Migration de processus dans les systemes massivement paralleles

Ahmed Elleuch

THESE

présentée par

Ahmed ELLEUCH

pour obtenir le grade de **DOCTEUR**

de l'INSTITUT NATIONAL POLYTECHNIQUE de GRENOBLE

(Arrêté ministériel du 30 Mars 1992)

Spécialité : **Informatique**

Migration de Processus dans les Systèmes Massivement Parallèles

Date de soutenance : 16 novembre 1994

Composition du jury :

Président :	Roland	BALTER
Rapporteurs :	Jean-Marc Claude	GEIB GIRAULT
Examinateurs :	Traian Denis	MUNTEAN TRYSTRAM

Thèse préparée au sein du Laboratoire de Génie Informatique

Je tiens à remercier ceux grâce à qui cette thèse a pu être menée à bien.

Monsieur Roland Balter, Professeur à l'Université Joseph Fourier, qui me fait l'honneur de présider le jury de cette thèse.

Monsieur Jean-Marc Geib, Professeur à l'Université de Lille, et Monsieur Claude Girault, Professeur à l'Université Paris VI, pour le temps qu'ils m'ont consacré, et pour avoir accepté d'être rapporteurs de ce travail.

Monsieur Denis Trystram, Professeur à l'Institut Polytechnique de Grenoble, pour l'intérêt qu'il a porté à mon travail en acceptant de participer à ce jury.

Monsieur Traian Muntean, Professeur à l'Université de Marseille II, pour m'avoir accueilli dans son équipe. Je lui exprime ma profonde gratitude pour son soutient tout au long de mon travail de recherche.

Tous les membres de notre équipe "SyMPa" du Laboratoire de Génie Informatique pour les encouragements qu'ils m'ont prodigués. Merci donc à Alba, El-Ghazali, François, Leila, Harold, Irina, Philippe, Robert, et mon collègue de bureau, Léon ami et confident. Je suis heureux que la plupart de cette équipe ait pue rester ensemble, dans leur nouveau environnement à Marseille, et qu'ils puissent ainsi poursuivre les activités de recherche menées maintenant depuis plusieurs années.

Je tiens également à exprimer ma profonde reconnaissance à tous ceux qui m'ont aidé tout au long de mes études, ma famille, mes amis et tous mes enseignants.

Résumé

Cette thèse traite de la migration de processus dans les systèmes massivement parallèles. L'intérêt d'une telle fonctionnalité est de permettre à un système d'exploitation une gestion efficace des ressources. Les critères de conception sont la transparence de la migration, la réduction des coûts induits, et l'adéquation entre les algorithmes de migration et l'extensibilité des architectures cibles.

La migration d'un processus vers un nouveau processeur d'exécution nécessite la suspension du processus, le transfert de son contexte d'exécution et la reprise de l'exécution sur le nouveau processeur. De plus, les protocoles de communication et d'accès doivent être reconsidérés afin de tenir compte de la migration de processus. Pour ces différentes actions et selon les critères de conception retenus, de nouveaux algorithmes ont été proposés. La mise en œuvre de ces algorithmes dans le noyau de système ParX nous a permis de montrer que la réalisation d'un mécanisme de migration de processus dans un système massivement parallèle peut s'effectuer sans pénaliser les performances du système de façon significative.

Enfin, nous avons proposé un algorithme de répartition de charge qui utilise la migration de processus. Comparé à un algorithme uniquement fondé sur le placement des processus, les expérimentations effectuées montrent que notre algorithme améliore les temps de réponse du système grâce à la migration de processus. Cette amélioration est obtenue lorsque les temps d'exécution et d'inter-création des processus sont variables et le coût de migration négligeable par rapport à la durée d'exécution des processus.

Mots clés : migration de processus, répartition de charge, systèmes massivement parallèles, extensibilité.

Abstract

This thesis discusses process migration in massively parallel systems. This capability is useful to manage efficiently the resources in a system. The issues that we deal with are the migration transparency, the minimisation of the migration overheads and the suitableness of migration algorithms to the scalability of the machine architecture.

To migrate a process a set of operations must be performed: freezing the process, transferring its state to the new processor and restarting the process execution. The interaction between the migrated process and other entities in the system must also be handled in order to insure correctness. For these operations we have proposed new migration algorithms which fulfil the requirements above. Our algorithms have been implemented in the ParX Kernel. The performance measurements show that supporting process migration in a massively parallel system do not inquire significant penalties in performance of the system.

Besides, we have proposed a load balancing algorithm which uses the process migration facility. Compared to an algorithm relying only on the placement of the processes, the experiments carried out show that thanks to the process migration facility our algorithm offers a potential benefit when the processes execution and inter arrival times are both variable and the overhead to migrate a process is negligible compared to its execution time.

Key words: process migration, load balancing, massively parallel systems, scalability.

Sommaire

Présentation

Cette thèse s'inscrit dans le cadre de la conception et du développement d'un prototype de système d'exploitation pour des machines massivement parallèles. Elle est consacrée à l'étude de la migration de processus en tant que mécanisme système amené à faciliter et optimiser la gestion des ressources, en particulier, le placement des processus sur les processeurs d'une telle machine.

La migration de processus est développée dans la plupart des systèmes distribués. En fait, les objectifs recherchés à travers la migration de processus sont multiples : la répartition de charge, l'accès aux ressources, la tolérance aux pannes ou encore pour accomplir certaines activités de maintenance.

La répartition de charge a été l'un des sujets les plus étudiés, mais aussi, des plus controversés quant à l'efficacité du recours à la migration pour la réduction des temps de réponse du système. D'une part, en permettant de transférer un processus ayant entamé son exécution d'un processeur à un autre, la migration de processus est un moyen pour ajuster dynamiquement l'état de charge du système. D'autre part, faire migrer un processus est une tâche qui peut s'avérer assez complexe et coûteuse, en particulier lorsque la migration d'un processus engage des actions à entreprendre au niveau de plusieurs mécanismes système. Nous pensons qu'une réponse, quant à la viabilité de la migration de processus pour la répartition de charge, ne peut se faire dans l'absolu. Il nous faut tenir compte des spécificités du système et des contraintes que doit respecter la migration de processus. En particulier, une des propriétés qui distingue les systèmes massivement parallèles est l'extensibilité. Celle-ci exprime l'aptitude d'un système à pouvoir étendre le nombre de processeurs, et à en tirer profit. Cette propriété n'est généralement pas respectée par les algorithmes de migration existants. Pour cause, ces algorithmes sont destinés à des architectures où le nombre de processeurs est réduit. Ainsi nous sommes amenés à élaborer de nouveaux algorithmes de migration.

Dans le contexte nouveau des systèmes massivement parallèles, le présent travail est une étude de la migration de processus sous ses différents aspects. Nous reconsidérons dans cette étude l'utilité d'un mécanisme de migration ainsi que ses propriétés qui doivent tenir compte des caractéristiques des systèmes visés. La proposition des algorithmes de migration ainsi que leur mise en œuvre est faite dans le cadre du système d'exploitation pour des machines parallèle Paros et de son noyau ParX [Lan91] [CEM93] [Mun94]. Un algorithme de répartition de charge utilisant la migration de processus est aussi proposé et évalué.

Plan du manuscrit

Le chapitre I est une introduction à la migration de processus. Dans ce chapitre nous présentons les principaux problèmes inhérents à la migration de processus. Les objectifs ainsi que le cadre de travail y sont précisés.

Dans le chapitre II, nous nous intéressons à la conception d'un mécanisme de migration dans les systèmes distribués et particulièrement dans les systèmes massivement parallèles. Un état de l'art sur la migration de processus dans les systèmes distribués est effectué en début de ce chapitre. Les algorithmes de migration utilisés dans ces systèmes sont par ailleurs présentés plus en détail - sur certains aspects - dans les chapitres suivants. Nous abordons ensuite l'approche micro-noyau qui tend à être une base de développement des systèmes d'exploitation pour les machines parallèles. En particulier, l'impact de cette approche sur la construction d'un mécanisme de migration est étudié. Dans le cadre du noyau de système Parallèle ParX - adoptant l'approche micro-noyau - nous décrivons l'architecture générale du mécanisme de migration et son interaction avec le système.

Les chapitres III, IV, V et VI sont consacrés à l'étude et à l'élaboration d'un mécanisme de migration. Dans chacun de ces chapitres nous traitons un aspect particulier de la migration de processus.

Dans le chapitre III, le problème de l'extraction d'un processus en exécution sur un processeur et sa remise en exécution sur un nouveau processeur est traité. Les problèmes qui se posent sont : la détermination de l'ensemble des structures de données nécessaires à l'exécution du processus et la restauration de ces structures sur le nouveau processeur d'exécution de manière cohérente.

Le coût de transfert d'un processus est le principal obstacle à l'utilisation de la migration dans l'objectif d'améliorer les performances d'un système. A cet égard, l'objet du chapitre IV est la réduction du coût de transfert.

Nous nous intéressons dans le chapitre V à la correction des protocoles d'échange de messages entre les processus. Nous considérons divers schémas de communication auxquelles nous proposons des protocoles de migration adéquats.

La désignation - qui consiste à identifier des entités et à les retrouver - est un service nécessaire pour faire face à la répartition et à la multitude des ressources. Dans ce contexte, le chapitre VI est réservé à l'étude des moyens pour assurer une désignation correcte à la suite de la migration de processus.

Le chapitre VII est consacré à un exemple d'application de la migration de processus qui consiste en la répartition de charge afin d'améliorer les temps de réponse. Pour cela, nous proposons un algorithme de répartition de charge utilisant la migration de processus. Une évaluation de l'algorithme est effectuée où nous mettons en évidence les conditions dans lesquelles la migration est bénéfique.

La conclusion de cette thèse présente les principaux enseignements que nous retenons ainsi que les perspectives envisagées.

Chapitre I

Introduction

Ce chapitre est une présentation du cadre de notre travail. Après avoir préciser quelques définitions, les principaux problèmes soulevés par la réalisation d'un mécanisme de migration sont abordés. Nous discutons ensuite les motivations qui justifient le développement d'un tel mécanisme dans les systèmes à mémoire distribuée et en particulier dans les systèmes massivement parallèles. Les objectifs de notre travail sont spécifiés à la fin de cette introduction.

I.1 Définitions et concepts de base

Dans cette section nous précisons quelques définitions et concepts de base utiles dans la suite du document.

* Un *processus* correspond à l'exécution d'une séquence d'instructions. Il est représenté par un état qui évolue suivant l'avancement de l'exécution. L'*état*, ou encore l'*image*, du processus constitue l'ensemble des informations propres au processus et nécessaires à son exécution, comme par exemple le contenu de l'espace d'adressage du processus, son contexte d'exécution, etc.

* Le *placement* d'un processus sur un processeur est l'installation initiale de l'image du processus sur le processeur de façon à lui permettre d'entamer son exécution.

* La *migration* d'un processus consiste à transférer son exécution d'un processeur *source* vers un processeur *destination*. Lors de la migration d'un processus plusieurs actions sont à entreprendre. Nous les regroupons en trois phases :

(1) l'extraction du processus. Cette action comprend la suspension (ou gel) du processus, la détermination de son image et la libération de certaines ressources sur le processeur source.

(2) le transfert de l'image du processus - ou d'une partie - vers le processeur destination de façon à permettre la réactivation du processus.

