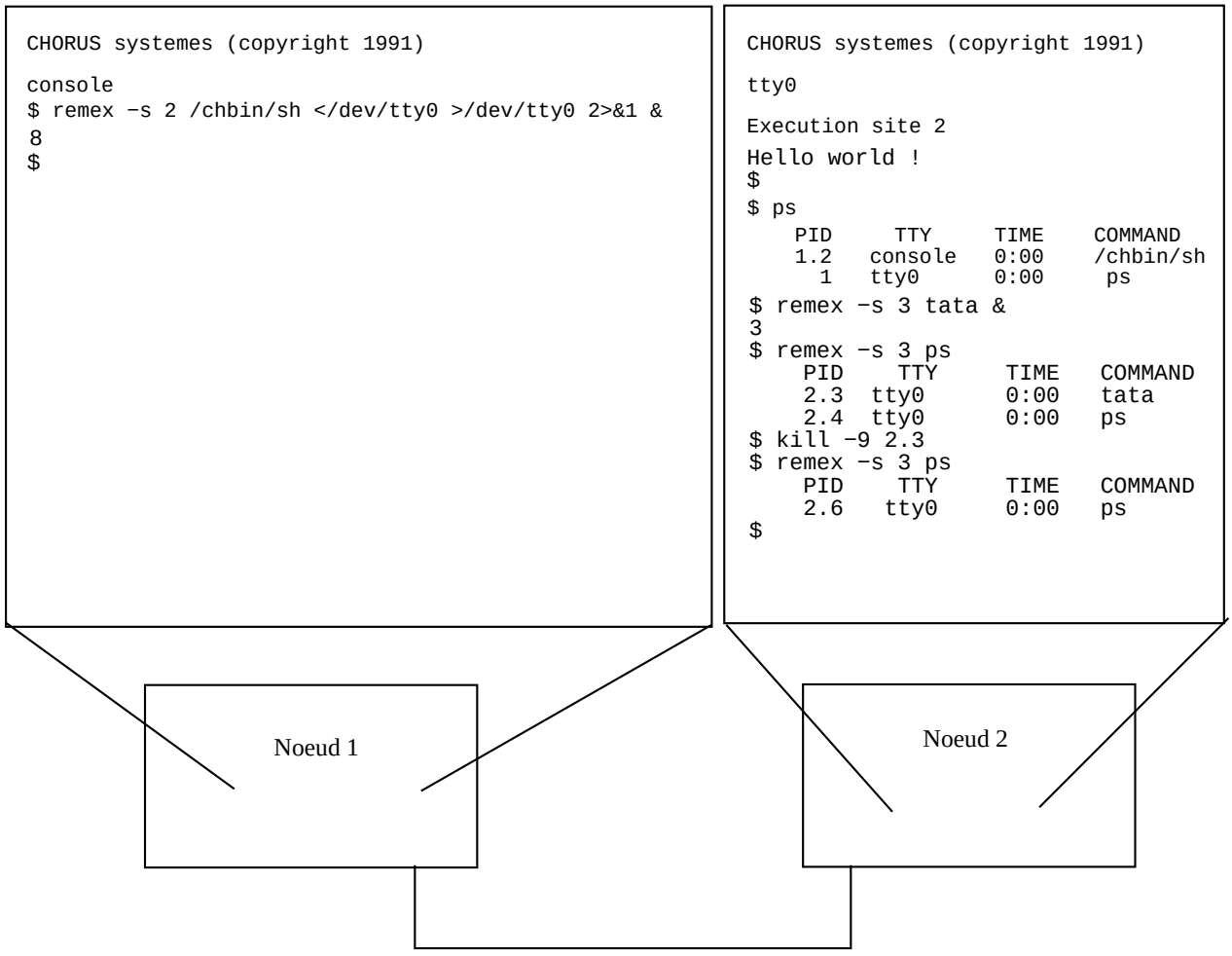

Figure III.13 : Exemple de contrôle distant

– soit en utilisant un groupe statique CHORUS dans lequel chacun met sa porte. Les processus peuvent alors communiquer entre eux par adressage fonctionnel sur le groupe de portes, le numéro de site servant de *cotarget* et déterminant le destinataire. Cette solution implique une dépendance vis-à-vis du placement des processus.
– soit en utilisant le processus père comme serveur de nom de porte. Le processus père crée une porte avant d'exécuter ses fils et met l'identificateur de la porte dans le contexte (variable `env`) de son fils. Lorsque le fils s'initialise il a l'identificateur de la porte de son père. Ceci lui permet d'envoyer ses propres identificateurs de porte au père et de demander ceux des portes de ses frères. Ainsi les processus se connaissent mutuellement et peuvent communiquer.

La figure III.14 montre un exemple de programmation parallèle au dessus de CHORUS/MiX. Le processus père, appelé `Père`, positionne le site d'exécution par défaut pour ses fils. Les processus `toto` et `tutu` utilisent ensuite l'IPC CHORUS pour communiquer.

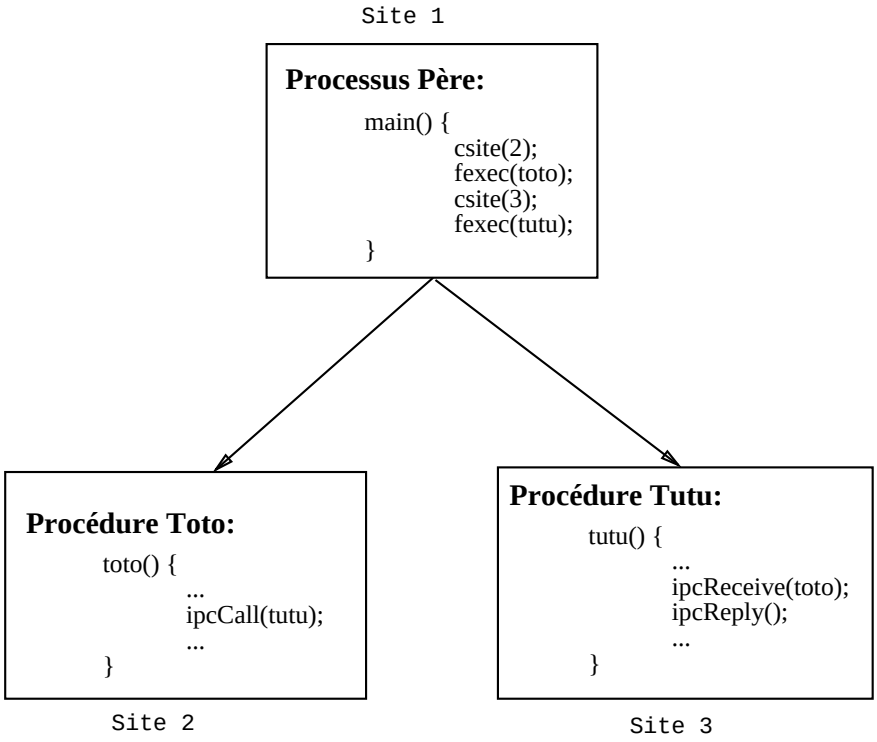

Figure III.14 : Exemple d'application parallèle

3.2 Analyse de l'implantation

Le but de ce portage est d'implanter, sur les noeuds du multicalculateur, un système d'exploitation réparti qui offre l'interface d'un système standard. Nous devons maintenant vérifier que les services offerts par ce système satisfont bien les besoins des utilisateurs habitués aux ordinateurs parallèles et permettent d'en faciliter l'usage pour des utilisateurs moins avertis. Nous comparons donc notre implantation tantôt aux systèmes natifs tantôt à un système UNIX traditionnel.

3.2.1 L'implantation

D'une manière générale, la remarque principale faite sur nos choix concerne les performances offertes par le système d'exploitation. Par exemple, notre implantation obtient de moins bonnes performances pour la communication que le système natif de l'iPSC/2 (NX). La dégradation de performances se situe entre 1,6 et 4 fois suivant le nombre d'octets transmis. Ceci est dû principalement au niveau de service offert par le système. Les choix d'implantation de NX ont été fait résolument dans le but d'offrir de bonnes performances avant tout. Le système est ainsi très dépendant de la machine. Au contraire, le système CHORUS/MiX a été conçu pour être

facilement portable. La comparaison de l'interface de communication entre les deux systèmes montre également une différence de services rendus : NX ne permet aucune transparence dans la communication (un message est toujours destiné à un noeud). Aucune conclusion ne peut être donnée pour le moment en ce qui concerne les performances puisque notre version n'a pas été optimisée. Néamoins il est probable qu'un niveau de service plus élevé engendre une détérioration des performances. Ce problème peut être résolu en implantant deux interfaces dans le système d'exploitation : l'une optimisée mais réduite et l'autre de haut niveau.

La gestion des fichiers et des périphériques n'est pas performante. Ceci peut également constituer un handicap à l'acceptation du système par les utilisateurs. Néanmoins plusieurs parties du code pourraient être optimisées afin d'améliorer les performances. Ainsi l'envoi de messages entre le gestionnaire de fichiers et le serveur de communication pourrait se faire par RPC-léger, comme cela est fait entre le gestionnaire de fichiers et le gestionnaire de processus. C'est principalement sur la station maître que l'on perd beaucoup de temps pour lire ou écrire les informations sur le disque. En effet, une requête passe par trois processus différents avant d'être traitée. Pour obtenir une implantation performante il faudrait pouvoir implanter l'accès au disque dans le processus de la station maître qui gère la communication. Ce problème n'est cependant pas général aux multicalculateurs mais plutôt dû à leur configuration en co-processeur d'une station maître. C'est le passage par le goulet d'étranglement que constitue cette station qui pose problème. Lorsque le multicalculateur sera une machine indépendante avec ses propres disques de tels problèmes ne se poseront plus.

3.2.2 Rapport aux systèmes natifs

Les défauts attribués aux systèmes natifs sont de ne pas être des systèmes indépendants puisqu'une partie de la gestion des applications se fait sur la machine maître et de ne pas offrir une interface standard. Le système CHORUS/MiX comble ces défauts. Le portage d'un système complet sur les noeuds du multicalculateur permet de voir la machine comme étant indépendante de la station maître. Nous pouvons donc penser que les prochaines générations de multicalculateurs seront des machines sans station maître qui seront accédées directement sur les noeuds à travers un réseau, de même que les stations de travail. Ceci permet d'éviter la petite gymnastique nécessaire au chargement d'un programme sur les noeuds, aussi bien au niveau de l'exécution de l'application que de sa programmation.

L'utilisation d'une interface standard de station de travail permet également de faciliter à l'utilisateur le passage d'un type de machine à un autre : d'une part car l'écriture de nouvelles applications ne nécessite pas l'apprentissage de nouveaux services ou de nouvelles commandes, d'autre part car les applications utilisées et développées sur une station de travail peuvent être facilement portées sur le multicalculateur. Les possibilités d'utilisation en multi-utilisateurs de la machine en facilite également l'accès et le rendement. Le multicalculateur est alors vu de la même manière qu'une station de travail, et non plus comme un co-processeur derrière une station maître.

Pour les applications très coûteuses en temps de calcul il peut être intéressant de pouvoir réserver certains noeuds lors de l'exécution. Nous garantissons ainsi à l'application un certain potentiel

d'exécution. Ce service est imposé par NX mais n'est pas implanté par CHORUS/MiX. Il sera probablement intéressant de proposer un tel service si le système est destiné a supporter des applications de calcul intensif.

3.2.3 Rapport à UNIX

Les intérêts principaux de CHORUS/MiX par rapport à UNIX sont l'extension des services à la répartition et la possiblité de configurer le système en fonction des ressources présentes sur le site. Un système UNIX traditionnel permet le développement d'applications parallèles à travers les primitives `fork(2)` et `exec(2)` pour la création dynamique de processus et à travers les tubes pour la communication. L'ajout des services TCP/IP permet également la création à distance de processus. Néanmoins il est clair qu'une interface par échange de messages est plus facilement utilisable qu'une communication par tube.

Comme nous l'avons vu, la possibilité de configurer le système en fonction des ressources présentes sur un noeud est très utile sur un multicalculateur : aussi, en configurant le système du noeud 0, il nous a été facile de le dédier à la gestion des disques. Cette manipulation n'aurait pas été possible avec un système UNIX traditionnel. De plus la possibilité de placer le gestionnaire d'un type de service sur un seul noeud et d'en faire profiter les autres noeuds de façon transparente nous permet d'offrir un grand nombre de services sans pour autant surcharger les noeuds.

Par rapport à un système CHORUS/MiX traditionnel, notre implantation de CHORUS/MiX ne permet pas d'avoir accès à la totalité des appels système UNIX. En particulier une gestion minimale des terminaux, implantée dans le gestionnaire de fichiers, a été portée sur l'iPSC/2. Notons qu'aucun problème technique ne s'oppose au portage du serveur de terminaux (DM). D'autres serveurs de CHORUS/MiX auraient également pu être portés pour gérer l'IPC system V, les tubes, etc.

3.3 Besoins des multicalculateurs

L'analyse et l'utilisation de notre implantation nous ont permis de mieux comprendre les besoins, en terme de systèmes d'exploitation, des multicalculateurs.

Le système CHORUS/MiX permet un accès transparent à la plupart des ressources sauf au processeur (le nom ou le numéro de site doit être précisé pour une exécution). Ainsi les programmes parallèles écrits au dessus de CHORUS/MiX dépendent de l'architecture du réseau sur lequel ils s'exécutent. Avec un nombre de noeuds limité il est possible à l'utilisateur de gérer la répartition de ses programmes, de placer lui-même ses processus sur les différents noeuds, par contre ce travail devient fastidieux lorsque le nombre de noeuds augmente. Il en est de même pour la gestion de la tolérance aux pannes et d'une manière générale pour tout ce qui touche aux ressources d'exécution.

Nous pouvons remarquer que l'architecture d'une station de travail est centralisée autour d'un processeur. Il est actuellement possible d'augmenter la capacité des autres composants de la station (unité de mémorisation, unité d'entrée/sortie) mais pas celle du processeur. Dans un environnement de multicalculateur, au contraire, la ressource processeur n'est plus considérée

comme une ressource particulière et elle est mise au même niveau que la mémoire, les disques, etc. Il est alors possible d'augmenter la puissance de calcul en augmentant le nombre de processeurs de la même manière que la capacité de mémorisation peut être augmentée en utilisant une extension mémoire). Cette nouvelle vue de la ressource d'exécution implique de définir une nouvelle politique pour la gérer. Il faut donc, à partir des ressources physiques, offrir un niveau de machine virtuelle grâce à une transparence renforcée pour l'accès au processeur.

Les utilisateurs potentiels des multicalculateurs connaissent bien les interfaces des systèmes d'exploitation gérant les architectures centralisées, c'est-à-dire qu'ils comprennent bien la gestion centralisée des ressources telle qu'elle est implantée dans ce cas. L'implantation d'un système d'exploitation dont l'interface offre une vision centralisée de la gestion des ressources doit faciliter l'adaptation de ces utilisateurs aux architectures parallèles. Ainsi les multiprocesseurs à mémoire partagée sont déjà utilisés dans une large gamme d'applications car ils offrent cette vue centralisée.

Les systèmes répartis actuellement proposés ne sont donc pas bien adaptés à la gestion des multicalculateurs. Nous devons développer de nouveaux services, en particulier la transparence d'accès aux ressources d'exécution, pour obtenir un système qui permettra à d'utiliser plus facilement les machines parallèles. Nous pensons que la définition donnée par Tanenbaum et Van Renesse dans [Mul89] sur les systèmes répartis doit convenir aux systèmes d'exploitation qui seront développés pour les multicalculateurs : *Un système d'exploitation réparti est un système qui apparait à ses utilisateurs comme un système d'exploitation centralisé traditionnel mais qui gère plusieurs processeurs indépendants. Dans ce cas le concept clé est la transparence, en d'autres termes, l'utilisation de plusieurs processeurs doit être transparente à l'utilisateur. Une autre manière d'exprimer ceci est de dire que l'utilisateur voit le système comme un "uniprocesseur virtuel" et non pas comme un ensemble de machines distinctes.* Ainsi le système d'exploitation, sur ces architectures, ne sera plus vu comme un ensemble d'instances de systèmes d'exploitation mais comme une instance unique et l'utilisateur ne verra de la machine qu'un serveur d'exécution puissant. Nous appelons ce système : un **système à image unique**.

Conclusion

Dans ce chapitre nous avons décrit les principaux problèmes à résoudre pour porter CHORUS/MiX sur un multicalculateur. L'utilisation de ce système sur l'iPSC/2 nous a permis d'analyser les avantages et les défauts d'un tel système sur un multicalculateur. Ainsi la baisse des performances semble être le point le plus négatif et le gain de confort et l'utilisation multi-utilisateurs, les principaux bénéfices pour l'utilisateur. Nous avons également pu établir que l'interface du système CHORUS/MiX, même si elle est plus complète que l'interface UNIX, n'offre pas un confort suffisant pour permettre le support d'utilisateurs non avertis.

Le développement de nouveaux services est donc nécessaire pour permettre l'utilisation des multicalculateurs dans une vaste gamme d'applications. Au niveau de la programmation le besoin le plus important est la transparence d'accès aux différents processeurs qui doit permettre au programmeur de ne pas se soucier du mapping de son programme sur le réseau de l'ordinateur.

De cette façon les programmes peuvent être portés sur un grand nombre de machines (y compris les monoprocesseurs) sans aucune modification, à condition d'avoir le même environnement.

Le but des systèmes d'exploitation répartis actuels n'est pas encore de prendre en compte les nouvelles architectures de multicalculateurs. Par contre ils peuvent constituer de bonnes bases pour construire les systèmes futurs qui gèreront les machines parallèles. Nous traitons, dans la partie suivante, des principaux problèmes liés à l'implantation des services nécessaires à un système à image unique dans un système réparti, en particulier dans le cas de CHORUS/MiX.

Deuxième partie

Introduction

Nous avons appelé un **système d'exploitation à image unique** un système d'exploitation qui gère un réseau de processeurs de manière à donner à l'utilisateur la vue d'une machine monoprocesseur virtuelle, c'est-à-dire qui cache la répartition des ressources à l'utilisateur.

4 Le système à image unique

Nous attendons d'un système à image unique qu'il gère la répartition des ressources du multicalculateur pour offrir la vue d'un système centralisé. L'implantation d'un système d'exploitation à image unique impose donc d'utiliser de nouvelles politiques pour gérer les ressources : le système d'exploitation doit tenir compte de leur répartition mais les utilisateurs doivent pouvoir y accéder comme si elles étaient locales. Pour la mise en oeuvre d'un tel système les principaux problèmes sont :

l'ordonnancement
: Dans l'optique d'un système à image unique, l'utilisateur ne se soucie pas de la distribution de ses entités d'exécution sur le réseau. C'est le système qui doit gérer leur ordonnancement en fonction de la disponibilité des processeurs, ce qui nécessite la mise en place d'une politique globale d'ordonnancement.

la transparence
: Puisque l'utilisateur ne connait pas le site d'exécution de ses processus, il doit pouvoir accéder à une ressource indépendamment de sa localisation. De plus toutes les ressources doivent être accessibles depuis tous les sites du système. Le système doit donc permettre un l'accès transparent à toutes les ressources.

la cohérence
: Plusieurs ressources équivalentes peuvent être gérées par différents noeuds du réseau. Il faut assurer leur cohérence pour garantir à l'utilisateur que le service rendu est le même quelque soit le site. Par exemple, un système à image unique implique la mise en place d'une horloge globale.

une distribution adaptée
: Une gestion centralisée des ressources génère fréquemment des goulets d'étranglement. Par contre, une gestion distribuée est plus difficile à mettre en oeuvre et entraine un surplus d'échanges de messages. Le développement d'un système à image unique implique de trouver un degré de distribution adapté.

Ces problèmes sont indépendants de l'interface offerte par le système d'exploitation. Comme nous l'avons vu, l'aspect standard de l'interface est un facteur important qui justifie un tel système. Or les interfaces standards actuelles sont des interfaces de systèmes d'exploitation centralisés. Il est donc fréquent que l'implantation de telles interfaces amène à résoudre des problèmes qui sont plus liés à la distribution de l'interface qu'à la distribution de la gestion des ressources. Par

exemple, la gestion de fichiers mappés UNIX en distribué impose la mise en place, par le système, de fonctionnalités de gestion de la cohérence en réparti.

Si un système à image unique doit permettre de diversifier l'utilisation des multicalculateurs il ne doit pas être incompatible avec l'utilisation qui en est faite actuellement, c'est-à-dire qu'il doit permettre d'optimiser une application parallèle, en particulier, en explicitant un placement pour une telle application et la réservation d'un certain nombre de noeuds pour son exécution.

Le système doit donc offrir deux niveaux compatibles d'interface : l'un permettant de s'abstraire de la répartition, l'autre permettant de la gérer au niveau de l'application.

5 Une base pour un système à image unique

Pour construire un système à image unique il est possible de réécrire entièrement un système d'exploitation. Ceci permet de concevoir le système en tenant compte des particularités des machines auxquelles il est destiné. Il est également possible de modifier un système d'exploitation existant pour l'adapter à la machine cible. C'est généralement la seconde solution qui est choisie car elle est moins coûteuse et elle permet d'offrir facilement une interface standard.

Nous avons choisi d'utiliser le système CHORUS/MiX pour tester les nouvelles fonctionnalités et offrir une interface de système d'exploitation adaptée aux multicalculateurs[GHP90]. Notre choix s'appuie sur les raisons suivantes :

- Les systèmes basés sur un micro noyau ont été conçus pour être facilement modulables et adaptables à plusieurs types d'architectures. Ils semblent particulièrement intéressants dans le cas des multicalculateurs car ils offrent une grande flexibilité. L'interface du micro-noyau permet en effet une programmation système de haut niveau qui facilite la conception de nouveaux services.
- L'implantation d'une interface standard de haut niveau (ex : UNIX) sous forme multiserveurs permet une intégration plus facile de nouveaux services. En effet, la définition d'une coopération entre deux serveurs n'implique pas de modification dans le comportement des autres serveurs.
- Parmi les systèmes décrits au chapitre II seul CHORUS/MiX offre à la fois une interface standard et une implantation multiserveurs. De plus il étend l'interface standard pour permettre à l'utilisateur de tirer parti de la répartition. Il n'impose donc pas une modification de l'interface pour la mise en place de nouveaux services.
- Le fait d'avoir entièrement réécrit le code qui implante l'interface UNIX a permis de mieux le structurer. En particulier en utilisant le modèle objet. Il est alors plus facile de faire évoluer les serveurs pour qu'ils rendent de nouveaux services. Ceci devrait permettre de compléter le sous-système UNIX pour le rendre adaptable à un plus grand nombre d'architectures.
- La transparence de l'IPC permet de mettre en place un service accessible par tous et facilement portable à différentes architectures de machines. D'autre part, la gestion de ressources telles que des fichiers bénéficie de cette transparence ce qui limite les développement nécessaires à la réalisation d'un système à image unique.
- Le système CHORUS/MiX est un système qui permet une bonne intégration (figure III.15) du multicalculateur dans un réseau local puisqu'il permet la communication entre sous-domaines,

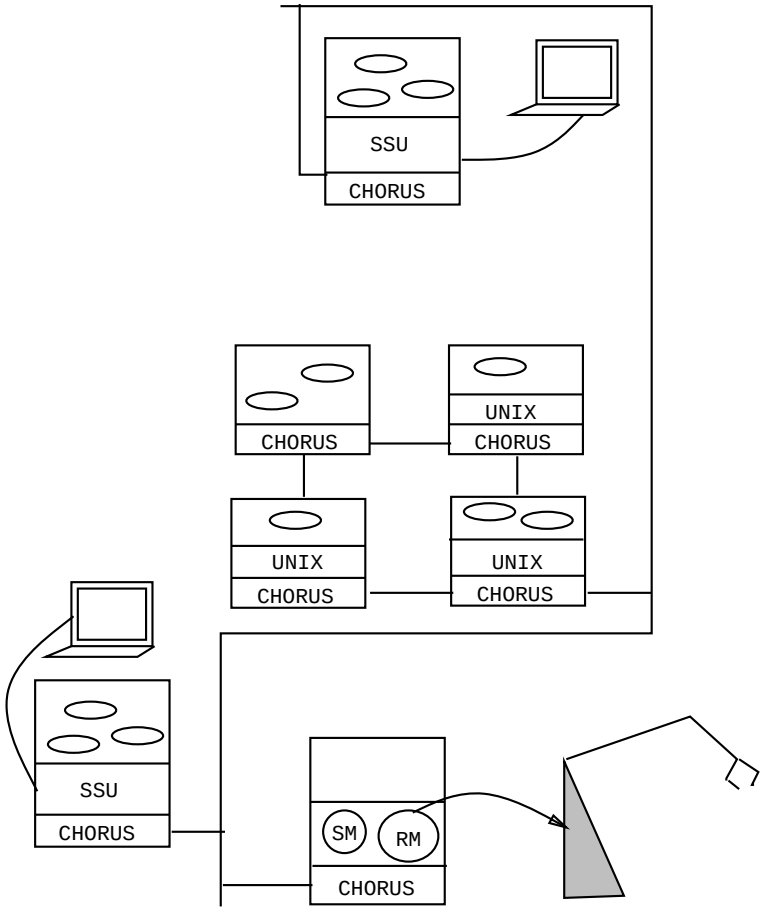

Figure III.15 : Intégration d'un multicalculateur

même si les protocoles sont différents.

Etant donné les caractéristiques de CHORUS/MiX et les problèmes d'implantation d'un système à image unique, tels que nous les avons présentés, les principaux services manquants à CHORUS/MiX sont : la gestion de charge répartie et la gestion cohérente de certaines ressources telles que le temps. Nous pensons que la gestion de charge est le problème déterminant dans l'évolution du système CHORUS/MiX vers un système à image unique. En effet le processeur est la seule ressource pour laquelle le système ne garantit pas un accès transparent et une gestion répartie. Dans les deux chapitres suivants nous traitons donc des problèmes liés à l'implantation d'un service de gestion de charge répartie. En particulier, nous traitons dans le chapitre IV de la migration de processus qui est utilisée par le gestionnaire de charge pour permettre une meilleure distribution des processus sur le réseau.

Chapitre 4

La migration de processus

Introduction

Dans un système réparti, le placement des entités logiques - les entités d'exécution, les données ou les ports de communication - dépend de la disponibilité des ressources physiques qu'elles requièrent. La disponibilité de ces ressources physiques pouvant évoluer, les entités doivent être déplacées pour assurer la continuité du service. Par exemple, l'optimisation d'une application parallèle peut être obtenue en déplaçant des données vers des sites peu actifs [Mig92] ou ayant de la place mémoire disponible. Un service de déplacement des entités logiques, pour être utilisable dans le cadre d'une politique globale d'affectation dynamique des ressources aux entités, ne doit pas poser d'hypothèses quant à la nature des entités et leur comportement futur. C'est le déplacement de processus - entité qui recouvre les trois précédentes - qui permet de définir un service utilisable par le plus grand nombre d'applications. En effet, le processus définit un ensemble complet de ressources indépendantes qui peut donc être déplacé. Nous utilisons ce service pour permettre la reconfiguration dynamique dans le système à image unique.

Dans la première partie nous donnons une vue synthétique des problèmes posés par la migration de processus, tels qu'ils sont décrits dans la littérature. Nous décrivons, dans la seconde partie, une migration de processus pour CHORUS/MiX et sa réalisation sur l'iPSC/2.

1 Présentation

La migration de processus est un sujet de recherche très actif. Cependant si le sujet a été approfondi en ce qui concerne les stations de travail, comme en témoigne la littérature, peu de propositions existent sur multicalculateurs. C'est pourtant dans ce domaine et celui de la tolérance au pannes qu'elle peut faire ses preuves. Cette partie donne un aperçu de la migration de processus telle qu'elle est décrite dans la littérature.

1.1 Motivations

Le développement d'un service de migration de processus est généralement motivé par l'un des besoins suivants [JLHB88] [Esk91] [JV88b] [Smi88] :

Partage de la charge En déplaçant des processus d'un site chargé vers un site moins chargé, nous pouvons obtenir une meilleure utilisation des ressources d'exécution. Ce partage peut aussi s'appliquer à d'autres ressources telles que la mémoire.

Continuité de service Avant l'arrêt d'un site, il est possible de migrer les processus qui s'y exécutent. Nous pouvons ainsi garantir la continuité d'exécution des processus et la continuité d'accès aux services rendus par ces processus (si les ressources qu'ils utilisent sont disponibles à partir d'autres sites).

Baisse du trafic réseau En plaçant sur un même site les processus qui communiquent intensément nous réduisons le trafic sur le réseau.

Utilisation d'une ressource Lorsqu'un processus accède fréquemment à une ressource, nous pouvons le placer sur le site qui gère cette ressource pour obtenir des gains de temps d'accès - pour le processus - et une baisse d'utilisation du réseau - pour le système.

Ces motivations justifient le développpement d'un service de migration mais leurs besoins différents induisent des caractéritiques différentes pour le service. Nous donnons quelques définitions habituellement reconues dans la littérature.

1.2 Définitions

1.2.1 Terminologie

Nous faisons référence au déplacement, d'un site à un autre, d'une entité d'exécution et des données qui y sont associées sous le nom de **migration de processus**. La migration peut se produire à un instant quelconque de l'exécution du processus. Nous la différencions du placement de processus qui ne peut avoir lieu qu'au moment de l'activation du processus.

La migration d'un processus consiste en l'extraction des données liées à ce processus d'une instance de système d'exploitation pour les replacer dans une autre instance. L'**état du processus** est l'ensemble des données qui caractérisent le processus. Cet état doit être suffisant pour créer un processus qui sera l'image exacte du processus modèle.

La migration est effectuée en déplaçant un processus d'un site d'**origine** pour l'exécuter sur le site **destinataire**.

1.2.2 Principales phases

D'une manière générale, la migration d'un processus peut être décomposée en trois phases principales (figure IV.1) :

– L'arrêt, ou **gel**, du processus. Pour migrer un processus nous avons besoin de le représenter sous forme de données significatives pour une autre instance de système d'exploitation. Le processus doit être stoppé avant d'être migré : ainsi son état ne change pas pendant la migration.

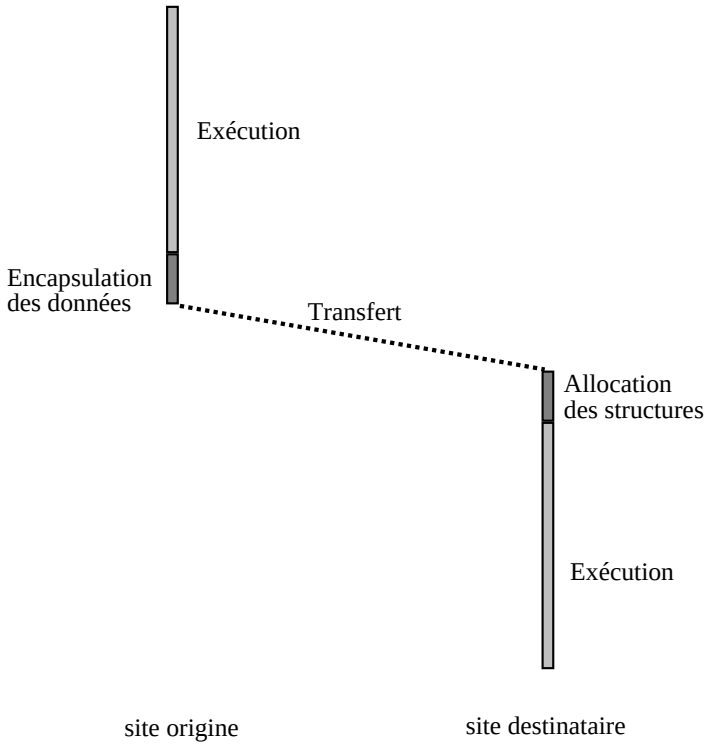

Figure IV.1 : Les phases d'une migration

– Le **transfert** du processus. Cette phase consiste à encapsuler les données liées au processus puis les envoyer au site destinataire. Elle repose donc sur un mécanisme de transfert de données entre deux sites auquel accède le service de migration. Le transfert peut être effectué en plusieurs échanges.
– Le **réveil** du processus. Le site destinataire doit créer toutes les structures nécessaires pour accueillir le processus, les initialiser avec les données qui sont reçues et, enfin, exécuter le processus.

Ces trois phases peuvent être exécutées concurremment pour des raisons d'optimisation. Elles sont cependant soumises aux contraintes temporelles suivantes : le gel doit avoir lieu avant la fin du transfert et le réveil du processus doit avoir lieu après le début du transfert.

1.2.3 Caractéristiques

Un service de migration peut être qualifié par les caractéristiques suivantes :

– La **transparence** : la migration d'un processus est dite transparente si le processus n'est pas averti de sa migration. La décision de migration est donc prise sans tenir compte de l'avis du processus.

– Les **dépendances résiduelles** : la migration d'un processus implique des dépendances résiduelles si une partie des données qui caractérisent le processus sont maintenues sur le site origine. Ces données peuvent être **critiques** si elles sont indispensables à l'exécution correcte du processus.
– L'**hétérogénéité** : les processus ne sont habituellement pas migrés entre des processeurs différents car les ensembles d'instructions ne sont généralement pas compatibles. La migration, dans un environnement hétérogène, peut être obtenue en définissant un ensemble d'instructions virtuelles, destinées à rendre le milieu homogène.
– L'**exactitude** : le processus du site destinataire est considéré comme une image exacte du processus du site d'origine, s'il n'y a aucune modification dans l'état du processus ni aucune perte de service. Sur le site destinataire, le processus migré doit bénéficier des mêmes services que sur le site origine et il doit les accèder avec les mêmes données.