(3) la restauration de l'exécution du processus. Cette action comprend la réallocation de l'image sur le processeur destination, le rétablissement des interactions entre le processus et le reste du système (au niveau des communications entre processus et des accès aux entités) et la reprise de l'exécution.

Il faut noter que cette décomposition de l'opération de migration ne représente pas un ordre logique du déroulement des actions citées. L'ordonnancement de ces actions dépend des techniques de migration utilisées. Par exemple, pour réduire la durée de transfert de l'image, on peut procéder à une pré-copie d'une partie de l'image avant de suspendre le processus.

* La migration de processus est dite restreinte à un *environnement homogène* si elle ne peut s'effectuer qu'entre deux processeurs identiques ayant une même configuration matérielle et logicielle. Inversement, la migration de processus supporte un *environnement hétérogène* si elle peut s'effectuer entre des processeurs différents.

* La migration d'un processus est dite *transparente* si elle peut avoir lieu sans que le processus concerné, ainsi que tout autre processus en exécution, ne soit amené à tenir compte de la migration, mis à part les processus système chargés d'effectuer la migration. Le processus ayant migré doit produire les mêmes résultats que s'il n'avait pas migré. Dans l'absolue, la migration d'un processus modifie forcément le déroulement de son exécution, notamment au niveau de sa durée d'exécution.

* Lorsque la migration d'un processus nécessite de garder sur le processeur source un ensemble d'informations relatives au processus, on dit que la migration induit des *dépendances résiduelles*. La migration de processus est alors non *tolérante aux pannes* qui surviennent sur le processeur source.

* Une architecture multi-processeurs est dite *massivement parallèle* ou *extensible* ("scalable") lorsqu'il est possible d'étendre le nombre de processeurs tout en augmentant la puissance de calcul. La programmation sur une telle architecture suppose en général l'indépendance algorithmique des applications du degré de parallélisme physique de l'architecture. Le système d'exploitation d'une machine massivement parallèle doit aussi fournir une puissance de calcul qui croit avec le nombre de processeurs.

* Nous nous intéressons aux architectures à mémoire distribuée où les nœuds d'exécution sont reliés par un réseau de communication. Nous désignons par *processeur* à la fois l'unité d'exécution et la mémoire locale associée.

I.2 Principaux problèmes

Nous présentons ci-dessous les principales difficultés que pose l'élaboration d'un mécanisme de migration de processus dans un système à mémoire distribuée.

I.2.1 Gel d'un processus

L'une des actions à entreprendre pour effectuer la migration d'un processus est la suspension, ou encore le gel, du processus. La principale contrainte qui se pose lors du gel d'un processus est que l'image du processus doit pouvoir être extraite et restaurer sur un nouveau processeur de manière cohérente. Or ceci n'est pas toujours possible à n'importe quel instant de l'exécution du processus. En particulier, le problème se pose dans le cas où le processus est engagé dans un appel de procédure système, ou encore, lorsque le processus exécute une section non interruptible.

I.2.2 Dépendances envers plusieurs mécanismes système

La migration de processus intervient au niveau des mécanismes de gestion de la mémoire, des processus et des communications ou encore des entrées / sorties. Les techniques utilisées pour ces différents mécanismes ont un impact important sur l'élaboration des alogorithmes de migration. Par exemple, l'existence d'une mémoire virtuelle pouvant être accédée à la demande, à partir de n'importe quel processeur, permet de ne pas transférer l'espace d'adressage du processus en entier (lors de la migration). Un autre exemple concerne les identificateurs de processus ; lorsqu'ils ne sont pas uniques dans tout le système, la migration d'un processus risque de créer un conflit d'identificateurs. Ce conflit se produit si le même identificateur du processus ayant migré est déjà utilisé pour un autre processus sur le processeur destination.

Le mécanisme de migration peut dépendre aussi d'autres services telle que la gestion des fichiers. Par exemple, lorsqu'un serveur de fichiers maintient la localisation des processus ayant ouvert un fichier. Le serveur doit alors tenir compte de la possibilité de migration de ces processus. Ce problème a été particulièrement souligné dans le cas du système Sprite [DoO91].

Enfin, il ressort de cette présentation que la conception d'un mécanisme de migration est fortement dépendante du système d'exploitation. En conséquence, l'évaluation des performances d'une technique de migration reste souvent relative au système sur lequel elle a été implantée. Afin de pouvoir comparer différentes techniques de migration et d'étudier l'influence des mécanismes système, certains travaux utilisent un même système construit pour être une plate-forme d'évaluation [ZGG90] [BaM91].

I.2.3 Accès aux prérequis

Les prérequis d'un processus sont toutes les entités nécessaires pour poursuivre l'exécution du processus sur un processeur quelconque. Par exemple : l'image du processus, une horloge système, un serveur de fichiers particulier, des bibliothèques système, etc. Lors de la migration d'un processus, il est nécessaire de déterminer l'ensemble des prérequis à un état cohérent, et de pouvoir les rendre accessibles à partir du processeur destination.

Une seconde difficulté est que certaines entités ne peuvent pas être transférées. Par exemple, un serveur de fichiers lié à un processeur particulier. Les solutions pour un tel problème s'appuient en général sur des techniques qui créent des dépendances résiduelles. Les inconvénients sont alors multiples, notamment, le problème de tolérance aux pannes et le fait que le processeur source n'est pas complètement déchargé de certaines activités induites par le processus ayant migré. Les dépendances résiduelles doivent ainsi être maintenues tout au long de l'exécution du processus.

L'étude de ces différents problèmes fait l'objet du chapitres III. Les solutions proposées dans ce chapitre font intervenir un mécanisme de désignation d'entités. Les moyens nécessaires pour supporter la migration de processus au niveau de ce mécanisme sont décrits au chapitre VI.

I.2.4 Acheminement correct des messages

Un autre problème que pose la migration de processus est celui d'assurer que les messages échangés avec un processus ayant migré soient correctement délivrés. En particulier, les propriétés assurées par les protocoles de communication doivent être respectées. Par exemple, conserver l'ordre d'envoi des messages. Notons que la correction des communications est plus délicate à réaliser lorsque deux processus communicants peuvent migrer simultanément [JMJ89]. Nous consacrons le chapitre V au traitement des divers problèmes que pose la migration de processus au niveau des échanges de messages entre processus.

I.3 Motivations

Divers besoins motivent le développement d'un mécanisme de migration dans un système parallèle. De façon générale, ces besoins consistent à optimiser l'utilisation des ressources en migrant des processus d'un processeur à un autre. Les décisions de migration se font selon des critères d'allocation des ressources spécifiques aux besoins. Des exemples de ressources sont les processeurs en tant qu'unité d'exécution, la mémoire, les bus et les liens de communication, les disques, etc. En ce qui concerne les critères d'allocation des ressources, les exemples sont nombreux : améliorer les temps de réponse d'un système, multiprogrammation des machines parallèles, assurer la continuité d'un service, etc.

Ces divers besoins de répartition de charge sont éxprimés aussi bien pour les systèmes répartis en tant que réseaux d'ordinateurs, que pour les machines multiprocesseurs. Cependant, l'exploitation de ces deux types d'architectures est différente. De plus ils se distinguent aussi par leurs caractéristiques matérielles. Dans ce qui suit, nous illustrons certains exemples de critères d'allocation de ressources dans le cas des deux types d'architectures. Nous précisons ensuite les motivations qui nous ont amenés à conduire ce travail de thèse.

I.3.1 Amélioration des temps de réponse

Les temps de réponse d'un système peuvent être réduits en distribuant la charge sur les processeurs. De nombreux algorithmes utilisant la migration de processus ont été proposés à ce sujet [NXG85] [Cab86] [GuG90] [SWo92]. Cependant le recours à la migration pour l'amélioration des temps de réponse est contesté [LeO86] [ELZ88], comparé à la répartition de charge lors du placement initial des processus. A travers les différents travaux à ce sujet, nous remarquons qu'il y a une concordance sur le fait que la migration est viable sous certaines conditions. Leland et Ott [LeO86] concluent que la migration n'est profitable que pour des processus longs en temps d'exécution. Eager, Lazowska et Zahorjan [ELZ88] précisent que la migration n'est avantageuse que lorsque les temps de calcul et d'inter-création de processus sont très variables. La question qui se pose alors est : ces conditions sont ils extrêmement rare? La réponse dépend particulièrement du type d'applications et du modèle de programmation utilisés. Les études comparatives, citées ci-dessus, s'appliquent plutôt dans le cas des systèmes répartis où le nombre de nœuds dans le réseau n'est pas important et où la communication n'est pas fréquente.

L'organisation de l'exécution des applications parallèles est souvent différente de celle dans les systèmes répartis. Plusieurs machines parallèles sont exploitées en effectuant une partition des processeurs et du réseau d'interconnexion (BBN Butterfly [CGT85], CM [TRo88],

IBMRP3 [PBG85], IPSC/2 [Nug88], Ncube [HMD86], Supernode [GLM91], etc). L'exécution d'une application est alors effectuée sur un sous ensemble de processeurs que nous désignons par *cluster*. Dans ces conditions, la migration de processus est utile non seulement pour répartir la charge entre les processeurs d'un cluster, mais aussi, pour répartir la charge entre clusters. Pour cela, un ou plusieurs processeurs, appartenant à un cluster faiblement chargé, sont libérés et alloués à d'autres clusters ayant une charge importante. En revanche, dans le cas des systèmes répartis, on peut avoir besoin de libérer une machine mais pour d'autres critères d'allocation de ressources. Par exemple, il est commun dans de tels environnements que les machines soient dédiées à des utilisateurs. Lorsque les utilisateurs d'une machine ne l'exploitent pas, des programmes appartenant à d'autres utilisateurs sont exécutés sur cette machine. La migration de processus sert alors à libérer la machine si elle est requise par ses utilisateurs [Dou89].

En ce qui concerne l'amélioration des temps de communication, la migration est un moyen pour rapprocher les processus lorsqu'ils s'échangent un volume important de données, par exemple, en les disposant sur un même processeur. Cependant, peu d'intérêt a été porté à ce type de problème. D'une part, dans le cas des systèmes répartis la communication entre processus distants est non fréquente. D'autre part, le temps de migration d'un processus risque d'être considérable. Néanmoins, dans le cas des machines parallèles sans mémoire commune, l'optimisation des communications grâce à la migration de processus présente un plus grand intérêt. En effet, certains modèles de programmation sur ce type de machines font que la communication entre les processus est importante (par exemple le modèle CSP [Hoa85]).

La réduction des temps de réponse du système grâce à la migration de processus sera plus amplement étudiée à travers la proposition d'un algorithme de répartition de charge qui fait l'objet du chapitre VII.

I.3.2 Multiprogrammation des machines parallèles

Jusqu'à présent les machines massivement parallèles n'ont pas encore connu un essor en rapport avec leurs capacités de cacul ; la complexité d'exploitation de telles machines et leur coût de revient en sont les principals obstacles. Surmonter ces difficultés requiert des supports système qui facilitent l'utilisation de ce type de calculateurs et garantissent des performances élevées. De plus, ces supports système doivent permettre la répartition du coût d'une telle machine, grâce à son partage entre plusieurs applications et en permettant de moduler la puissance de calcul selon les besoins [KBC94]. Il s'agit donc bien de développer un véritable système d'exploitation capable de gérer efficacement et simultanément l'exécution de plusieurs applications.