1.3 Principaux problèmes

Plusieurs problèmes se posent pour l'implantation d'un service de migration. Parmi ceux qui sont décrits dans la littérature, nous avons sélectionné ceux qui sont d'ordre général et qui ne couvrent pas un problème d'implantation spécifique à un système d'exploitation.

1.3.1 Etat du processus

Le choix des données composant l'état du processus doit être fait de manière précise. Si une des données manque, l'état du processus sur le site destinataire est modifié et les contraintes d'exactitude de la migration ne sont plus satisfaites. Généralement l'état d'un processus peut être décomposé comme suit :
– le contexte de mémoire. C'est le contenu de la mémoire utilisée par le processus au moment du gel. Il est composé du code, des données et de la pile.
– le contexte du processeur. C'est l'état des registres du processeur au moment où le processus est gelé. Ce contexte est dépendant du processeur il ne peut donc être réutilisé que sur un processeur acceptant le même jeu d'instructions.
– le contexte système. Ce sont toutes les données qui caractérisent le processus, dans le système d'exploitation. Parmi ces données certaines doivent être conservées, comme l'identificateur du processus, et d'autres, comme le site d'exécution, seront obsolètes sur le site destinataire. D'autre part la migration d'un processus engendre des modifications dans les données propres au système d'exploitation. C'est, par exemple, le cas du nombre de processus exécutés localement. Ces données ne sont pas migrées mais doivent être mises à jour.
– le contexte de communication. Au processus sont associées des structures destinées à gérer la communication (tampons, portes, etc). Il faut s'assurer que ces structures sont migrées avec le processus pour garantir l'exactitude de la migration.

Les données qui caractérisent l'état du processus sont généralement toutes gérées par le système d'exploitation. Pour cette raison, la migration de processus est très dépendante du système d'exploitation sur lequel elle est implantée.

1.3.2 Gel du processus

Un processus étant, par définition, une entité active son état se modifie constamment. Pour le migrer nous devons obtenir une image fixe qui représente le processus à un instant de son exécution. Il faut donc le geler.

Le principal problème, posé par le gel d'un processus, est la nécessité d'obtenir un état du processus qui soit cohérent. Les contextes de mémoire et du processeur ne posent pas de problème : ils peuvent être considérés comme étant dans un état cohérent quelque soit l'instant d'exécution du processus. Le processeur garantit généralement qu'une instruction n'est jamais interrompue. Ainsi, lorsque le processus est gelé, les registres du contexte du processeur ne sont pas en cours de modification et une adresse mémoire n'a jamais été partiellement modifiée.

Par contre les données du système d'exploitation, le contexte système du processus, ne sont pas forcément dans un état cohérent. Si le processus exécute un appel système il accède aux données du système. Lorsque l'ordre de migration est donné le processus peut avoir commencé une modification des données du système qu'il ne peut continuer sur l'autre site. Par exemple, il peut détenir le verrou d'accès à une ressource : si ce verrou n'est pas relâché, sur le site d'origine, la ressource est inutilisable après la migration du processus. Un problème équivalent se pose avec le contexte de message : si le processus est migré à l'instant où il exécute la réception du message il risque d'en perdre une partie. La migration d'un processus en cours de communication synchrone pose également plusieurs problèmes : comment rediriger la réponse ? comment resynchroniser les processus une fois la migration achevée ?

Pour geler un processus il faut donc trouver un point cohérent de son exécution. Dans Condor [BL90], les processus ne peuvent être déplacés qu'à des instants précis de leur exécution appelés *checkpoint*. Lorsqu'un processus est engagé dans une communication synchrone, les systèmes Charlotte [AF89], MOS [BW89] et Sprite [DO91] attendent la fin de cette communication pour migrer le processus.

1.3.3 Le transfert du processus

Il est souvent reproché à la migration de processus d'être peu intéressante car le temps de transfert est trop long. Pour optimiser cette durée il est possible d'utiliser différentes techniques qui supposent, pour la plupart, l'existence d'une mémoire virtuelle paginée à la demande. Les principaux algorithmes utilisés pour transférer un processus sont les suivants :

- L'algorithme le plus simple consiste à encapsuler les données qui caractérisent le processus dans des messages. Un seul message peut ne pas être suffisant dans la mesure où la taille du processus n'est pas fixée et que le système limite généralement la taille des messages. Cet algorithme est utilisé par les systèmes Charlotte [AF89], Demos/MP [PM83], Locus [BP86] et Mosix [BP86].
- La pré-copie [TLC85]. Cette technique est itérative. Pendant les itérations le processus continue son exécution. Au premier pas, la totalité du code et des données du processus sont envoyées au site destinataire. Après ce premier transfert, nous pouvons supposer que le processus a seulement modifié une partie des pages de données. Dans la seconde itération, seules les pages de données modifiées sont envoyées au site destinataire. Le nombre de pages, envoyées au

cours de la seconde itération, étant plus petit que le nombre total de pages, le temps de transfert diminue. Par conséquent, le nombre de pages modifiées, dans cette deuxième phase, doit diminuer également. L'algorithme itère jusqu'à obtenir un nombre de pages inférieur à un seuil donné. Lorsque le seuil est atteint, le processus est gelé, le contexte système est envoyé avec les dernières pages modifiées et le processus est définitivement installé sur le site destinataire. Ainsi le temps pendant lequel le processus est arrêté ne dure que le temps du transfert du contexte système, de quelques pages de données et de l'installation du processus.
Du point de vue du processus cette technique fonctionne très bien puisque le temps d'arrêt est réduit. D'autre part cette technique n'engendre pas de dépendances résiduelles, elle doit donc tolérer les pannes. Le principal reproche que nous pouvons faire à la pré-copie est l'augmentation du trafic réseau et de l'activité du système d'exploitation qu'entraîne le transfert répété d'une même page. Le gain est visible au niveau du processus, l'augmentation du trafic réseau et de l'activité du système d'exploitation limite le gain au niveau du système entier.

– La copie paresseuse [Zay87]. Cette technique repose principalement sur un service particulier de la mémoire virtuelle : la copie sur référence. La copie sur référence évite le transfert effectif des données lors de l'envoi d'un message. Le transfert a lieu lorsque le destinataire du message réfère explicitement les données. En particulier, dans le cas d'une mémoire paginée, le transfert a lieu page par page lorsque celles-ci sont accédées. Sur la base de ce service, il est possible de développer une fonctionnalité de migration de processus qui n'envoie que le contexte système du processus au site destinataire. Les pages de données sont envoyées sur le principe de la copie paresseuse.
Le gain de temps est le principal bénéfice de cette technique. En effet au moment de la migration du processus seul le contexte système est déplacé. Par contre, le coût réel de la migration est difficile à calculer dans la mesure où nous ne pouvons savoir quelles sont les pages qui seront déplacées, ceci dépendant des pages accédées par la suite. Si le processus migré n'accède que peu de pages alors le gain de temps est important. Mais ce gain peut être réduit si nous tenons compte du temps de traitement supplémentaire dû aux défauts de page. D'autre part, les dépendances résiduelles entraînées par cette technique ne permettent pas de l'utiliser pour supporter une fonctionnalité liée à la tolérance aux pannes (cf. 1.3.7).
– La copie sur disque [DO91]. Cette technique repose également sur un service de copie sur référence. Elle consiste à sauvegarder sur un disque les données qui caractérisent le processus. Le contexte système est alors envoyé au site destinataire qui exécute le processus. Les défauts de pages sont résolus en chargeant, depuis le disque, les pages accédées.
Dans un contexte où chaque site possède un disque, cette technique peu s'avérer intéressante car les accès au disque sont rapides. Par contre dans un contexte où il y a peu de disques, cette technique perd son intérêt. Par exemple, si les sites origine et destinataire ne possèdent pas de disque, la migration fait appel à un troisième site, possèdant un disque. Les données du processus sont sauvegardées sur le disque pour être lues par le site destinataire. Elles transitent alors deux fois par le réseau. D'autre part les disques, s'il sont en nombre insuffisant, peuvent devenir des goulets d'étranglement.

1.3.4 Réveil du processus

Pour garantir une migration de processus qui ne risque pas de pénaliser le propriétaire du processus déplacé, le site destinataire ne doit lancer l'exécution du processus que s'il est sûr qu'elle peut avoir lieu dans des conditions satisfaisantes. Sinon il doit prévenir le site origine que la migration ne s'est pas déroulée correctement. De son côté le site d'origine doit conserver l'image du processus migré tant qu'il n'est pas assuré que le processus s'exécute sur le site destinataire. Si un problème intervient pendant la migration, il est ainsi prêt à re-lancer l'exécution du processus.

1.3.5 L'exactitude

Après la migration, le processus doit continuer son exécution sur le site destinataire sans erreur. Il doit pouvoir accéder à toutes les ressources auxquelles il accédait sur le site d'origine. De plus, il ne doit pas y avoir de différence notable entre le comportement du système d'exploitation du site d'origine et celui du site destinataire. Ceci pose les problèmes suivants :

- la gestion du temps : les horloges de deux sites évoluent indépendamment l'une de l'autre. Aussi, après une migration, le processus peut obtenir une date inférieure à celle qu'il possédait avant la migration.
- la disponibilité des ressources. Deux sites n'offrent pas forcément les mêmes ressources : périphériques, mémoire libre, puissance du processeur, etc. Le problème est différent suivant la ressource considérée :

  l'accès aux périphériques : le système doit mettre en place un système de nommage global qui garantisse l'accès transparent depuis un site quelconque du réseau.

  la place mémoire disponible : le site destinataire ne dispose pas forcément d'autant de mémoire libre que le site origine. Il faut donc s'assurer avant de déplacer le processus que cet espace est bien suffisant.

  les temps de communication : entre deux processus, ils ne seront plus forcément les mêmes. En effet, si le processus migré communiquait avec un processus local, alors les temps de communication seront plus longs après la migration.

  les alarmes : elles sont retardées par la migration du processus.

- En aucun cas la migration ne doit entraîner la destruction du processus.

Ainsi un environnement d'exécution exactement équivalent à celui du site origine est impossible à obtenir sur le site destinataire. Nous nous préoccuperons donc d'obtenir un environnement aussi exact que possible et qui garantisse l'accès transparent aux ressources.

1.3.6 La transparence

La migration d'un processus peut se faire de façon transparente au processus. Elle est alors opérée par le propriétaire du processus, un adminisateur système (par exemple pour arrêter un site) ou encore par un serveur tel qu'un gestionnaire de charge. Dans ce cas le processus concerné n'est pas informé de son déplacement. Son exécution doit se dérouler sans changement.

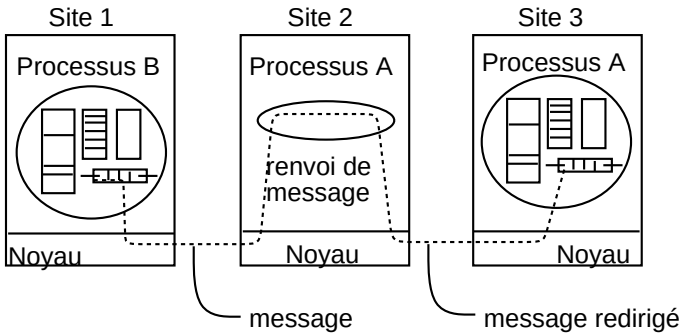

Figure IV.2 : Les dépendances résiduelles dans DEMOS/MP

Pour des raisons d'optimisation ou d'accès non-transparent aux ressources, il peut être intéressant de prévenir le processus du fait qu'il va être déplacé. Ainsi le processus est en mesure de préparer son déplacement. C'est à dire de modifier ses accès aux ressources qu'il ne pourra accéder de façon transparente ou de libérer les données dont il n'a plus besoin, pour faciliter la migration. En prévenant le processus nous lui donnons également l'occasion de participer à sa migration et au choix du site destinataire.

1.3.7 Les dépendances résiduelles

Les dépendances résiduelles peuvent être nécessaires pour permettre la transparence du service de migration. Par exemple, l'envoi de message est souvent dépendant du site sur lequel se trouve le récepteur. Il est alors nécessaire de laisser une trace sur le site origine pour rediriger les messages sur le nouveau site du processus. Cette technique est utilisée dans le système Demos/MP [PM83] comme le montre la figure IV.2. Si le système est capable de retrouver dynamiquement le processus migré, sans l'aide de cette trace, celle-ci peut être conservée dans un but d'optimisation. Elle n'est alors plus considérée comme une dépendance critique.

Les dépendances résiduelles posent des problèmes de tolérance aux pannes et de complexité pour les migrations multiples. En effet, si le site origine est arrêté, les données appartenant au processus migré sont perdues. Le processus ne peut plus alors s'exécuter convenablement. D'autre part, après plusieurs migrations, le processus aura disséminé des données sur plusieurs sites. Ce qui implique que son existence sera fragilisée face à l'arrêt d'un site et que la complexité et le temps d'accès aux données seront augmentés.

1.3.8 Les performances

Il est souvent reproché à la migration de processus d'être lente et complexe par rapport à des techniques telles que l'exécution à distance [JV88b]. Une implantation performante doit alors permettre d'étendre l'utilisation de la migration de processus. Notons tout de même que, comme

nous l'avons vu au chapitre IV en 3.7, les critiques sont parfois abusives.

Il est difficile de mesurer exactement le temps pris par la migration d'un processus puisque celle-ci se déroule sur deux sites qui n'ont pas la même horloge. Par contre, il est possible de calculer le temps d'envoi d'un processus : la durée écoulée entre l'instant où le système reçoit l'ordre de migration et l'instant où il reçoit, du site destinataire, l'autorisation de détruire la copie locale.

Les implications exactes d'une migration sont, elles aussi, difficiles à mesurer. Par exemple, lorsqu'un processus migre vers un site moins chargé, il va s'exécuter plus vite mais il va aussi permettre aux processus de son site d'origine de s'exécuter plus vite. D'autre part, les temps d'accès aux ressources distantes sont changés, le trafic réseau engendré par le processus modifie également le comportement de l'ensemble du système, etc. Pour ces raisons, les gains, ou les pertes, exactes de la migration de processus sont généralement calculées statistiquement sur un ensemble de processus.

2 La migration de processus dans Chorus/MiX

Les problèmes évoqués précédemment nous permettent de spécifier un service de migration destiné aux multicalculateurs. Dans l'implantation nous définissons ensuite un service global de migration pour CHORUS/MiX qui supporte différentes utilisations. La réalisation décrit une première expérience de migration qui a été développée sur l'iPSC/2. Cette implantation étant destinée à supporter la répartition de charge nous en avons réduit la portée.

2.1 Spécification

Le service de migration que nous voulons mettre en place doit servir de base à l'implantation d'un service de répartition de charge et doit satisfaire les besoins de reconfiguration nécessaires à l'administration du système d'exploitation.

2.1.1 Le choix des processus

Il n'est pas souhaitable de migrer les processus dépendant d'un site. Par exemple, un processus chargé de gérer une ressource, telle qu'un périphérique, ne doit pas être migré, même pour des raisons de tolérance aux pannes. Ainsi nous limitons la migration aux processus s'exécutant dans l'espace d'adressage utilisateur, les processus de l'espace d'adressage système étant généralement dédiés à la gestion d'une ressource locale. De même, les processus ayant verrouillé leur mémoire sur un site sont supposés avoir besoin de ressources locales.

2.1.2 Activation de la migration

Il y a deux solutions permettant de geler proprement un processus. La première consiste à envoyer un signal au processus désigné. Un signal a pour effet d'obliger le processus à exécuter un traitement spécifique, correspondant au type du signal reçu. Dans notre cas, nous pouvons associer le traitement de gel du processus à un signal de migration. La seconde solution consiste à activer le

gel du processus depuis un processus qui n'est pas le processus migré. Par exemple, ce processus peut être celui qui donne l'ordre de migration.

La première solution laisse la possibilité au processus de préparer sa migration. Elle permet également de bénéficier de la transparence à la répartition offerte par la gestion des signaux dans CHORUS/MiX. Par contre, elle est plus difficile à mettre en oeuvre que la seconde et elle ne permet pas d'obtenir un compte rendu de l'exécution de la migration. Nous choisissons d'utiliser la seconde solution pour des raisons de simplicité.

2.1.3 Choix d'un algorithme de transfert

Le choix d'un algorithme de transfert est soumis à plusieurs contraintes :

– pour satisfaire différentes configurations de multicalculateurs, en particulier celles qui possèdent peu de disques, l'algorithme ne doit pas reposer sur l'utilisation systématique d'un disque.
– pour satisfaire les besoins de la reconfiguration dynamique [AHP91] l'algorithme doit tolérer les pannes donc ne pas avoir de dépendances résiduelles critiques.
– pour limiter l'occupation du réseau, nous privilégions un algorithme qui limite les communications.

Compte tenu de ces contraintes et des propriétés des algorithmes de transfert décrit en 1.3.3, le transfert simple des données semble la meilleure solution pour implanter notre migration.

2.1.4 L'interface

Les nécessités précédentes impliquent le dévelopement d'une interface d'accès au service de migration qui soit différente :

– pour la répartition de charge, le service doit pouvoir être appelé, soit au moyen d'un appel système, en précisant l'identificateur du processus à migrer, soit au moyen d'une commande directe (un message) échangé entre deux serveurs du système.
– pour les besoins de l'administration système, un ordre de migration doit pouvoir être donné interactivememt.

2.1.5 La transparence

Dans les deux cas, cités précédemment, le niveau de transparence doit être suffisant pour que le processus ne puisse pas, a priori, s'apercevoir de la migration : nous garantissons l'accès aux mêmes ressources sauf pour quelques cas particuliers tels que le temps d'accès à une ressource.

2.2 L'implantation dans CHORUS/MiX

Cette implantation veut répondre aux besoins de la migration tels qu'ils sont énoncés précédemment. Elle définit un cadre pour le développement d'un service de migration général. Pour permettre l'évolution de ce service nous supposons l'utilisation de fonctionnalités qui ne sont pas encore offertes par CHORUS.

2.2.1 L'état du processus

Dans le sous-système UNIX, un processus est principalement décrit par un objet *Proc*. Cet objet regroupe les informations propres au processus. Dans cet objet nous trouvons les informations nécessaires à l'identification du contexte mémoire : il contient les adresses des différents segments de mémoire. Le contexte système du processus est également obtenu à partir le l'objet *Proc*. Chacune des activités d'un processus possède son propre contexte du processeur. Ce contexte est décrit dans un objet *u_thread*. Le contexte de message est géré directement par le noyau : en supposant que ce dernier nous offre les services nécessaires à la migration de ce contexte, nous ne nous en occupons pas. L'état du processus est donc obtenu à partir de l'instance *Proc* associée au processus et des différentes instances des objets *u_thread* associées à ses activités.

Dans la figure IV.3, nous donnons les champs et sous-structures qui composent l'instance *Proc* d'un processus. Les données marquées ** sont les données qui doivent être restaurées sur le site destinataire. Etant donné le nombre de champs nécessaires à la migration du processus, il est plus rapide d'envoyer toute l'instance *Proc* en utilisant les primitives de communication de CHORUS, plutôt que de la reconstruire sur le site cible. De même, lorsque nous migrons les instances *u_thread* nous envoyons la totalité des données qu'elles contiennent.

Pour satisfaire aux besoins propres à la migration, nous complétons l'objet *Proc* par un objet de la classe `ProcMigrate`. Un objet de la classe `ProcMigrate` est composé de :

cntUThread : Le nombre d'activités du processus non-encore endormies,

migSite : L'identificateur du site destinataire.

Dès que la migration du processus commence, sa structure *Proc* est verrouillée pour ne pas autoriser de modification de l'état du processus.

2.2.2 Le gel du processus

Comme nous l'avons vu précédemment, un processus ne peut être migré à un instant quelconque de son exécution. Tant que le processus est en cours d'exécution d'un appel système nous ne pouvons être sûrs que son état est cohérent. Le système CHORUS/MiX ajoute un problème supplémentaire à la migration des processus : l'arrêt des processus multi-activité. Le processus doit attendre que toutes ses activités soient gelées pour commencer sa migration.

Pour arrêter de façon cohérente une activité en cours d'exécution le noyau offre la possibilité de l'avorter. Si l'activité exécute son propre code, elle est simplement arrêtée. Si l'activité exécute un appel système, le système garantit qu'elle est arrêtée en laissant les données du système dans un état cohérent. En particulier, tous les appels système non-bloquants sont menés à terme. Dans le sous-système UNIX, l'activité d'un processus peut être bloquée dans les appels systèmes suivants :

- réception d'un message (*u_ipcReceive*) ou d'une réponse à un appel synchrone (*u_ipcCall*),
- mise en attente (*u_threadDelay*) ou attente d'un verrou (*u_mutexGet*),
- ouverture d'un fichier (*open*), lecture (*read*) ou écriture (*write*) sur l'écran ou sur un tube, attente d'un événement (*pause* et *wait*) et contrôle des entrées/sorties (*ioctl*).

Il y a donc deux possibilités pour geler un processus dans un état cohérent :

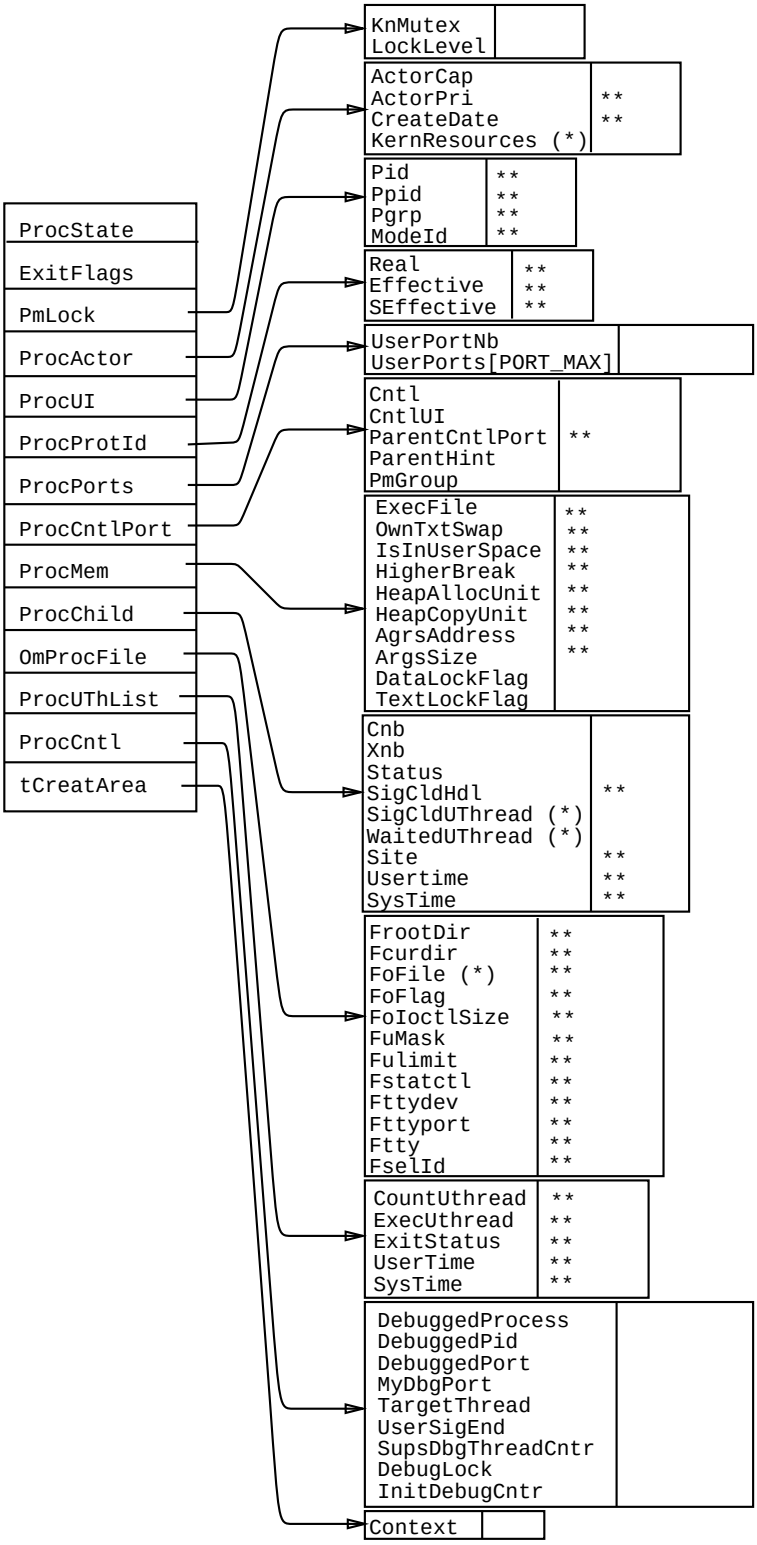

Figure IV.3 : La classe Proc

Avorter ses activités Ceci pose uniquement un problème à celles qui sont impliquées dans un appel système bloquant. En mémorisant le fait que l'activité est impliquée dans un appel système, il est alors possible de relancer l'appel sur le site cible. Cela pose les problèmes suivants :

- *open()* : rend une erreur lorsqu'il est avorté. Cet appel n'étant pas interruptible du point de vue du serveur de fichier, nous pouvons le relancer sur le site cible.
- *read()* et *write()* sur l'écran ou sur un tube ne donnent aucune garantie sur l'état de leur déroulement, s'ils sont avortés. En les relançant sur le site destinataire nous risquons de voir apparaître deux fois la même donnée ou d'avoir à la lire deux fois. Si l'utilisateur peut corriger de tels défauts sur l'écran - par exemple en entrant deux fois ses données en lecture - les implications sur les tubes sont difficilement maîtrisables. Nous avons donc besoin d'un support de la part des serveurs concernés.
- *pause()* et *wait()* peuvent être relancés sur le site cible.

Nous devons mémoriser l'état des activités (active, endormie, bloquée dans un appel, etc.) pour les remettre dans cet état à l'arrivée et éventuellement relancer l'appel avorté.

Positionner son état Nous enregistrons la demande de migration dans le champ qui décrit l'état du processus. Le processus est alors en attente de migration. Les activités testent l'état de leur processus à la sortie de tous les appels système, donc lorsqu'elles sont dans un état cohérent. Si le processus attend d'être migré, l'activité exécute la migration du processus, sur le site d'origine. Dans le cas multi-activité c'est la dernière activité qui teste l'état du processus qui exécute la migration. Si il reste des activités non endormies dans un processus en attente de migration, l'activité courante s'endort après avoir testé l'état du processus. Le compte des activités non endormies est tenu dans le champ `cntUThread` de l'objet `ProcMigrate`.

Dans le cas où les activités sont avortées, la migration a lieu immédiatement après l'ordre de migration. Par contre, dans le second cas, nous ne sommes pas sûrs que le processus sera migré rapidement puisqu'il faut attendre un appel système pour engager la migration. Ainsi cette seconde solution peut être mal adaptée pour certaines classes de processus. Par exemple, un processus de calcul ne fait pas beaucoup d'appels système, sa migration risque donc d'être retardée. Le cas critique est celui d'un processus dont l'une des activités ne fait aucun appel au système. D'autre part, cette solution ne permet pas de rendre un code d'erreur au processus ayant initialisé la migration puisque la migration est différée jusqu'à l'arrêt de toutes les activités.

Malgré ces considérations nous avons choisi d'utiliser la seconde solution car elle est plus rapide à implanter. L'objectif idéal serait de mettre en oeuvre une solution hybride où nous n'avortons que les activités impliquées dans des appels systèmes non bloquants et où l'activité teste l'état du processus après chaque appel système. Nous satisferions ainsi un éventail plus large de processus. L'interface entre le gestionnaire de processus (PM) et les serveurs UNIX pourrait également évoluer de manière à prendre en compte la migration, ce qui permettrait d'avorter les activités dans tous les cas.

2.2.3 Le transfert

Avant le transfert, nous vérifions que le processus satisfait bien les contraintes imposées par la migration. Le site origine doit également vérifier si les ressources nécessaires au processus sont disponibles sur le site destinataire. Il envoie donc une requête de migration au site destinataire pour s'en assurer. A la réception de ce message, le site destinataire réserve les ressources pour le processus et compose sa réponse en fonction du déroulement. Cette réponse sert donc à la fois d'acceptation de la migration et de compte rendu de la réservation. Le transfert du processus commence après réception de la réponse sur le site d'origine.

Lorsque le site destinataire est assuré d'avoir initialisé correctement le processus - il a reçu toutes les données et les ressources sont allouées - il envoie un message au site d'origine demandant la destruction du processus. Par contre, si un incident se produit dans le transfert, le site origine arrête la migration et réveille le processus gelé.

Le transfert du processus se déroule de la façon suivante :

Sur le site d'origine Lorsque un processus est en attente de migration, c'est la dernière activité, non endormie, qui exécute la migration. Cette activité regroupe les informations concernant le processus : structure *Proc*, liste des activités, liste des portes, etc. Elle les encapsule dans des messages et les envoie au site destinataire d'après l'algorithme suivant :

```
envoie(demande migration + Proc + activité)
attend la réponse
si (ressources réservées)
   alors envoie(autres activités)
         envoie(code)
         envoie(données)
         migre(portes)
         attend(fin de migration)
si (migration ok)
   alors détruit le processus
   sinon réveil le processus
```

Tous les échanges de messages entre les deux sites sont synchrones et reposent sur la fiabilité du RPC. Le délai d'attente en réception est borné sur les deux sites. Nous évitons ainsi une attente infinie en cas de panne d'un site. Si la réponse ne parvient pas au site d'origine dans le temps d'attente, le processus est réveillé localement.

La structure *Proc* est envoyée en premier car elle contient les données qui servent à réserver les ressources nécessaires au processus. La structure décrivant l'activité exécutant la migration est envoyée dans le même message pour optimiser le cas des processus mono-activité. Ces structures sont envoyées en même temps que la requête de migration pour des raisons d'optimisation.

La pile du processus est inclue dans ses données, il n'y a donc pas lieu de l'envoyer explicitement. Le transfert de la mémoire est réalisé en itérant sur la taille de celle-ci puisque notre service

de communication ne permet pas l'envoi de message dont la taille excède une page de mémoire virtuelle. L'option utilisée pour l'envoi des messages est la recopie car nous devons garder une copie locale du processus pour pouvoir le réveiller si la migration se déroule mal.

D'autres parties de code, telles que les bibliothèques partagées ne sont pas envoyées car elles sont automatiquement chargées sur tous les sites, pour être utilisées par les processus.

Sur le site destinataire Sur le site destinataire c'est une activité du PM, dédiée au traitement des requêtes, qui exécute la réception puis le réveil du processus. La demande de migration est reçue, par cette activité, sur la porte dédiée aux requêtes (figure IV.4). L'algorithme suivant est utilisé :

```
attend ( message )
Si ( message = migration )
  alors
        allouer ( Proc ),
                l'initialiser à partir du message
        allouer ( acteur )
                ( activités ), mettre à jour Proc
        allouer ( région code , initialisée avec segment)
        allouer ( région données )
        créer ( porte)
        envoyer ( compte rendu, porte )

        recevoir ( activités, code, données)
        positionner les signaux
        envoyer ( fin de migration )
        attend ( réponse )
        si ( réponse positive )
          réveiller le processus
Sinon
        autres traitements
```

Une fois les ressources allouées une porte de contrôle est créée pour le processus. Pour éviter que les échanges avec le site d'origine n'interfèrent avec d'autres requêtes adressées au PM, les messages suivants sont reçus sur la porte de contrôle du processus. L'identificateur de la porte est donc ajouté au message qui rend compte de l'allocation.

Les activités sont créées dans l'état suspendu. Elles ne s'exécutent pas tant que la totalité du contexte du processus n'est pas reçu. Lorsque le processus est complet les activités sont lancées.

Pour optimiser le transfert du code du processus, nous utilisons la fonction `rgnMap` qui permet d'initialiser une région de mémoire virtuelle à partir d'un segment. Cette optimisation n'est pas en contradiction avec les choix et contraintes énoncées en 2.1.3. En effet, un processus UNIX dépend toujours d'un disque pour la pagination à la demande. Nous initialisons ainsi le processus à partir du disque dont il dépend plutôt qu'à partir de son site d'origine. Si une copie du code se

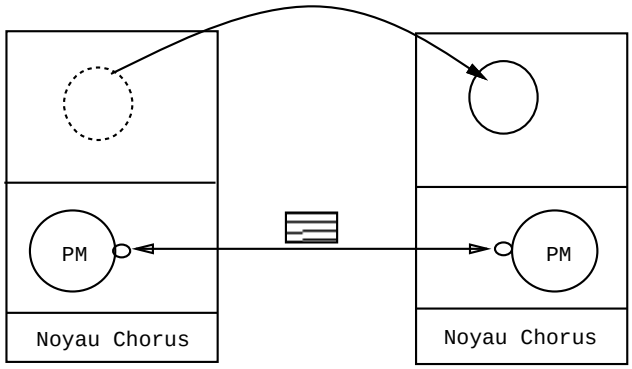

Figure IV.4 : Migration d'un processus

trouve déjà sur le site, celle-ci n'est pas rechargée depuis le disque, réduisant ainsi le temps du transfert.

D'autre part, puisqu'il s'agit de code, la représentation mémoire ne peut être modifiée. Le site d'origine n'a donc pas besoin de décharger sa mémoire sur le disque et nous n'augmentons pas le coût du transfert. Enfin, si le site d'origine n'avait pas chargé la totalité du code, cela évite de charger des pages de code depuis le site du disque sur le site origine avant de les envoyer au site destinataire. Pour ce qui est de la tolérance aux pannes, nous n'augmentons pas ainsi les dépendances résiduelles ni le degré de dépendances du processus vis à vis d'un autre site, nous respectons donc les contraintes que nous nous sommes fixées.

2.2.4 Le réveil du processus

Avant de réveiller le processus nous devons initialiser certaines variables de la structure *Proc*. Par exemple, la capacité de l'acteur accueillant le processus a changé. Nous devons ensuite correctement repositionner son contexte de signal. Une fois le processus prêt, il est réveillé. Il reprend alors son exécution à l'instruction suivant l'appel système qui a précédé la migration.

2.2.5 La transparence

Dans CHORUS/MiX la transparence d'accès aux serveurs est garantie par l'utilisation de la communication dans les échanges entre serveurs. Ainsi aucun traitement n'est nécessaire pour garantir aux processus la transparence d'accès aux ressources telles que les fichiers ou les périphériques. De même un processus qui accèdait un serveur directement, par envoi de messages, ne modifie pas ses requêtes après la migration.

Le problème de la transparence d'accès aux portes du processus sera résolu en migrant ses portes en même temps que celui-ci. La migration de portes distantes n'étant pas encore implantée dans

CHORUS, nous devons imposer aux processus éligibles pour la migration de ne pas posséder de portes. L'ajout de cette fonctionnalité est prévu dans l'algorithme de migration des processus : pour migrer une porte il nous suffit de connaître la capacité de l'acteur destinataire de la porte : le site destinataire pourra retourner cette capacité en même temps que l'accord de migration.

Notons que la migration de portes entre sites est un service qui sera offert par une prochaine version du noyau. Elle n'a donc pas été implantée dans ce travail.

2.3 Réalisation sur l'iPSC/2

L'implantation réalisée sur l'iPSC/2 est une restriction des spécifications. En particulier nous avons défini un sous ensemble de processus qui sont migrables facilement. Ce sous ensemble doit servir de base à une première expérimentation de gestion de charge. Pour valider plus largement les premières expériences, nous devrons étendre les possibilités de migration de notre service.

2.3.1 Restrictions

En plus de la restriction que nous avons posée en 2.1.1, nous imposons des restrictions destinées à limiter les développements. La plupart de ces limitations sont engendrées par l'absence du service de migration de porte entre deux sites. Nous expliquons les principales raisons de chaque limitation :

les processus debogués : le contrôle d'un processus en cours de déboguage est réalisé à travers une porte (`MyDbgPort`). Puisque cette porte ne peut pas être migrée, nous ne migrons pas les processus débogués.

les processus multi-activités : pour simplifier le transfert, nous ne migrons que des processus mono-activités. Ceci nous permet d'envoyer tous les contextes système et processeur du processus dans le premier message.

les régions mémoire : les régions mémoire étant gérées par le noyau, le PM ne dispose pas de la description des régions créées par le processus. Ceci peut facilement être obtenu en mémorisant ces régions dans le contexte *Proc*.

les portes : La migration de porte n'est pas implantée entre des processus résidant sur des sites différents. Pour garantir l'exactitude de la migration, nous ne migrons donc pas de processus ayant des portes.

les fils : Lorsqu'un processus fils finit son exécution il prévient son père en envoyant un message sur sa porte de contrôle. Si le processus père migre, l'identificateur de la porte de contrôle possédé par le fils n'est plus valide. Lorsque le noyau supportera la migration distante, la porte de contrôle pourra être migrée plutôt que d'être re-créée à l'arrivée.

Néanmoins l'extension des fonctionnalités du service de migration a été prévue. Le code est donc structuré de manière à pouvoir lever ces restrictions sans modification importante.

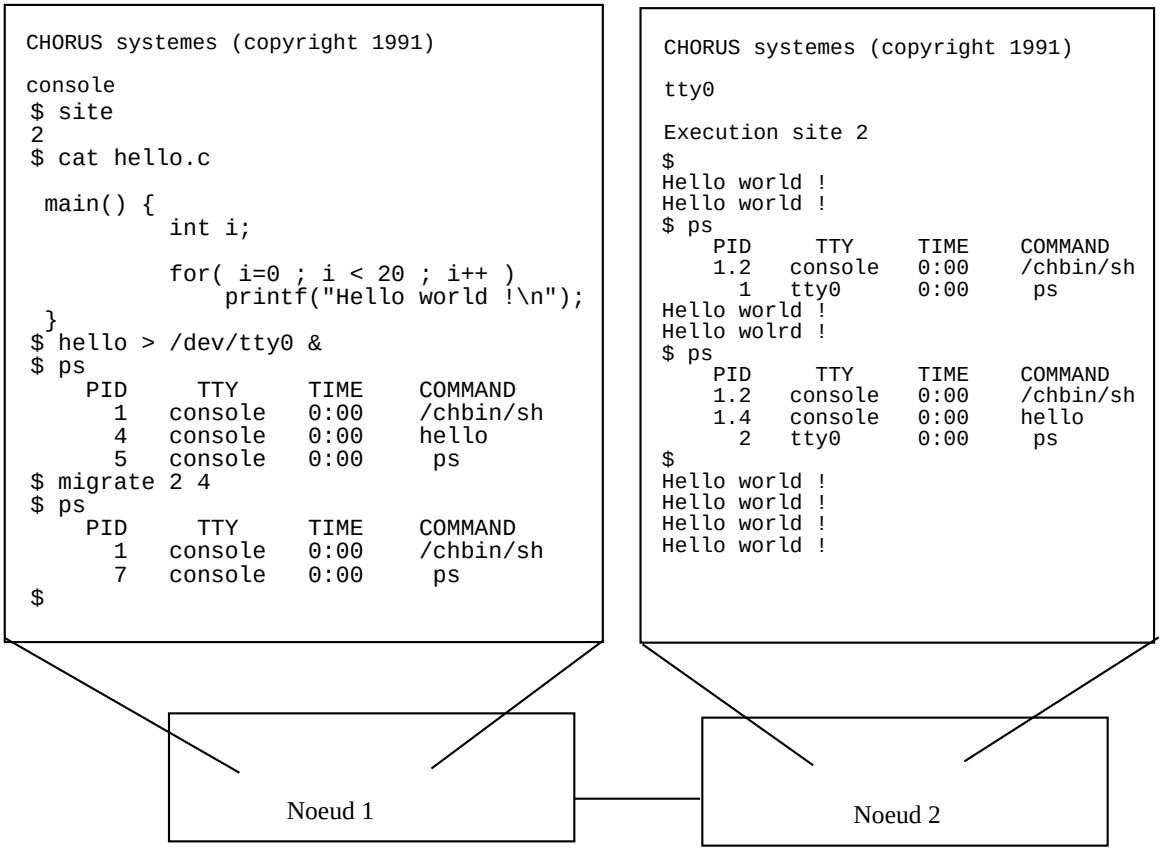

Figure IV.5 : Exemple de migration

2.3.2 L'interface

Une migration de processus peut être invoquée seulement sur un processus local.
– depuis le shell au moyen de la commande : `migrate` *pid site*,
– dans un processus au moyen de l'appel système : `migrate (` *pid*, *site* `)`,
– depuis un serveur système en envoyant un message au PM.
Dans les deux premiers cas, nous utilisons un appel système qui est exécuté soit par une commande du shell, soit par un processus. Dans le dernier cas, c'est une activité du PM qui reçoit le message de migration. Le processus est d'abord testé pour vérifier s'il est migrable et son état est positionné à l'attente d'une migration. Un code d'erreur est alors retourné à l'initiateur de la migration.

2.3.3 L'utilisation

La figure IV.5 montre le déplacement d'un processus du site 1 au site 2. La même migration peut être répétée plusieurs fois.

Conclusion

Nous avons décrit, dans ce chapitre, la conception et la réalisation d'un service de migration de processus. Les contraintes propres aux multicalculateurs nous ont conduits à concevoir un service très simple mais efficace qui n'engendre pas de dépendances résiduelles, contrairement à la plupart des algorithmes décrits dans la littérature qui cherchent à optimiser le transfert.

Pour satisfaire les besoins d'autres applications de la migration de processus ce service pourrait être complété par quelques fonctionnalités supplémentaires.

- L'extension des formats standards d'exécutables (COFF, ELF) doit permettre aux programmeurs de décrire les besoins du processus. Par exemple, le programmeur peut préciser sur quel site se trouve la principale ressource accédée par le processus. En utilisant ces données le choix du placement du processus sera plus facile. Le programmeur peut également préciser un temps d'exécution supposé ce qui éviterait de migrer les processus à la fin de leur vie.
- La migration d'un groupe de processus doit permettre de déplacer simultanément plusieurs processus fortement liés. Par exemple, si ces processus communiquent beaucoup nous gagnerons en temps de communication, entre les processus, mais aussi en intensité de trafic sur le réseau.
- Il peut être intéressant de migrer, en même temps que le processus, certaines ressources logicielles particulièrement liées au processus. Par exemple, le déplacement de deux processus communiquants par *tube* sur un même site peut également entraîner le déplacement du *tube*.
- Un processus doit être capable de traiter un ordre de migration qui le concerne s'il n'est pas le responsable de cet ordre. Il doit alors pouvoir empêcher la migration ou exécuter des traitements spécifiques avant et/ou après la migration. Ceci permet d'éviter de déplacer des processus fortement liés à un site, ou d'engager un traitement qui facilite la migration (relâchement de ressources inutiles). Cette possibilité pourra être donnée à travers une activation pour signal.

Néanmoins cette réalisation est suffisante pour servir de base à l'implantation d'un gestionnaire de charge, telle que nous la décrivons au chapitre suivant.

Chapitre 5

La gestion de charge répartie

Introduction

Traditionnellement un des défis de l'informatique consiste à tirer le meilleur parti du matériel pour offrir plus de possibilités et de confort à l'utilisateur. Ceci est particulièrement vrai au plus bas niveau pour les systèmes d'exploitation puisqu'ils consituent la première couche d'interface entre la machine et l'utilisateur. Dans ce cas, tirer le meilleur parti du matériel signifie : s'assurer qu'il n'existe pas une ressource inoccupée alors qu'un utilisateur est en attente de cette ressource. Le système d'exploitation doit alors gérer les ressources disponibles de façon optimale afin de garantir à l'utilisateur un plus grand confort et de meilleures performances d'exécution. Les ressources gérées par le système d'exploitation sont : la mémoire, le processeur, les éléments de mémorisation secondaires (disques, bandes, etc.), les autres périphériques ou encore les accès aux connexions vers d'autres systèmes. Sur les architectures traditionnelles - monoprocesseur - l'allocation de ces ressources est bien maîtrisée par les développeurs de système d'exploitation. Nous pouvons citer des techniques telles que la pagination et la mémoire virtuelle pour gérer l'allocation de la mémoire, les ordonnancements en temps partagé ou en temps réel qui permettent de minimiser le temps d'inoccupation du processeur, le découpage des disques en blocs pour réduire les pertes de place, etc.

L'apparition de nouvelles architectures - réseaux et multiprocesseurs - engendre de nouvelles classes de problèmes dans l'allocation des ressources, du fait du nombre et de la répartition de ces ressources. Les solutions proposées différencient les multiprocesseurs à mémoire partagée, où une seule instance du système d'exploitation gère la totalité des ressources, des architectures réparties (réseaux et multicalculateurs), où plusieurs instances de système d'exploitation coopèrent pour permettre l'accès aux ressources. Dans le cas de partage de mémoire, le système dispose d'une image exacte de l'état des ressources, ce qui permet de mettre en place, à moindre frais, des politiques efficaces et adaptées aux besoins de l'utilisateur. Parmi celles-ci nous pouvons citer les politiques permettant un ordonnancement par groupe [Bla90]. Par contre le système réparti ne connait de façon sûre que l'état des ressources locales à un site. Dans ce chapitre, nous nous intéressons au cas des architectures réparties qui nécessitent une instance du système pour chaque

ensemble de ressources fortement couplées.

La complexité des problèmes d'allocation des ressources, dans le cas réparti, est dépendante du niveau de fonctionnalité et de confort que le système d'exploitation veut offrir à l'utilisateur. Ainsi les premiers réseaux d'ordinateurs ont été conçus comme des ensembles de machines totalement disjointes, chacune gérant ses propres ressources de manière centralisée. Dans ce cas, le système d'exploitation est un système centralisé traditionnel auquel sont ajoutées quelques fonctionnalités telles que TCP/IP [Zim80]. L'utilisateur doit gérer la distribution et l'accès à ses ressources. Il peut posséder un fichier sur un site distant mais c'est à lui de le placer explicitement sur ce site et de gérer l'accès explicite à ce site. Par la suite sont apparus des systèmes intégrant une gestion plus globale et plus transparente (par exemple : NFS, exécution distante, etc.) pour faciliter à l'utilisateur la prise en compte de la répartition de ces ressources. L'évolution actuelle tend à rendre transparent, à la répartition des ressources, l'accès à tous les services offerts par un système d'exploitation.

Notre but est le développement d'un système à image unique pour les architectures réparties. C'est-à-dire la mise en place d'une politique globale et transparente de gestion de toutes les ressources. L'élaboration d'un tel système requiert la résolution de nombreux problèmes tels que la connaissance de l'état des ressources sur les autres instances du système, la mise en place d'une politique d'allocation commune à toutes ces instances, le nommage des ressources, etc.

Un tel sytème peut être utilisé sur un réseau de stations de travail pour faciliter la programmation multisite. Mais il peut être mal accepté puisqu'il partage les ressources entre tous les utilisateurs et ne respecte pas la notion de possession de base que représente une station de travail. Par contre il se justifie pleinement dans le cas des multicalculateurs, dans la mesure où nous ne pouvons considérer chacun des noeuds comme une entité indépendante. Sur une machine disposant d'un grand nombre de noeuds il devient nécessaire car les problèmes d'allocation se complexifient.

Nous décrivons ici les problèmes posés par une politique d'ordonnancement global sur un multicalculateur dans le cadre d'un système à image unique. Dans la première partie nous classons les différents modes d'ordonnancement dans un contexte réparti. En deux, nous présentons la gestion globale de la charge et les problèmes posés par son implantation sur multicalculateur. Nous décrivons en trois les différentes solutions proposées dans la littérature pour résoudre ces problèmes. Dans la quatrième partie nous présentons une mise en oeuvre d'un serveur de charge permettant la répartition de processus UNIX sur un multicalculateur en utilisant les services du noyau CHORUS.

1 Les algorithmes d'ordonnancement global

Plusieurs types d'ordonnancement global sont utilisés dans un système réparti. Comme dans tout domaine récent, nous rencontrons dans la littérature une grande variété de vocabulaire. Nous précisons ici le sens attribué par la suite aux termes concernant l'ordonnancement réparti.

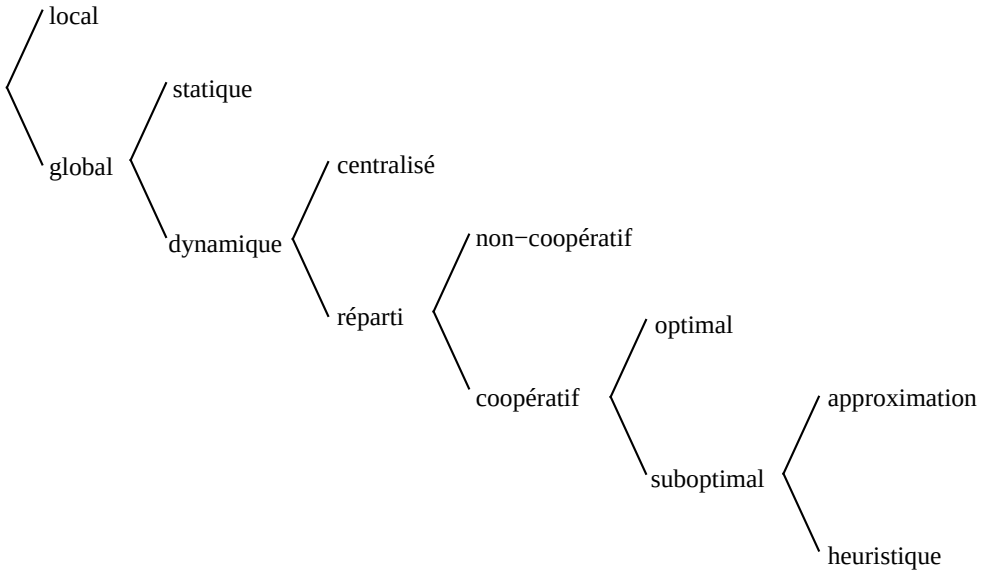

Figure V.1 : Classification des stratégies d'ordonnancement

1.1 Terminologie

Nous employons le terme **ordonnancement** pour désigner en général toute décision d'exécution d'un processus sur un site. Un **ordonnancement réparti** ou **global** sera donc une décision d'exécution d'un processus sur un site parmi un ensemble de sites. Pour les algorithmes d'ordonnancement réparti statiques - qui définissent l'attribution des processus aux processeurs pendant la compilation - nous parlerons de **placement** de processus sur le réseau. Nous avons volontairement employé le terme **gestion de charge** au cours de cette partie pour désigner d'une manière générale tous les algorithmes dynamiques, c'est-à-dire qui définissent l'attribution des processus aux processeurs au plus tôt au moment d'exécuter ceux-ci.

Un processus peut être exécuté sur un site distant au moment de son activation - nous parlons alors d'**exécution distante** - ou en cours d'exécution - et nous parlons de **migration**. Parmi les algorithmes de gestion de charge, nous différencions donc les algorithmes de **répartition de charge** qui attribuent un processus une seule fois en utilisant l'exécution distante et les algorithmes d'**équilibrage de charge** qui permettent de re-attribuer un processus en le migrant. Une description et un classement d'algorithmes d'ordonnancement ont été donnés dans [BSS91a].

1.2 Classification

Une classification des algorithmes d'ordonnancement a été proposée dans [CH88a], nous la reproduisons dans la figure V.1. Les algorithmes d'ordonnancement global peuvent alors être classés suivant l'arborescence proposée sous le terme `global`. Nous utilisons cette classification pour étudier nos besoins en terme d'ordonnancement sur un multicalculateur.

1.2.1 Statique ou dynamique

La première distinction que nous pouvons faire parmi les algorithmes d'ordonnancement global est entre les stratégies **statiques** et les stratégies **dynamiques**. C'est-à-dire entre le placement et la gestion de charge. Contrairement aux stratégies dynamiques, les stratégies statiques [Lo84] ne considèrent aucune information sur l'état du système pendant l'exécution de l'application : le placement des processus composant l'application est fait avant sa compilation, soit par le programmeur, soit par un paralléliseur. Le placement a l'avantage de permettre l'exécution des applications d'après une répartition optimale, lorsque celle-ci peut être calculée statiquement. Le code de l'application est alors souvent dépendant des choix de placement et de l'architecture. Le placement ne permet pas de prendre en compte des informations sur l'évolution du système en temps d'exécution. Il est mal adapté aux machines multi-utilisateurs puisque les autres utilisateurs introduisent des inconnues dans la charge des sites alors qu'il nécessite que tous les paramètres soient contrôlés. Ceci est particulièrement vrai pour la charge du processeur mais peut l'être également pour la place mémoire occupée, l'accès à des entrées/sorties, etc. Avec cette méthode il est possible que deux utilisateurs décident de placer leurs processus les plus longs sur le même site, il y aura donc un temps pendant lequel les deux processus devront se partager le temps d'exécution alors que d'autres sites sont libres.

Les stratégies dynamiques ont l'avantage de permettre d'adapter la répartition de la charge en cours d'exécution. De plus, une stratégie dynamique peut-être **adaptative**, c'est-à-dire qu'elle peut modifier ses critères de décision en fonction des conclusions tirées de l'analyse de la charge sur le réseau. Les stratégies dynamiques ont le désavantage de perdre du temps pour acquérir les données significatives sur l'état du système et pour élaborer leurs décisions. Comme tout confort supplémentaire la gestion dynamique de charge amène un surcoût.

Pour les applications très coûteuses en temps d'exécution il doit être possible de combiner les deux méthodes. Pour la mise au point, une stratégie dynamique est suffisante en terme de performances. Dans un second temps un outil d'aide à la distribution peut être utile pour déterminer un placement optimal des processus. Enfin des possibilités d'ordonnancement adaptatives (réservation d'un certain nombre de sites) doivent permettre l'exécution de manière optimale. Pour toutes les autres applications, l'utilisateur se contentera probablement d'écrire une application parallèle sans s'occuper du placement. Dans ce cas une gestion dynamique de la charge est préférable. Notons également que la programmation d'un grand nombre de sites est complexe Dans ce cas, une gestion globale de la charge peut simplifier la programmation parallèle.

1.2.2 Centralisée ou répartie

Cette distinction fait allusion à la répartition du centre de décision pour l'ordonnancement des processus. Elle ne peut exister que pour une stratégie dynamique puisque dans le cas statique il n'existe pas d'unité de décision de l'ordonnancement global.

Au moment de l'exécution, l'unité de décision peut être centralisée sur un des sites du réseau et commander l'ordonnancement des tâches pour les autres sites. Il existe dans ce cas un site maître parmi l'ensemble des sites. L'approche centralisée offre probablement un meilleur rendement en

terme d'utilisation des sites dans la mesure où elle permet une gestion optimum de la charge : l'unité de décision connait la charge de chacun des sites et peut ainsi décider d'une affectation optimale pour l'exécution d'un processus. Notons également que cette méthode permet automatiquement une connaissance exacte de l'état des autres sites dans la mesure où cet état est déterminé par le site maître. De plus les échanges de messages sont moins importants que dans le cas réparti car chaque site ne correspond qu'avec le site maître. Bien sûr les inconvénients majeurs de cette solution sont les mêmes que pour toute application centralisée : dès que le nombre de sites à gérer augmente, l'unité de décision devient rapidement un goulet d'étranglement et son arrêt entraîne des perturbations sur l'ensemble du réseau. La solution centralisée peut être acceptable pour un petit nombre de processeurs, de l'ordre de la dizaine.

Si nous répartissons l'unité de décision de l'ordonnancement, nous évitons le problème du goulet d'étranglement puisque chaque décision se fait sur le site concerné (ou entre les sites concernés). Il est évident que pour gérer la charge de réseaux connectant plusieurs centaines de processeurs la solution distribuée est mieux adaptée. La répartition des unités de décision permet aussi d'acquérir des propriétés de tolérance aux pannes puisque l'arrêt d'une de ces unités ne stoppe pas, a priori, le travail des autres.

Dans [TL88] deux implantations de gestion de la charge sont réalisées au dessus du système V. L'article compare une implantation centralisée à une implantation répartie. En utilisant largement la diffusion de la charge, service qui dépend des caractéristiques du réseau, et les services de groupes de processus du système, les auteurs obtiennent de meilleures performances avec l'algorithme centralisé. Paradoxalement, l'implantation centralisé d'un ordonnancement réparti est, dans ce cas, plus extensible qu'une implantation décentralisée. Cette propriété provient de l'utilisation systématique de la diffusion pour la mise en place de l'algorithme d'ordonnancement, comme le montre [ZF87]. L'implantation est donc dépendante du support de communication. Un algorithme distribué est plus simple à mettre en oeuvre dans cet environnement et supporte mieux les pannes.

Un algorithme, à la fois centralisé et réparti, permettant de tolérer les pannes est donné dans [BS91] : une liste des sites sous-chargés est formée à travers le réseau, la tête de cette liste est connue de deux sites : le *manager* et la tête de liste. Le site *manager* distribue alors la charge sur les sites sous-chargés de la liste. Si le *manager* s'arrête, la tête de liste prend sa fonction et en notifie la nouvelle tête de liste. Le faible taux de centralisation doit permettre d'offrir les qualités d'extensibilité nécessaires tout en maintenant un bon niveau de simplicité. De plus la tolérance aux pannes est prise en compte pour éviter une trop forte dépendance vis-à-vis du site *manager*. Par contre, le problème du goulet d'étranglement n'est pas résolu.

Une autre solution hybride consiste à ne placer des unités de décision que sur certains sites d'un réseau. Nous réduisons ainsi les risques d'étranglement puisque le nombre de sites gérés est petit. Cette solution ne réduit cependant pas les problèmes de tolérance aux pannes. Son implantation ne permet pas la simplicité d'un algorithme distribué et implique le développement de protocoles de coopération entre les serveurs, ce qui augmentent à la complexité.

Une implantation centralisée peut être également bien adaptée à une architecture particulière. Par exemple, l'implantation décrite dans [BFS89] utilise des 3B4000 commercialisés par AT&T, constitués d'éléments de calcul indépendants sauf pour la gestion de la charge qui est assumée

par un processeur maître. Le système décrit dans [Tho87] a des caractéristiques équivalentes. La gestion de charge a cependant, dans ce cas, une forte dépendance vis-à-vis de l'architecture et ne peut être utilisée dans un cadre plus général.

1.2.3 Coopératif ou non coopératif

Dans le cas d'unités physiquement réparties sur un réseau celles-ci peuvent coopérer ou non pour mettre en place une politique globale de gestion de la charge. Dans le cas où les unités ne coopèrent pas, le gain en échange de messages est évident puisqu'aucune information n'est échangée entre les sites. La décision d'exécution d'un processus est locale et le choix d'affectation est aléatoire, dans la mesure où le site ne possède aucune information sur l'autre site mis en cause par l'exécution. La coopération peut être effectuée à différents niveaux entre les sites du réseau. Nous dirons qu'un algorithme de gestion de charge est non coopératif s'il ne s'occupe que de son état local pour déterminer si un processus doit être exécuté sur un autre site. Dans ce cas, chaque site agit comme une unité indépendante sans se soucier de tout ou partie de l'état global du système. L'algorithme est coopératif dans le cas contraire, c'est-à-dire, lorsqu'un site base sa décision d'exécuter un processus à distance sur des données de charge ou de disponibilité de ressources d'au-moins un autre site. La coopération mise en place avec les autres sites peut être de voisinage avec un ensemble de sites proches, aléatoire avec une partie quelconque des sites du système, ou encore globale si la totalité des sites coopèrent pour mettre en oeuvre leur politique de gestion de charge.

Dans [ELZ86], les auteurs comparent les performances de deux politiques coopératives - l'une d'elles est très simple - et d'une politique non coopérative. La figure V.2 reproduit les résultats obtenus par simulation du comportement de ces trois algorithmes. D'après cette figure, la politique non-coopérative, même si elle permet un gain par rapport à une exécution sans gestion de charge, se montre moins adaptée à un système chargé. Elle se comporte bien tant que le système est peu chargé puisque le site destinataire, choisi aléatoirement, est probablement peu chargé ; il donc peut accepter une charge supplémentaire.

1.2.4 Affectation optimale ou suboptimale

Une affectation optimale ne peut être obtenue que si l'on connaît toutes les informations caractérisant le système. Le choix de l'affectation d'un processus se fait alors, en fonction de ces informations, pondérées par des critères d'importance, dans le but de minimiser le temps d'exécution et optimiser l'utilisation des ressources du système. Le processus tire ainsi le meilleur parti du système. Dans [CA82] un placement optimal de tâches est obtenu en utilisant un algorithme basé sur les résultats de la théorie de décision de Markov.

L'affectation optimale d'un processus pose cependant le problème de l'acquisition des données utilisées et de leur traitement pour obtenir un choix. Le système étant réparti, une unité de décision ne peut connaître l'évolution des informations caractérisant les sites distants : elle doit donc les demander. Cette acquisition pose les problèmes suivants :

– son coût croît avec le nombre de sites concernés. Pour maintenir sur chaque site des informa-

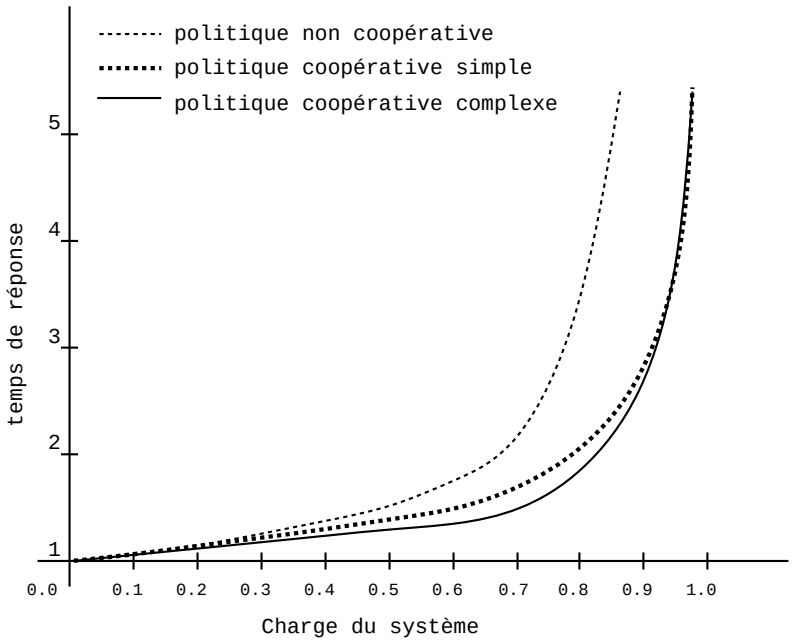

Figure V.2 : Comparaison de trois gestions de charge

tions sur l'ensemble du système, le nombre de messages échangés est proportionnel au carré du nombre de sites. Le réseau risque d'être fortement encombré du fait de ces échanges d'information. Nous pénalisons ainsi l'exécution des applications.

– la validité des informations n'est pas garantie puisque les autres sites restent libres d'évoluer indépendamment. Il est donc nécessaire de mettre en place un protocole pour obtenir des informations exactes : chaque modification de l'état d'un site doit être transmis aux autres. La lourdeur de ces protocoles rend la connaissance exacte de l'état de tous les sites quasiment impossible à obtenir et sûrement trop coûteuse pour un grand nombre d'applications.

Le traitement de ces informations induit également un coût proportionnel au nombre de sites. En plus de ces problèmes de coût, l'affectation optimale pose un problème de fonctionnement : il n'est pas sûr que l'exécution d'un processus dans des conditions optimales permette l'exécution optimale d'un groupe de processus.

La figure V.2 compare deux politiques coopératives de complexité différente. Nous voyons que la politique la plus complexe permet un léger gain de performance par rapport à la politique simple. D'après les auteurs [ELZ86], ce gain de performance est minime par rapport à la difficulté d'implantation.

Pour pallier à ces problèmes les unités de gestion de charge utilisent des méthodes d'affectation suboptimales : seules certaines informations, choisies pour leur représentativité, sont utilisées dans le choix d'une affectation. Par exemple, une unité peut baser ses décisions sur les valeurs

caractérisant la charge des sites voisins plutôt que sur celles qui caractérisent la totalité des sites.

1.2.5 Approximation ou heuristique

Le cas suboptimal peut encore être divisé en deux catégories suivant le choix que l'on fait sur les critères de charge d'un site. Nous dirons d'un algorithme suboptimal qu'il utilise une approximation pour faire ses choix si les décisions concernant la gestion globale de charge s'appuient uniquement sur les données possédées par le site. Dans ce cas, nous nous contenterons d'une fonction générant une bonne solution par rapport à une solution optimale. L'intérêt de la fonction peut être vérifié à partir du temps pris pour évaluer une solution et la capacité de la fonction à fournir une solution proche de l'optimal. Nous essayons alors de générer des solutions qui sont capables de minimiser le premier critère et de maximiser le second.

Nous dirons qu'un algorithme utilise des heuristiques s'il pose des "a priori" quant à l'exécution des applications ou l'évolution du système. Par exemple, une unité de décision peut utiliser, pour ses choix, des informations qu'elle acquiert cycliquement, en supposant que l'évolution de ces valeurs est suffisamment lente pour que les dernières valeurs reçues soient significatives. Les suppositions doivent être faites prudemment car elles peuvent s'opposer à une utilisation générale de l'algorithme d'ordonnancement si elles ne sont pas vérifiées dans tous les cas.

Pour développer une gestion de charge sur un multicalculateur, dans le but de l'intégrer à un système à image unique, nous nous intéressons plus particulièrement aux algorithmes d'ordonnancement globaux de cette dernière classe. Le choix de la dynamicité est justifié par notre objectif d'étendre l'utilisation de ce type de machines à un ensemble plus grand d'utilisateurs et surtout par l'aspect multi-utilisateurs qui en découle. La nature même des machines auxquelles nous nous intéressons impose un algorithme réparti, pour permettre à notre implantation d'être valable sur d'autres machines que celle utilisée pour le développement de cette fonctionnalité et pour répondre aux besoins de tolérance aux pannes imposés par le système à image unique. Comme nous l'avons vu en 1.2.3, les performances des algorithmes coopératifs justifient la prise en compte d'informations d'autres sites. Par contre, l'implantation d'un algorithme optimal est trop chère. Nous utiliserons également des heuristiques pour faciliter l'utilisation des informations issues d'autres sites.