L'un des services que doit offrir un tel système est l'allocation des ressources, ou encore, la distribution automatique des programmes. Cependant la réalisation de ce service est rendue complexe de par le caractère dynamique de l'exécution des applications. Grâce à la migration de processus, il est possible de reconsidérer le placement des processus et donc d'adapter dynamiquement l'allocation des ressources.

Par exemple, considérons le critère d'allocation où il s'agit d'exécuter le plus d'applications en utilisant un même ensemble de ressources. Reprenons le cas où l'exécution des applications se fait sur une partition de la machine. Etant donnée une certaine topologie du réseau d'interconnexion, si l'on désire allouer un cluster avec une disposition particulière des processeurs et des interconnexions, même si le nombre de processeurs nécessaires est suffisant, la création du cluster n'est pas toujours possible. Il faut que ces processeurs soient disposés d'une manière adaptée à la topologie du cluster que l'on veut créer. En revanche, moyennant la migration de certains processus, la création du nouveau cluster devient possible. L'exemple de la figure 1 illustre cette situation. Cette restructuration du placement des processus est à comparer aux techniques de ramasse-miettes utilisées pour résoudre le problème de fragmentation de la mémoire.

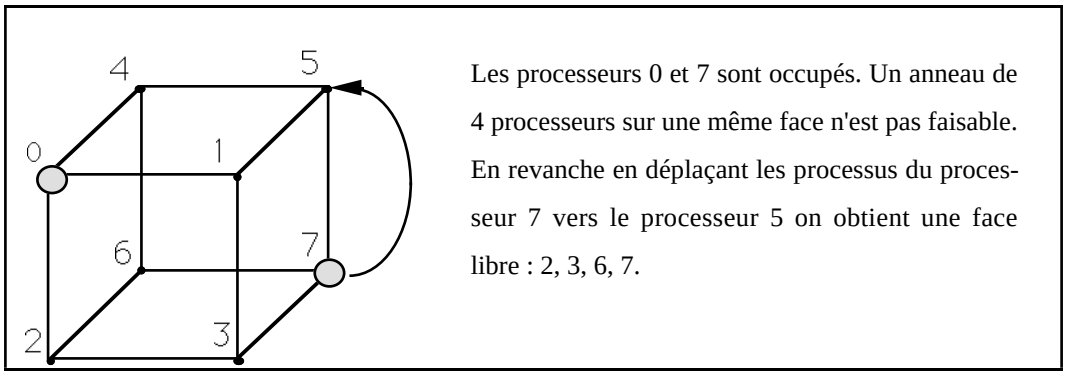


Figure 1 : Restructuration du placement des processus

I.3.3 Accès direct à des ressources

La migration de processus est utile pour l'accès direct à une ressource distante. En effet, certaines ressources (disque, terminal, accès réseau local, etc.) sont souvent liées à des processeurs spécifiques du système. Cette situation est fréquente dans le cas des machines parallèles où la majorité des processeurs sont dépourvus de périphériques d'entrées/sorties. Le besoin d'accéder directement à une ressource est aussi motivé par le fait que le coût des accès à ces ressources peut dépasser largement celui de la migration du processus effectuant les accès.

Par exemple, dans le cas où un processus doit manipuler une partie importante d'une base de données stockée sur un disque particulier.

I.3.4 Tolérance aux pannes et maintenance

Considérons une panne qui survient sur un processeur mais qui n'altère pas l'image des processus en exécution, par exemple, un lien de communication sur lequel il n'est plus possible d'échanger des messages. La migration de processus permet de poursuivre l'exécution des processus concernés par cette panne, en les transférant sur un autre processeur fonctionnant correctement. Ceci est d'autant plus utile lorsqu'il s'agit d'un processus long en temps d'exécution ayant presque terminé. En outre, lors d'opérations de maintenance sur un processeur, il est utile de faire migrer ailleurs les processus qui s'exécutent sur ce processeur sans avoir à les abandonner.

BiriliX [LHK93] et Condor [LiS92] sont des exemples de systèmes où la migration de processus est utilisée pour des besoins de tolérance aux pannes. La migration de processus a été par ailleurs proposée pour la reconfiguration d'un système à la suite d'une panne survenant sur un processeur [SSF92].

I.3.5 Motivations et objectifs de ce travail

Notre travail s'inscrit dans le cadre de la conception et le développement de mécanismes système permettant de mieux exploiter les machines massivement parallèles. Devant la complexité de telles machines pouvant atteindre un nombre important de processeurs, la gestion des applications en exécution de manière à maximiser l'utilisation des ressources est un problème complexe. L'automatisation de l'allocation des processeurs reste encore un objectif non atteint par les environnements de programmation existants, d'où les difficultés que confrontent les utilisateurs à exploiter les machines massivement parallèles. De nouveaux supports système sont nécessaires pour faciliter l'utilisation de ces calculateurs tout en garantissant des performances élevées. De plus en accord avec la propriété d'extensibilité, ces supports système doivent permettre la répartition du coût de revient d'une telle machine, grâce à son partage entre plusieurs applications et en permettant de moduler la puissance de calcul selon les besoins. Il s'agit donc bien de développer un véritable système d'exploitation capable de gérer efficacement et simultanément l'exécution de plusieurs applications.

En permettant de reconsidérer le placement des processus à tout moment, la migration de processus est un moyen approprié pour réduire la complexité du problème. Notons que l'on peut

aussi envisager la migration d'autres types d'entités comme par exemple des segments de données en mémoire [ScD88] [BlS89], ou des fichiers [Hac89].

Les exemples d'application de la migration de processus, décrits précédemment, montrent l'apport que celle ci peut avoir dans la résolution d'un certain nombre de problèmes. C'est dans cette optique que nous avons engagé l'étude et la réalisation d'un mécanisme de migration pour le noyau ParX [Lan91] [CEM93] du système parallèle Paros [Mun93]. Ce système a été développé dans le cadre du projet Esprit SuperNode II [Sup91] auquel notre équipe a participé. L'architecture de ce système, décrite plus loin dans le chapitre II, est structurée de façon à pouvoir bâtir différents systèmes au dessus du noyau ParX. L'existence d'un mécanisme de migration de base contribue donc à cette approche puisque un grand nombre de systèmes distribués offrent la migration de processus.

Par ailleurs, la tolérance aux pannes n'est pas un besoin déterminant comparé à la nécessité de réduire les coûts de migration. En outre, la migration de processus dans un environnement hétérogène est difficile à réaliser sans introduire des surcoûts non négligeables au niveau de l'exécution des processus [MaS88a] [ThH92]. Nous avons donc limiter notre étude de la migration à un environnement homogène. Aucune traduction de codes ni conversion de données n'est nécessaire. Notons aussi que seuls les processus applications peuvent migrer. Les processus systèmes dédiés à la gestion de ressources ne sont pas sujet à la migration.

I.3.6 Critères de conception

Etant donné que la migration est destinée à être utilisée pour des besoins d'optimisation de l'utilisation des ressources, la réduction des coûts de migration est un critère essentiel. En particulier, le rajout du mécanisme de migration dans un système massivement parallèle ne doit pas augmenter ses temps de réponse. Ce critère n'est pas déterminant dans le cas des systèmes répartis où le surcoût rajouté par le fonctionnement du système est acceptable les temps de réponse de ces systèmes, ce qui n'est pas le cas des systèmes massivement parallèles.

Deux autres critères de conception sont aussi à respecter. Le premier critère est la transparence de la migration. Ce critère est nécessaire étant donné que la migration est prévue pour que le système puisse gérer automatiquement la répartition des processus (sans que ces derniers n'interviennent). Le second critère concerne l'extensibilité des mécanismes de migration. Ce critère propre aux architectures massivement parallèles n'est généralement pas respecté par les algorithmes de migration existants, qui sont destinés à des architectures où le nombre de processeurs est réduit.

I.4 Travail réalisé

Suivant les critères que nous venons de décrire, de nouveaux algorithmes de migration, adaptés aux systèmes massivement parallèles, ont été réalisés. Ces algorithmes se distinguent de ceux utilisés dans les autres systèmes distribués, d'abord par le fait qu'ils n'introduisent pas des surcoûts significatifs au niveau des délais de réponse du système, ensuite, par le fait que sur chaque processeur la complexité de ces algorithmes est indépendante de la taille du réseau et ceci au niveau des besoins en espace mémoire, des traitements effectués ainsi que du nombre de messages de contrôle que nécessitent ces algorithmes.

La construction du service de migration s'est faite au dessus du micro-noyau de ParX. Seuls les protocoles de communication ont du être adaptés pour supporter la migration, sans pour autant pénaliser les temps de réponse du micro-noyau. Ces résultats montrent que les caractéristiques des systèmes massivement parallèles ne s'opposent pas à la réalisation d'un mécanisme de migration de processus dans ces systèmes.

La réalisation du mécanisme de migration souffre du fait que la machine d'expérimentation, le SuperNode [MuW90], utilise comme processeur de base le transputer T800 [Inm89] et que ce dernier ne permet pas l'adressage logique. En conséquence, un processus ayant migré ne peut être installé qu'aux mêmes adresses physiques occupés par le processus sur le processeur origine. Ce qui diminue les possibilités d'utilisation pratique du mécanisme de migration.

Enfin, nous nous sommes intéressés à déterminer dans quelles mesures la migration de processus permet de réduire les temps de réponse du système. Pour cela nous avons proposé un algorithme de répartition de charge qui utilise la migration de processus. Comparé à un algorithme uniquement fondé sur le placement des processus, des simulations que nous avons menées montrent que notre algorithme améliore les temps de réponse du système. Cette amélioration est obtenue lorsque les temps d'exécution et d'inter-création des processus sont variables et le coût de migration négligeable par rapport à la durée d'exécution des processus.

Chapitre II

Principes des Techniques de Migration de Processus

Ce chapitre décrit les concepts généraux liés à la migration de processus dans les systèmes distribués. Pour cela nous commençons par présenter la migration de processus à travers les systèmes existants. Nous abordons ensuite l'approche micro-noyau souvent utilisée dans la construction de systèmes d'exploitation pour les machines massivement parallèles. En particulier, les contraintes qu'impose cette approche sur la construction d'un mécanisme de migration sont dégagées. Dans le cadre du système parallèle Paros - adoptant l'approche micro-noyau - nous décrivons son architecture générale et comment s'insère le mécanisme de migration dans ce système.

II.1 Migration de processus dans les systèmes distribués

La migration de processus est mise en œuvre dans un grand nombre de systèmes distribués : Accent [Zay87 a, b] [Ras86], Amoeba [MRT90], Charlotte [ACF86] [ArF89], Chorus/Mix [Phi93] Chorus[CTV94], Demos/MP [PoP83], Emps [DGi92], Locus [PCW81] [WPR83], Mach [MZD93], Mos [BaL85], Mosix[BGW93], Sprite [Dou89] [DoO91] [OCD88], Unix & NFS [MaS88], V [TLC85] [The86], etc. A travers les réalisations des mécanismes de migration dans ces systèmes se dégagent différentes techniques de migration. Ces techniques concernent notamment (1) le transfert de l'image, (2) l'acheminement correct des messages entre les processus communicants et la (3) désignation et l'accès aux entités. La suite de cette section décrit les principales techniques de migration existantes. L'objectif de cette description n'est pas d'évaluer ni de comparer ces techniques, mais d'en donner l'éventail et de voir comment elles peuvent être intégrées dans un système distribué. Ces techniques sont par ailleurs étudiées et évaluées dans les chapitres suivants.