2 Présentation de la gestion de charge

2.1 Formalisation d'une présentation intuitive

Il est généralement admis que les processeurs de stations de travail sont faiblement occupés. Par contre lorsqu'il s'agit d'exécuter une application importante le processeur est rapidement saturé. La même constatation peut être faite pour les multiprocesseurs à mémoire distribuée : si le placement d'une application parallèle n'est pas optimisé certains noeuds peuvent être surchargés alors que d'autres sont inoccupés ou peu chargés. Il est donc tentant d'utiliser les processeurs libres, ou peu chargés, pour exécuter la surcharge des processeurs saturés. Ceci peut être réalisé directement par l'utilisateur s'il dispose des outils nécessaires pour trouver le site le moins chargé

et exécuter à distance ses processus. Cette solution permet d'obtenir des gains de performances mais elle est fastidieuse. La solution proposée par la gestion de charge consiste à fournir un utilitaire qui réalise ces opérations à la place de l'utilisateur.

Nous avons principalement traité les problèmes liés à la charge du processeur mais les autres ressources du système doivent également être prises en compte. Par exemple un site peu chargé peut offrir un espace de mémoire réduit. Dans ce cas, il est bon de vérifier si la place est suffisante pour le processus qui est envoyé sur ce site. Le but attribué à un algorithme de gestion de charge est alors de permettre aux processus de s'exécuter en tirant parti de toutes les ressources disponibles du système [GB90] et d'éviter les étranglements. Les ressources prises en compte sont des ressources physiques comme le processeur, l'espace mémoire, les périphériques, le réseau ou des ressources logiques du système d'exploitation. La disponibilité de ces ressources et les besoins des processus évoluent : la répartition doit donc être dynamique pour être performante. Le problème général posé par la gestion de charge peut être formalisé de la façon suivante :

"Etant donné un ensemble de ressources distribuées dont la disponibilité évolue, étant donné un ensemble de processus dont le nombre et les besoins évoluent au cours du temps, la gestion de charge doit définir et faire évoluer la répartition de ces processus sur les sites en fonction des changements intervenant dans le système - pour permettre aux processus de tirer le meilleur parti des ressources".

Il est facile de se persuader qu'il n'existe pas d'algorithme optimal ou qui puisse satisfaire toutes les applications. En effet il existe toujours un cas particulier qui sera favorisé ou défavorisé par un algorithme. Un gestionnaire de charge devra donc garantir de meilleures performances pour l'ensemble du système.

2.2 Intérêts de la gestion de charge

Le développement d'un service de gestion répartie de la charge est justifié par trois arguments principaux :

- l'argument majeur repose sur le gain de temps obtenu en utilisant les sites non actifs, ou moins chargés, pour exécuter les processus des sites trop chargés [DO91].
- la gestion de charge doit permettre d'obtenir une moins grande dispersion des temps d'exécution des applications dans un environnement multi-utilisateurs [ZF87]. Puisque les processus peuvent accéder aux sites les moins chargés - quand le site local est surchargé - ils ne sont plus dépendants de la valeur de charge locale mais d'une valeur de charge globale qui varie statistiquement moins.
- la gestion de charge permet aux applications parallèles d'être plus indépendantes de l'architecture sur laquelle elles s'exécutent. Puisque le gestionnaire de charge établit la répartition des processus, l'utilisateur n'a plus à définir leur placement, dépendant de la machine sur laquelle ils s'exécutent. Cette dernière justification s'applique plus particulièrement aux machines parallèles mais elle peut être généralisée pour tout réseau supportant des applications parallèles. Elle est rarement développée dans la littérature.

La justification de la gestion de charge varie suivant les applications auxquelles elle est destinée. Cet intérêt peut alors induire des choix d'algorithmes et d'implantations qui rendent souvent

l'expérience difficilement utilisable pour des cas plus généraux, par exemple lorsque la gestion de charge fait des suppositions sur la nature et le comportement des processus.

2.3 Incidence de l'architecture

L'architecture du système induit également des motivations différentes pour la mise en place d'un service de gestion de charge. Ainsi de nombreux algorithmes et implantations qui ont été développés pour des stations de travail ne sont pas applicables aux multicalculateurs et réciproquement. Nous recensons ici les principales contraintes qui déterminent l'implantation du service dans les deux cas.

2.3.1 Cas d'un réseau de stations de travail

Parmi les réalisations marquantes de gestionnaires de charge beaucoup concernent les stations de travail. Nous pouvons citer [ZF87], [DO91], [BS91], [AC88], [FR89], [TL88] et [HJ87]. Le principal reproche que nous pouvons faire aux implantations sur stations de travail est qu'elles utilisent trop fréquement la méthode de diffusion de la charge. Cette méthode est efficace sur un réseau du type bus ou anneau mais ne peut être utilisée sur des topologies plus complexes. Les conclusions tirées de ces implantations ne sont valables que pour les stations de travail.

Sur un réseau de stations de travail il est rare que tous les processeurs soient utilisés au maximum de leurs possibilités de traitement. Par contre, l'exécution de grosses applications dépasse vite la capacité du processeur local. Un exemple qui illustre bien ces affirmations est le cas du développeur : il travaille la majeure partie de son temps avec un éditeur de texte - peu gourmand en temps d'exécution - mais ne peut exécuter rapidement une compilation importante. Il est donc tentant de faire bénéficier les grosses applications du temps de traitement disponible sur d'autres stations. Dans ce cas, la gestion de charge répartie est un moyen d'augmenter la puissance de calcul offerte aux utilisateurs. Le bénéfice généralement constaté [DO91][AF89] [TL88] est le gain d'un grand nombre d'heures de calcul à l'échelle du réseau. Ces réalisations ne cherchent pas à équilibrer complétement la charge entre les différents sites mais plutôt à déplacer les applications des sites très chargés vers les sites peu chargés. Pour atteindre ce but la mise en place d'une répartition de charge semble suffisante [BS91].

Dans un réseau de stations de travail où chaque utilisateur travaille sur une machine, il existe souvent une notion de propriété : une station est le minimum garanti à chaque utilisateur pour ses besoins. Les utilisateurs souhaitent alors ne pas être importunés sur leur machine par l'exécution d'applications autres que les leurs. Ces considérations doivent donc être prises en compte par un gestionnaire de charge. Par exemple dans le système Sprite [DO91], la gestion de charge utilise les stations distantes inoccupées pour exécuter des processus. Lorsqu'une activité locale apparaît sur la machine les processus distants sont immédiatement migrés, vers le site dont ils sont issus, pour laisser la disponibilité de la machine à l'utilisateur.

Un environnement de stations de travail n'a pas été conçu pour exécuter des applications parallèles. Par contre certaines applications peu couplées, ou à gros grain de parallélisme, peuvent tirer bénéfice d'une parallélisation (par exemple la commande `make` d'UNIX). Nous pensons qu'un

gestionnaire de charge, dans un réseau de stations de travail, sert principalement à distribuer les commandes lancées par un utilisateur depuis un shell de telle sorte que celles-ci soient exécutées plus rapidement que localement. Pour les stations de travail c'est donc principalement le gain de temps qui est l'intérêt prédominant. Les deux autres, cités précédemment, n'étant pas vraiment adaptés à ce cas puisqu'ils concernent plutôt les applications parallèles.

2.3.2 Cas des multicalculateurs

Les conclusions apportées par la gestion globale de la charge sur des réseaux de stations de travail sont, en partie, utilisables pour les multicalculateurs. Néanmoins il faut noter que les applications qui s'exécutent sur les deux types d'architectures sont différentes : il est plus probable de devoir gérer une application à base de processus communiquants dans le cas du multicalculateur que dans le cas du réseau de stations de travail. D'autre part, le couplage entre les processus d'une application et le grain de parallélisme ne sont pas non plus les mêmes dans les deux cas. De ces différences nous pouvons déduire que les besoins de gestion de charge sont différents.

Indépendance par rapport à l'architecture Au niveau de la programmation, l'allocation statique des processus aux noeuds pose un problème de portabilité. La distribution de l'application est souvent intégrée à sa conception ce qui la rend dépendante de l'architecture de la machine. La prise en compte du nombre de noeuds peut, par exemple, être une contrainte supplémentaire dans la programmation. Il est, au contraire, préférable d'écrire des programmes parallèles en utilisant le parallélisme naturel des structures et des données, sans se soucier de l'architecture sur laquelle ils vont s'exécuter. Pour cela, il faut offrir une gestion de la distribution des processus au niveau du système d'exploitation, transparente à l'utilisateur. Le bénéfice apporté par la gestion de charge est alors la portabilité des applications parallèles et une facilité accrue pour le développement de telles applications.

Dispersion des temps d'exécution Dans [ZF87] une gestion de charge répartie doit permettre une moins grande dispersion des temps d'exécution d'une application. A ce titre, nous pouvons comparer un tel service sur un multicalculateur à un service de temps partagé sur un ordinateur classique. En effet, les deux services sont destinés à mieux répartir les temps de calcul entre les applications. Dans le cas du placement, le temps d'exécution d'une application est très dépendant des applications qui s'exécutent concurrement : si la plupart des sites choisis par l'application sont fortement occupés, alors le temps d'exécution est plus long. Par contre, la gestion de charge répartie alloue les noeuds en fonction de leur charge ; le temps d'exécution de l'application ne dépend alors plus de son placement mais d'une valeur globale de charge.

Faciliter et rationnaliser l'utilisation des multicalculateurs Les multicalculateurs sont actuellement des machines dédiées, réservées à un petit nombre d'initiés, comme le furent les multiprocesseurs à mémoire partagée. Ceux-ci se sont cependant popularisés et équipent maintenant de nombreux serveurs. Comme eux l'utilisation des multicalculateurs peut être étendue à

une gamme plus vaste d'utilisateurs. Il faut pour cela qu'ils offrent une interface standard facilement accessible aux utilisateurs. Nous pensons que les multicalculateurs peuvent être utilisés comme serveurs d'exécution tels que ceux décrits dans [PPTa91]. L'intérêt pressenti est le rapport coût/performances, les possibilités d'évolution et les possibilités intrinsèques de tolérance aux pannes de ces machines. Dans un tel environnement l'utilisateur ne peut gérer lui-même la répartition de ses processus et l'administration de la répartition des noeuds doit être pris en compte par le système d'exploitation. Ces systèmes d'exploitation ne répondent alors plus aux même contraintes que sur les stations de travail : dans le cas de stations de travail interconnectées par un réseau, chaque station est susceptible d'apporter sa part de charge même si ce n'est pas de façon uniforme. Dans le cas d'un serveur d'exécution, quelques noeuds gèreront les échanges avec l'extérieur et il sera possible d'avoir une grande part des noeuds dédiés à l'exécution. Avec une telle configuration la politique de répartition des processus doit prendre en compte les risques d'engorgement des noeuds gérant les échanges avec l'extérieur. Par exemple une politique qui privilégie uniquement l'envoi de processus sur les noeuds voisins [LK87] peut être mal adaptée car elle ne permet pas une assez grande dispersion géographique des processus. Dans ce cas, le but d'une gestion de charge répartie n'est pas d'accroître la puissance de calcul mais plutôt de faciliter et de rationnaliser l'utilisation du multicalculateur.

Gain de temps En permettant une meilleure utilisation du multicalculateur, la gestion de charge permet également un gain de temps. Les utilisateurs de ce type de machines ont généralement besoin d'une forte puissance de calcul. A ce titre, ils seront également intéressés par les gains de temps.

Bien sûr, comme beaucoup d'outils système, la gestion de charge risque en offrant plus de confort de faire baisser les performances. Pour cette raison, il sera probablement souhaitable de laisser le choix à l'utilisateur de profiter, ou non, de ces services. De plus, dans le cadre d'un ordonnancement global d'une application sur un multicalculateur, il sera intéressant d'offrir à l'utilisateur la possibilité de se réserver un nombre de sites dont il est sûr qu'il ne les partagera pas, ou encore la possibilité d'exécuter simultanément plusieurs processus sur des sites différents, pour améliorer les performances et permettre l'exécution d'applications scientifiques. Nous devrons donc offrir des politiques d'ordonnancement différentes, l'utilisateur choisissant celle qui convient le mieux à son application.

2.4 Problèmes fondamentaux de la gestion de charge

Le problème général de la gestion de charge répartie peut être scindé en sous problèmes indépendants. Nous les présentons de manière à respecter la chronologie suivant laquelle se déroule une gestion répartie de charge :

Estimation de la charge La charge d'un site dépend de l'utilisation de ses ressources : processeur, mémoire, réseau, etc. Pour estimer la charge, cette utilisation doit être traduite sous forme de valeurs. Parmi ces valeurs nous choisissons celles qui sont significatives pour les combiner et obtenir une ou plusieurs valeurs de charge.

Estimation de la surcharge Une fois la charge du site calculée, nous décidons si le site est surchargé ou s'il peut augmenter sa charge. Nous devons donc choisir les critères sur lesquels reposent les décisions d'engager un déplacement de la charge.

Initialisation de la gestion Lorsque nous avons détecté une sous- ou surcharge, nous devons choisir un protocole pour initialiser l'échange : est-ce le site sous-chargé qui demande de la charge ou le site surchargé qui propose de la charge ?

Echange des valeurs Une fois l'échange initialisé, nous devons choisir le mode d'échange des valeurs de charge, entre les sites. Ce mode dépend des instants d'échange, des sites impliqués dans l'échange, de la répartition des unités de décision, de l'architecture du système, etc.

Choix du processus Dans le cas d'une migration, nous devons choisir le processus qui tirera le plus grand bénéfice de sa migration.

Politique de déplacement Une fois la différence de charge établie, il devient nécessaire de déplacer une partie de la charge depuis le site surchargé vers le site sous-chargé. Nous devons donc choisir l'unité de déplacement. Jusqu'à maintenant nous avons parlé de gestion de charge en terme de processus, il est également possible de déplacer des activités ou d'autres entités d'exécution. En fonction de l'entité utilisée, nous choisissons la politique de déplacement qui lui convient le mieux entre la migration et l'exécution à distance.

Les algorithmes choisis peuvent alors être caractérisés en fonction de leur adéquation à répondre aux besoins des utilisateurs et des nouveaux problèmes qu'ils peuvent engendrer. Ainsi, les algorithmes utilisés doivent satisfaire les contraintes propres aux environnements répartis en particulier ils doivent être **extensibles** (capables de gérer quelques processeurs aussi bien que plusieurs centaines). Nous devons également vérifier que les algorithmes choisis n'induisent pas un surcoût trop important afin que les avantages générés par le service ne soient pas réduits à néant. Enfin ces algorithmes doivent être **stables** c'est-à-dire éviter de déplacer continuellement les processus d'un site à l'autre sans leur laisser le temps de s'exécuter.

3 Etudes et réalisations de gestion de charge

La décomposition du problème de gestion de charge répartie en sous-problèmes permet l'implantation d'un tel service sous forme modulaire - basé sur un ensemble de sous services indépendants - et facilite l'évaluation de différents algorithmes. Ainsi, certains projets de recherche se sont fixés pour but de tester, dans leur environnement, différents algorithmes de gestion de la charge pour déterminer expérimentalement celui qui est le mieux adapté, à chaque sous-problème. Parmi ces projets nous pouvont citer [JVV91], [DTJ89] et [BBHL90] qui décrivent un berceau de tests plutôt qu'un choix d'algorithme. Ces implantations posent un problème quant à la validité de leurs résultats. En effet, si l'algorithme choisi est celui qui obtient les meilleurs performances sur l'architecture supportée, nous ne connaissons pas son comportement sur d'autres architectures. Dans ce cas, il serait intéressant d'avoir des comparaisons. Dans cette partie, nous présentons les solutions, proposées dans la littérature, aux problèmes énoncés précédement. Il nous a également semblé intéressant de citer quelques études sur le comportement des processus. De telles études

permettent d'obtenir, à la fois, des données utiles et une justification au développement d'un service de gestion de charge.

3.1 La vie d'un processus

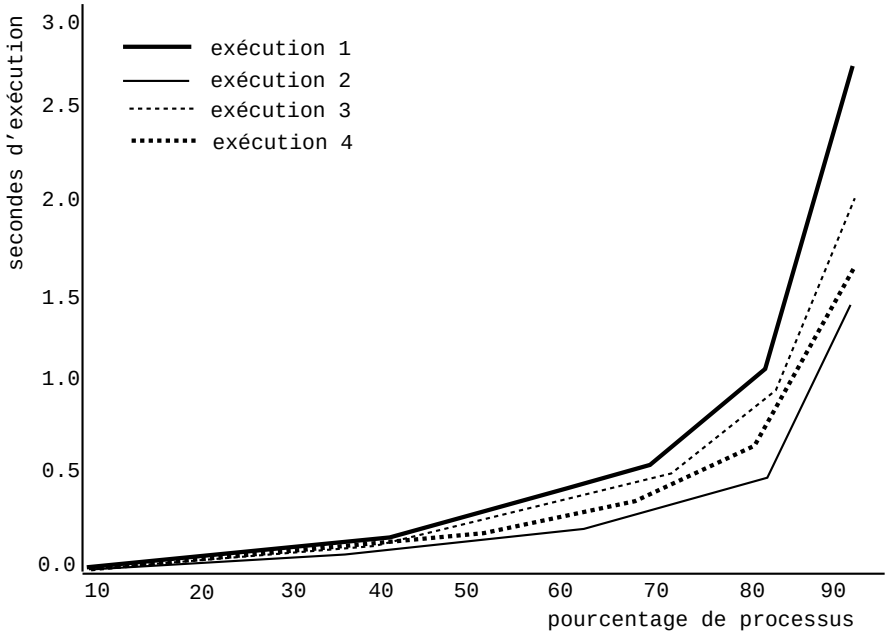

Figure V.3 : Pourcentage de processus en fonction de leur temps d'exécution

Quelques articles relatent des études sur le comportement des processus dans un environnement standard de stations de travail interconnectées. Nous ne possédons pas de données sur le comportement des processus sur un multicalculateur. Les données que nous possédons nous paraissent suffisantes dans la mesure où nous nous intéressons aux applications d'ordre général. Parmi ces études nous pouvons citer [LO86]. Nous reproduisons ici quelques unes des figures qui y sont données. Ainsi la figure V.3 donne la répartition des processus en fonction de leur temps d'exécution. Nous pouvons en déduire que 70% à 80% des processus consomment moins d'une demi seconde de temps CPU pour leur exécution et que 80% à 95% des processus consomment moins d'une seconde. Parmi ceux-ci, il est montré que 24% à 42% des processus qui ont une durée de vie d'une seconde n'utilisent en fait que 0.1 seconde de temps CPU et 73% à 84% des processus n'en utilisent que 0.4 seconde. Le reste du temps est passé dans des entrées/sorties.

La population des processus peut être divisée en trois catégories : ceux qui consomment beaucoup de temps CPU, ceux qui font beaucoup d'entrées/sorties et les processus "ordinaires" qui utilisent les deux avec parcimonie mais sans relation particulière entre les deux valeurs. La figure V.4 montre la dispersion des processus en fonction de leur utilisation du processeur et de leurs accès aux disques. Ainsi 98% des processus n'utilisent que 35% du temps d'exécution total du processeur. Par contre 0,1% des processus les plus longs constituent 50% de la charge totale des

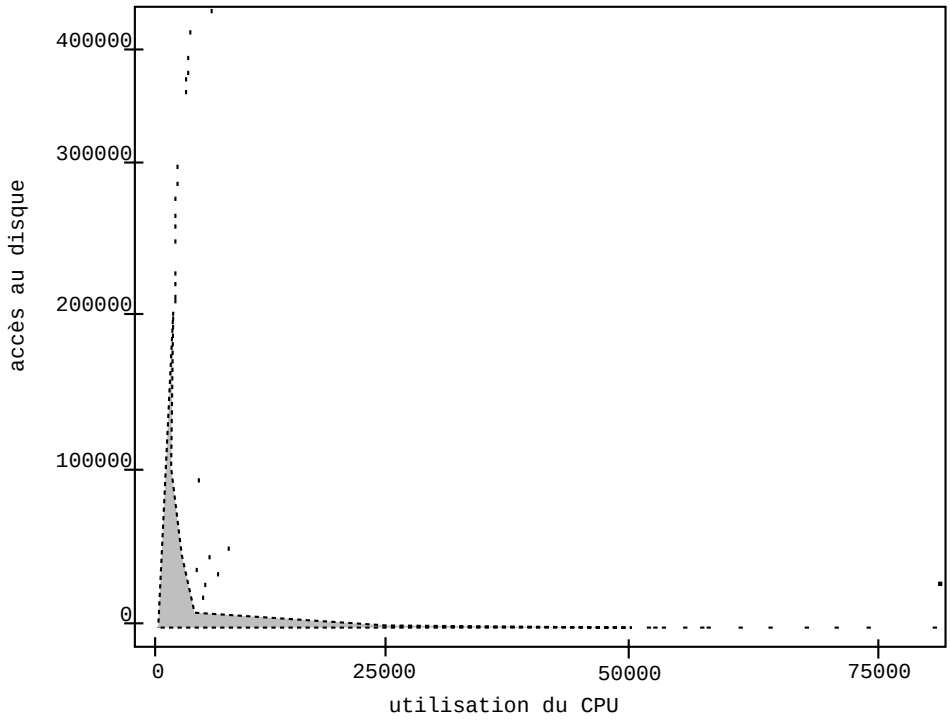

Figure V.4 : Dispersion de l'utilisation du processeur et du disque

processeurs. De ces données nous déduisons que, pour mettre en place une gestion globale de la charge, nous devons prendre en compte principalement les processus les plus longs. D'autre part les processus qui consomment le plus d'entrées/sorties sont également parmi ceux qui consomment le moins de temps de calcul.

D'autre part, une étude [Cab86] sur la répartition temporelle de la création des processus montre que celle-ci est très irrégulière. Les tests sont faits sur une durée de vingt minutes, ils sont découpés en intervalles d'une seconde. Il y a approximativement 90% de ces intervalles pendant lesquels au plus un processus est activé. Ceci tendrait à prouver qu'une politique de gestion répartie de la charge, basée sur l'exécution à distance, ne doit échanger ses valeurs de charge qu'à la création d'un processus et non périodiquement. Cette affirmation n'est pas valable pour une politique qui repose sur la migration dans la mesure où elle cherchera à déplacer les processus longs. Nous en déduisons également que des déséquilibres se créent par la répartition temporelle de la création des processus.

Dans [BF81] trois approches de calcul du temps d'exécution restant à un processus, que nous appelons **résidu d'exécution**, sont comparées à une méthode optimale. La première approximation suppose que tous les processus ont une durée de vie équivalente. Dans la deuxième, le résidu d'exécution est estimé par sa durée de vie au moment de l'estimation. La dernière approximation est basée sur la connaissance d'une valeur supposée par processus, le résidu d'exécution est alors déduit de cette valeur supposée moins le temps écoulé. Bien sûr la dernière méthode obtient de

meilleurs résultats que les deux autres dans un calcul d'erreur. Par contre, celle qui est basée sur le temps déjà écoulé est jugée satisfaisante du fait de sa simplicité. Cette estimation peut également être vérifiée d'après les conclusions données dans [LO86].

Nous nous intéressons maintenant à la résolution des différents problèmes liés à la gestion répartie de charge.

3.2 Estimation de la charge

L'estimation de la charge d'un site permet au gestionnaire de charge de décider si le site doit se décharger d'un ou plusieurs processus ou si, au contraire, il peut accepter une charge plus importante. Les valeurs significatives de la charge d'un site sont toutes les valeurs qui traduisent l'occupation des ressources locales à ce site. Ainsi nous devons prendre en compte l'occupation de la mémoire, le taux d'activité du processeur, les accès aux entrées/sorties, la charge locale du réseau, etc. Une généralisation des algorithmes de gestion globale de la charge doit également permettre de placer les processus en fonction des ressources disponibles dans le système d'exploitation. Dans ce cas les valeurs traduisant l'occupation des ressources système (nombre de processus, le nombre de fichiers ouverts, etc.) d'un site peuvent également être prises en compte. Remarquons que, si l'allocation des ressources du système d'exploitation est dynamique, l'occupation de ces ressources dépend uniquement de la place mémoire disponible.

Activité du processeur La valeur qui a le rôle principal dans la littérature est le taux d'occupation du processeur (*idle time*). Ceci est justifié puisque le but recherché est principalement l'optimisation de l'allocation des processeurs. Cette valeur est souvent représentée par la longueur moyenne de la file d'attente de l'ordonnanceur [BP86]. Dans [CA86] l'évolution de cette file est estimée en fonction de son comportement passé. Cette méthode est difficilement applicable à un environnement standard dont il est difficile d'extraire des régles de comportement, comme nous l'avons vu en 3.1.

Le résidu d'exécution des processus présents sur le site influe également sur le comportement du site, en particulier sur la durée d'exécution du nouveau processus. En effet, plus l'exécution des processus est longue moins le processeur se déchargera rapidement, le processus arrivant aura donc une durée d'exécution plus longue dans ce cas. Nous avons vu en 3.1 que la durée de vie restant à un processus peut être approximée par sa durée de vie actuelle. Ainsi le temps d'exécution restant au processeur peut être approximé par la somme des résidus d'exécution de ses processus. Les deux valeurs, longueur de la file d'attente et temps d'exécution restant au processeur, traduisent assez bien l'occupation de celui-ci.

Dans[BS85] la charge est calculée en fonction de la vitesse du processeur. Ce choix permet d'intégrer la gestion d'environnements hétérogènes.

Occupation de la mémoire L'occupation de la mémoire [SS84] peut facilement être obtenue auprès du système d'exploitation qui la gère. Néanmoins les propriétés de la gestion mémoire peuvent influencer l'utilisation de données significatives. En effet, l'espace mémoire disponible n'est pas significatif lorsque la gestion repose sur une pagination à la demande. Dans ce cas, la

taille de l'espace libre sur le disque est plus représentatif. D'autre part, décharger la mémoire depuis un site sans disque est coûteux puisque la décharge se fait à travers le réseau. Remarquons que le problème de la place mémoire est moins critique sur les ordinateurs actuels qui offrent des mémoires vives pouvant accueillir les données de plusieurs processus. De plus, il n'y a pas de corrélation entre le temps d'utilisation du processeur et la place mémoire d'un processus [BF81]. Il est donc plus utile de demander au site destinataire d'un processus si sa place mémoire est suffisante au moment d'effectuer le déplacement. Cette précaution est de toutes façons nécessaire pour être sûr que nous pouvons exécuter ce processus sur le site destinataire.

Accès aux entrées/sorties Les accès aux entrées/sorties doivent également être pris en compte lors de l'évaluation de la charge d'un site. En effet, il y a aproximativement un rapport inverse entre le taux d'utilisation du processeur et le taux d'utilisation des périphériques [Cab86], comme nous l'avons vu en 3.1. Il est donc intéressant d'intégrer ces valeurs à la charge du site puisque les processus concernés consomment une ressource mais apparaissent peu dans l'occupation du processeur. Ainsi l'indicateur de charge du système BSD [LMKQ89] est calculé à partir de la somme des longueurs de la queue du processeur et de la queue des entrées/sorties ; ces valeurs sont échantillonées puis lissées. Cet indicateur est utilisé dans plusieures réalisations, par exemple [ZF87] et [TL88].

Charge du réseau La charge locale du réseau peut aussi être utilisée dans le calcul de la charge d'un site. En effet, un réseau surchargé rend les communications distantes plus difficiles et donc ralentit l'exécution des processus. Néanmoins cette valeur est difficile à obtenir sans support direct du matériel. Par exemple si une facilité de routage automatique, ne nécessitant pas l'intervention du processeur, est offerte ou si le réseau est un bus (du type Ethernet) ou encore un anneau, alors des messages peuvent utiliser les même brins de réseau que le site sans que celui-ci s'en rende compte. Le calcul du taux local d'utilisation du réseau n'étant généralement pas significatif de l'utilisation réelle du réseau, nous ne pouvons calculer localement la charge du réseau. Certains réseaux offrent des statistiques sur l'utilisation du support de communication. Ces statistiques peuvent être intégrées dans une politique de gestion de charge répartie si elles ne sont pas trop coûteuses à obtenir.

La multiplication des valeurs de charges entraîne des surcoûts importants lors de leur acquisition auprès du système et lorsqu'elles sont échangées entre les sites. Il ne peut donc être question de prendre en compte et d'échanger toutes les valeurs de charge d'un site avec les autres sites. Seules quelques valeurs, suffisamment significatives, sont retenues. Il est également possible de calculer un indice de charge, représentatif de la charge du site, en combinant ces valeurs. Toutes les données n'ayant pas la même importance vis à vis du site, elles peuvent être pondérées pour obtenir une estimation de la charge plus représentative. Ces méthodes permettent de réduire les coûts d'échanges entre les sites.

Dans [ELZ86], les auteurs comparent différents algorithmes de gestion de charge. Ils montrent que des techniques extrêmement simples - qui utilisent peu d'informations - de gestion de la charge donnent de très bons résultats, aussi bons que ceux d'algorithmes compliqués (voir figure V.2). Ainsi un algorithme basé sur un seul seuil de charge peut obtenir des performances proches de

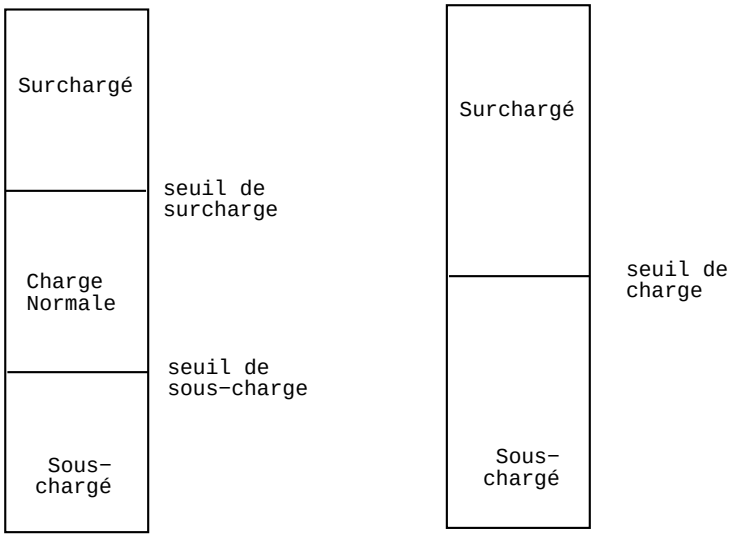

Figure V.5 : Seuils de charge

l'optimum malgré sa simplicité. La même conclusion est donnée dans [LO86]. Les simulations décrites dans [CH87] montrent également qu'une grande quantité d'informations n'apporte rien dans la mesure où leur validité dure moins longtemps.

Les valeurs de charge qui auront été préssenties comme significatives sont échangées entre les sites afin d'élaborer les décisions de déplacement de processus.

3.3 Estimation d'un déséquilibre de charge

L'initialisation d'une migration ou d'une exécution distante de processus pour répartir la charge est effectuée par une unité de décision. C'est l'unité de décision qui mémorise les valeurs de charge des autres sites. Elle doit être capable de prendre une décision rapidement quelque soit l'état de la charge sur l'ensemble du réseau. En particulier, elle doit détecter les déséquilibres de charge entre les sites.

Dans de nombreux cas, un écart de charge est détecté en comparant la charge locale à un ensemble de valeurs provenant d'autres sites. Soit à partir de constantes, soit en utilisant les valeurs des autres sites, l'unité de décision peut classer le site comme faiblement, modérement ou très chargé. Une décision de déplacement de charge est généralement prise lorsque le site est considéré comme surchargé. De nombreux algorithmes utilisent des valeurs de seuils (figure V.5) pour déterminer si le site est souschargé ou surchargé. Dans [ELZ85] et [DTJ89] un seul seuil est utilisé pour déterminer si le site peut accepter des processus d'autres sites. Le site est alors soit en état de sous-charge donc près à recevoir une charge supplémentaire, soit en état de surcharge donc près à distribuer sa charge. D'autres algorithmes tels que ceux décrits dans [NXG85] ou [AC88] utilisent

deux seuils pour classer la charge des sites en trois classes : faible, moyenne ou forte. Les sites de faible charge peuvent accepter une charge supplémentaire, les sites de charge moyenne n'acceptent par de surcharge mais n'en fournissent pas non plus et les sites de forte charge cherchent à se décharger d'une partie de leurs processus. Les algorithmes basés sur deux seuils souffrent moins d'instabilité que les algorithmes basés sur un seul seuil puisque un état de charge modéré ne génère pas de déplacement de processus. Par contre ils admettent des différences de charge entre les sites - tant que cette différence est comprise entre les deux seuils - et ainsi ne permettent pas un équilibrage aussi précis que les techniques n'utilisant qu'un seul seuil.

Les seuils utilisés peuvent être fixes, c'est-à-dire déterminés pendant le développement du gestionnaire de charge, ou variables [HJ87] [PTS88], c'est-à-dire déterminés dynamiquement en fonction de la charge moyenne du système. Les algorithmes basés sur des seuils variables permettent de mettre en oeuvre un meilleur équilibrage de la charge puisqu'ils prennent en compte l'état du système.

Dans [BP86] le système ne fixe pas de seuil de charge. Une unité de décision mémorise les valeurs de charge de plusieurs sites dans une table. La charge locale est ensuite comparée à ces valeurs pour déterminer l'état de charge du site.

3.4 Initialisation de la gestion de charge

La gestion de charge peut être initialisée soit par un processeur surchargé offrant à d'autres processeurs d'exécuter quelques unes de ses tâches, soit par un processeur sous-chargé, éventuellement inoccupé, en demandant du travail. Ces politiques ne sont pas exclusives et peuvent être mises en oeuvre ensemble.