II.1.1 Techniques de transfert de l'image d'un processus

L'image d'un processus est souvent de taille importante, son transfert constitue ainsi une surcharge significative pour le réseau de communication. De plus, le temps de transfert pénalise le processus ayant migré puisque ce dernier est a priori bloqué en attente de l'accomplissement du transfert. Malgré ces inconvénients, certains systèmes n'opèrent aucune optimisation de l'opération de transfert ; l'image du processus est complètement recopiée pendant la phase de gel. C'est le cas des systèmes Amoeba, Charlotte, Demos/MP, Emps, Locus et Mosix[1].

Afin de réduire les coûts de transfert de l'image, d'autres techniques ont été élaborées. Nous distinguons en particulier deux approches : l'une anticipe l'opération de transfert et l'autre retarde le transfert de certaines parties de l'image qui sont ramenées lorsque le processus les référence.

La technique de pré-copie utilisée dans le système V anticipe le transfert de l'image en entamant le transfert avant la suspension du processus. Les zones mémoire copiées, mais modifiées pendant la copie, doivent être transférées une nouvelle fois. Cette technique nécessite donc que le gestionnaire mémoire puisse permettre la détermination des zones mémoire modifiées.

La copie sur référence utilisée dans le système Accent retarde le transfert d'une partie de l'image. Elle n'entreprend un transfert d'une zone mémoire, non encore recopiée, que lorsque le processus effectue un accès à cette zone. Cette technique s'appuie donc sur les mécanismes de gestion mémoire.

Le système Sprite utilise aussi la copie sur référence : lors de la migration, l'espace mémoire associé au processus est sauvegardé sur un système de stockage, les pages mémoire sont chargées sur le nouveau processeur lorsqu'elles sont référencées. Cette solution a été aussi adoptée pour Chorus/Mix. Là encore, l'opération de transfert repose sur les mécanismes de gestion mémoire, mais aussi les mécanismes de stockage.

A travers ces différentes réalisations, nous constatons que l'opération de transfert de l'image d'un processus est réalisée en grande partie grâce aux services de communication entre processeurs, de gestion mémoire et de stockage offerts par le système.

---

[1] Dans le cas de Mosix, seul l'espace mémoire ayant été modifié est transféré directement vers le processeur destination.

II.1.2 Acheminement correct de messages

La migration de processus nécessite la mise en œuvre d'algorithmes adaptés pour assurer que les messages échangés entre les processus soient correctement délivrés ; même si un processus change de processeur d'exécution les messages qui lui sont déstinés doivent lui parvenir. Pour cela diverses techniques ont été élaborées.

Les systèmes Amoeba et V utilisent un mécanisme qui rejette un message lorsque le processus destinataire n'est plus présent. Après avoir déterminé la nouvelle adresse du processus destinataire, le message est à nouveau retransmis par le processeur émetteur du message. La détermination de l'adresse est faite grâce à la diffusion d'une requête de localisation à la suite du rejet.

Les systèmes Accent, Demos/MP, Locus, Mach, Mosix et Sprite font suivre les messages vers le processeur où est supposé se trouver le processus destinataire du message. Une suite de redirections peut ainsi se constituer vers un processus s'il migre plusieurs fois. Afin de réduire la longueur de cette suite, des techniques de mise à jour des adresses de redirection sont en plus utilisées.

Pour le système Charlotte, les processus pouvant échanger des messages avec un autre processus sont connus. De ce fait lors de la migration d'un processus, il est possible de mettre à jour la nouvelle adresse du processus chez ses correspondants. Ainsi les communications se poursuivent sans traitements particuliers à la suite de la migration.

Le système Emps réalise les opérations de communication à travers des boites aux lettres. Une boite aux lettres est une structure de données qui maintient les adresses des processus qui communiquent à travers elle. Lors de la migration d'un processus, les boites aux lettres utilisées par ce processus sont mises à jour selon la nouvelle adresse du processus.

Nous constatons ainsi que ces diverses techniques, assurant l'échange correct de messages, sont fortement dépendantes des protocoles de communication. Notons enfin que le problème des communication a été par ailleurs traité dans de nombreux travaux que nous étudierons plus tard [Bar91] [DGi92] [Fow86] [JLH88] [LCL87] [RaJ88].

II.1.3 Désignation et accès aux entités

A la suite de la migration d'un processus, les identificateurs utilisés par l'ensemble des processus en exécution doivent désigner les mêmes entités que si la migration n'avait pas eu lieu. Pour cela, les propriétés de ces identificateurs doivent être respectées.

Un identificateur est dit *global* s'il permet de référencer la même entité à partir de n'importe quel processeur. Ainsi, lorsque les identificateurs manipulés par un processus sont globaux, ces derniers restent valides à la suite de la migration du processus.

Un identificateur est dit *local* s'il ne peut être utilisé que sur un processeur déterminé. Dans le cas où un processus manipule des identificateurs locaux, la migration du processus doit permettre au processeur destination de pouvoir interpréter correctement ces identificateurs.

Un identificateur est dit transparent à la localisation s'il ne traduit pas en lui même l'adresse de l'entité désignée. Lorsque l'identification d'un processus est non transparente à la localisation, la migration du processus induit la modification de l'identificateur. Ceci pose un problème si le processus ayant migré est désigné à travers cet identificateur par d'autres processus dans le système. Lorsqu'un identificateur est transparent à la localisation, il est nécessaire de disposer d'un mécanisme permettant de déduire l'adresse de l'entité. La localisation d'entités ayant migré se base en général sur les mécanismes de communication supportant la migration de processus.

Dans les systèmes Amoeba, Locus, Mos, Sprite et V, les identificateurs utilisés par les processus sont globaux et transparents à la localisation, ils ne sont donc pas altérés par la migration de processus.

Dans le cas des systèmes Accent, Charlotte, Demos/MP, Emps et Mach, un processus ne manipule que des noms locaux. Le système maintient une correspondance entre les identificateurs locaux avec des identificateurs globaux non transparents à la localisation permettant ainsi de retrouver les entités désignées. Lors de la migration d'un processus, l'utilisation des identificateurs locaux est rendue possible sur le processeur destination.

II.2 Approche micro-noyau

L'approche micro-noyau consiste à implanter le plus possible de services système - au même niveau que les processus utilisateur - au dessus d'un micro-noyau ayant un nombre réduit de fonctionnalités. Différents sous-systèmes peuvent ainsi être construits en utilisant les fonctionnalités de base d'un micro-noyau. Cette structuration du système présente plusieurs avantages. En particulier, elle favorise la portabilité des sous-systèmes sur d'autres architectures. Elle permet aussi d'adapter les services système aux besoins des utilisateurs. De plus, la minimisation des fonctionnalités d'un micro-noyau permet d'optimiser ses temps de réponse. Cet avantage est non des moindres dans le cas des machines massivement parallèles

utilisés, le plus souvent, pour des applications qui nécessitent des puissances de calcul importantes.

Le système V [Che88] a été un précurseur des systèmes utilisant l'approche micro-noyau. Cette approche s'est plutôt développée à travers les micro-noyaux des systèmes tels que Amoeba [vRT92], Chorus [Roz91], Mach [Tev87] [Bla92] et plus récemment Plan 9 [PPT92].

Les micro-noyaux des systèmes Amoeba, Chorus et Mach manipulent des concepts assez similaires. Les services offerts par ces micro-noyaux concernent la gestion des processus, de la mémoire et des communications. Dans le cas de ces micro-noyaux, un processus comporte plusieurs flots de contrôle d'exécution ("thread") partageant un même espace d'adressage. Le contexte d'exécution propre à ces flots de contrôle est réduit, d'où une commutation de contexte rapide entre flots. Alors que pour Chorus et Mach la gestion mémoire est paginée, celle-ci ne l'est pas dans le cas du système Amoeba où on met l'accent sur les performances d'accès mémoire. En ce qui concerne les communications inter-processus, elles sont basées sur l'échange de messages à travers des ports. Un port représente une adresse globale à laquelle on peut rattacher des messages ou les consommer.

Dans Plan 9, toutes les ressources appartiennent à un espace unique de désignation et sont accédées uniformément de la même manière. Pour cela, une ressource est représentée par une arborescence de répertoires et de fichiers. Cette représentation a permis de simplifier l'écriture du code du système et a réduit considérablement sa taille.

La réduction de la taille du code d'un micro-noyau permet de le charger dans une mémoire cache ("on chip CPU") et de diminuer ainsi ses temps de réponse, QNX [Hil92] en est un exemple. En particulier, il est important que l'espace mémoire nécessaire à l'exécution du micro-noyau soit de taille limitée. L'allocation dynamique de la mémoire doit être ainsi évitée. Ceci simplifie le fonctionnement du micro-noyau qui n'aura plus à gérer les problèmes d'insuffisance mémoire. En particulier et dans le cas des systèmes massivement parallèles, étant donné le nombre important de processeurs et la dépendance qui existe entre ces processeurs, il est indispensable que le micro-noyau sur chacun des processeurs puisse s'exécuter sans être contraint à s'interrompre à cause d'une insuffisance mémoire.

Suivant l'approche micro-noyau, l'implantation d'un mécanisme de migration de processus doit se faire au dessus du micro-noyau. En revanche, le rétablissement de l'interaction entre un processus à faire migrer et les autres entités dans le système peut nécessiter des modifications, notamment, au niveau des mécanismes de désignation et de communication. Il est alors essentiel que les modifications portées au micro-noyau n'induisent pas des surcoûts

significatifs au niveau des temps de réponse, et que la taille des structures de données rajoutées dans le micro-noyau soit limitée.

II.3 Mise en œuvre d'un service de migration dans Paros

Nous présentons dans cette section l'architecture du système Paros [Sup91] [[Mun94] et le modèle d'exécution supporté par son micro-noyau [Lan91] [CEM93]. Nous proposons ensuite une conception générale du service de migration de processus dans ce système. En particulier, nous situons les mécanismes supplémentaires à fournir et les modifications à effectuer dans d'autres mécanismes système.

II.3.1 Architecture du système Paros

Le système Paros a été conçu pour l'exploitation de machines massivement parallèles. L'un des principaux objectifs de ce système est d'être à la fois non dédié et capable de répondre aux besoins en performances que nécessitent les applications parallèles.

La figure 1 décrit l'architecture générale du système Paros. Celui-ci comporte un noyau appelé ParX au dessus duquel sont construits différents sous-systèmes. ParX est structuré en trois niveaux d'abstractions correspondant chacun à une machine virtuelle.

Le niveau le plus bas, appelé HEM ("Hardware Extension Machine"), cache les caractéristiques et l'hétérogénéité du matériel. Il offre une vue d'une machine abstraite unique grâce à un ensemble de primitives permettant la communication et le lancement de processus à distance entre toute paire de processeurs.

Au dessus du HEM se trouve le π-Nucleus qui est un micro-noyau mettant en œuvre un modèle d'exécution adapté aux applications parallèles. Le π-Nucleus assure la gestion des processus, des communications / synchronisations et des ressources.