Si le système utilise uniquement la répartition de charge pour mettre en oeuvre sa politique d'ordonnancement global, une implantation où le processeur cherche à se décharger - ou **initialisée par l'émetteur** de la charge - semble plus adaptée. A la création d'un processus le processeur est prêt à entrer dans une phase de déplacement de sa charge. Par contre il ne peut se permettre d'attendre les propositions émanant d'autre sites pour exécuter le processus car rien ne prouve qu'il en recevra une dans un intervalle de temps assez court. S'il attend une telle requête il risque donc de pénaliser l'exécution du nouveau processus.

Par contre, si le système supporte la migration de processus il peut choisir entre les deux politiques puisqu'une migration peut être effectuée à la demande. Dans ce cas, une implantation initialisée par l'émetteur est mieux adaptée pour des réseaux peu ou modérément chargés : si la plupart des sites sont surchargés, ils génèrent un grand nombre de messages de demande d'exécution de processus qui encombrent le réseau et le peu de processeurs sous-chargés. Réciproquement, une stratégie où le processeur cherche du travail, que nous qualifions d'**initialisée par le récepteur** de la charge, convient mieux à un réseau très chargé puisque peu de messages de migration sont générés. Mais elle est mal adaptée à un réseau peu chargé puisqu'elle génére un travail supplémentaire important. La méthode d'initialisation par le récepteur a l'avantage de faire supporter le poids du calcul induit par la décision d'un déplacement de charge au site le moins chargé, ce qui pénalise moins l'exécution générale.

Dans [ELZ85] les techniques d'initialisation par l'émetteur et celles d'initialisation par le récepteur

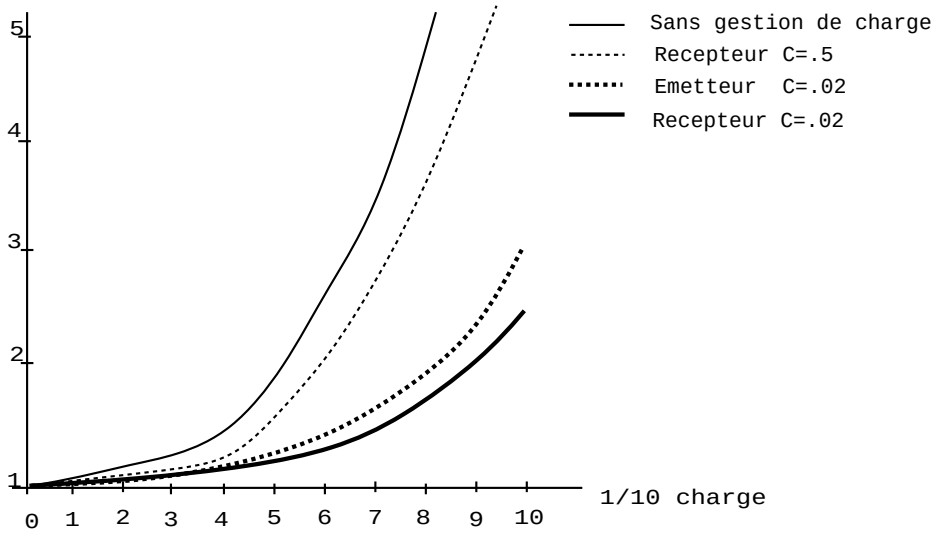

Figure V.6 : Comparaison de deux techniques de gestion de charge

sont comparées. Les résultats sont applicables à un grand nombre d'architectures dans la mesure où le modèle utilisé est basé sur des communications directes entre les sites mais sans utilisation d'un service de diffusion. Les conclusions, tirées de la figure V.6 sont les suivantes : les deux techniques offrent des gains de performances substantiels, l'initialisation par l'émetteur se comporte mieux pour une charge faible ou modérée, l'initialisation par le récepteur se comporte mieux avec une charge forte - si les coûts de transfert sont égaux dans les deux cas. Par contre, cette technique n'apporte pas un gain significatif même lorsque le réseau est très chargé.

Une méthode hybride qui intègre les deux stratégies et s'adapte à la charge du réseau peut alors s'avérer intéressante. Le problème est de savoir si le travail supplémentaire généré par la nécessité de décision entre les deux algorithmes compense les désavantages de ces stratégies lorsqu'elles sont utilisées seules. Nous pouvons remarquer que les deux stratégies sont compatibles et le fait que tous les sites n'utilisent pas le même algorithme ne gêne pas à la mise en oeuvre d'une politique globale de gestion de charge.

Une méthode hybride, utilisant l'initialisation par l'émetteur et par le récepteur, est décrite dans [GG91]. Le réseau est divisé en partitions au sein desquelles une première répartition de la charge, engagée par le récepteur, est effectuée. Si cela n'est pas suffisant une partition cherche à se débarasser de sa surcharge. Cet algorithme est comparé à deux autres pour montrer le gain qu'il entraîne. Cependant cette comparaison porte sur l'exécution d'une application dont le comportement est connu ce qui réduit la validité des conclusions. De plus la constitution de partitions est basée sur des informations statiques concernant cette application (charge supposée des liens de communications). Il semble donc difficile de la mettre en oeuvre dynamiquement dans un environnement exécutant plusieurs applications.

3.5 Echange des valeurs de charge

Pour mettre en place une politique de gestion de charge qui permette une utilisation des processeurs proche de l'optimum il faut connaître de façon précise les caractéristiques et l'état du système : emplacement des différentes ressources, état de charge des processeurs, etc., ceci à tout instant de l'exécution. Une telle politique est trop coûteuse à la fois dans la prise en compte d'un grand nombre de valeurs, engendrant des calculs trop importants, et pour les échanges de messages qu'elle entraîne. L'échange systématique d'une seule valeur entre tous les sites engendre, si le réseau n'offre pas de facilité de diffusion [LO86], un nombre de communications de l'ordre du carré du nombre de sites. Dans ces conditions l'algorithme ne peut plus être qualifié d'extensible. Notons également qu'un service de diffusion sur un réseau dont la topologie n'est pas adaptée (hypercube, grille, etc.) ne peut pas non plus servir de support à un échange des valeurs de charge.

La gestion de charge doit donc se contenter d'un ensemble réduit de valeurs qui sont communiquées partiellement. Le choix des sites auxquels les valeurs de charge locales sont communiquées constitue généralement un compromis entre une connaissance uniquement locale et une connaissance globale sur chaque site. L'approche la plus simple [ZF87] ne se soucie pas des valeurs des autres sites et choisit aléatoirement le site destinataire d'un trop plein de charge. Un premier raffinement [ELZ86] consiste à échanger les valeurs de charge avant un transfert pour éviter un déplacement pénalisant. Si le système base sa gestion répartie sur les valeurs de charge des autres processeurs celles-ci doivent être échangées entre les sites. Parmi les techniques décrites ou développées nous différencions :

Techniques de temps fixe pour lesquelles le site communique ses valeurs de charge à chaque période de temps. Le site doit choisir les destinataires de ces valeurs. Il ne peut les envoyer à tous les sites, même si le réseau offre un service de diffusion : cela génère un trafic de messages trop important si chacun émet simultanément. Les techniques de temps fixe ne permettent pas d'avoir une connaissance exacte de la charge à l'instant de déplacement d'un processus mais seulement une estimation. Ce défaut peut être compensé en vérifiant la charge du site destinataire, avant le déplacement, afin de s'assurer qu'elle est restée dans un ordre de grandeur équivalent à celui de la charge supposée.

Techniques au coup par coup pour lesquelles les valeurs de charge ne sont échangées qu'au moment où elle sont nécessaires. Par exemple lorsqu'un processeur se considère comme surchargé il peut demander les valeurs de charge d'autres sites afin d'engager un processus de répartition de charge. L'avantage de cette technique est de permettre une connaissance assez précise et sûre de la charge du site distant au moment du transfert. Par contre, elle entraîne une plus forte activité au moment de la demande.

Techniques basées sur la diffusion qui utilisent ou supposent qu'une fonctionnalité particulière permet à un site de diffuser sa charge à d'autres sites. Bon nombre d'algorithmes décrits font cette supposition [DO91] [LO86] car ils destinent leurs fonctionnalités à des réseaux de stations de travail connectées par un réseau en bus - Ethernet étant le plus classique - ou en anneau sur lesquel la diffusion est offerte. Par contre, ces techniques ne sont pas applicables actuellement au réseau des multicalculateurs où les connexions point à point rendent la diffusion très coûteuse : d'une part car elle doit actuellement être gérée

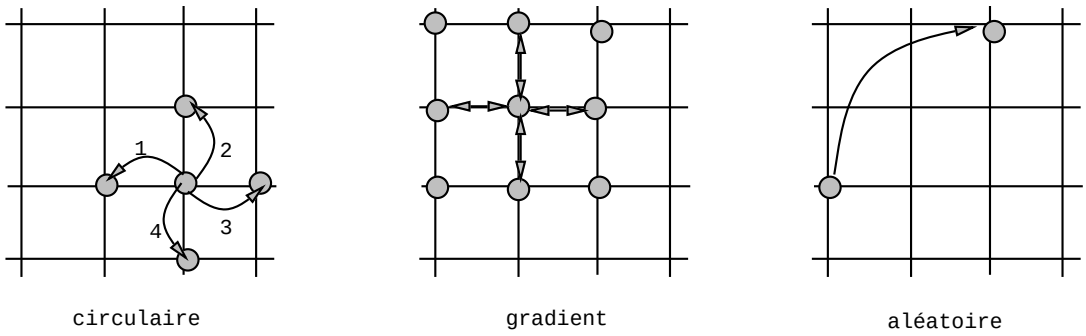

Figure V.7 : Techniques d'échange des valeurs de charge

au niveau du logiciel et d'autre part car elle engendre un nombre de messages de l'ordre du nombre de sites (elle n'en engendre qu'un sur un bus).

Dans [Zho88] le système est soumis à différentes charges pour étudier son comportement. Dans ce cadre les algorithmes utilisant des échanges périodiques fournissent des performances équivalentes à ceux basés sur une technique au coup par coup.

Sur les réseaux n'offrant pas de facilités de diffusion le site doit choisir un ensemble restraint de sites auxquels il envoie ses valeurs de charge. Différentes techniques (figure V.7) peuvent alors être utilisées :

Echange par paires L'échange par paires consiste à envoyer la charge d'un site à un seul autre site. Cet échange ne permet pas une large diffusion de la charge du site puisqu'aux instants d'échange un seul autre site connaît la charge. De ce fait la répartition de la charge est plus lente (donc moins performante) mais plus stable. Par exemple, si un site est peu chargé proportionnellement à ses voisins il va chercher à acquérir de la charge. Lorsqu'il communique ce besoin à plusieurs sites il a, statistiquement, plus de chances de trouver un site intéressé à se décharger que s'il communique sa charge à un seul site. Par contre puisque l'échange se fait moins vite il y a moins de risque que les processus soient déplacés trop souvent (cf. 3.10). L'avantage de l'échange par paires est son extensibilité. En effet l'algorithme ne fait jamais intervenir plus de deux sites dans un échange. Le nombre de messages échangés est donc indépendant de la taille du réseau.

Le choix du second site, constituant la paire, est variable. Le site peut communiquer sa charge à un seul autre site choisi de façon aléatoire [BS85]. Nous appelerons cette technique échange par **paires aléatoires**. Elle permet d'obtenir une bonne dispersion des processus sur le réseau quelque soit sa taille. Dans [GV91] les paires sont établies tour à tour avec chacun des voisins du site. Cette technique est appelée **paires circulaires** (round-robin).

Le principal problème posé par l'échange par paires est le peu de connaissance sur l'état du réseau obtenu par un site. Celui-ci ne peut alors fixer son comportement qu'en fonction de la charge d'un autre site et non en fonction de la totalité du système. Cette vue, trop restrainte, de l'ensemble du système peut amener un processus à être déplacé plusieurs fois avant de trouver un site qui est réellement peu chargé. Une solution est proposée par le système Mosix[BP86]. Chaque site

conserve une table dans laquelle il mémorise une dizaine de valeurs de charges et les sites auxquels elles correspondent. Suivant une technique de temps fixe les sites échangent leurs tables avec un site choisi de façon aléatoire. La mise à jour est faite en conservant les deux moitiées les plus à jour des tables. Ces tables ne reflètent pas parfaitement l'état du système mais elles permettent de donner une vue plus globale du système. Notons que celui-ci n'évolue pas rapidement puisque les échanges se font sur le modèle des paires aléatoires donc les informations contenues dans les tables conservent, en partie, leur validité.

Echanges avec le voisinage Le gestionnaire de charge peut aussi baser ses décisions sur un nombre fixe de valeurs de charge en provenance d'autres sites. Ainsi l'algorithme décrit dans [LK86] - appelé algorithme du *gradient* - utilise les valeurs de charge des voisins (situés à une distance inférieure ou égale à une constante d) pour choisir le site destinataire d'une surcharge éventuelle. Ces valeurs peuvent être obtenues soit en utilisant une technique de temps fixe (plutôt dans le cas d'initialisation par l'expéditeur) soit une technique au coup par coup (plutôt dans le cas d'initialisation par le recepteur). Le site définit alors un gradient de pression - représentée par la charge - et l'équilibrage est obtenu par raffinements successifs. Cet algorithme suppose une connaissance de la topologie du réseau, même si elle est partielle. En effet, la connaissance des voisins d'un site est nécessaire pour l'échange des valeurs. Il s'adapte bien, par exemple, aux architectures du type transputer puisque le modèle physique correspond au modèle de l'algorithme. Par contre lorsque le réseau est complet (même virtuellement comme l'iPSC/2) l'algorithme implique la définition d'un maillage pour le réseau afin de définir des distances entre sites ; le nombre de voisins et le maillage peuvent être choisis en fonction de l'architecture pour de meilleures performances. Notons également que l'algorithme du gradient se comporte bien à partir d'une configuration initiale où tous les sites sont chargés et pour compenser les déséquilibres qui apparaissent en cours d'exécution d'une application. En nécessitant plusieurs migrations pour un même processus - puisque les processus ne peuvent être envoyés au delà du voisinage - il devient assez coûteux dans le cas d'un fort déséquilibre.

La technique d'échange avec le voisinage permet de mettre en place un équilibrage réel de la charge. Cet équilibrage est obtenu en plusieurs périodes de temps. Il se fait donc assez lentement. Par exemple dans le cas d'un multicalculateur utilisé comme serveur de calcul, la charge est générée principalement par les sites sur lesquels sont connectés des utilisateurs. Le gestionnaire de charge doit alors avoir une grande capacité à disperser les processus pour assurer une bonne utilisation de la machine. Pour compenser ce défaut l'algorithme ACWN (Adaptative Contact Within a Neighbourhood) [Tur90] oblige les processus à se déplacer d'au moins *min.hops* et d'au plus *max.hops*. Ainsi en fixant - éventuellement dynamiquement - ces deux paramètres le gestionnaire de charge peut forcer la dispersion des processus.

Echanges à répétition La technique des échanges à répétition est dérivée de l'échange par paires. Dans [ELZ86] elle consiste à envoyer, depuis un site surchargé, un processus sur un site aléatoirement choisi. Si ce site est lui-même trop chargé, il renvoie le processus à un troisième site. Il est alors montré que cette technique peut s'avérer très instable dans un environnement surchargé : les processus ne trouvant aucun site sous-chargé sont continuellement déplacés. Par

contre en limitant le nombre de déplacements du processus à un seuil fixe (qui peut être choisi dynamiquement) elle retrouve sa stabilité. Cette technique est utilisée pour des applications réelles dans [GV91]. De même que l'échange par paires, cet algorithme offre une bonne extensibilité puisqu'il est indépendant de la configuration du système. Notons qu'il n'y a pas, dans ce cas, d'échange réel de valeurs de charge puique cet échange est implicite.

Pour améliorer les performances de cet algorithme il est possible de ne pas déplacer le processus sur tous les sites mais directement du site origine au site qui décide de l'accepter. En effet le transfert de processus étant assez lourd il vaut mieux simuler ce transfert par un simple échange de message. Cette optimisation est utilisée dans le système Falcon [GV91]. De même cet algorithme qui est prévu pour fonctionner avec initialisation par l'émetteur peut être adapté pour utiliser une initialisation par le récepteur : lorsqu'un site est sous-chargé il peut demander à un site de lui envoyer un processus ou de faire suivre la requête s'il n'est pas surchargé.

Mise aux enchères De même que l'échange à répétition, la mise aux enchères ne constitue pas vraiment un échange de valeurs de charge. La mise aux enchères consiste à annoncer à plusieurs sites la sous-charge ou sur-charge locale. Les sites qui reçoivent le message proposent alors leurs services, s'ils sont intéressés, en donnant une valeur (par exemple leur valeur de charge) qui constitue leurs enchères. Le site origine choisit alors l'enchère la plus intéressante pour déplacer son processus. Cette technique fonctionne bien si elle est couplée à un service de diffusion car elle permet, ainsi, de contacter beaucoup d'intéressés aux enchères. Un inconvénient de cette technique est qu'elle nécessite l'échange de nombreux messages pour la mise aux enchères comme pour les propositions d'enchères. Une variante de cet algorithme fait attendre le site qui propose une enchère un temps inversement proportionnel à cette enchère. Le premier message qui arrive au site d'origine est donc probablement celui qui propose la meilleure enchère ou, à défaut, un site qui propose une enchère intéressante. De même l'envoi des messages de mise aux enchères peut être limité à un voisinage ou un ensemble de sites aléatoirement choisis afin de réduire le nombre de messages.

Certaines implantations n'échangent pas explicitement leurs valeurs de charge. Ainsi la gestion de la charge du système V [TL88] repose sur le service de groupes de processus. Les sites du réseau sont classés en deux groupes : l'un contenant les sites surchargés et l'autre les sites peu chargés. Les sites moyennement chargés ne participent pas. En utilisant des seuils de charge, l'unité de décision d'un site met ou retire sa porte de l'un des groupes. La diffusion sur les groupes est alors utilisée pour trouver une machine libre ou un processus à exécuter.

Le mode d'échange des valeurs de charges est une part importante d'une gestion de charge. Ceci explique probablement pourquoi les algorithmes sont généralement nommés d'après leur technique d'échange de charge.

3.6 Le choix du processus

Le gestionnaire de charge doit choisir le processus à déplacer lorsque la charge locale du site est trop importante. Dans le cas d'une politique de déplacement des processus qui repose sur l'exécution à distance, le choix se portera sur le prochain processus devant s'exécuter sauf si le

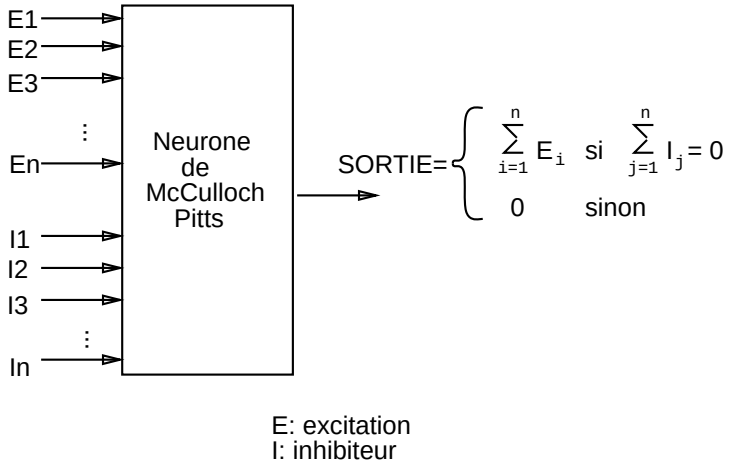

Figure V.8 : Neurone de McCulloch Pitts

gestionnaire de charge dispose d'informations sur le processus. Ces informations peuvent être des statistiques sur les exécutions précédentes du processus [ZF87] ou encore fournies par le compilateur. Dans le cas d'une politique de déplacement reposant sur la migration le choix se fera parmi la liste des processus du site. Nous nous intéressons au choix du processus dans le cas de la migration.

L'approche la plus classique consiste à considérer tous les processus comme étant équivalents et donc à migrer un processus quelconque. Cette politique est simple et donc rapide : elle ne nécessite ni l'acquisition ni la mémorisation de données sur le processus. Par contre, il se peut que l'exécution du processus sur le site destinataire s'avère plus lente que son exécution locale si celui-ci accède fréquement une ressource du site d'origine. De même un processus dont le temps d'exécution restant est inférieur au temps de migration ne gagnera pas à être migré. Pour optimiser le choix du processus nous utilisons les données qui le caractérisent. Idéalement il faudrait migrer les processus pour lesquels le résidu d'exécution est le plus long sur les sites dont ils utiliseront le plus de ressources. Malheureusement il n'est pas possible de déterminer à l'avance le comportement d'un processus et les ressources qu'il utilisera. Pour choisir le processus à déplacer différentes solutions sont envisageables :

Le processus le plus long Comme nous l'avons vu en 3.1 le résidu d'exécution d'un processus peut être approximé par sa durée d'exécution actuelle. Ainsi, en choisissant le processus qui a le plus long temps d'exécution, nous choisissons celui qui a, statistiquement, le plus important résidu d'exécution. Notons que le plus vieux processus n'est pas forcément celui qui a le temps d'exécution le plus long. En effet ce processus peut attendre une ressource depuis longtemps sans s'être exécuté depuis.

Les ressources utilisées Pour faciliter le travail du gestionnaire de charge il est envisageable de donner au processus la possibilité de préciser les ressources auxquelles il est plus particulièrement attaché. Ainsi le gérant de tâches Gatos [FR89] suppose la connaissance d'un

graphe de tâches pour respecter la précédence d'exécution dans une application parallèle. De même nous pouvons envisager la modification du compilateur afin que l'exécutable du processus contienne ces informations : cela entraîne un changement du format des exécutables donc une incompatibilité avec les systèmes standards. Une manière de déterminer dynamiquement certaines des ressources utilisées par un processus est alors de l'exécuter localement un certain temps pour voir ce qu'il utilise (fichier ouvert, mémoire, etc.) et avec quelle fréquence par rapport à l'utilisation du processeur ou du réseau. Ceci doit permettre de déterminer quelle est la ressource importante pour le processus et de placer le processus sur le site qui offre cette ressource. Ainsi, dans le système Mosix, chaque site conserve des statistiques sur les processus qu'il exécute. Les processus sont tenus de s'exécuter, sur un site, pendant une durée minimale avant d'être déplacés ce qui permet d'établir ces statistiques. Quand un processus migre, ses statistiques sont remises à jour. Le système utilise ces statistiques pour choisir le processus à migrer. Cependant cette stratégie s'applique bien au cas des processus demandeurs d'une seule ressource, ce n'est pas le cas général des processus. Ainsi nous pouvons avoir à choisir entre plusieurs ressources sur des sites différents. L'approche, parfois utilisée dans les langages à objets, qui consiste à déplacer le processus à chaque requête sur le site où réside la ressource, ne peut être utilisée ici car un processus est trop lourd à migrer. Dans [ZF87], en imposant l'immobilité à certains processus les performances ne sont que légèrement affectées. Nous pouvons donc contraindre certains processus à rester sur les sites dont ils utilisent les ressources sans implications importantes sur le comportement global de la gestion de charge.

Proximité des ressources Certains paramètres qui semblent intéressants, comme la prise en compte de la proximité de deux sites, sont en fait dépendants du support de communication. En effet ce paramètre n'a pas d'incidence pour les réseaux du type Ethernet, ni dans le cas de réseaux point à point qui utilisent un routage aléatoire [Int87b] puisqu'une communication passe toujours par un noeud intermédiaire aléatoirement choisi, avant d'arriver au destinataire. Dans ce cas seul le partage d'un site peut permettre à deux processus communiquants de réduire les temps d'échange de données. Nous pensons que ce paramètre peut avoir un intérêt pour d'autres types de réseaux. Les simulations et les réalisations dont nous disposons ne prennent pas ce paramètre en compte et travaillent sur des processus indépendants. Les fonctionnalités décrites dans [GV91] permettent de situer des processus sur un même site mais ne donnent pas de notion de proximité.

Neurone de McCulloch Une approche originale pour le choix d'un processus est décrite dans [SS84]. Le gestionnaire de charge utilise un neurone de McCulloch Pitts (figure V.8) pour choisir le processus à migrer parmi ceux qui sont disponibles. Le neurone de McCulloch-Pitts est une cellule de décision qui agit en fonction d'excitations et d'inhibiteurs. La sortie de cette cellule est une seule valeur : soit la somme des excitations, soit zéro. Si un des inhibiteurs est positionné, la sortie est nulle. Dans le contexte de la gestion de processus, les inhibiteurs peuvent être : le processus a besoin d'une ressource locale, il a déjà été migré plusieurs fois, le processus a été fixé sur ce site par l'utilisateur, etc. Les excitations sont tous les paramètres importants pouvant intervenir dans l'ordonnancement global : temps d'exécution actuel et global, nombre de migrations, état du processus, taille du processus,

priorité, ressources spéciales nécessaires, position dans la file d'attente de l'ordonnanceur local, etc.

Parmi les données à prendre en compte pour le choix d'un processus à migrer certaines sont primordiales - si le processus ne peut pas être migré - et d'autres facultatives. Une donnée primordiale peut être, par exemple, le placement d'un processus sur un site car il accède à une ressource particulière : horloge temps réel, gestion d'un périphérique, etc. Les données facultatives sont celles qui ont été décrites précédemment : proximité des fichiers, proximité d'un autre processus, etc. Notons que, comme pour la stratégie de choix du site destinataire, la stratégie de choix du processus ne doit pas non plus être trop coûteuse par rapport au temps d'exécution des processus. Dans [SS84], la gestion de charge est testée avec différentes contraintes sur les groupes de processus. Les résultats obtenus montrent qu'une implantation tenant compte des dépendances entre processus n'apporte pas de gains substantiels de performances, par rapport à la quantité d'informations supplémentaires à prendre en compte.

3.7 Choix de la politique de déplacement des processus

Dans la littérature nous pouvons différencier deux écoles en ce qui concerne le choix de la politique de déplacement des processus : l'une est pour la migration de processus [LO86] et l'autre qui est contre [JV88b]. Des études [ELZ88] et [KL88] ont tenté de déterminer les bénéfices apportés par la migration de processus en simulant le comportement d'un système réparti gérant la charge globalement. Nous n'avons pas trouvé de preuve convaincante pour l'une ou l'autre des solutions. Dans cette partie nous comparons les intérêts de chacune. Nous pensons que seule une réalisation permettant la mesure des performances dans les deux cas peut apporter une réponse. Celle-ci ne serait cependant pas définitive, d'une part car elle ne s'appliquera qu'au cas de l'environnement choisi et d'autre part car certaines applications particulières pourront tirer un bénéfice de l'une ou l'autre politique. Pour cette raison une politique mixte ou adaptative nous semble mieux répondre aux particularités de chacun. Notons tout de même que ce sont généralement les réalisations de gestion de charge destinées aux réseaux de stations de travail qui récusent la migration de processus.

La répartition de charge permet qu'aucun site ne reste inactif tandis que la charge augmente sur un autre. Dans ce cas le site d'exécution d'un processus est déterminé à sa création en fonction de données de charge. Pour implanter ce partage de charge, le système doit offrir une fonctionnalité d'exécution distante. L'intérêt de la répartition de charge réside dans le fait que les processus ne se déplacent qu'une fois au cours de leur exécution, au moment de leur création. Le système limite ainsi le temps passé en échange de processus. Par contre cette méthode ne nivelle pas complètement la charge, si elle ne fait aucune supposition sur la nature des processus. Dans une application parallèle, il peut se produire que les deux plus longs processus se trouvent sur le même site en fin d'exécution. Nous aurons alors un site traitant deux processus alors que les processeurs voisins sont inoccupés. Le résultat sera probablement une forte dégradation des performances. Ceci parce que le lieu d'exécution a été choisi sans tenir compte des caractéristiques des processus et qu'il n'y a pas de possibilité de rattrapage. D'une manière plus générale, la répartition de charge ne permet pas de résorber un déséquilibre qui se crée s'il n'y a pas de création de processus.

L'équilibrage de charge permet un nivellement permanent de la charge entre tous les sites quelque soit l'état des processus. Dans ce cas les processus seront migrés pendant leur exécution pour être déplacés. Une politique correctement mise en place assure qu'aucun site n'est inactif tant qu'un autre site exécute plusieurs processus. L'équilibrage de charge n'est donc qu'une tentative d'amélioration de la répartition de charge : cette technique doit permettre de mettre en place une meilleure dispersion de la charge sur les processeurs. Il convient donc mieux pour obtenir une moins grande divergence dans les temps d'exécution d'une même application. Les reproches principaux fait à la migration de processus sont son coût et la complexité des développements qu'elle entraîne. Cette complexité dépend du système dans lequel la migration est implantée (cf. chapitre IV) et ne peut être prise comme un argument général pour ne pas utiliser la migration. Le coût d'une migration est à comparer au coût d'une exécution distante. Pour migrer un processus il faut envoyer au site distant le code, les données, la pile et le contexte système du processus. Pour exécuter un processus à distance il suffit d'envoyer le code et les données du processus (certaines exécutions distantes, telle que celle qui est implantée dans CHORUS/MiX, nécessitent également l'envoi du contexte système). Le coût supplémentaire correspond donc à l'envoi de la pile et du contexte système du processus. Certaines optimisations peuvent modifier ce coût :

– si tous les sites possèdent un disque il peut y avoir une instance de l'exécutable de ce processus sur chacun des sites. Dans ce cas l'exécution à distance n'a pas besoin d'envoyer ni le code ni les données. Par contre la migration doit tout de même transférer les données puisqu'elles peuvent avoir été modifiées depuis le lancement du processus.
– si une instance du processus existe déjà sur le site destinataire, le code peut être partagé au moyen de mécanisme de mémoire virtuelle, il n'a donc pas besoin d'être transmis. Cette optimisation est valable dans les deux cas.
– certaines implantations de migration de processus [TL88] optimisent le transfert du code et des données en continuant à exécuter le processus localement, tant que ce transfert n'a pas atteint un stade avancé (cf. chapitre IV, 1.3.3). Le coût de transfert est alors plus élevé du point de vue général mais l'est moins du point de vue du processus.

L'utilisation de la migration de processus peut s'avérer intéressante dans le cas de processus à longue durée de vie, dans la mesure où le gestionnaire de charge peut ne déplacer que les processus qui tireront un bénéfice de la migration. Les statistiques citées en (cf. 3.1) semblent prouver que la dynamicité d'une répartition de charge n'est pas suffisante pour implanter seule une gestion de charge performante. En effet, la moitié du temps d'exécution est utilisée par les processus les plus longs. Un mauvais choix d'exécution à distance pour un de ces processus risque d'entraîner des conséquences importantes et à long terme. La migration permet alors modifier le placement et de limiter les conséquences de ce choix d'exécution à distance.

Nous pouvons remarquer que le choix entre différentes politiques de déplacement n'est pas toujours indépendant des autres choix d'implantation ou des besoins du service de gestion de charge répartie. En effet, certaines politiques de gestion de la charge sous-entendent une utilisation de la migration de processus. Par exemple l'algorithme du gradient suppose que les processus peuvent être déplacés plusieurs fois avant de trouver un site sur lequel ils se fixent. Avec un seul déplacement possible la charge se disperse insuffisamment en utilisant une politique d'échange de valeurs de charge avec les voisins. La gestion de charge du système Sprite utilise l'exécution

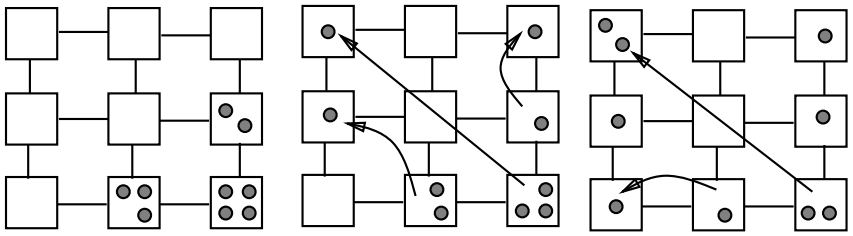

Figure V.9 : Dispersion des processus par les paires aléatoires

à distance pour tirer parti des stations de travail inoccupées du réseau. Par contre lorsqu'une station est utilisée par son propriétaire les processus en exécution distante qui s'y trouvent sont migrés sur leur station de travail d'origine. L'initialisation par le récepteur entraîne généralement la migration de processus. Dans ce cas, une étude [ELZ86] montre que, pour les algorithmes à seuil, si la migration est beaucoup plus chère que l'exécution à distance alors l'initialisation par le récepteur obtient de moins bonnes performances.

Remarque Certains systèmes, tels qu'UNIX, permettent la création de processus par duplication (*fork*). L'exécution à distance de l'instance dupliquée d'un processus pose plus de problèmes que la création à distance. En effet, le processus lorsqu'il est dupliqué possède des données qui sont modifiées dans son espace d'adressage et un contexte système. L'exécution à distance d'un processus dupliqué est donc équivalente à la migration de processus. Nous pourrons alors parler de **duplication à distance**. Nous revenons sur ce problème en 4.3.1.

3.8 Dispersion

Une considération importante pour un système de gestion de charge est le degré de globalité avec lequel un site voit le système. Or il n'est pas possible à un site de connaître la charge de l'ensemble du système pour des raisons d'extensibilité. Sur une architecture de multicalculateur, nous risquons alors de voir apparaître des accumulations géographiques de processus. L'algorithme choisi doit alors montrer des qualités de dispersion des processus sur le réseau. Nous pensons que la capacité de dispersion d'un algorithme est très importante pour les multicalculateurs dans la mesure où elle accélère la répartition de la charge or nous avons vu que les algorithmes d'échange avec le voisinage n'ont pas de bonnes qualités de dispersion 3.5. Dans la figure V.9, chaque noeud ne déplace qu'un processus à la fois vers un site choisi aléatoirement. Le choix aléatoire des sites formant une paire n'assure pas, en toutes circonstances, l'accès au site le moins chargé. Ainsi le noeud le plus chargé envoie deux fois au même site.