Le niveau supérieur de ParX est réservé pour la construction de serveurs offrant des interfaces pour diverses machines virtuelles qui correspondent à différents modèles de support d'exécution pour les applications. Par exemple, les machine virtuelles supportant l'allocation automatique des processeurs [Tal93] [EKM94], la diffusion [DeM93], ou encore la mémoire virtuelle partagée [Cas91] [BaM94].

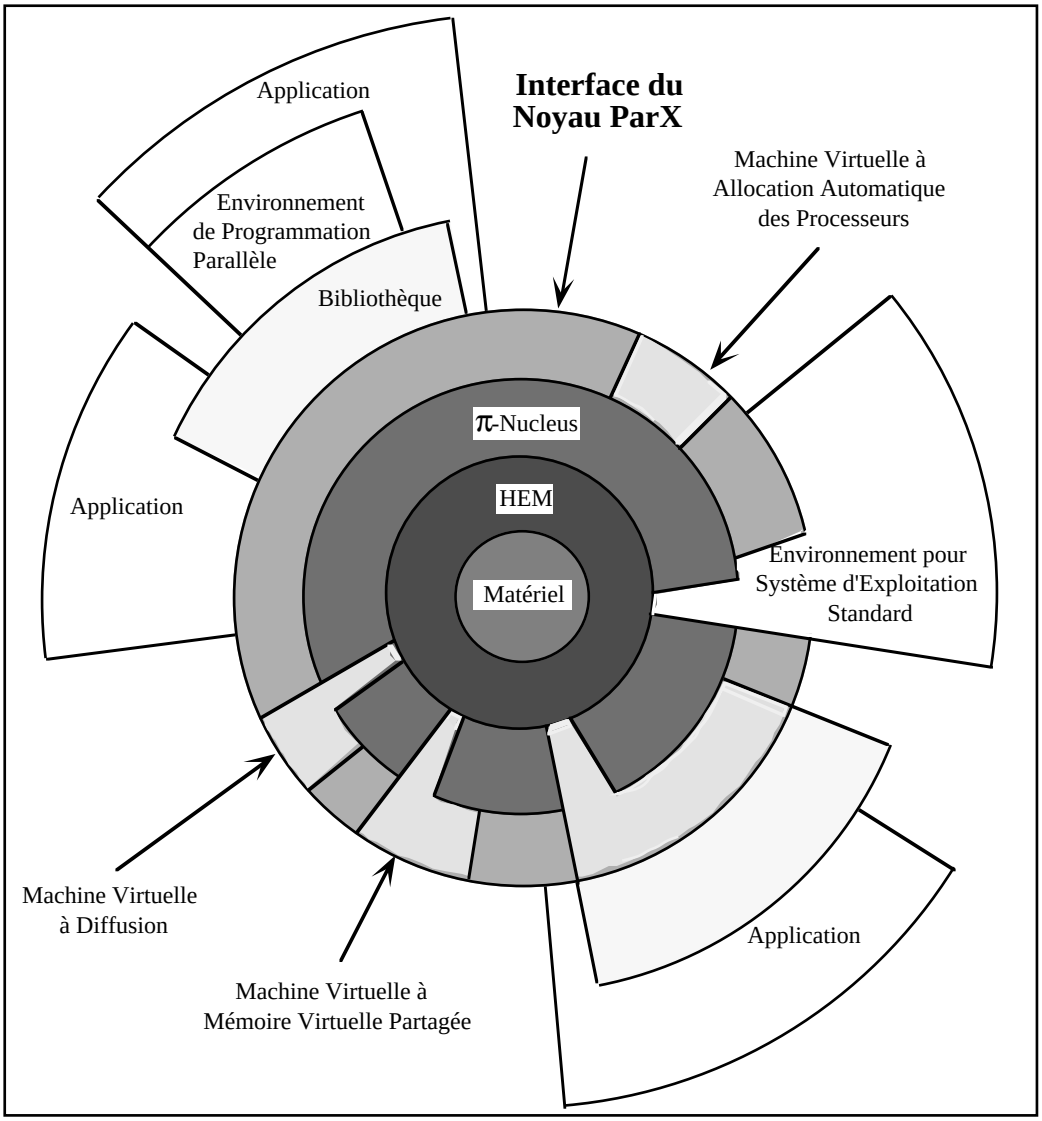


Figure 1 : Architecture générale de Paros

II.3.2 Modèle d'exécution

Le micro-noyau (π-Nucleus) supporte un modèle de processus adapté à l'exécution d'applications parallèles sur des architectures distribuées. Ce modèle comporte trois types d'entités : la *Ptask*, la *task*, et le *thread*. Une Ptask est l'entité de contrôle attachée à l'exécution d'un programme parallèle. Elle permet de gérer le lancement et la terminaison du programme, l'ordonnancement des processus, l'allocation des ressources nécessaires à cette exécution, etc. Elle définit aussi un domaine de communication et de protection. Une Ptask est composée d'un ensemble d'espaces d'adressages distincts, chacun correspondant à une task. Une task est un contexte d'exécution commun à des flots de contrôle concurrents qui sont des processus légers appelés threads. Vu isolément, un thread ne conserve pas de descripteurs des ressources

qu'il utilise comme la mémoire, les fichiers ouverts, les canaux, etc. Son contexte se limite à quelques informations telles que le compteur ordinal et l'état de la pile d'exécution. La manipulation d'un tel processus est ainsi simplifiée. Notons aussi que dans la version actuelle du π-Nucleus, l'espace mémoire entier d'une task est chargé sur le processeur où s'exécute la task.

L'exécution d'une Ptask s'effectue sur un ensemble de processeurs, appelé *cluster*. Un cluster est une machine virtuelle ayant le support système nécessaire pour l'exécution de la Ptask (à laquelle le cluster est alloué).

Le π-Nucleus offre aussi un mécanisme pour la construction générique de protocoles de synchronisation, de communication et d'accès mémoire. Les protocoles ainsi construits obéissent à un certain nombre de règles afin de respecter les contraintes qu'impose l'approche micro-noyau.

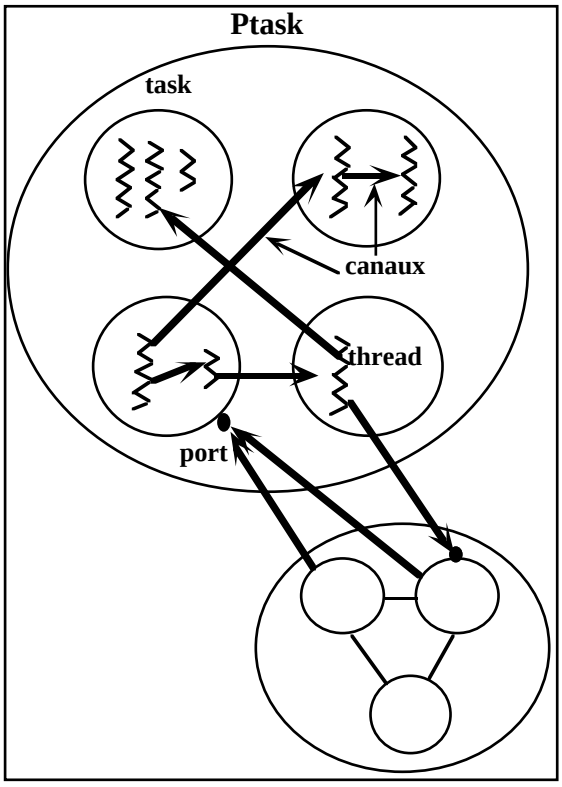


Figure 2 : Objets de communication de base : les canaux et les ports

Des exemples de protocoles ayant été développés sont : le rendez-vous entre un ensemble de processus, le transfert de données entre différents espaces d'adressage et divers protocoles de communication point à point synchrones ou asynchrones. Parmi ces derniers, deux protocoles de communication de base - adaptés au modèle de processus - sont fournis : les canaux et les

ports (figure 2). Ils sont tous deux synchrones à l'émission et à la réception. Un canal est un objet de communication entre deux tasks d'une même Ptask. Un port est un objet de communication de plusieurs tasks vers une, celles-ci pouvant appartenir à différentes Ptasks. Le port est notamment utilisé pour la communication entre différents sous-systèmes.

II.3.3 Unité de migration

La structuration du modèle de processus en trois niveaux implique trois types de migration envisageables : celle d'un thread, d'une task ou d'une Ptask.

La migration d'un thread est écartée, celui-ci ne pouvant s'exécuter que s'il fait partie d'une task qui représente son environnement d'exécution. Cependant si l'on suppose qu'un nœud d'exécution est un multiprocesseurs à mémoire partagée, la migration d'un thread vers un nouveau processeur du même nœud est possible. En effet, le thread est toujours rattaché à la même task englobante. La migration du thread revient alors à ordonnancer son exécution sur le nouveau processeur.

Le système Amber [FBC91] envisage la migration de threads d'un processeur à un autre. Comme dans le cas de ParX, chaque thread est rattaché à un espace d'adressage appelé *nœud logique* (équivalent à la task). La migration d'un ensemble de threads se fait en créant sur le processeur destination un nouveau nœud logique auquel le programmeur rattache les threads à faire migrer. Cette technique ne permet donc pas une migration transparente à l'utilisateur.

En ce qui concerne la task, elle ne peut migrer qu'à l'intérieur du cluster de processeurs occupée par la Ptask englobante. Cette limitation est due au fait que la protection des Ptasks est assurée en les séparant physiquement. Dans le cas où un nœud d'exécution offre une protection complète entre les tasks qu'il exécute, il est possible d'allouer deux tasks appartenant à des Ptasks différentes sur ce même nœud.

Nous écartons la possibilité de rattacher une entité (i.e., thread, task) à une nouvelle entité englobante (i.e., task, Ptask), car ceci implique une modification de la sémantique de ces différentes entités. L'unité de migration est donc la tâche. La migration de plusieurs tâches, voir d'une Ptask entière, est aussi possible.

II.3.4 Service de migration de processus

Suivant l'approche micro-noyau, le service de migration de processus doit être construit au dessus du π-Nucleus. Ce service peut faire partie de la machine virtuelle à allocation automa-

tique des processeurs. Il peut aussi être fourni par un sous-système qui le nécessite dans une instanciation dédiée du π-Nucleus générique.

Lors de la migration d'une task, le serveur de migration doit entreprendre les actions suivantes : l'extraction, le transfert et la restauration de l'image de la task tout assurant que les communications et la désignation des entités puissent se poursuivre correctement. L'ensemble de ces actions doit être, autant que possible, réalisé grâce aux primitives offerts par le π-Nucleus sans avoir à modifier ce dernier.

L'extraction et la restauration de l'image du processus est traitée dans le chapitre III. Le transfert de l'image fait l'objet du chapitre IV. Le chapitre V est consacré à la correction des communications. A cet égard, notons que le mécanisme générique de construction de protocoles de ParX offre une interface permettant de spécifier un protocole sans avoir à effectuer des modifications dans le micro-noyau. Dans le chapitre VI, nous reconsidérons le service de désignation de façon à supporter la migration de processus. Ce service étant construit au dessus du π-Nucleus, nous ne sommes donc pas amenés à modifier ce dernier.