Par contre, la figure V.10 montre autant de migrations que la figure V.9 mais l'équilibrage obtenu est moins bon. Cette méthode est néanmoins extensible et peut donc convenir à un réseau ne bénéficiant pas d'un routage direct entre les sites.

La figure V.10 montre un autre problème posé par la méthode de l'échange avec le voisinage. Ici chacun des sites communique avec ses voisins et déplace un processus à la fois pour équilibrer

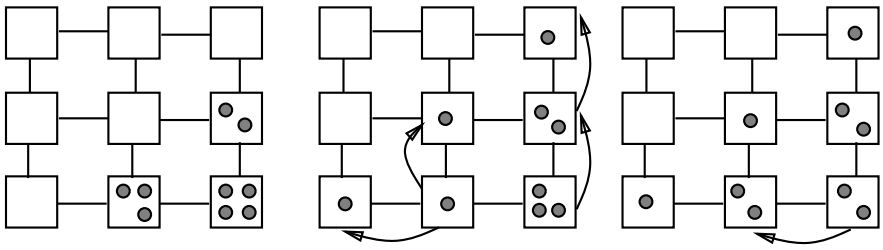

Figure V.10 : Dispersion des processus avec le voisinage

la charge. Dans le troisième schéma, chacun des noeuds à une différence d'un processus avec ses voisins. Cette différence n'est pas asez grande pour jusitfier un déplacement, la situation n'évolue plus. Elle est considérée comme un équilibre puisqu'il y a équilibre entre chaque noeud et ses voisins mais ce n'est pas un équilibre global. Si nous comparons les figures V.9 et V.10 nous constatons facilement l'avantage de la dispersion : le nombre de migration est le même entre les deux figures, mais la répartition est meilleure dans la figure V.9.

3.9 Extensibilité

En théorie, la distribution des unités de décision permet de fournir un service plus extensible, c'est-à-dire qu'il est possible de l'adapter à un grand nombre de sites sans modification. Des études telles que [TL88] prouvent que dans un environnement où la diffusion est possible (en particulier pour des réseaux de stations de travail) cette affirmation n'est pas vérifiée. Nous ne disposons pas d'études équivalentes dans le cas de calculateurs parallèles dont le réseau ne supporte pas la diffusion. Il n'est donc pas possible de généraliser. Il est également envisageable de réduire le nombre de sites possédant des gestionnaires de charge. Cette solution n'entraine pas une surcharge importante en échange de messages. Par contre elle doit permettre une meilleure dispersion des processus dans la zône d'influence du gestionnaire. Nous ne possèdons pas d'exemple d'une telle implantation.

3.10 Stabilité

La stabilité d'un système de gestion de charge est sa capacité à réagir à un changement dans la distribution de la charge sans produire une charge supplémentaire, due aux migrations. C'est-à-dire la capacité à éviter les déplacements de charge inutiles : par exemple la migration d'un processus puis son retour sur un même site. Le processus (et le système d'exploitation) risque alors de passer la majeur partie de son temps en migrations, sans s'exécuter. Le problème ne se pose pas lorsque la politique de déplacement des processus choisie est une répartition de charge. En effet un processus n'est déplacé qu'une seule fois au cours de son exécution, c'est le seul surcoût à son exécution. Il faut cependant éviter que les sites échangent systématiquement leurs processus lorsque cela n'est pas nécessaire.

Le risque de déplacement continuel des processus - ou effet **ping-pong** - est significatif quand

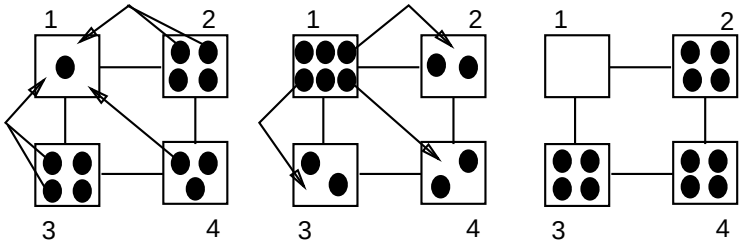

Figure V.11 : Exemple d'instabilité

apparaîssent des écarts de charge importants. En effet un tel déséquilibre risque d'entraîner une sur-réaction du système de gestion de la charge qui déplace une charge importante vers un site sous-chargé. Ce site passe ainsi en état de surcharge et veux donc, à son tour, se décharger du surplus sur un autre site. Un effet de ping-pong peut alors se produire entre les deux sites qui se chargent et se déchargent tour à tour. Notons que, lorsque le gestionnaire de charge est distribué et possède une connaissance partielle de la charge des autres sites, plus l'équilibrage de charge veut être précis plus les risques d'instabilité sont grands [CH88b]. Dans la figure V.11, chacun des sites 2, 3 et 4 s'aperçoit de la sous charge de 1, il n'y a pas consertation et chacun envoie le nombre de processus nécessaire à compenser le déséquilibre. Nous pouvons constater qu'avec un tel algorithme nous obtenons un résultat inverse de celui que nous attendions, puisque le déséquilibre n'est pas réduit.

– Dans Mosix [BS85] et [BF81], les processus doivent s'exécuter un minimum de temps sur un site avant de pouvoir être à nouveau migrés. Cette restriction permet de ne pas déplacer les processus trop souvent et ainsi de garantir la stabilité de la gestion de charge. Pour permettre une meilleure dispersion des processus, en particulier dans le cas où une application en crée beaucoup, le système Mosix compte le nombre de duplications du processus par unité de temps. Si ce nombre est élevé, le gestionnaire de charge lève la contrainte sur le temps de résidence minimum d'un processus et utilise l'exécution distante pour ne pas surcharger le site. Mosix ne permet pas non plus de migrer plus d'un processus à la fois pour éviter les échanges de charge trop importants et soudains qui déstabilisent le système.
– La technique à double seuil a été développée pour compenser les risques d'instabilité [AC88] de la technique à seuil unique. En établissant un état dans lequel le site ne cherche ni à se charger ni à se décharger l'algorithme réduit les possibilités de migration. Ainsi un site qui se charge par déplacement de processus (donc dans un état de charge faible) n'en accepte plus dès qu'il a atteint une charge moyenne. Il ne va alors pas chercher à s'en débarrasser, la stabilité est donc garantie.
– Certains articles prouvent mathématiquement la convergence d'algorithmes vers une charge uniforme en modélisant le processus de gestion de charge [Sta85] et en simulant les processus par des distributions.

3.11 Analyse

La littérature propose également des études intéressantes qui peuvent compléter cette bibliographie. Un bref aperçu des problèmes généraux de la gestion de charge est donné dans [Hac89]. Différentes techniques de gestion de charge sont analysées dans [BSS91a], [JV88a], [JV88b], [MPV91], [HLMB90]. Ces articles donnent une vue générale sur plusieurs algorithmes avec une vue différente de la nôtre car elles ne sont pas ciblées sur les multicalculateurs. Ces études nous permettent de conclure sur quelques problèmes généraux de l'implantation d'une gestion de charge :

- la mise en oeuvre d'un gestionnaire de charge diffère en fonction des caractéristiques du système visé. L'implantation d'un tel service, capable d'intégrer les contraintes à la fois des stations de travail et des multicalculateurs, semble donc difficile.
- il n'est pas nécessaire de prendre en compte un trop grand nombre de paramètres - aussi bien sur l'état du système que sur les caractéristiques des processus. Trop d'informations peuvent mener à une déstabilisation du système.
- la migration de processus s'avère intéressante dans quelques cas extrêmes. Ces cas extrêmes occupent la moitié du temps d'exécution global, il faut donc les prendre en compte.
- la capacité de dispersion de la gestion de charge doit être prise en compte sur un réseau reliant un grand nombre de noeuds, en particulier sur les mutlicalculateurs.

4 Un gestionnaire de charge pour CHORUS/MiX

Nous nous proposons de réaliser un gestionnaire de charge pour multicalculateur. Ce gestionnaire de charge fait partie des services que nous devons ajouter à CHORUS/MiX pour en faire un système à image unique.

4.1 Le cadre

Nous décrivons les objectifs fixés pour le gestionnaire de charge et les hypothèses qui caractérisent l'environnement qu'il doit gérer.

4.1.1 Objectifs

La gestion de charge repose sur un serveur, intégré au système UNIX, qui répond aux objectifs suivants :

- les intérêts d'un tel service sont différents selon que ce service est destiné aux multicalculateurs ou aux stations de travail. Nous ne prenons donc pas en compte les contraintes liées à ces dernières, ce qui entraîne quelques différences dans la conception du serveur. Par exemple, nous voulons obtenir un équilibrage de charge plutôt qu'une répartition de charge.
- nous prenons en compte les contraintes générales des systèmes répartis : le service doit être extensible, sans dépendance vis-à-vis de la topologie du réseau d'interconnexion ou de caractéristiques particulières à un multicalculateur.

– le gestionnaire de charge doit offrir une dispersion des processus qui permette de tirer parti des possibilités d'exécution du multicalculateur.
– l'interconnexion de plusieurs multicalculateurs demande la mise en place d'un service qui gère chacune des machines indépendamment. Les processus d'une application parallèle doivent être exécutés sur une même machine pour faciliter la communication entre ces processus. Par contre, un serveur peut choisir, pour l'utilisateur, la machine la moins chargée pour exécuter son application parallèle. Ainsi, nous différencions le service d'allocation d'une application à une machine, du service de gestion de charge répartie sur une machine. Le premier est plus proche de la répartition de charge sur un réseau de stations de travail. Nous nous intéressons au second.
– nous nous imposons d'implanter ce service de façon modulaire, c'est-à-dire avec un minimum d'interactions entre les parties du service chargées de résoudre les sous-problèmes énoncés en 2.4. Nous pourrons ainsi faire évoluer indépendamment les algorithmes utilisés. L'évaluation et le raffinement de ces algorithmes se feront d'après les tests effectués en "grandeur nature".

4.1.2 Hypothèses

Les caractéristiques des multicalculateurs sont très variables en ce qui concerne les organes d'entrée/sortie, le réseau de communication ou encore le type des noeuds. Puisque Ces caractéristiques influencent (cf. 3.5) le choix d'un algorithme d'échange des valeurs de charge, nous devons faire des choix quant au type de machine visé pour mettre en place la gestion globale de charge. Nous essayons d'imposer le moins de contraintes possibles.

Généralement les réseaux sont différenciés d'après leur topologie mais nous préférons prendre en compte leurs fonctionnalités de routage car elles déterminent la vue du réseau, locale ou globale, dont dispose le gestionnaire de charge. Ces fonctionnalités ont une influence importante sur le choix des algorithmes pour le gestionnaire de charge. Ainsi, Le routage direct convient bien aux techniques d'échange des valeurs de charge par paires aléatoires car il donne une vue globale du réseau, permettant ainsi d'obtenir une bonne dispersion de la charge en contactant des noeuds éloignés. Par contre, sans routage direct ce mode d'échange est réduit à un échange avec les voisins et perd ses propriétés. Nous choisissons de travailler pour des réseaux offrant un service de communication direct parce que d'une part beaucoup de réseaux offrent ce service, et d'autre part, ce service peut être simulé par un noyau de système d'exploitation. Nous ne faisons, par ailleurs, aucune supposition quant à la topologie du réseau.

Nous supposons que la gestion de charge est réalisée sur une machine ou un réseau homogène. Ainsi nous ne prenons pas en compte les contraintes liées à l'hétérogénéité des sites : ceci nous permet de supposer qu'un processus quelconque peut s'exécuter sur un site quelconque.

4.2 Choix des solutions

Avant de décrire en détails l'implantation du gestionnaire de charge dans CHORUS/MiX nous en donnons une vue indépendante du système qui lui sert de support. Ceci nous permet d'analyser chacune des solutions par rapport aux questions, posées en 2.4, sans considération extérieure à celles de la gestion de charge. L'ordre des propositions ne respecte par celui utilisé auparavant

mais plutôt un ordre naturel de dépendance entre les solutions.

4.2.1 Politique de déplacement

L'implantation de la migration de processus (chapitre IV) est moins performante que l'exécution à distance. Nous utilisons donc a priori cette dernière pour limiter les coûts de déplacement des processus. Ainsi un processus sera placé sur le site qui lui offre les meilleurs possibilités avant d'être exécuté. Cependant, comme nous l'avons vu en 3.7, cette politique ne satisfait pas pleinement les besoins des multicalculateurs. Pour compenser ces désavantages et mieux utiliser la capacité du multicalculateur, nous utilisons également la migration de processus lorsque le déséquilibre, mesuré entre deux sites, est trop important. L'implantation d'une politique hybride nous permettra également de comparer les performances de chacune et de comprendre quand utiliser l'une ou l'autre.

Pour limiter le coût de l'équilibrage nous essayerons de limiter le nombre des migrations. D'après les chiffres cités en 3.1, la migration ne doit par concerner plus de 1% des processus. Par contre, même avec un service de migration qui agit rarement, nous devons obtenir des gains importants, ce qui justifie la mise en place d'un tel service.

4.2.2 Echange des valeurs de charge

L'algorithme d'échange des valeurs de charge est probablement une des parties les plus sensibles de l'ordonnancement réparti. Sur les multicalculateurs, celui-ci doit satisfaire aux contraintes d'extensibilité tout en permettant une bonne dispersion des processus sur l'ensemble du réseau. Dans notre environnement, chacune des techniques décrites précédemment possède des avantages et des inconvénients. Nous avons essayé de les analyser :

Echange avec le voisinage Comme nous l'avons vu, les techniques d'échange avec le voisinage utilisent largement la migration pour obtenir l'équilibrage de charge. Nous pensons qu'une telle technique n'est pas bien adaptée à un multicalculateur offrant un routage direct entre les noeuds. Elle nous oblige à définir les voisins de chaque noeud et donc un maillage dans le réseau. Elle ne tire pas parti de la communication directe entre chaque noeud. De plus la dispersion des processus nécessite un grand nombre de migrations, elle est donc trop coûteuse.

Mise aux enchères La mise aux enchères, pour permettre une bonne diffusion, implique la distribution des messages d'enchères à un grand nombre de sites, ce qui ne permet pas une bonne extensibilité. Pour limiter les échanges de messages, nous pouvons utiliser l'une des deux solutions suivantes :

- définir un voisinage pour le site proposant les enchères. Les difficultés d'implantation sont alors les mêmes que pour les échanges avec le voisinage, cités précédemment.
- proposer les enchères à un nombre fixe de sites, aléatoirement choisis. Cette technique aurait l'avantage de permettre une bonne dispersion des processus sur le réseau.

Cependant la mise aux enchères n'est pas utilisable pour l'exécution à distance puisqu'elle suppose une initialisation de l'échange par le récepteur. Elle ne peut donc convenir aux choix de politique de déplacement des processus que nous avons faits.

Echanges à répétition L'échange à répétition, dans sa forme originale, ne convient pas à notre environnement. En effet, la migration de processus est trop coûteuse pour que nous déplacions des processus dont nous ne sommes pas sûrs qu'ils s'exécuteront dans de meilleures conditions. En échangeant des messages à la place des processus, l'algorithme devient une variante des paires aléatoires. Le fait de consulter plusieurs sites lui permet, statistiquement, de trouver un site moins chargé que le site fourni par l'algorithme des paires aléatoires. Le problème est alors de mesurer le surcoût, engendré par un nombre plus important de messages et des temps de décision plus longs, pour déterminer s'il apporte malgré tout des gains.

Les paires aléatoires La technique basée sur l'échange par paires aléatoirement choisies, telle qu'elle est mise en oeuvre dans Mosix, nous semble bien adaptée aux multicalculateurs. Elle permet une bonne extensibilité puisqu'elle n'implique jamais plus de deux sites dans un échange. De plus, le choix aléatoire d'un site permet d'obtenir une vue dispersée de la charge du système, dans la mesure où les valeurs dont nous disposons proviennent de noeuds quelconques du réseau. Ainsi, la notion de proximité n'est pas prise en compte ce qui évite l'engorgement d'une partie du réseau et permet d'obtenir rapidement une bonne dispersion des processus. D'autre part l'utilisation de tables de charge (cf. 3.5), améliorant les choix de placement en diffusant plus largement l'état des sites, engendre un trafic réseau moins important que l'algorithme des échanges à répétition tout en donnant une bonne connaissance de l'état du réseau.

Pour conclure, dans la mesure où nous ne souhaitons pas compliquer le développement du gestionnaire de charge en utilisant une technique différente pour l'exécution à distance et la migration, nous ne pouvons utiliser un échange des valeurs de charge basé sur la mise aux enchères. Nous écartons également l'échange avec le voisinage qui est destiné à un environnement particulier. Nous choisissons donc, pour limiter le trafic réseau, d'utiliser un échange basé sur les paires aléatoires, en envoyant une table de charge lorsque la paire est constituée. Nous utilisons une technique à temps fixe pour l'échange des valeurs de charge.

4.2.3 Echange des tables de charges

L'algorithme utilisé pour la construction d'une table est le suivant. A chaque pas de temps, un site envoie la moitié de sa table de valeurs à un site aléatoirement choisi. Sur le site récepteur, une nouvelle table est constituée à partir de la charge locale (dans la première entrée de la table) et en intercalant à la suite une valeur de l'ancienne table locale et une de la table reçue. Cet algorithme est illustré par la figure V.12. La première valeur de la table locale n'est pas prise en compte car elle représente la charge du site au pas de temps précédent.

Avec l'algorithme d'échange des tables de charge utilisé, une valeur est transmise à un autre site au premier pas de temps. Au deuxième pas de temps, ces deux sites envoient leurs tables, contenant cette valeur, à deux sites choisis aléatoirement. La valeur est donc connue d'au plus

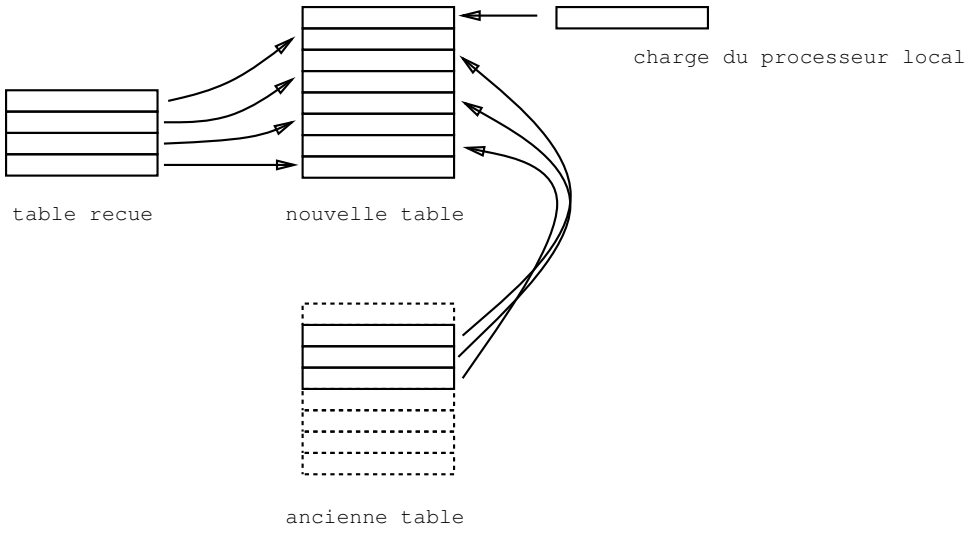

Figure V.12 : Evolution d'une table de charge

quatres sites. Ainsi, au pas de temps n, elle sera connue d'au plus 2^n sites. Cette valeur est donc diffusée rapidement. Or, plus une valeur est diffusée, plus elle est imprécise. Pour cette raison, le nombre d'entrées dans la table doit être limité pour conserver une bonne validité des valeurs.

4.2.4 Estimation de la charge

Comme la plupart des gestionnaires de charge décrits dans la littérature il nous semble intéressant d'utiliser la longueur de la file d'attente du processeur dans l'estimation de la charge. En effet cette valeur permet d'obtenir une bonne vue de la charge absolue - elle permet de savoir quelle part du processeur sera attribuée aux processus arrivant sur le site - et elle est facilement comparable avec celle des autres sites. Par contre, puisqu'elle est mesurée instantanément, elle ne traduit pas l'évolution du site. La connaissance du temps d'exécution restant est alors nécessaire. Nous utilisons la somme des résidus d'exécution des processus pour traduire ce temps. Cette somme nous semble plus intéressante que le lissage du nombre de processus dans la file d'attente, dans la mesure où elle cherche à approximer le futur proche sans hypothèse sur le passé. Le lissage cherche uniquement à réduire la portée d'éventuels comportements inhabituels mais ce sont, au contraire, ces comportements que nous devons traiter. Nous utilisons ces deux valeurs pour caractériser l'activité du processeur.

Dans un premier temps nous n'utilisons pas d'autres valeurs significatives :

– L'occupation de la mémoire peut facilement être obtenue auprès du système. Elle est vérifiée à chaque déplacement de processus. Par contre il ne nous semble pas intéressant de la mémoriser dans la mesure où la valeur obtenue traduit une occupation instantanée et doit à nouveau être vérifiée avant de déplacer le processus.
– La charge du réseau peut rarement être obtenue si le médium n'offre pas, lui-même, un tel

service (cf. 3.2). Ce service n'étant pas courant nous ne l'utilisons pas.
– Enfin, sur un multicalculateur, où tous les sites ne disposent pas de périphériques, il est difficile de prendre en compte la file des processus en attente d'entrées/sorties dans une évaluation de la charge puisque cette valeur ne sera pas significative sur tous les sites. De plus, beaucoup d'accès aux périphériques se font à travers le réseau donc peuvent être confondus avec des échanges de messages, ce qui rend plus difficile l'utilisation de cette valeur.

4.2.5 Estimation de la surcharge

En recevant une table de charge, le service d'ordonnancement réparti peut comparer sa charge à celle des autres sites et décider si elle est proportionnellement trop grande ou trop petite. Dans ce cas la technique du seuil de charge, simple ou multiple, n'est pas nécessaire puisque les seuils de charge sont utilisés comme des valeurs de références lorsque le site ne possède pas de valeurs pour comparer sa propre charge. Néanmoins nous utilisons également trois états pour le site : peu, modérement et fortement chargé. Ceci nous permet de ne pas recalculer la charge du site entre deux pas de temps.

4.2.6 Initialisation des échanges

Puisque nous avons choisi d'utiliser l'exécution à distance pour supporter la plus grosse part de l'équilibrage de charge, c'est l'émetteur du processus qui doit proposer l'envoi de processus aux sites moins chargés. Nous pouvons cependant utiliser une technique différente pour les processus qui doivent être migrés. En effet, en connaissant les valeurs de charge d'autres sites, nous pouvons choisir soit de proposer à un site surchargé de déplacer un processus sur notre site, soit de proposer à un site sous-chargé de lui envoyer de la charge. Les avantages pour chacune des techniques ont été présentés en 3.4. La technique d'initialisation par le récepteur ne semble pas, dans les simulations que nous avons présentées, fournir des résultats significativement meilleurs que l'initialisation par l'émetteur. Cependant nous pensons que la combinaison avec une technique d'initialisation par l'émetteur permet d'obtenir plus rapidement un équilibre de la charge : deux sites agissent alors pour l'équilibrage au lieu d'un. De plus le risque d'instabilité n'est pas accru dans la mesure où deux sites doivent s'accorder pour engager une migration. Dans un premier temps nous n'implantons qu'une intialisation par l'émetteur pour vérifier les idées de base. Nous désirons tester l'initialisation par le récepteur dans une seconde phase d'évaluation de différents algorithmes. Nous traitons donc par la suite des problèmes qui y sont liés.

4.2.7 Choix du site destinataire

Le site le moins chargé est celui qui possède moins de processus et un temps restant plus court que ceux des autres sites. Nous utilisons toujours le nombre de processus comme le critère principal de décision. En effet, avec l'équilibrage de charge, nous pouvons supposer qu'il évolue lentement. Il nous permet d'approximer, dans un futur proche, la part du processeur qui peut être attribuée à un processus. Nous utilisons le temps total d'exécution pour choisir entre deux sites ayant la même charge.

4.2.8 Choix du processus

Les paramètres de choix d'un processus sont totalement indépendants de l'algorithme d'échange des valeurs de charge. Nous pourrons donc facilement modifier ces paramètres s'ils s'avèrent mal choisis, sans influer sur les caractéristiques du gestionnaire de charge. La sélection du meilleur processus à migrer utilise les mêmes critères que le choix du site le moins chargé, c'est-à-dire le temps d'utilisation du processeur.

D'une manière générale les processus sont placés à leur création, sans utiliser de critères particuliers. Lorsque le système décide de migrer un processus, il choisit le processus le plus long puisque c'est celui qui a potentiellement le plus long résidu d'exécution (cf. 3.1). Nous devons donc conserver le temps d'exécution pour chaque processus. Ce choix peut poser deux problèmes si le système n'est pas très stable :

– les processus longs risquent d'être systématiquement déplacés au profit des processus courts, c'est-à-dire que, plus un processus veillit, plus il risque de passer son temps à être déplacé.
– un processus est considéré comme long par rapport aux processus qui s'exécutent en même temps que lui. S'il n'y a que des processus courts, nous pouvons être amené à migrer un processus qui n'en vaut pas la peine.

Cette constatation nous conduit à imposer un temps minimum d'exécution locale, au moins équivalent, au temps moyen de migration d'un processus. De plus cette contrainte nous permet d'assurer la stabilité du système.

4.2.9 Extensibilité et dispersion

Comme nous l'avons déjà dit, l'extensibilité est garantie par les échanges entre les gestionnaires de charges - de valeurs de charge ou de processus - qui n'impliquent jamais plus de deux sites à la fois. Ainsi le nombre de protagonistes est indépendant du nombre de sites connectés au réseau.

En ne déplaçant qu'un seul processus lorsqu'un déséquilibre de charge est constaté, nous contraignons les sites très chargés à placer leurs processus dont il se décharge sur des sites différents, donc à les disperser.

4.2.10 Stabilité

L'imprécision dans l'équilibrage de charge peut avoir des conséquences opposées : un surplus d'instabilité ou, au contraire, engendrer un meilleur comportement. Puisque nous avons choisi d'utiliser des heuristiques pour gérer la charge nous ne devons pas chercher à "trop optimiser" les algorithmes car cela peut être nuisible à la stabilité du système. Par exemple, nous avons vu que la migration simultanée de plusieurs processus, pour arriver plus rapidement à l'équilibre, entraîne des risques d'instabilité. Ainsi nous pensons que l'imprécision de la mise en oeuvre du serveur peut être un gage de sa stabilité.

La stabilité des algorithmes choisis est assurée de différentes façons :

– nous utilisons principalement l'exécution à distance pour répartir la charge : la migration ne sert qu'à déplacer des processus longs. Le nombre de déplacements est donc réduit.

– nous ne déplaçons qu'un seul processus à la fois sur un site donné pour éviter les transferts de charge trop importants entre les sites. De même, un site n'accepte qu'un processus à la fois. Ainsi l'équilibrage se fait plus lentement.
– avant le déplacement d'un processus nous vérifions si la différence de charge entre les sites impliqués justifie toujours ce transfert.
– les processus, une fois déplacés, sont tenus de s'exécuter localement pendant une durée minimale. Cette contrainte permet d'assurer aux processus qu'ils ne passent pas la totalité de leur temps en déplacement.

Nous ne pouvons cependant pas garantir, sans test, la stabilité de l'algorithme. Des mesures sont nécessaires pour vérifier les hypothèses posées.

4.3 Spécification d'un serveur

La description précédente donne les caractéristiques générales d'un serveur de charge destiné à un système à image unique. Nous n'avons pris en compte aucune contrainte propre au système d'exploitation ou à une machine. Nous décrivons dans cette partie l'intégration du serveur dans le système CHORUS/MiX. Notre souci a été de maintenir l'indépendance entre les différentes parties pour permettre de faire évoluer un tel serveur d'après les résultats obtenus à l'exécution.

La gestion de la charge pourrait facilement être intégrée dans le PM, permettant ainsi un accès aisé aux données décrivant les processus. Ceci est envisageable dans le cadre du développement d'un gestionnaire de processus dédié au système à image unique. Puisque le PM implante l'ordonnancement local des processus, il n'est en effet pas illogique, qu'en étendant sa gestion aux réseaux, il s'occupe également de l'ordonnancement réparti. Cependant il est plus facile de développer un acteur indépendant pour la mise au point. De même cette implantation, plus modulaire, doit permettre de tester d'autres algorithmes, sans modification fondamentale dans le protocole d'échange entre les deux serveurs.

La gestion de charge est donc implantée dans un acteur indépendant, intégré dans un sous-système UNIX, que nous nommons gestionnaire de charge ou LM (*Load Manager*). Le LM a plusieurs fonctions :

– mesurer la charge, recevoir et envoyer les tables de charge,
– obtenir des données sur les processus, faire le choix du processus à déplacer et ordonner au PM le déplacement d'un processus,
– fournir au PM les noms des sites les moins chargés pour exécuter à distance les processus nouveaux.

4.3.1 Les contraintes de CHORUS/MiX

Le système CHORUS/MiX offre deux possibilités pour créer un processus :

la primitive `fork(2)` Le processus créé est une duplication du processus père qui exécute cette primitive. Cette primitive ne peut être exécutée à distance, il n'y a donc pas de **duplication à distance**. Ceci est justifié car les processus exécutent généralement l'appel `exec(2)` après cette primitive.

la primitive `fexec(2)` Le processus créé hérite du contexte système du père. Cette primitive utilise la variable *csite* du contexte système du processus. Si le site désigné par *csite* est différent du site local, la primitive entraîne la **création à distance** d'un processus dont l'espace d'adressage est initialisé d'après les paramètres de l'appel. Le positionnement de la variable *csite* ne peut être réalisé que par le père.

Ces fonctions ne peuvent être exécutés que par le processus père. Le système offre également la possibilité de déplacer un processus après sa création :

la primitve `exec(2)` Cette primitive utilise la variable *csite* pour l'exécution à distance du processus. Elle entraîne l'**exécution à distance** d'un processus ayant le même contexte système mais dont l'espace d'adressage est initialisé à partir des paramètres de la primitive. L'exécution à distance d'un processus est alors équivalente à la création à distance obtenue avec la primitive `fexec(2)` puisque le processus distant n'hérite pas des données du processus local. Le positionnement de la variable *csite* ne peut être réalisé que par le père.

la primitive `migrate(2)` Cette primitive permet le déplacement d'un processus sur un site distant. L'identificateur du processus et du site sont des paramètres de l'appel. Un processus n'est pas obligatoirement migré par son père, l'appelant doit simplement posséder les droits nécessaires.

Le LM utilise ces possibilités pour répartir la charge sur le réseau. L'interface de ces primitives pose alors les problèmes suivants :

- des processus peuvent être créés, au moyen de la primitive `fork(2)`, sans pouvoir utiliser l'exécution à distance. Si une application exécute alors plusieurs `fork(2)` elle risque de surcharger le site. Ce problème devrait être compensé pas la migration de la surcharge.
- le gestionnaire de charge doit posséder des droits étendus pour utiliser la primitive `migrate(2)`. De plus certains processus ne peuvent être migrés, du fait de leurs caractéristiques. Il faut donc donner la possibilité au LM, pour lui permettre d'implanter une politique efficace, de s'assurer d'une migration et d'essayer un autre processus en cas d'erreur.
- le gestionnaire de charge ne peut modifier la variable *csite* des processus pour permettre la création ou l'exécution à distance. Mis à part le père du processus, seul le gestionnaire de processus (PM) est autorisé à modifier cette variable. Certaines modifications doivent alors être introduites dans le PM pour qu'il coopère avec le LM afin de positionner le site d'exécution.

Le calcul de la charge est basé sur la connaissance du nombre de processus présents sur le site et du temps écoulé dans chaque processus. Le gestionnaire de processus est à même de fournir de telles informations. Avec l'appel système `utime(2)`, le LM peut obtenir les temps d'exécution des processus. Par contre il n'existe pas d'appel système pour obtenir le nombre de processus s'exécutant actuellement sur le site. Nous devons donc définir un protocole de communication entre les deux serveurs.

4.3.2 Coopération avec le PM

La coopération entre le PM et le LM est basée sur des échanges par messages. La définition d'une interface fixe entre le PM et le LM nous permet de modifier facilement le gestionnaire de charge

sans implications sur le PM. Le LM utilise les services du PM pour obtenir des informations sur les processus et sur la charge du système. Le PM utilise les services du LM pour obtenir l'identificateur d'un site faiblement chargé, dans le cas de l'exécution distante.

Les services du PM Pour échanger les données concernant les processus et la charge du système nous avons plusieurs solutions. Il est en effet possible de faire effectuer plus ou moins de traitements au gestionnaire de processus :

Nombre de processus en attente Pour maintenir à jour cette donnée, le LM doit être tenu au courant de toutes les créations, mises en sommeil et destructions de processus. Cela implique beaucoup d'échanges de messages entre les deux serveurs. Le PM possède toutes les données nécessaires au calcul, il peut donc facilement maintenir cette information.

Temps d'exécution restant Ce calcul peut être fait par le LM en utilisant la fonction `utime(2)` sur chaque processus ou par le PM lors de la mise à jour des temps d'exécution pour chaque processus (variables *sysTime* et *userTime* de la structure *Proc*). La première solution est lente et coûteuse en calcul. C'est donc le PM qui maintient cette valeur.

Processus le plus long Pour tenir à jour l'identificateur du processus le plus long, le LM doit être informé de chaque allocation du processeur aux processus. Il doit ensuite maintenir des informations sur chaque processus et utiliser un algorithme de calcul du processus le plus long. Ceci est difficilement envisageable. C'est donc le PM qui est chargé de la gestion de cet identificateur.

Pour l'échange des données du PM vers le LM, nous définissons deux types de requêtes : GET_LOAD et MIG_PROC. La requête GET_LOAD rend au LM une structure composée du nombre de processus et du temps d'exécution total. La requête MIG_PROC rend le *Pid* du processus à migrer ou un code d'erreur qui rend compte du déroulement de la migration.

Les services du LM Le PM utilise les services du gestionnaire de charge pour obtenir l'identificateur du site sur lequel est exécuté un nouveau processus. Il envoie une requête au LM local pour obtenir l'identificateur du site. Nous définissons donc un type de requête (LIGHTLOADED_SITE) pour l'échange entre le PM et le LM.
Le LM et le PM s'accèdent réciproquement à travers des groupes statiques.

4.3.3 Coopération entre les LM

Les LM établissent entre eux un protocole d'échange des informations nécessaires à leur coopération :

Echange des tables La coopération entre les LM des différents sites s'inscrit principalement dans l'échange des valeurs de charge. A des instants fixés, le LM choisit aléatoirement un site distant. L'adressage entre les LMs est assuré au travers d'un groupe statique : un message est envoyé à la porte du LM du site choisi. Le message a alors le type NEW_TABLE.

Vérification de la charge Les LM coopèrent également pour vérifier leurs charges respectives avant de déplacer un processus (figures V.14 et V.15). Après avoir trouvé un site prêt à accueillir une surcharge, un LM envoie au gestionnaire de charge de ce site une requête, de type CHECK_LOAD, pour vérifier sa charge. Il le prévient d'une migration ou d'une exécution à distance imminente. La requête du LM émetteur contient la charge de son site. Ainsi le LM récepteur compare les deux charges et décide si la migration doit avoir lieu. Dès qu'il reçoit une requête CHECK_LOAD, un LM n'accepte plus de déplacement de charge avant la fin du traitement de la requête.

Pour assurer la tolérance aux pannes du LM nous ne faisons aucune supposition sur l'état des sites avec lesquels nous échangeons des données ainsi l'envoi de la table de charge n'est pas bloquant. Si le site choisi est en panne, l'envoi de la table est perdu. Le site doit attendre le pas de temps suivant pour faire parvenir ses données à un autre site. Ceci ne gêne pas l'exécution de l'algorithme de répartition de charge. De même, le LM ne suppose pas recevoir une table de valeur à chaque pas de temps. Il se met simplement en attente de réception d'une telle table. Si la table ne parvient pas au LM, celui-ci reste en attente de réception. Pour les échanges entre LM, tel que la vérification d'une valeur de charge, le temps d'attente d'une réponse est fixé. Si la réponse ne parvient pas au serveur, celui-ci revient au début de son traitement et ne tient pas compte de l'échec.

4.3.4 Structure du LM

Les fonctions traitées par le LM (recevoir les tables de charge, demander des données au PM, envoyer sa table, etc.) sont indépendantes les unes des autres, elles peuvent donc être traitées par plusieurs activités aux sein d'un même acteur. Le LM est donc un serveur multi-activités. Le LM possède une porte sur laquelle il reçoit indifféremment les requêtes LIGHTLOADED_SITE du PM et les tables de valeurs de charge.

Les données principales du gestionnaire de charge sont celles contenues dans la table des valeurs de charge. Le nombre d'entrées dans cette table peut être fixé en fonction du nombre de sites connectés au réseau. Il ne faut pourtant pas la surdimensionner car les informations qu'elle contient perdent de leur valeur : les dernières informations de la table seraient calculées depuis trop longtemps pour être encore significatives. Chaque entrée dans la table contient trois champs :
– l'identificateur du site,
– le nombre de processus s'exécutant sur ce site,
– le temps d'exécution supposé restant au site.
La valeur de la charge locale est mémorisée dans la première entrée de la table.

Le LM contient très peu de données en dehors de la table des valeurs de charge des autres sites. Le LM peut recevoir, à tout instant, une requête du PM lui demandant l'identificateur du site d'exécution d'un processus qui doit être créé. Pour y répondre, il doit être en mesure de savoir si le site est trop chargé ou non. Or à la réception d'une table de charge, le LM détermine son état de charge. Nous mémorisons donc cette information, dans une variable *loadState*, pour la réutiliser lorsque le PM envoie une requête LIGHTLOADED_SITE.

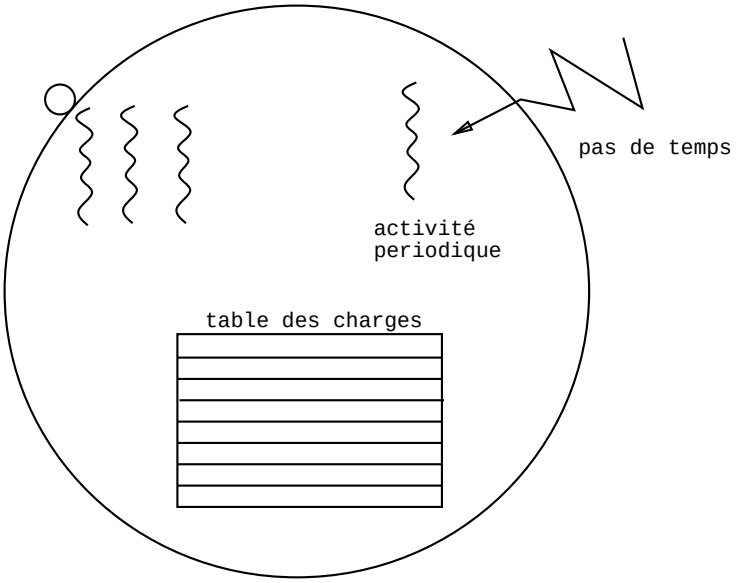

Figure V.13 : Structure du gestionnaire de charge

4.3.5 Algorithmes du LM

Le LM répond principalement à trois types de requêtes :

NEW_TABLE : issue d'un autre gestionnaire de charge ; elle est reçue à des instants fixes et entraîne la mise à jour de la table des valeurs de charge,

CHECK_LOAD : provient aussi d'un LM mais elle peut être reçue et émise à des instants quelconques,

LIGHTLOADED_SITE : émise par le PM, à des instants quelconques. Elle entraîne la recherche d'un site peu chargé.

D'autre part le LM doit s'activer périodiquement pour envoyer sa table de charge à un site aléatoirement choisi.

Traitement périodique

L'activité de traitement périodique est généralement endormie en attente des instants d'échange de la charge. Lorsqu'elle se réveille, elle demande au PM local les valeurs significatives de la charge du site pour compléter sa table de charge. La valeur locale est toujours mise en tête de table car cette valeur est considérée comme la plus à jour. L'activité choisit ensuite le site destinataire et construit un message à partir de la moitié de sa table de charge. Lorsque ce traitement est fini elle s'endort pour un temps fixé.

Notons que le calcul actuel de la charge est peu coûteux car les valeurs sont maintenues continuellement à jour par le PM. Par contre, pour pouvoir prendre en compte un plus grand nombre

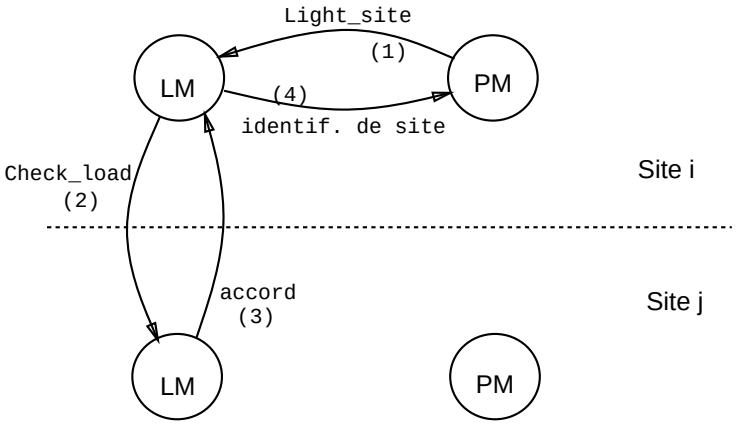

Figure V.14 : traitement d'une requête d'`exec(2)`

de paramètres, le LM doit éviter de demander systématiquement la charge au PM : il ne doit pas le faire à chaque requête. Nous supposons donc que la charge n'est pas significativement modifiée entre deux pas de temps et seul le traitement périodique met à jour la charge dans le LM.

La requête NEW_TABLE

Lorsque l'activité reçoit une table de charge, elle la mémorise en intercalant les valeurs qu'elle possède et celles qu'elle a reçues (figure V.12). En fonction de cette nouvelle table, l'activité détermine l'état du site et positionne la variable *loadState*. L'activité calcule également, d'après les valeurs reçues, l'identificateur du site le moins chargé dans la variable *lightSite* et celui du site le plus chargé dans *highSite*. D'autre part, si le site est dans l'état HIGHT, l'activité essaie de migrer un processus vers ce site peu chargé.

La variable *loadState* peut prendre trois valeurs : LIGHT, MODERATE et HIGHT. D'une manière générale nous essayons de réaliser le maximum de traitements lors de la réception de requêtes NEW_TABLE : le LM sélectionne et mémorise le site qui est le moins chargé. De cette façon, les réponses à la requête LIGHTLOADED_SITE sont rapides et ne pénalisent pas le traitement d'une exécution distante.

Requête CHECK_LOAD

Pour migrer un processus, l'activité contacte le serveur de charge du site le moins chargé, en envoyant une requête CHECK_LOAD. Elle vérifie, avec ce LM, que la différence de charge entre les deux sites justifie la migration d'un processus, c'est-à-dire que le site origine est dans l'état HIGH et le site destinataire dans l'état LIGHT. Sur le site origine, après avoir reçu un accord, le LM utilise l'appel système `migrate(`*pid*`)` pour déplacer un processus. La variable *loadState* est alors positionnée à BUSY. Si la migration d'un processus n'est pas possible, c'est-à-dire que la requête de migration rend une erreur, le LM envoye une nouvelle requête MIG_PROC au PM.

Lorsqu'une migration est acceptée les LM impliqués n'en acceptent pas d'autres tant que la

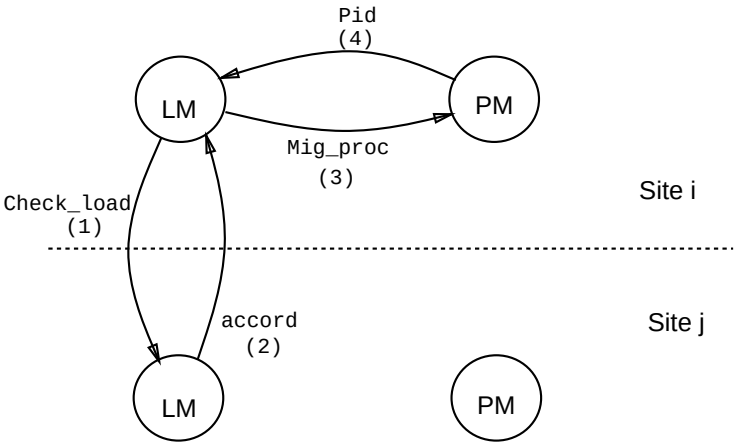

Figure V.15 : traitement d'une requête de migration

première n'est pas achevée et tant que ses conséquences ne sont pas connues. Donc aucune migration ne sera acceptée si la variable *loadState* du LM est positionnée à BUSY. Le problème est alors de savoir quand la requête est terminée puisque le LM émetteur n'est pas tenu au courant de la fin du transfert. Nous considérons que le site peut accepter une charge supplémentaire dès le prochain pas de temps. La variable *loadState* est repositionnée par l'activité périodique.

Requête LIGHT_LOADED_SITE

Le traitement de la requête LIGHTLOADED_SITE consiste à rendre au PM l'identificateur du site le moins chargé, si l'état du site est MODERATE ou HIGH. Si l'état du site est LIGHT le LM retourne l'identificateur du site local au PM.

Choix aléatoire d'un site

Le choix du site se fait dans un ensemble représentant la totalité des sites du réseau. Le nombre et les identificateurs des sites avec lesquels le LM coopère, c'est-à-dire la description de tous les sites du réseau, est fixé dans les données du LM. Néanmoins la numérotation des sites permet de calculer ces valeurs plutôt que d'en donner une description explicite. Nous définissons une fonction `getRandSite()` qui rend l'identificateur d'un site choisi aléatoirement, parmi les sites connectés au réseau. Cette fonction est dépendante du réseau sur lequel le LM s'exécute, puisqu'elle dépend de la numérotation des sites.

Cependant, Il est possible de définir dynamiquement les données de la fonction `getRandSite()` en établissant un protocole qui, au moment de l'initialisation des LM, leur permet d'échanger des informations sur chaque sites et de calculer leur nombre. L'implantation d'une telle fonction n'est pas réalisée actuellement.

4.3.6 Modifications du PM

L'implantation de l'équilibrage de charge dans le sous-système UNIX implique des modifications du PM : maintient des valeurs de charge, réponses aux requêtes du LM et demande du site d'exécution à chaque appel système `exec(2)`. Ce sont les activités du PM qui répondent aux messages du LM. Différentes modifications sont apportées au gestionnaire de processus afin de réaliser la gestion de charge :

- Le PM gère des variables de charge pour satisfaire au requêtes du LM :
  - la variable *load*, mise à jour au cours des traitements effectués sur les processus, permet de maintenir un indice de charge. Elle est composée du nombre de processus s'exécutant actuellement sur le site et du temps total d'exécution restant.
  - la liste des processus en cours d'exécution.
  - les variable *maxExecTime* et *maxExecPid* permettent d'identifier le processus le plus long. Le PM, à chaque mise à jour des variables de temps d'exécution des processus, compare la valeur obtenue à celle contenue dans une variable *maxExecTime*. Si cette valeur est supérieure à la variable de référence, la variable *maxExecPid* est mise à jour et le processus est placé en tête de la liste des processus en cours d'exécution. Lorsque le processus ayant le plus long temps d'exécution meurt il faut supprimer la tête de la liste.

  La mise à jour des variables de charge se fait dans chaque appel système par l'activité qui exécute l'appel.
- Le processus choisi lors du traitement de la requête MIG_PROC n'est pas forcément migrable. Pour éviter de ne migrer aucun processus, tant que le processus le plus long n'est pas migrable, il faut donner au LM la possibilité d'essayer un nouveau processus. Pour cette raison, lorsque le PM donne le *Pid* d'un processus au LM, il le supprime de la liste.
- Le traitement sur l'exécution distante du PM est modifié uniquement en demandant un identificateur de site au LM et en positionnant la variable *csite* pour le nouveau processus.

4.3.7 Gestion de charge sur l'iPSC/2

Le cas de l'iPSC/2 ne peut être pris comme une généralisation pour l'ensemble des multicalculateurs. Pour ajuster la gestion de charge nous devons donc la porter sur différentes plateformes. L'iPSC/2 nous sert néanmoins de première base de test. La plupart des suppositions faites pour l'implantation du LM doivent pouvoir être testées dans ce cas.

Notre machine de test ne possède ni disque, ni autres périphériques connectés aux noeuds. Seul le noeud 0 accède au disque du frontal. Dans cet environnement nous ne pouvons pas tester s'il est nécessaire de prendre en compte des paramètres tels que la proximité des périphériques ou des ressources. Les noeuds d'entrée/sortie sont dédiés à l'exécution des serveurs pour garder de bonnes performances. Les processus sont exécutés uniquement sur les noeuds de calcul ce qui réduit l'éventail des paramètres à prendre en compte. Notre plateforme de test dispose de 16 noeuds.

L'implantation actuelle de la migration de processus limite le nombre des processus qui peuvent être migrés. Cette limitation a forcément des conséquences sur le comportement de la gestion de

charge répartie. Pour tester le LM et effectuer des mesures de performances nous devons donc prendre une population de processus qui ne pose pas de problèmes à la migration. Nous utiliserons des applications développées spécialement pour mesurer les gains de performance obtenus par le gestionnaire de charge.

La fonction *getRandSite()* n'a pas besoin de contenir une description explicite des sites du réseau. Sur l'iPSC/2 ceux-ci sont numérotés à partir de 256 par pas de deux. La fonction *getRandSite* calcule un nombre aléatoire (entre 0 et 1). Elle le multiplie par 32 (16x2), y ajoute 256 et en prend la partie entière. Nous pensons que 8 entrées dans la table est correct pour un multicalculateur à 16 noeuds. Les valeurs les plus vieilles de la table datent de 3 pas de temps au maximum.

Nous avons choisi d'utiliser un pas de temps d'une seconde pour notre implantation. Cette valeur est celle qui est utilisée par Mosix. Nous espèrons tester différentes valeurs pour déterminer celle qui convient le mieux dans un cadre général. Suivant les résultats obtenus il pourra être envisageable de la faire évoluer dynamiquement en fonction de la charge des sites.

Conclusion

Nous avons essayé de donner une description globale des problèmes posés par la gestion de charge dans un environnement réparti, avant de décrire les solutions proposées dans la littérature. Cette étude nous a permis d'établir qu'il est difficile de juger exactement de la valeur d'un algorithme de décision dans la mesure où celui-ci est toujours très dépendant de valeurs intrinsèques au matériel et du type des applications qui sont prises en compte. En particulier, un algorithme de gestion de charge, destiné aux multicalculateurs, ne peut pas convenir à un réseau de stations de travail, sauf si celles-ci sont utilisées dans des conditions identiques.

Nous nous sommes ensuite intéressés à une solution plus particulièrement adaptée aux multicalculateurs, que nous implantons dans le cadre du système CHORUS/MiX. Les spécifications de cette solution, telles qu'elles ont été données, conduiront à une réalisation sur un iPSC/2.

Beaucoup de choix doivent encore être vérifiés avant d'affirmer que le service conçu est adapté aux multicalculateurs. Nous devons faire varier les paramètres du LM pour étudier la façon dont ils agissent sur la gestion de charge. Par exemple, nous pensons que le pas de temps n'a pas besoin d'être le même sur chaque site. Il peut même être choisi de façon aléatoire dans un intervalle. Ceci permettra d'éviter l'échange de messages simultané en provenace de tous les sites, qui risque d'engorger le réseau. Le nombre d'entrées dans la table de charge a également une incidence. Nous le ferons varier pour étudier le comportement des serveurs de charge.

La mise en place d'une politique hybride d'initialisation pour le déplacement de charge mérite d'être approfondie. Il est possible qu'elle génère peu d'avantages et une plus grande instabilité. Néanmoins les limites fixées (temps d'exécution local minimum) doivent permettre à cette politique d'obtenir un meilleur équilibrage. Nous désirons également tester d'autres algorithmes par rapport à celui qui est proposé. Par exemple, un échange des valeurs de charge, basé sur la répétition, est également adapté aux multicalculateurs. L'implantation d'un tel algorithme, dans un LM, sera réduite car nous pouvons utiliser des structures équivalentes.

Le gestionnaire de charge est actuellement vu comme un service particulier qui permet une

meilleure exécution des applications parallèles, en particulier en évitant aux processeurs de rester inactifs. La généralisation de l'utilisation de ce service - peut être en l'implantant dans le micro-noyau - permettra de le faire évoluer vers un véritable ordonnanceur réparti chargé de l'allocation des processeurs, comme un ordonnanceur centralisé est chargé de cette allocation sur une architecture centralisée. Pour cela il devra également fournir un éventail de services plus large. Par exemple, la gestion de priorités, l'ordonnancement de groupe ou encore le placement d'un processus sur un site précis.

Conclusion

Dans cette thèse nous avons étudié la potentialité des ystèmes d'exploitation répartis généraux à offrir une interface adaptée aux besoins des multicalculateurs. Nous avons montré que le système CHORUS/MiX permet de combler certaines des lacunes des systèmes natifs implantés sur ces ordinateurs mais que ses services n'offrent pas un niveau de transparence suffisant. Nous avons alors spécifié les besoins d'un système à image unique qui permet à l'utilisateur de voir un multicalculateur comme un uniprocesseur virtuel. La conception d'un service de migration de processus et d'un getsionnaire de charge répartis constituent un premier pas vers la définition d'un tel système.

Ce travail nous a permis de constater l'intérêt du système CHORUS/MiX dans le développement de tels services. En effet il permet, par sa conception, une intégration aisée de nouveaux services sans altération de l'existant. D'autre part la transaprence de son IPC à la répartition simplifie de nombreux problèmes dans la mesure où les traitements distants sont les mêmes que les traitements locaux.

Les premières expérimentations de gestion de charge répartie nous ont permis de mieux comprendre l'intèrêt et la puissance d'un tel service, indispensable à l'exploitation des multicalculateurs. Outre son effet sur les performances, il permet en effet à l'utilisateur de développer des applications parallèles sans se soucier de l'architecture sur laquelle elles seront exécutées et de garantir leur portabilité.

Nous avons mené une vaste étude bibliographique sur les algorithmes de gestion répartie de la charge et leur réalisation. Les contraintes n'étant pas les mêmes dans un environnement de staions de travail que sur les multicalculateurs, nous avons cerné les besoins spécifiques des seconds en ce domaine : extensibilité, capacité de dispersion et indépendance de la topologie du réseau.

D'un point de vue pratique, notre travail a été effectué sur iPSC/2 à seize noeuds, ce qui relativise nos conclusion sur le comportement du système CHORUS/MiX et des services que nous avons ajoutés. Cependant l'évolution des ordinateurs parallèles comprend, dès aujourd'hui, des machines dont l'ordre de grandeur est le millier de noeuds. Il est alors difficile d'extrapoler nos conclusions pour ce type de machines car l'extensibilité des algorithmes que nous avons utilisés ne peut être vérifiée qu'expérimentalement. de plus, il est difficile d'imaginer ce que représente la gestion d'une telle machine : nous en sommes actuellement à un stade très expérimental. Cependant il est certain que la maîtrise de la progrmmation de ces machines passe par des outils tels que la gestion de charge répartie ou une interface du type "système à image unique", dont nos travaux sont une première ébauche.

Annexe A

Portage de Chorus sur l'iPSC/2

1 L'environnement du portage

Lorsque nous avons commencé le portage [1], le système CHORUS était disponible entre autre sur des stations de travail basées sur un micro-processeur Intel 80386 du type PCAT (COMPAQ386). Ceci nous a permis de réutiliser toute une partie du code du système qui est dépendante du processeur : par exemple, la gestion de la mémoire ou le superviseur. Néanmoins, certaines parties dépendantes de la machine ne pouvaient être réutilisées car le matériel accédé par le processeur n'est pas le même (le contrôleur d'interruptions, par exemple).

Les versions système utilisées ont été successivement les versions 3.2 et 3.3 du noyau. Nous avons réalisé le portage sur les noeuds de l'iPSC2 installé à l'ONERA. La version du sous-système UNIX porté offre une interface SVR3.2.

2 Chargement de Chorus sur l'hypercube iPSC/2

Chaque noeud de l'iPSC/2 constituant un site indépendant, une instance du système d'exploitation doit s'exécuter sur chaque noeud. Dans le cas du système CHORUS nous devons donc avoir un noyau sur chaque noeud. La version du noyau CHORUS qui a servi de base au portage est prévue pour des machines possédant une unité d'entrée/sortie (disquette ou disque dur) sur laquelle est lu le binaire du système CHORUS pour le charger en mémoire puis l'exécuter. Aucun des noeuds de l'iPSC/2 ne possédait de tel périphérique au début du portage, de plus le micro-code de leur initialisation n'est pas prévu pour lire sur ce périphérique, si il existait. Le premier problème a donc été de modifier le mode de chargement du noyau CHORUS sur les noeuds. Afin de limiter les développements, le maximum de code issu du système natif NX, développé par Intel, a été réutilisé. Cette solution a été adoptée pour des raisons de rapidité, de simplicité et de savoir faire. Nous expliquons donc le chargement du système NX et la manière dont il a été réutilisé pour charger le noyau CHORUS.

1. Le portage a été réalisé en collaboration avec L.Philippe stagiaire de thèse à Chorus systèmes.

2.1 Chargement du système NX

Le système NX est constitué d'un seul fichier binaire exécutable. C'est donc ce fichier binaire qui est chargé sur tous les noeuds. La station maître possède un fichier dans lequel est décrite la configuration matérielle de l'iPSC/2 (nombre de noeuds, type de périphérique connecté à un noeud, type de coprocesseur flottant, etc) ce qui lui permet d'adapter le chargement du système NX à cette configuration. Trois programmes différents s'exécutent pour ce chargement du système d'exploitation sur les noeuds : un programme sur la station maître, un premier programme sur les noeuds, appelé *chargeur*, et le système NX. Le déroulement du chargement est entièrement dirigé depuis la station maître, il se fait en plusieurs étapes et, à chacune d'elles, les noeuds se synchronisent avec la station maître. L'USM est utilisée à la fois pour transmettre des données et pour synchroniser les noeuds avec la station maître.

(1) Arrêt et remise à zéro des noeuds. Les noeuds se mettent en attente de lecture sur la ligne USM pour charger le code en mémoire centrale.

(2) La station maître envoie le programme *chargeur* sur la ligne USM (la ligne USM est lue simultanément par tous les noeuds). Ce programme *chargeur* va servir à initialiser les structures logicielles de base nécessaires à la gestion du matériel et à ré-initialiser les cartes DCM des noeuds, dans les étapes suivantes. En effet ces cartes doivent être ré-initialisées après chaque arrêt de l'iPSC/2.

(3) Initialisation de la carte DCM du frontal, les noeuds n'interviennent pas.

(4) Initialisation des cartes DCM des noeuds. C'est une phase très délicate : il faut envoyer à la carte DCM une séquence longue de 32Ko dans un ordre bien défini car le matériel est très sensible. Les données proviennent d'un fichier qui est stocké sur le frontal. Ce fichier est envoyé par la station maître à tous les noeuds en utilisant l'USM. C'est le programme *chargeur* des noeuds qui, grâce à ces données, initialise les DCMs.

(5) Test des noeuds pour vérifier le fonctionnement des DCM. Pour cela les noeuds échangent entre eux des messages puis, lorsqu'un noeud a réussi à joindre tous ses voisins il envoie un message de synchronisation à la station maître par la ligne USM.

(6) Lorsque la station maître a reçu autant de messages de synchronisation qu'il y a de noeuds, elle lance les processus gérant la communication entre la station maître et les noeuds.

(7) Le binaire du système NX est formaté en message puis envoyé au noeud 0, en utilisant la ligne DCM. Le noeud 0 diffuse alors ce message à tous ses voisins, qui font de même. Ainsi le système est diffusé à tous les noeuds. Chaque noeud charge le message en mémoire pour exécuter le système. Lorsque le système est chargé les noeuds se synchronisent avec la station maître en utilisant la ligne USM.

(8) La station maître finit son initialisation : elle termine en lançant l'exécution des processus chargés de gérer l'IPSC/2 (gestion des fichiers, gestion des communication, ...).

Le programme de la station maître peut prendre en paramètre un binaire différent de NX (étape 7) pour le charger sur les noeuds. Il s'est avéré, après essais, que la taille maximum que l'on pouvait charger était insuffisante pour le binaire de CHORUS.

2.2 Chargement de CHORUS

Le programme de la station maître peut aussi prendre en paramètre un programme *chargeur* (étape 2) différent de celui utilisé pour NX. Malheureusement, à cette étape, les DCMs ne sont pas encore initialisés. Pour être sûr d'initialiser correctement les cartes de communication nous avons intégré le programme *chargeur* de NX au système CHORUS.

A la différence du chargement de NX qui est fait d'abord par l'USM puis par le DCM, la totalité du système CHORUS est envoyée sur les noeuds à travers la ligne USM, en même temps que le programme *chargeur*. Après l'initialisation des cartes DCMs, nous commençons l'exécution d'un code propre à CHORUS sans attendre un message sur la ligne DCM pour le chargement du binaire. Cela oblige à arrêter le programme de la station maître avant la fin de son exécution, en particulier avant qu'il n'envoie le binaire de NX aux noeuds.

2.3 Constitution de l'archive

Contrairement au système NX (et aux systèmes d'exploitation monolithiques) le système CHORUS/MiX n'est pas consitué d'un seul binaire. Il est composé du binaire du programme d'initialisation, du binaire du noyau CHORUS et des binaires des différents acteurs implantant CHORUS/MiX. Comme le noyau CHORUS ne gère pas de périphériques, il ne peut pas dynamiquement charger des acteurs. C'est pourquoi nous devons charger en même temps que le noyau et le programme d'initialisation, les acteurs que nous voulons exécuter au départ. Le code que nous chargeons sur les noeuds contient donc les binaires du noyau et de plusieurs acteurs, structurés en archive.

Une fois le système chargé, le noyau CHORUS a besoin d'informations décrivant l'archive afin de pouvoir la relire et exécuter les acteurs qui la composent. Nous définissons dans les données du binaire d'initialisation une structure, appelée *TcLoad*, où sont mémorisées les informations décrivant les exécutables du noyau et des acteurs : l'adresse et la taille du code, des données initialisées et non-initialisées (BSS). Ceci permet au noyau de connaître l'adresse à laquelle il exécute chaque acteur, la place mémoire qu'il doit réserver, etc.

Nous avons mis au point un utilitaire qui génère automatiquement une archive CHORUS lorsqu'on lui passe en paramètres les binaires du programme d'initialisation, du noyau et des différents acteurs à charger. Cet utilitaire recopie, dans les données du programme d'initialisation, le code, les données initialisées et non-initialisées du noyau et des différents acteurs. Grâce à la table des symboles du binaire d'initialisation on peut trouver l'adresse de la structure *TcLoad* et la mettre à jour pour chaque binaire. Ceci permet de construire l'espace virtuel nécessaire à l'exécution du noyau et de savoir où commencer à exécuter le noyau à l'étape d'initialisation. De même le noyau utilise ce tableau pour connaitre les acteurs à lancer et leur adresse. Pour les acteurs, nous ne recopions dans l'archive, que les code et les données : les données non initialisées sont allouées dynamiquement en temps d'exécution.

Remarque : Les binaires dont nous disposons après compilation sous UNIX sont au format COFF. Ce format décrit des fichiers exécutables auxquels le compilateur ajoute une entête. Cette entête mémorise les informations nécessaires au système pour exécuter le code contenu dans

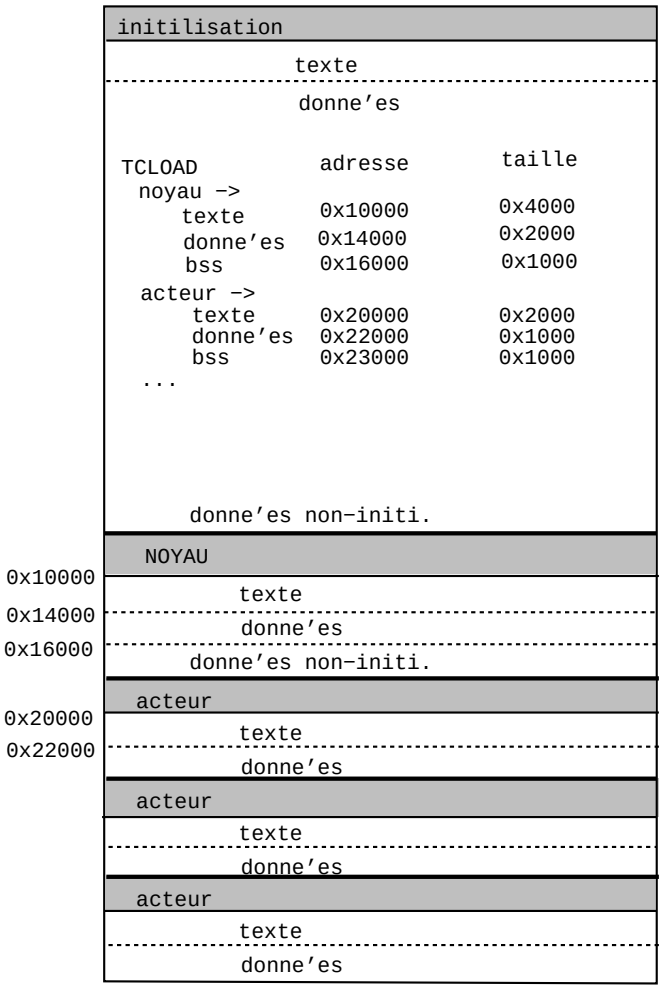

Figure I.1 : Structure d'une archive

le fichier. Elle contient une date de création, l'adresse dans le fichier du code et des données initialisées ainsi que leurs tailles et leurs adresses virtuelles, la taille des données non initialisées et l'adresse à laquelles elles doivent être allouées, etc. Nous utilisons les informations contenues dans cette entête pour compléter les champs de la structure *TCLoad*.

2.4 Constitution du binaire chargeable

L'archive, une fois constituée, n'est pas chargeable directement par l'USM. Il est nécessaire d'y inclure des caractères de synchronisation qui permettent de fiabiliser le chargement. D'autre part un petit protocole permet d'éviter le chargement de plus de trois caractères nuls consécutifs, ceci diminuant considérablement le temps de chargement des binaires. Les utilitaires NX générant le binaire chargeable utilisent un fichier au format COFF. C'est la raison pour laquelle tous les

binaires sont chargés dans les données du programme d'initialisation.