Chapitre III

Extraction et Restauration d'un Processus

L'objet de ce chapitre est l'étude de l'extraction du contexte d'exécution d'un processus et sa restauration sur un nouveau processeur. Ces actions requièrent la détermination du contexte à extraire de façon à ce qu'il soit suffisant pour poursuivre l'exécution du processus. Elles nécessitent aussi la disponibilité de certaines ressources (bibliothèques système, serveur mémoire, etc.) sur le processeur destination de la migration. De plus, l'ensemble des entités requises pour la réactivation du processus doit avoir un état cohérent avec la connaissance qu'a le processus sur l'état de ces entités. Par exemple, si le processus a déjà consulté l'horloge sur le processeur source de la migration, une nouvelle consultation de l'horloge sur le processeur destination doit rendre une valeur supérieure à celle du premier appel.

Afin de déterminer les conditions de cohérence suffisantes pour reprendre l'exécution d'un processus sur un nouveau processeur, nous sommes amenés à étudier la structure du processus et à tenir compte des caractéristiques du système. En particulier, il faut considérer l'interaction du processus avec le système : invocation du système et accès aux ressources.

Dans ce chapitre nous présentons des techniques pour l'extraction et la restauration d'un processus de façon à respecter la cohérence de son exécution. Ces techniques concernent la suspension d'un processus, l'invocation du système et l'accès aux différentes entités locales ou distantes. Nous proposons des solutions que nous appliquons au noyau ParX ; l'accent est mis sur la réduction des surcoûts introduits sur le fonctionnement du système.

III.1 Description d'un processus et de son interaction avec le système

III.1.1 Image d'un processus

Comme nous l'avons déjà défini, l'*image* (ou l'*état*) d'un processus est l'ensemble des informations permettant au système d'exécuter le processus. Ces informations peuvent être décomposées de la manière suivante :

- l'espace mémoire adressable par le processus, où sont stockés le code exécutable, les piles et les données,

- les structures de données du système où sont conservées les informations de contrôle de l'exécution du processus :

  ~ les valeurs des registres du processeur : le compteur ordinal, les registres généraux, les registres d'état du processeur, etc.

  ~ les informations nécessaires pour la gestion du processus (identification du processus, priorité, etc.) et pour la gestion des communications et l'accès aux ressources (tables de pagination, descripteurs de fichiers, descripteurs d'objets de communication, etc.).

Pour faire migrer un processus, il est a priori nécessaire de transférer son image sur le nouveau processeur d'exécution. Le transfert de l'image fera l'objet du chapitre suivant. Nous nous intéressons dans ce qui suit à retrouver l'image d'un processus, à l'extraire et à la restaurer sur le processeur destination de manière transparente au processus.

III.1.2 Prérequis d'un processus

Les *prérequis* d'un processus représentent l'ensemble des entités nécessaires à son exécution. Par exemple, la représentation de l'image du processus dans le système, l'horloge système, un serveur de fichiers, des bibliothèques système, etc.

L'image d'un processus est la *vue* du processus que perçoit le système. Elle ne traduit pas la sémantique rattachée à l'exécution du processus. Inversement, le processus perçoit aussi une vue sur l'état du système. Par exemple, la valeur de l'horloge du système, l'identificateur du processeur d'exécution, etc. Cette vue concerne l'état d'un certain nombre de prérequis du

processus. Elle n'englobe pas forcément l'état complet de tous les prérequis du processus. La figure 1 est une schématisation de cette notion de vues entre le processus et le système.

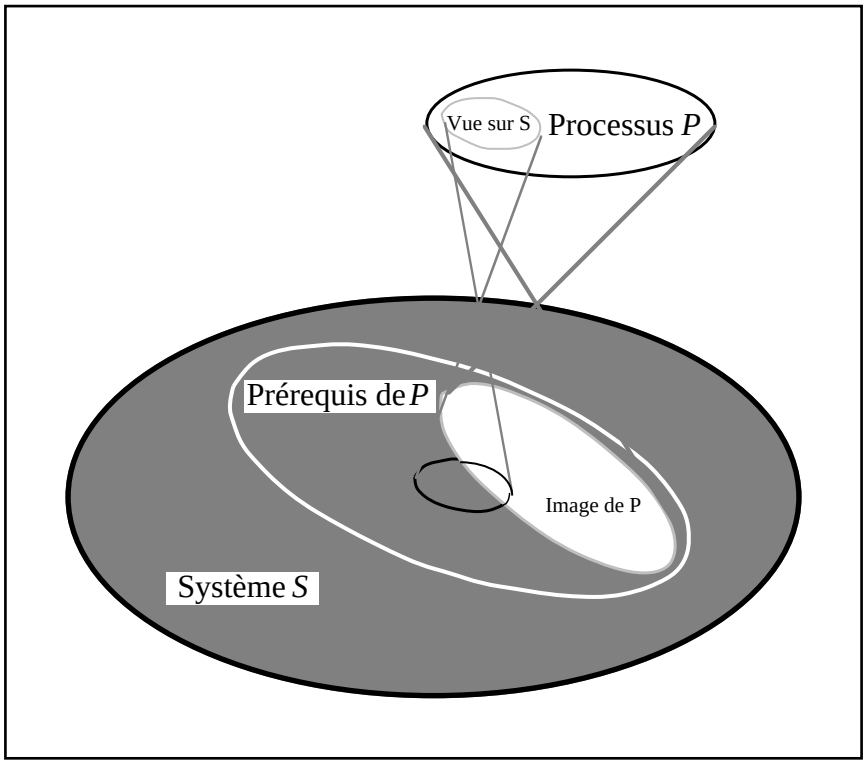


Figure 1 : Illustration des vues entre un processus et le système

Etant donné que les prérequis d'un processus sont nécessaires à son exécution, il est indispensable lors de la migration du processus d'assurer la disponibilité de ses prérequis sur le nouveau processeur d'exécution. Ainsi, à moins de pouvoir y accéder à distance, certains prérequis doivent être transférés sur le processeur destination, telles que les structures de données du système permettant le contrôle de l'exécution du processus. D'autres prérequis - dupliqués sur les différents processeurs d'exécution à travers plusieurs instances - ne doivent pas être transférés. Par exemple c'est le cas de l'horloge d'un processeur, des serveurs de communication, etc. Si le processus peut percevoir une même vue des deux instances qui représentent un prérequis sur les processeurs source et destination de la migration, l'accès à ce prérequis doit alors se faire par rapport à l'instance locale au processeur d'exécution. En revanche, si le processus perçoit différentes vues de ces instances, un problème de cohérence se pose. Nous verrons par la suite comment on peut résoudre ce problème.

Nous sommes donc amenés à assurer, au processus ayant migré, l'accès à ses prérequis de manière transparente en tenant compte des aspects illustrés dans le paragraphe précédent. Pour cela, deux volets sont à étudier :

- le premier volet concerne l'accès aux prérequis où il est question de revoir les moyens de désignation et de localisation de ces prérequis. Le chapitre VI est consacré à ce sujet.

- le second volet traite de la cohérence de la vue qu'a un processus sur chacun de ses prérequis. La section III.4 étudie les conditions de cohérence entre deux instances représentant un même prérequis, l'un se trouvant sur le processeur source et l'autre sur le processeur destination de la migration.

Par ailleurs, la détermination des prérequis ne faisant pas partie de l'image d'un processus n'est pas évidente. Nous divisons ces prérequis en deux groupes :

- ceux qui sont accédés par des appels de procédures système. Nous supposons que ces prérequis sont dupliqués sur les différents processeurs du réseau, car ils correspondent à un ensemble de ressources nécessaires à l'exécution des processus. Mis à part le problème de la cohérence des vues, ces prérequis sont ainsi disponibles.

- ceux qui sont propres au processeur d'exécution du processus ; exemple : des serveurs, des processus application, des segments mémoire, etc. Pour de telles entités nous utilisons le mécanisme de désignation permettant les accès distants ; il n'est donc pas nécessaire de les faire migrer.

Notons cependant que dans les systèmes orientés objets, les prérequis d'un processus peuvent être facilement déterminés, car les dépendances entre les objets sont maintenues dans des structures de données du système. Par exemple, Emerald [JLH88] [Jul88] et SOS [Hab89] [BBK91] maintiennent explicitement pour chaque objet, les références à ses prérequis.

III.1.3 Invocation du système

Un service système peut être invoqué soit par un appel de procédure ("system procedure call"), soit à travers l'envoi d'un message ("message based system") [LaN79]. Dans le cas où le service est sollicité par un appel de procédure, il existe une zone mémoire (pile) partagée par le processus appelant et le système. Lors d'une migration, cette zone mémoire ne peut pas être traitée par une simple copie sur la destination. En effet, il faudrait mettre à jour l'état du système du processeur destination afin que l'invocation du système puisse être poursuivie. Ce problème se pose aussi bien pour les appels système effectués au cours de la migration, que pour les appels système effectués après la migration. En effet, si les processus ont une vue sur l'état du système, celui-ci risque d'être différent d'un processeur à un autre. Ce problème peut se résoudre de deux manières différentes. Lorsqu'un processus ayant migré effectue un appel système, soit cet appel est exécuté sur le processeur destination, soit il est redirigé vers le

processeur source pour être exécuté. Dans le cas de la première solution, le service appelé est considéré comme un prérequis sur lequel le processus doit avoir une vue cohérente. La seconde solution rejoint la technique de masquage par le processeur origine[1] utilisée dans le système Sprite [DoO91] pour un certain nombre d'appels de procédures système. Etant donné les délais rajoutés par le traitement distant de ces appels, la tendance actuelle dans Sprite est de rendre le résultat des appels de procédures système indépendant du processeur d'exécution.

Il est à remarquer que pour les systèmes qui ne sont invoqués qu'à travers des messages, comme par exemple le système V [Che88], le problème des échanges de paramètres et de leur cohérence ne se pose pas. Il est cependant nécessaire d'assurer un acheminement correct des messages entre des processus pouvant migrer. Nous traitons cet aspect dans le chapitre V.

III.2 Suspension et réactivation d'un processus

L'une des actions à entreprendre pour effectuer la migration d'un processus est la suspension, ou encore le gel du processus. Elle consiste à arrêter l'exécution du processus et à le mettre dans un état, dit bloqué, en attente de l'achèvement de la migration. C'est à la suite de la suspension du processus que son image est figée.

Le choix de l'instant de suspension d'un processus dépend de deux aspects :

(1) d'une part, il y a un choix conceptuel qui influe sur la durée de suspension. Par exemple, on peut choisir d'effectuer la suspension après avoir sélectionné le processus, mais avant le début du transfert de l'image du processus ou après avoir transféré certaines parties de l'image du processus, etc.

(2) d'autre part, il n'est pas toujours possible à n'importe quel instant d'extraire et de restaurer l'image complète d'un processus. Ceci est particulièrement le cas lorsqu'un processus est engagé dans un appel de procédure système, ou encore lorsqu'un processus exécute une section non interruptible. Par ailleurs, on peut avoir aussi à certaines phases de l'exécution des registres non accessibles en lecture et/ou en écriture. Ceci est le cas de certains registres du transputer [Inm88], mais il convient de noter qu'en général les processeurs garantissent la suspension d'un processus de façon à ce qu'il soit possible d'obtenir en entier l'état du processeur associé à un processus.

[1] Où a été créé le processus

Le choix de l'instant de suspension en fonction de l'avancement de la migration sera traité au chapitre IV. Nous nous intéressons ici au second aspect auquel il est possible d'appliquer l'une des techniques suivantes.

III.2.1 Retour en arrière

L'idée consiste à reprendre l'exécution à partir d'un état où l'on puisse extraire et remettre le contexte complet du processus. Une analyse du code exécutable est nécessaire afin de déterminer un point de reprise. Ce point doit être choisi de façon à ce que le résultat produit par la séquence d'instructions reprise soit le même que si la séquence avait été exécutée une seule fois.

Cette solution a été adoptée dans [JoC90] pour suspendre un processus en cours de communication. Un inconvénient de cette technique est que l'analyse du code entraîne forcément des délais supplémentaires au niveau de la migration du processus.

III.2.2 Définition de points de reprise

Une autre technique utilisée dans [BaM91] est de laisser au programmeur le soin d'indiquer les points où le processus peut migrer. Ainsi lors de l'écriture du code relatif à un processus pouvant migrer, des lectures sur un canal doivent être insérées régulièrement. Il est alors possible de faire migrer le processus à la suite d'une de ces lectures.

Le principal avantage de cette solution est la simplicité de sa mise en œuvre. En revanche, elle n'assure pas la transparence de la migration puisque le programmeur est obligé de prendre en compte la possibilité de migration en indiquant les points de reprise. Dans Condor [LiS92] les points de reprise sont générés automatiquement sans l'intervention du programmeur. Mais cette approche nécessite la sauvegarde de l'état complet du processus à chaque point de reprise. De plus, les opérations que peut exécuter un processus ont été limitées, par exemple, la communication entre processus n'est pas permise.

III.2.3 Suspension non immédiate

Une troisième technique consiste à retarder la suspension du processus jusqu'à atteindre un état que l'on peut extraire et restaurer sur le processeur destination. C'est ainsi que dans les systèmes Charlotte [ArF89] et Sprite [DoO91], un processus ne peut migrer qu'après avoir achever une communication déjà entamée.

L'inconvénient de cette solution est que le retard au niveau de la suspension risque d'être important, et par conséquent, la décision de migration obsolète.

III.3 Modélisation du système et des applications

Afin d'exprimer les conditions de cohérence pour la migration d'un processus, nous allons modéliser les différentes entités qui peuvent être concernées par la migration. Le modèle utilisé est celui des machines abstraites d'état qui permet de représenter l'évolution de l'état d'une entité et les conditions de transition de cet état.

III.3.1 Spécification d'une entité

Chaque entité *E* est composée d'un ensemble de variables. Les valeurs de ces variables caractérisent l'état S_E de l'entité. L'espace d'états auquelle appartient les valeurs de ces variables est noté $\mathcal{E}$. Formellement on a :

$S_E(t) = v_i(t)_{(1 \le i \le N(t))}$
$\mathcal{E} = \mathcal{E}_1 x \mathcal{E}_2 x ... x \mathcal{E}_{N(t)}$ où

t est l'instant où l'on observe l'entité,
N(t) est le nombre de variables associées à *E* à l'instant *t*,
v_i la valeur d'une variable appartenant à $\mathcal{E}_i$.

Suivant la modélisation par une machine d'état abstraite, une entité *E* est représentée par le quadruplet $<\mathcal{E}$, S0 , $\mathcal{Q}$, $\mathcal{T}>$, avec :

$\mathcal{E}$: espace des valeurs d'état,
S0 : état initial,
$\mathcal{Q}$: ensemble des opérations applicables sur *E*,
$\mathcal{T}$: relation de transition d'état qui détermine le nouveau état de *E* lorsqu'une opération est appliquée sur *E* (de $\mathcal{Q}$ x $\mathcal{E}$ vers $\mathcal{E}$)

Pour la manipulation de l'entité *E*, nous distinguons de la même façon que dans [Lu89] deux types de fonctions : Les V-fonctions et les O-fonctions.

Une *V-fonction* permet d'obtenir la valeur d'une variable à un instant donné. Ainsi il existe pour chaque variable une V-fonction associée. Elle est dite *dérivée* si elle s'exprime en fonction d'un ensemble de V-fonctions.

Une *O-fonction* est une opération appartenant à $\mathfrak{O}$. Elle a pour effet de modifier l'état de *E* et donc d'opérer une transition d'état. A une O-fonction *O* est associé un ensemble d'assertions *A(O)* qui traduisent les relations entre les valeurs des V-fonctions avant et après l'application de *O*. L'ensemble des assertions décrit la relation $\mathfrak{T}$.

III.3.2 Spécification du système et des applications

L'ensemble des entités système nécessaires pour l'exécution des applications, à un instant *t* sur un processeur *X*, est noté $E_X(t)$. Ces entités peuvent être créées et détruites dynamiquement.

$E_X(t) = \{ E_{1,X}, E_{2,X}, ..., E_{n(t),X}\}$ où

$n(t)$ représente le nombre d'entités système à l'instant *t*

Les entités $E_{i,X}$ et $E_{i,Y}$ de même indice sont des entités équivalentes, la première se trouvant sur un processeur *X* et la seconde sur un processeur *Y*. Notons que ces deux entités peuvent avoir des états différents.

L'état de $E_X(t)$ est représenté par :

$S_X(t) = S_{1,X}(t) \times S_{2,X}(t) \times ... \times S_{n(t),X}(t)$ où

$S_{i,X}(t)$ est l'état de l'entité $E_{i,X}$ se trouvant sur le processeur *X* à l'instant *t*.

Nous supposons que l'état courant d'une entité spécifie aussi l'évolution de son état dans le passé.

Par ailleurs, le sous-ensemble d'entités système qui interagissent avec une entité application *A* s'exécutant sur un processeur *X*, à un instant *t*, est noté par $E_X(A, t)$. Rappelons qu'une application accède à l'état des entités système à travers des V-fonctions.

Définition formelle d'une vue

Soit *P* un processus application s'exécutant sur un processeur *X* ; l'ensemble des valeurs lues par le processus *P* jusqu'à l'instant *t* concernant l'état des entités de $E_X(P,t)$ est appelé *vue* du système perçue par le processus *P*, cette vue est notée $V_X(P,t)$.

III.3.3 Exemple

Soit un système qui gère un processeur *S*, considérons une horloge *H* appartenant à $E_S(t)$, où *t* est la date courante. L'horloge *H* est définie par < $\mathfrak{E}$, S0, $\mathfrak{O}$, $\mathfrak{T}$ >, avec :

$\mathcal{E}$ = *1 .. MaxInt*, *MaxInt* étant l'entier maximum,
A $\mathcal{E}$ est associée la variable *date* lue grâce à la V-fonction *V_lire_horloge*,
S0 = 0,
$\mathcal{O}$= *{O_initialise_horloge, O_incrémente_horloge}*,
$\mathcal{T}$: $\mathcal{O}$ x $\mathcal{E}$ -> $\mathcal{E}$ tel que

A(O_initialise_horloge) = {(date' = 0)}
A(O_incrémente_horloge) = {(date' = date + 1)} où

date' désigne la nouvelle valeur de la variable date.

Considérons un processus application *P* qui invoque à l'instant *t* la V_fonction *V_lire_horloge*, sa vue sur le système est alors :

$V_S(P,t) = V_S(P,t') \cup \{V\_lire\_horloge(t)\}$ où

t' est la date de la dernière invocation d'une V-fonction par le processus.

III.4 Conditions de cohérence

Soit *P* un processus qui s'exécute sur un processeur *S* jusqu'à l'instant t_S où il est suspendu pour être migré sur un processeur *D*. Il reprend son exécution sur *D* à l'instant t_r.

L'image de *P* lors de la suspension est représentée par l'état $S_P(t_S)$. Nous omettons dans cette représentation les informations relatives aux adresses physiques d'installation du processus. Une première condition pour que la migration de *P* soit cohérente est que l'image du processus soit la même à la reprise de l'exécution, ce qui revient à :

(C1) $S_P(t_r) = S_P(t_S)$

Notons que cette condition n'exige pas forcément que toute l'image du processus soit transférée sur le processeur destination. En effet, certaines parties de l'espace d'adressage du processus peuvent être ramenées à la demande au cours d'exécution. Le coût pour satisfaire la condition C_1 est donc celui du transfert de l'image d'un processus que nous étudierons au chapitre IV.

La condition C_1 ne suffit pas, le processus *P* peut avoir une vue $V_S(P,t_S)$ sur l'état du système incohérente avec $V_D(P,t_r)$. Considérons l'ensemble des entités système $E_S(P,t_S)$ ayant été utilisées par *P* avant la suspension :

$$E_S(P,t_S) = \{E_{1,S}, E_{2,S}, ..., E_{n(t_S),S}\}$$

L'état de ces entités est noté :

$$S_S(P,t_S) = S_{1,S}(t_S) \times S_{2,S}(t_S) \times ... \times S_{n(t_S),S}(t_S)$$

Une condition suffisante pour avoir la cohérence de vue du système par le processus *P* à la suite de sa migration, est de mettre l'état de l'ensemble des entités se trouvant sur *D*, équivalentes à ceux de $E_S(P,t_S)$, à l'état $S_S(P,t_S)$. Cet ensemble est noté $E_D(P,t)$. Notons que si une entité appartenant à $E_S(P,t_S)$ n'a pas d'équivalent sur *D*, elle est alors créée. Ainsi, la condition de cohérence de la vue par le processus *P* s'énonce comme suit :

(C_2) Pour tout type d'entité *i* indexant un élément de $E_D(P,t_r)$, on a :

$$S_{i,D}(t_r) = S_{i,S}(t_S)$$

La condition C_2 est en pratique difficile à satisfaire. En effet, si d'autres entités sur le processeur *D*, ont déjà une vue sur certaines entités appartenant à $E_D(P,t_r)$, la modification de leurs états peut créer une nouvelle incohérence.

Si l'on reprend l'exemple de l'horloge (cf. §III.3.3), la modification de la valeur de l'horloge d'un processeur donné introduit une incohérence vis-à-vis des processus ayant déjà invoqué *V_lire_horloge* et qui sont encore en exécution.

Dans ce qui suit nous décrivons deux solutions pour résoudre ce problème d'incohérence. La première solution, consiste à donner une vue du système propre à chaque processus. La seconde solution détermine un état pour les entités système qui donne une vue cohérente pour chacun des processus en exécution.

III.4.1 Masquage de l'état des entités système

Une solution, pour pouvoir satisfaire la condition C_2, est de masquer l'état de toutes les entités système de façon à ce que chaque entité ait sa propre vue sur l'état du système. Le résultat d'une V-fonction est donc dépendant du processus ayant invoqué la V-fonction.

La technique la plus simple pour l'implantation de cette solution est de calculer la valeur d'une V-fonction en fonction de la valeur courante de l'état de l'entité et d'un décalage propre au processus ayant invoqué la V-fonction. Cette technique s'applique pour toute variable qui évolue de façon linéaire. Ceci n'est pas toujours le cas, par exemple si l'on considère une variable qui pointe sur une adresse de lecture dans un fichier indexé (i.e., à accès sélectif). La

valeur de cette variable ne suit pas une fonction que l'on peut déterminer. Dans ces conditions, il faut maintenir pour un processus ayant migré sa propre *version* de cette variable. A chaque invocation d'une O-opération affectant la valeur de cette variable, il faut mettre à jour toutes les versions de la variable.