2.5 L'initialisation de CHORUS sur chaque noeud

Une fois que le binaire est dans la mémoire de chaque noeud, le processeur s'exécute automatiquement à l'adresse 0x00, adresse à laquelle est chargé le programme *chargeur* de NX (intégré au programme d'initialisation de CHORUS). Une fois l'initialisation des DCMs faite, les structures dépendantes du matériel et nécessaires à l'exécution du noyau CHORUS sont construites. Les tables de pages sont contruites en fonction de la taille de la mémoire centrale (4 à 16 Mo.) puis la pagination est activée. Une première ébauche de la gestion des interruptions est mise en place. Le programme d'initialisation construit les différents segments mémoire nécessaires à l'exécution du noyau et enfin la tâche du noyau (TSS) qu'il exécute.

3 Portage du noyau CHORUS et mise au point

Comme nous l'avons dit le noyau CHORUS que nous avons porté s'exécutait déjà sur un processeur Intel du type i386. Le travail de portage a donc été réduit pour cette partie puisque le code dépendant machine a pu être réutilisé. Néanmoins les particularités physiques des noeuds de l'iPSC/2 nous ont obligé à modifier les parties correspondantes dans le noyau.

3.1 Portage du noyau CHORUS

L'environnement mis en place par le programme d'initialisation permet d'offrir au noyau CHORUS la base nécessaire à son exécution. Néanmoins nous avons dû modifier la gestion des interruptions pour prendre en compte de nouveaux matériels tels que le DCM et l'USM. Cela se traduit par une modification de la table des interruptions du système : les entrées correspondant aux disques ou autres périphériques disparaissent au profit d'entrées pour le DCM et l'USM. D'une manière plus générale, nous sommes intervenus sur les parties du noyau qui étaient dépendantes du matériel annexe (horloge, ports d'entrée/sortie, etc.) plutôt que du processeur.

La configuration assez particulière de l'iPSC/2 par rapport à une station de travail a nécessité que les primitives d'affichage soient modifiées. Nous avons implanté les fonctions *printf* et *scanf* pour permettre l'affichage de traces d'une manière standard. Ce n'est normalement pas le rôle du noyau de gérer l'affichage puisque celui-ci doit se libérer d'un maximum de traitement matériel autre que celui de la mémoire et du processeur. Cependant, dans sa phase de mise au point le noyau ne peut se passer de gérer au moins un écran et un clavier pour permettre une interactivité avec l'utilisateur. C'est pourquoi le noyau implante les deux appels systèmes *printf()* et *scanf()*. Dans notre cas ces deux fonctions sont basées sur les fonctionnalités de l'USM. Nous traitons ceci plus en détail dans le paragraphe relatif au devéloppement d'outils de mise au point.

3.2 Environnement de mise au point

La mise au point est une partie très délicate d'un travail de portage de système. En effet, toute fausse manoeuvre est fatale en ce sens qu'elle engendre des erreurs qui entraînent souvent l'arrêt du processeur. Aucune information n'est alors disponible sur les données qui ont engendré l'arrêt, puisque leur adresse est perdue à ce moment là. Le travail de mise au point en est d'autant plus compliqué. Le moyen le plus simple, pour trouver les erreurs est alors l'affichage de traces d'exécution sur un terminal. L'IPSC/2 ne dispose pas de terminal sur les noeuds, nous avons donc eu recours aux leds équipant chaque noeud.

3.2.1 Avec les LEDs

Notre premier outil de mise au point a donc été un stylo composé d'un phototransistor et d'un amplificateur de signal relié à l'entrée RS232 d'une console. L'affichage des traces se fait alors en les envoyant, caractères par caractères sur une led : les codes ASCII des caractères sont envoyés bit à bit sur la led qui s'allume ou s'éteint en fonction de la valeur du bit. Le signal est alors capté par le stylo puis transmis à la console qui affiche la trace. Cet outil nous a permis de mettre au point l'initialisation de CHORUS.

3.2.2 Avec l'USM

Une fois la gestion des interruptions mise au point nous avons pu utiliser l'USM pour afficher nos traces sur la console de la station maître et lire des caractères depuis les noeuds, donc utiliser le debogueur du noyau. Physiquement tous les noeuds et la station maître peuvent écrire sur la ligne USM, mais il y a un risque de collision si deux émetteurs écrivent en même temps sur la ligne. Il n'y a pas de moyen annexe pour savoir si quelqu'un émet sur la ligne il faut donc mettre en place un protocole qui assure l'accès à l'USM.

Lorsque la station maître envoie un caractère sur la ligne de l'USM il est lu par tous les noeuds, ce qui a motivé le développement d'un protocole centralisé. La station maître choisit le noeud avec lequel elle communique. Le choix peut être réalisé par l'utilisateur grâce à un utilitaire, appelé *kt*, de communication avec l'USM. Le frontal envoie alors, sur la ligne USM, un caractère de contrôle signalant la sélection d'un noeud puis le numéro du noeud sélectionné. Le noeud est alors le seul à pouvoir émettre sur la ligne USM. Mais il y a toujours risque de collision entre l'émission sur le noeud et émission sur la station maître. Les protagonistes mettent alors un autre protocole en place pour leurs échanges de données : chaque réception d'un caractère est suivie de l'envoi d'un caractère de contrôle. Pendant ce temps les autres noeuds se mettent en attente de lire un nouveau caractère de contrôle annonçant une nouvelle sélection sur l'USM. Les noeuds ne pouvant émettre sur l'USM écrivent dans un tampon circulaire qui est envoyé lorsqu'ils sont selectionnés. Ceci permet de dialoguer avec chaque site indépendamment depuis le frontal, voir à distance depuis une station de travail.

Ce moyen d'affichage et de mise au point reste assez lent si on le compare aux moyens disponibles sur les stations de travail. Il nous a tout de même permis de mener à bien le portage du noyau et des premiers serveurs.

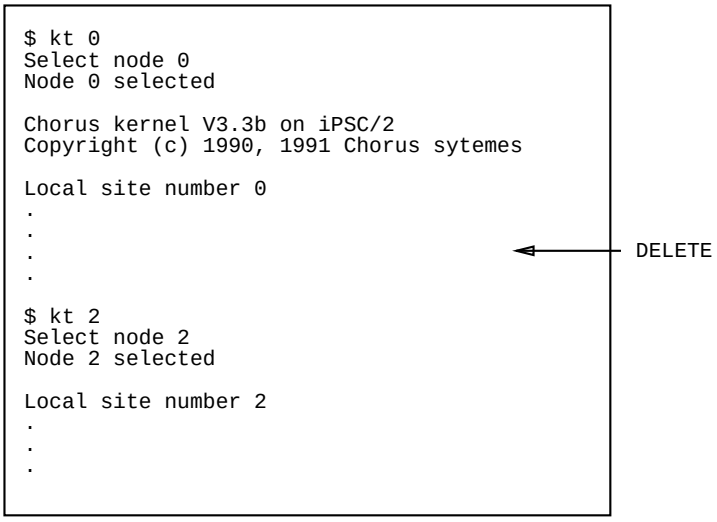

Figure I.2 : Séquence de communication avec l'USM

Annexe B

Comparaison des performances de communication

Dans les tableaux suivants, les temps sont donnés en millisecondes pour des messages de type RPC ayant la même taille en envoi et en reception. Les temps correspondent aux temps moyen obtenus pour mille échanges de message et les tailles sont données en octets.

Le tableau I correspond au cas optimum pour les deux systèmes : la taille et l'adresse des messages sont alignées sur des frontières d'entiers. Les mesures sont effectuées entre deux noeuds voisins.

Taille du message	**NX/2**	**Chorus**	**Rapport**
12	745	2400	3,2
500	1775	4150	2,3
1000	2128	4950	2,3
4096	4340	9600	2,2

Tableau I - Messages alignés sur des frontières d'entiers

Ce tableau montre que notre implantation de la communication sur l'iPSC/2 est moins performante que celle de NX/2. La principale raison de cette différence tient au niveau de service offert par les systèmes. En effet, lors de l'envoi d'un message CHORUS nous précisons uniquement l'identificateur de la porte destinatrice et nous ne tenons pas compte de sa localisation. Par contre, dans NX, le numéro de site doit être précisé à l'envoi du message. De même le système NX n'offre pas des services tels que la diffusion sur un groupe, l'accès en mode fonctionnel, etc.

La mesure des mêmes performances dans le cas de noeuds distants (par exemple les noeuds 0 et 15) montre une légère dégradation des performances, de l'ordre de 1% pour la communication de CHORUS entre 5 et 10 % pour NX/2. Il est possible que cette différence provienne de la différence de service offert par les deux systèmes.

Le tableau II compare les performances de la communication des deux systèmes dans des circonstances moins favorables : les adresses des messages sont volontairement fixées de manière à ne pas être alignées sur une frontière d'entier (de même pour la taille).

Taille du message	**NX/2**	**Chorus**	**Rapport**
101	1898	3550	1,9
501	2592	4250	1,6
4095	7513	9600	1,3

Tableau II - Messages non alignés sur des frontières d'entier

Dans ce tableau les différences de performances entre les deux systèmes sont moins importantes. Ceci peut être justifié par le fait que notre implantation de la communication n'a pas été optimisée.

Les différences de performances obtenues entre les deux systèmes laisse suposer qu'une version optimisée de la communication sous CHORUS pourrait satisfaire les utilisateurs des mutlicalculateurs en offrant un débit raisonnable et un confort supplémemtaire.

Bibliographie

[ABBa86] Mike Accetta, Robert Baron, William Bolosky, and all. Mach : A new kernel foundation for unix development. In *Proc. of USENIX Summer'86 Conference*, pages 93–112, Atlanta, GA, 9-13 June 1986. USENIX Association. Z/0771.

[ACF87] Yeshayahu Artsy, H-Y Chang, and Raphael Finkel. Interprocess communication in charlotte. *IEEE software*, pages 22–28, January 87.

[AC88] Rafael Alonso and Luis L. Cova. Sharing jobs among independently owned procesors. In *Proc. of the 8th International Conference on Distributed Computing Systems*, pages 282–288, San Jose, CA, Juin 88. IEEE Computer Society.

[AF89] Yeshayahu Arsty and Raphael Finkel. Designing a process migration facility ; the charlotte experience. *Computer*, 22(9) :47–56, September 1989.

[AGHR89] Francois Armand, Michel Gien, Frédéric Herrmann, and Marc Rozier. Revolution 89 or "distributing unix brings it back to its original virtues". In *Pro. of Workshop on Experiences with Building Distributed (and Multiprocessor) System"*, pages 153–174, Ft. Lauderdale, Oct. 5-6 1989. USENIX.

[AHP91] Francois Armand, Bénédicte Herrmann, and Laurent Philippe. Dynamic reconfiguration in chorus/mix. Puma deliverable, Chorus systèmes, October 1991.

[ARG89] Vadim Abrossimov, Marc Rozier, and Michel Gien. Virtual memory management in chorus. In *Lecture Notes in Computer Sciences*, page 20, Berlin, Germany, 18-19 April 1989. Springer-Verlag.

[Arm90] Francois Armand. Mix svr4. Technical report, Chorus Systèmes, St Quentin en Yvelines, France, Juillet 1990.

[Arm91] Francois Armand. Offrez un processus à vos drivers ! In *Proc. of AFUU'91 Conference*, pages 15–32, Paris, France, January, 91. AFUU.

[ARS89] Vadim Abrossimov, Marc Rozier, and Marc Shapiro. Generic virtual memory management for operating system kernels. In *12th ACM Symposium on Operating Systems Principles*, page 27, Litchfield Park, Arizona, December, 1989. ACM Press.

[Bab91] Ozalp Babaoglu. Fault tolerance support in distributed systems. *Operating Systems Review*, 25(1) :83–84, January 1991.

[BB88] Hans-Jorg Beier and Thomas Bemmerl. Software monitoring of parallel programs. In *Proc. of COMPAR 88*, page 8, Manchester, UK, September 1988.

[BB90] Joffroy Beauquier and Béatrice Bérard. *Systèmes d'exploitation, concepts et algorithmes*. McGraw-Hill, Paris, France, 1990.

[BBHL90] Thomas Bemmerl, Arndt Bode, Olav Hansen, and Thomas Ludwig. A testbed for dynamic load balancing on distributed memory multiprocessors. Technical report, Technical University Munich, August 1990.

[BBMa87] Jean-Pierre Banâtre, Michel Banâtre, G. Muller, and all. Quelques aspects du système gothic. *T.S.I.*, 6(2) :170–174, May 1987.

[BF81] Raymond M. Bryant and Raphael A. Finkel. Analysis of three dynamic distributed strategies with varying global informations requirements. In *Proc. of 2th International Conference on Distributed Computing Systems*, pages 314–323, Paris, FR, Mars, 1981. IEEE Computer Society.

[BFS89] F. Bonomi, P.J. Fleming, and P.D. Steinberg. Distributing processes in loosely-coupled unix multiprocessor systems. In *Proc. of Summer 1989 USENIX Conference*, pages 61–72, Baltimore, MD, June 12-16, 1989. The Usenix Association.

[Bir91] Andrew Birell. Position paper for sigops workshop on fault-tolerance support in distributed systems. *Operating Systems Review*, 25(1) :83–84, January 1991.

[BJMa91] Michel Banâtre, Ph. Joubert, Ch. Morin, and all. Stable transactionnal memories and fault tolerance architectures. *Operating Systems Review*, 25(1) :68–72, January 1991.

[BL90] Allan Bricker and M.J. Litzkow. Condor technical summary. Technical report, University of Wisconsin, WI, 1990.

[Bla90] David L. Black. Scheduling and resource management techniques for multiprocessors. Thesis, Carnegie Mellon University, Pittsburgh, PA, July, 1990.

[BP86] David A. Butterfield and Gerald P. Popek. Networking tasking in the locus distributed unix system. Technical report, Locus Computing Corp., January 1986.

[BP86] Amonon Barak and On G. Paradise. Mos - a load balancing unix. In *EUUG Autumn'86*, pages 273–280, Manchester, UK, 24 September 86. EUUG.

[BS91] Guy Bernard and Michel Simatic. A decentralized and efficient algorithm for load sharing in networks of workstations. In *Proc. of EurOpen Spring 1991 Conference*, pages 139–148. EurOpen, 20-24 May, 1991.

[BSS90] Kenneth Birman, Andre Schipper, and Pat Stephenson. Fast causal mutlicast. Ccs technical report tr-1105, Cornell University, Ithaca, NY, April 13 1990.

[BSS91a] Guy Bernard, Dominique Steve, and Michel Simatic. Placement et migration de processus dans les systèmes rèpartis faiblement couplés. *Technique et Sciences de l'informatique*, 10(5) :375–392, November 1991.

[BSS91b] Kenneth Birman, Andre Schipper, and Pat Stephenson. Lightweight causal and atomic group multicast. Ccs technical report tr-91-1192, Cornell University, Ithaca, NY, February 1991.

[BS85] Amnon Barak and Amnon Shiloh. A distributed load-balancing policy for a multicomputer. *Software Practice and Experience*, 15(9) :901–913, July 85.

[BW89] Amonon Barak and Richard Wheeler. Mosix : An integrated multiprocessor unix. In *Proc. of 1989 Winter USENIX Conference*, pages 101–112, San Diego, CA, January 30-February 3, 1989. Usenix Association.

[CA86] Timothy C. Chou and Jacob A. Abraham. Distributed control of computer systems. *IEEE Transactions on Computers*, C-35(6) :564–567, June 1986.

[Cab86] Luis-Felipe Cabrera. The influence of workload on load balancing strategies. Technical report, IBM Almaden reseach center, August 1986.

[CA82] Timothy C. Chou and Jacob A. Abraham. Load balancing in distributed systems. *IEEE Transactions on Software Ingineering*, SE-8(4) :401–412, July 82.

[CH87] Thomas L. Casavant and Jon G. Huhl. Analysis of three dynamic distributed strategies with varying global informations requirements. In *Proc. of 7th International Conference on Distributed Computing Systems*, pages 185–192, Berlin, DE, September, 1987. IEEE Computer Society.

[CH88a] Thomas L. Casavant and Jon G. Huhl. A taxonomy of scheduling in general-purpose distributed computing systems. *IEEE Transactions on Software Ingineering*, SE-14(2) :141–154, February 1988.

[CH88b] Thomas L. Casavant and Jon G. Huhl. Effects of reponse and stability on scheduling in distributed computing systems. *IEEE Transactions on Software Ingineering*, SE-14(11) :1578–1588, November 1988.

[Che88] David R. Cheriton. The v distributed system. *Communications of the ACM*, 31(3) :314–333, March 1988.

[Clo88] Paul Close. The IPSC/2 node architecture. Technical report, Intel Scientific Computers, Portland, OR, june 88.

[Com88] Douglas Comer. *Internetworking with TCP/IP ; Principles, Protocols and Architecture*. Prentice Hall, West Lafayette, IN, 1988.

[Cra91] Cray Research, Inc., Eagan, MN. *The* CRAY Y-MP *Supercompter System*, 1991.

[Des91] Yves Deswarte. Tolérance aux fautes, sé'curité et protection. In *Construction des systèmes d'exploitation répartis*, chapter 9. INRIA, 1991.

[DO91] Fred Douglis and John Ousterhout. Transparent process migration : Design alternatives and the sprite implementation. *Software Practice and Experience*, 21(8) :757–785, Augusr 91.

[DTJ89] Piyush Diskhit, Satish K. Tripathi, and Pankaj Jalote. Sahayog : A test bed for evaluating dynamic load-sharing policies. *Software Practice and Experience*, 19(5) :411–435, May 1989.

[Eck90] Horst Eckardt. Investigation of distributed disk I/O concepts. Puma deliverable no 2.3.1, Siemens AG, Munich, Germany, Augusr 1990.

[ELZ85] Derek L. Eager, Edward D. Lazowska, and John Zahorjan. A comparison of receiver-initiated and sender-initiated adaptive load sharing. In *ACM SIGMETRICS 1985*, pages 53–68, Austin, Texas, 1985. Elsevier Science Publishers B.V.

[ELZ86] Derek L. Eager, Edward D. Lazowska, and John Zahorjan. Adaptive load sharing in homogeneous distributed systems. *IEEE transactions on software engineering*, SE-12(5) :662–675, May 1986.

[ELZ88] Derek L. Eager, Edward D. Lazowska, and John Zahorjan. The limited performance benefits of migrating active process for load balancing. In *Performances Evaluation Review*, volume 16, pages 63–72. ACM, May 1988.

[Esk91] Rasit Eskicioglu. Design issues of process migration facilities in distributed systems. Technical report, University of Alberta, 1991.

[Fly74] M.J. Flynn. Directions and issues in architecture and language. *IEEE computer*, 13(10) :948–960, October 1974.

[Fou90] Open Software Foundation. Multi-server interface. Technical report, OSF and CMU, Cambridge, MA, November 1990.

[Fou91] Open Software Foundation. The osf/1 operating system. In *EurOpen '91*, pages 33–41, Tromso, Norway, May 20-24 1991.

[FR89] Bertil Foliot and Michel Ruffin. Gatos : gérant de tâches dans un système distribué. Rapport technique 263, Laboratoire MASI, Universite P. et M. Curie, Janvier, 1989.

[Gar87] N. H. Garnett. Helios - an operating system for the transputer. In *Proc. of OUG-7, 7th occam User Group Technical Meeting*, pages 411–419, Grenoble, France, 14-16 September 1987. IOS press.

[GB90] Andrzej Goscinski and Mirion Bearman. Resource management in large distributed systems. *Operating Systems Review*, 24(4) :7–45, October 1990.

[GG91] Prabha Gopinath and Rajiv Gupta. A hybrid approach to load balancing in distributed systems. In *Proc. of SEMDS II Symp. on Experiences with Distributed and Multiprocessor Systems*, pages 133–147, Atlanta, GA, March 21-22, 1991. The Usenix Association.

[GHP90] Michel Gien, Bénédicte Herrmann, and Laurent Philippe. Unix single system image on a multi-computer. Puma deliverable, Chorus systèmes, July 1990.

[GV91] Andrew S. Grimshaw and Virgilio E. Vivas Jr. Analysis of three dynamic distributed strategies with varying global informations requirements. In *Proc. of SEMDS II Symp. on Experiences with Distributed and Multiprocess or Systems*, pages 149–164, Atlanta, GA, March 21-22, 1991. The Usenix Association.

[Hac89] Anna Hac. Load balancing in distributed systems : A summary. *Performance Evaluation Review*, 16(2) :17–19, February 1989.

[HAR+88] Frederic Herrmann, Francois Armand, Marc Rozier, Michel Gien, and all. Chorus, a new technology for building unix systems. In *Proc. of EUUG Autumn'88 Conference*, pages 1–18, Cascais, Portugal, 3-7 October 1988. EUUG.

[Hil88] Daniel Hillis. *La machine à connexions*. Masson, Paris, France, November 1988.

[HJ87] Anna Hac and Xiaowei Jin. Dynamic load balancing in a distributed system using a decentralized algorithm. In *Proc. of the 7th International Conf. on Distributed Computing Systems*, pages 170–177, Berlin, Germany, September, 1987. IEEE Computer Society.

[HLMB90] Olav Hansen, Thomas Ludwig, Roy Milner, and Sally Baker. Load balancing strategies. Technical report, Technical University Munich, December 1990.

[HP88] Bénédicte Herrmann and Laurent Philippe. Un serveur de tubes unix pour chorus. In *Proc. of EUUG Autumn'88 Conference*, pages 1–18, Cascais, Portugal, 3-7 October 1988. EUUG.

[HP91] Bénédicte Herrmann and Laurent Philippe. Multicomputers UNIX based on CHORUS. In *EDMCC2, Distributed Memory Computing*, pages 440–449, Munich, Germany, April, 1991. LNCS 487.

[HP92a] Bénédicte Herrmann and Laurent Philippe. UNIX on a multicomputer : the benefits of the CHORUS architecture. In *Transputer's 92*, pages 142–157, Arc et Senans, FR, May 20-22, 1992. IOS Press.

[HP92b] Bénédicte Herrmann and Laurent Philippe. Building a distributed UNIX for multicomputer. In *PACTA'92*, pages Part1, 663–671, Barcelonna, SP, september, 1992. IOS Press.

[Inm89] Inmos, Bristol, UK. *The Transputer Databook*, 1989.

[Inm91] Inmos, Bristol, UK. *The* T9000 *Transputer, Product Overview*, 1991.

[Int87a] Intel Scientific Supercomputers Division, Portland, OR. iPSC/2 *System, Product and Market Information*, August 87.

[Int87b] Intel Scientifique Supercomputers Division, Portland, OR. *Paragon* XP/S*, product overview*, August 87.

[JLHB88] Erice Jul, Henry Levy, Norman Hutchinson, and Andrew Black. Fine-grained mobility in the emerald system. *ACM transactions on Computer Systems*, 6(1) :109–133, February 1988.

[JV88a] Wouter Joosen and Pierre Verbaeten. Design issues in load balancing techniques. Technical report cw 81, Katholieke Universtseit Leuven, April 1988.

[JV88b] Wouter Joosen and Pierre Verbaeten. On the use of process migration in distributed systems. Technical report cw 83, Katholieke Universtseit Leuven, November 1988.

[JVV91] Wouter Joosen, Bruno VandenBorre, and Pierre Verbaeten. Design and implementation of an experimental laod balancing environment. In *Proc. of EurOpen Spring 1991 Conference*, pages 123–138, Tromso, Norway, May 20-24, 1991. EurOpen.

[KL88] Philip Krueger and Miro Livny. A comparison of preemptive and non-preemtive load distributing. In *8th Internatinnal Conference on Distributed Computing Systems*, pages 123–130, San Jose, CA, Juin 88. IEEE Computer Society.

[Kra87] Sacha Krakowiak. Introduction aux systèmes informatiques répartis. In *Principes des systèmes d'exploitation des ordinateurs*, chapter 11. Dunod informatique, mai 1987.

[Kra91] Sacha Krakowiak. Communication et machines parallèles. In S.Krakowiak R. Balter J.-P., Banâtre, editor, *Construction des systèmes d'exploitation répartis*, chapter 2, pages 2–14,2–2–20. INRIA, 1991.

[LH89] Kai Li and Paul Hudak. Memory coherence in shared virtual memory systems. *ACM Transactions on Computer Systems*, 7(4), November 1989.

[LK87] Franck C. H. Lin and Robert M. Keller. The gradient model load balancing method. *IEEE Transactions on Software Ingineering*, SE-13(1) :32–38, January 1987.

[LK86] Franck C. H. Lin and Robert M. Keller. Gradient model : a demand-driven load balancing scheme. In *6th international Conference on Distributed Computing Systems*, pages 329–336, Cambridge, MA, May, 86. IEEE Computer Society.

[LMKQ89] S.J. Leffer, M.K. McKusick, M.J. Karels, and J.S. Quaterman. *The Design and Implementation of the 4.3BSD Unix Operating System.* Addison-Wesley Publishing Co., 1989.

[Lo84] Virginia M. Lo. Load-balancing heuristics and process behavior. In *4th international Conference on Distributed Computing Systems*, pages 30–39, San Fransisco, CA, May, 1984. IEEE Computer Society.

[LO86] Will E. Leland and Teunis J. Ott. Load-balancing heuristics and process behavior. In *ACM SIGMETRICS 1986*, pages 54–69, Raleigh, NC, 30 May, 1986. ACM.

[LP91] Zakaria Lahjomri and Thierrry Priol. Koan : a shared virtual memory for the ipsc/2 hypercube. Technical report, IRISA, Rennes, France, Juillet 1991.

[Mer89] Philippe Mercier. *La maîtrise du MS/DOS et du BIOS.* Marabout service, 1989.

[Mig92] Serge Miguet. *Redistribution élastique pour équilibrage de charge en imagerie*, chapter 16, pages 229–240. ??, 1992.

[MPV91] Dejan S. Milojicic, Milan Pjevac, and Dusan Velasevic. Load balancing survey. In *Proc. of the Autumn 1991 EurOpen Conference : Distributed Open Systems in Perspective*, pages 157–172, Budapest, Hungary, September 16-20, 1991. EurOpen.

[MT84] Sape J. Mullender and Andrew S. Tanenbaum. Protection and resource control in distributed operating systems. *Computer Networks*, 8 :421–432, 1984.

[Mul87] S. J. Mullender. *The Amoeba Distributed Operating System : Selected papers 1984 - 1987.* CWI tract 41, Amsterdam, 1987.

[Mul89] S.J. Mullender, editor. *Distributed Systems.* ACM Press, New York, NY, 1989.

[nCU90] nCUBE, Beaverton, OR. nCUBE 2 *Supercomputers, systems*, 1990.

[Nel81] Bruce Jay Nelson. *Remote Procedure call.* PhD thesis, Carnegie-Mellon University, Pittsburgh, Pa, 1981.

[nMKS91] Holger Herzog nad Markus Kolland and Juergen Schmitz. Evaluation of distributed operating systems in open network. In *EurOpen'91*, Tromso, Norway, May 20-24, 1991.

[NXG85] Lionel M. Ni, Chong-Wei Xu, and Thomas B. Gendreau. A distributed drafting algorithm for load balancing. *IEEE Transactions on Software Ingineering*, SE-11(10) :1153–1161, October 1985.

[Ort91] Maria Inés Ortega. Distributed shared memory and unix : Experimentation in the chorus/mix system. Technical report, Chorus Systemes, September 1991.

[Ort92] Maria Inés Ortega. *Mémoire virtuelle répartie au sein du système* CHORUS. PhD thesis, Université de Paris 7, Paris, FR, 1992.

[Par91] Parsytec Computer GmbH, Aachen, Germany. *Beyond the supercomputer, Parsytec GC*, 1991.

[Pie06] Paul Pierce. The nx/2 operating system. Technical report, Intel Scientific Computers, Portland, OR, 88/06.

[PM83] Michael L. Powell and Barton P. Miller. Process migration in demos/mp. *Operating Systems Review*, 17(3) :110–119, November 1983.

[PPTa90] Rob Pike, Dave Presotto, Ken Thompson, and all. Plan 9 from bell labs. *EUUGN*, 10(3) :2–11, Autumn 1990.

[PPTa91] Dave Presotto, Rob Pike, Ken Thompson, and all. Plan 9, a distributed system. In *EurOpen '91*, pages 43–50, Tromso, Norway, May 20-24 1991.

[PR91] Eric Pouyoul and Ellen Reier. Integrating x-kernel into chorus - first draft. Note technique chorus systèmes - unisys, Chorus systèmes, St Quentin en Yvelines, France, Février 1991.

[PTS88] Spiridon Pulidas, Don Towsley, and John A. Stankovic. Imbedding gradient estimators in load balancing algorithmes. In *Proc. of 8th International Conference on Distributed Computing Systems*, pages 482–490, San Jose, CA, June, 1988. IEEE Computer Society.

[RA90] M. Rozier and V. Abrossimov. Chorus kernel v3.3 programmer's manual. Chorus systèmes technical report, Chorus systèmes, October 1990.

[RAA+88] Marc Rozier, Vadim Abrossimov, Francois Armand, Ivan Boule, Michel Gien, and all. Chorus distributed operating systems. *Computing Systems Journal*, 1(4) :305–370, December 1988.

[Ras88] R.F. Rashid. From rig to accent to mach : The evolution of a network operating system. Technical report, Carnegie Mellon University, Pittsburgh, PA, August 1988.

[RFHa86] A. Rifkin, M. Forbes, R. Hamilton, and all. Rfs architectural overview. In *Proc. of the summer USENIX Conference*, page 8, Atlanta, GA, June 1986. The USENIX Association.

[RR81] R.F. Rashid and G. Robertson. Accent : a communication oriented network operating system kernel. In *Proc. 8th Symposium on Operating System Principles*, pages 64–75. ACM Press, December, 1981.

[RT74] Denis M. Ritchie and Ken Thompson. The unix time-sharing system. *Commucations of the Association for Computing Machinery*, 17(7) :17, July 1974.

[Sa78] D.P. Siewiorek and all. A case study of c.mmp, cm*, and c.vmp. *Proceedings of IEEE*, 66(10) :1178–1199, October 1978.

[Smi88] Jonathan M. Smith. A survey of process migration mechanisms. *ACM Press : Operating System review*, 22(3) :28–40, July 1988.

[SS84] John A. Stankovic and Indrjit S. Sidhu. An adaptive bidding algorithm for processes, clusters and distributed groups. In *Proc. of 4th International Conference on Distributed Computing Systems*, pages 49–59, San Fransisco, CA, Mai, 1984. IEEE Computer Society.

[Sta85] John A. Stankovic. Stability and distributed scheduling algorithms. *IEEE Transactions on Software Ingineering*, SE-11(10) :1141–1152, October 1985.

[Ste86] M. Stein. The network file system (nfs). In *Usenix Summer'86 Conference, Tutorial notes*, page 434, Atlanta, GA, June 1986. Usenix Association.

[Tab90] Daniel Tabak. *Multiprocessors*. Prentice-Hall International Editions, 1990.

[Thi92] Thinking Machines Corporation, Cambridge, MA. CM-5*, Technical summary*, January 1992.

[Tho87] Alexander Thomasian. Dynamic load balancing in a distributed system using a decentralized algorithm. In *Proc. of the 7th International Conf. on Distributed Computing Systems*, pages 178–184, Berlin, Germany, September, 1987. IEEE Computer Society.

[TL88] Marvin M. Theimer and Keith A. Lantz. Finding idle machines in a workstation-based distributed system. *IEEE Transactions on Software Engineering*, 15(11) :1444–1458, November, 1988.

[TLC85] Marvin M. Theimer, Keith A. Lantz, and David R. Cheriton. Preemptive remote execution facilities for the v system. *ACM Press : Operating System review*, 19(5) :2–12, December 1985.

[TMvR86] Andrew S. Tanenbaum, Sape J. Mullender, and Roobert van Renesse. Using sparse capabilities in a distributed operating system. In *6th Inter. Conf. on Distributed Computing Systems*, pages 558–563, Cambridge, MA, May 1986.

[TR79] Ken Thompson and Denis M. Ritchie. *UNIX Programmer's Manual (version 7)*. Bell Labs, January 1979.

[TRYa87] A. Tevanian, R.F. Rashid, M.W. Young, and all. A unix interface for shared memory and memory mapped files under mach. In *Proc. of the summer Usenix Conference*, pages 53–67, Phoenix, AZ, July 1987.

[Tur90] Stephen J. Turner. A dynamic load balancing toolkit for transputer networks. In *Proc. of the Second Workshop on Parallel Computing Action*, page 5, Ispra, Italy, December 5-6, 1990. CEC/DG XIII.

[Zay87] Edward Zayas. Attacking the process migration bottleneck. In *Proc. of the 11th ACM Symposium on operating systems principles*, pages 13–24, Austin, Texas, 8-11 November 1987. ACM Press.

[ZF87] Songnian Zhou and Domenico Ferrari. A measurement study of load balancing performance. In *Proc. of the 7th. Internat. Conf. on Distributed Computing Systems*, pages 490–497, Berlin, GER, September 21, 1987. IEEE Computer Society.

[Zho88] Songnian Zhou. A trace driven simulation study of dynamic load balancing. *IEEE Transactions on Software Ingineering*, SE-14(9) :1327–1341, September 1988.

[Zim80] Hubert Zimmermann. Osi reference model - the iso model of architecture for open systems interconnection. *IEEE transactions on Communication*, 28(4) :425–432, April 1980.