Application dans le cas de l'horloge

Si l'on reprend l'exemple de l'horloge, il suffit de garder un décalage qui est rajouté à la valeur courante de l'horloge à chaque invocation de la V-fonction *V_lire_horloge*. Ce décalage est propre au processus ayant invoqué la V-fonction.

III.4.2 Introduction de critères de cohérence entre les vues

Lu [Lu89] propose de définir des critères de cohérence entre les vues sur une entité. Ces critères sont moins restrictifs que la condition C_2. A chaque V-fonction est associé un invariant entre l'état actuel et l'état précédent, qu'il faut respecter. Cet invariant est déduit de la sémantique du comportement de l'entité. Il ne spécifie pas forcément le comportement complet de l'entité, mais ce qui est supposé au niveau d'un processus application. Cette solution peut s'énoncer de la manière suivante.

Définition des invariants

Soit *V* une V-fonction et *O* une O-opération ; soit deux valeurs de *V*, respectivement avant et après l'application de *O*, représentées par v_i et v_i'. On définit l'invariant de *V* vis-à-vis de *O* et on note *I(V, O)* l'ensemble des prédicats qui doivent être vérifiés à la suite de l'appel de *O*. Ces prédicats s'expriment en fonction de v_i et v_i'.

L'invariant de la V-fonction *V* est :

$I(V) = \bigcup_{O \in \mathcal{O}_V} I(V,O)$ où $\mathcal{O}_V$ est l'ensemble des opérations pouvant affecter le résultat de *V*.

Cohérence des vues

L'évolution des valeurs d'une vue doit respecter un certain nombre d'invariants qui représentent les *contraintes de cohérence* de la vue. Ces invariants sont ceux des V-fonctions invoquées dans la vue.

Soit *S* et *D* deux processeurs, *P* un processus, *t* et *t'* deux dates telles que $t \leq t'$. Une vue $V_S(P,t)$ est cohérente avec une vue $V_D(P,t')$ si et seulement si quelle que soit la V-fonction *V* utilisée à la fois dans les deux vues, *I(V)* est vérifié entre les valeurs de $V_S(P,t)$ et de $V_D(P,t')$. On note alors : $V_S(P,t) \Rrightarrow V_D(P,t')$.

Cohérence de la migration d'un processus

Soit *P* un processus ayant migré d'un processeur source *S* vers un processeur destination *D*. La migration est cohérente si la condition (C_2') suivante est respectée.

(C_2') Pour tout t tel que $t \geq t_r$ on a :

$V_S(P, t_s) \Rightarrow V_D(P,t)$ où

t_s et t_r sont respectivement les dates de suspension et de remise en exécution de *P*.

Cette condition exprime que pour les variables consultées par *P*, sur *S* puis sur *D*, les valeurs lues vérifient les invariants associés aux V-fonctions invoquées pour la lecture de ces variables.

Application dans le cas de l'horloge

Pour illustrer la condition de cohérence C_2', reprenons l'exemple de l'horloge. Soit *P* un processus qui s'exécute sur *S* puis migre vers le processeur *D*. La date de suspension est t_s et la date de reprise de l'exécution sur *D* est t_r ($t_s \leq t_r$). Supposons qu'à la date t_s on a :

$V_S(P,t_s) = \{V\_lire\_horloge(t_1)\}$, avec t_1 une date inférieure à t_s.

L'invariant de *V_lire_horloge* est :

$I(V\_lire\_horloge) = \{(t' \geq t) \wedge ((V\_lire\_horloge(t') \geq V\_lire\_horloge(t))\}$

Pour que la migration soit cohérente il faut que :

$\forall t, t \geq t_r, \quad (V_S(P,t_s) \Rightarrow V_D(P,t))$

Pour cela, l'invariant de *V_lire_horloge* doit être vérifié, ce qui revient à :

$V\_lire\_horloge(t) \geq V\_lire\_horloge(t_1)$

Il suffit donc que la valeur de l'horloge sur *D* soit supérieure à $V\_lire\_horloge(t_1)$. En tenant compte des autres vues sur l'horloge par les divers processus en exécution sur *D*, une valeur pouvant être affectée à l'horloge est :

$Max\{V\_lire\_horloge(t_1), Max\{V\_lire\_horloge(t) \in \{\bigcup_i V_D(P_i,t_r)\}\}$ où

i est un indice permettant d'énumérer tous les processus sur *D* ayant invoqué *V_lire_horloge*,
t un instant quelconque.

III.4.3 Critiques de ces solutions

Le principal avantage des deux solutions, décrites dans les sections III.4.1 et III.4.2, est qu'elles n'induisent pas de dépendances résiduelles sur le processeur source d'une migration.

Cependant, il est nécessaire pour chaque processus de maintenir les vues sur toutes les entités système ayant été invoquées par chacun des processus en exécution. Ainsi, une surcharge du système est introduite à chaque appel d'une V-fonction.

De plus la première solution masquant les entités (cf. §III.4.1) introduit un surcoût pour la gestion des différentes versions d'une même variable. Quant à la solution proposée par Lu, elle nécessite une spécification complète de la sémantique de chaque entité système, ce qui n'est pas toujours simple pour des entités complexes. Par ailleurs, les contraintes de cohérence ne peuvent pas être satisfaites dans tous les cas, ceci dépend des invariants retenus[1]. Lorsqu'ils ne sont pas vérifiés, une demande de migration est refusée. En outre, la détermination des conditions de cohérence entre les vues, avant et après la migration d'un processus, introduit un délai supplémentaire au niveau de la suspension du processus, ce qui retarde l'achèvement de la migration. L'exemple de l'horloge nous montre que pour une entité simple n'ayant qu'une variable d'état, la détermination de la valeur de réinitialisation de l'horloge dépend du nombre de processus en exécution ayant lu à un moment donné une valeur de l'horloge. Ce nombre peut être important.

La conclusion principale de cette étude est que si un processus a une vue sur un nombre important d'entités système, la migration du processus respectant les conditions de cohérences (C_2 ou $C_{2'}$) est coûteuse. Ce coût est d'autant plus critique lorsqu'on se place dans le cadre des systèmes parallèles où les temps de réponse du système doivent être minimisés.

III.5 Solution proposée

Les solutions aux problèmes d'extraction et de restauration d'un processus d'une manière cohérente sont proposées dans le cadre du noyau ParX.

III.5.1 Image d'une Ptask

L'image d'une Ptask est distribuée sur l'ensemble des processeurs du cluster sur lesquels elle s'exécute. Cette image est organisée suivant le modèle de processus de manière hiérarchique :

---

[1] Dans le cas extrême, les invariants peuvent être équivalents à la condition C_2.

- contexte d'exécution d'un thread : identificateur, valeurs des registres (compteur ordinal, pointeur sur l'espace de travail, registre de travail),

- image associée à une task : identificateur, liste des threads, liste des segments mémoires, liste des canaux de communication accessibles par la tâche, liste des ports de communication, identificateur du processeur logique sur lequel s'exécute la tâche,

- image associée à une Ptask : identificateur, liste des processeurs logiques, liste des processeurs physiques, liste des tâches de la Ptask, description du réseau logique, description du réseau physique, correspondance entre le réseau logique et le réseau physique.

Ces informations sont maintenues dans des structures appropriées qui facilitent la détermination des prérequis.

III.5.2 Accès aux prérequis système et invocation du système

Après le transfert de l'image d'une task, il reste à assurer l'accès aux prérequis système. L'invocation du noyau ParX se fait par un appel de procédure. En conséquence, il reste à assurer l'invocation correcte des appels système pour lesquels il faut vérifier les critères de cohérence des vues (C_2).

Comme nous l'avons déjà souligné, les solutions présentées dans les sections III.4.1 et III.4.2 introduisent des surcoûts au niveau du fonctionnement du système. L'intégration de telles solutions dans le noyau ParX serait pénalisante au niveau du temps de réponse du système. Nous proposons une autre solution qui évite ces surcoûts mais qui peut induire des dépendances résiduelles vis à vis du processeur origine.

Au cours de la migration d'une task *T*, si un thread de *T* a déjà effectué un appel de procédure non achevé, l'appel est poursuivi sur le processeur source avant d'être suspendu. A la suite de la migration de la task *T*, nous nous servons du mécanisme de désignation pour résoudre le problème de cohérence des vues. Toute entité système étant désignée par un nom logique, le mécanisme de désignation permet de localiser et d'accéder à l'entité à partir de son nom logique. Si la task *T* a accès à une entité qui peut donner une vue sur son état, alors *T* continue à accéder à la même entité à travers le mécanisme de désignation. En ce qui concerne les entités équivalentes et dupliquées sur le processeur destination, elles sont désignées grâce à un nom de groupe. Un tel nom permet d'accéder à l'entité locale au processeur d'exécution. De cette manière nous minimisons le nombre de prérequis auxquels il faut accéder de manière distante.

Cette solution, à la différence des autres techniques, évite de rajouter dans le noyau une couche intermédiaire pour le contrôle de la cohérence d'éventuelles migrations de processus. Elle utilise plutôt le mécanisme de désignation qui permet de s'abstraire de la localisation physique des entités manipulées. La solution retenue n'introduit donc pas de surcoût sur le fonctionnement du noyau.

III.5.3 Suspension et réactivation d'une *task*

La suspension d'une task revient à suspendre tous ses threads. Inversement, la réactivation d'une task, consiste à remettre tous les threads à leurs états avant la suspension. Le thread est donc l'unité d'exécution sur laquelle il faut agir.

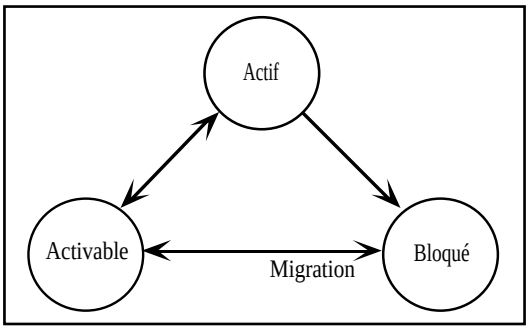


Figure 2 : Graphe de transition d'états d'un thread

La suspension d'un thread consiste à le faire passer à l'état bloqué. Pour cela, nous rajoutons au graphe de transition d'états le passage de l'état activable vers l'état bloqué lors d'une migration (figure 2). ParX a été conçu de façon à exploiter des machines où l'on peut avoir différents types de processeurs, en particulier les transputers [Inm88]. Afin d'augmenter la puissance de calcul et le débit de communication, le transputer intègre au niveau matériel l'ordonnancement des threads, la gestion de l'horloge et des communications. Ces fonctionnalités sont mises en œuvre grâce à des programmes micro-codés utilisant des registres non accessibles en lecture : les registres indiquant la date du prochain processus à réveiller, les registres internes utilisés pour le contrôle du transfert sur les liens de communication, etc. Il en est de même pour les registres sauvegardant l'état de l'unité de calcul flottant[1]. Sans rentrer plus dans le détail des restrictions dues au matériel, la solution que nous proposons est de retarder l'instant de gel du thread jusqu'à ce qu'il atteigne un état que l'on puisse l'extraire et le remettre. La technique de retour en arrière nécessite une analyse du code. La définition de points de reprise est aussi écartée, car elle altère la transparence de la migration.

[1] L'unité de calul flottant n'est interruptible que si le thread qui s'exécute est en basse priorité (un transputer gère deux priorités).

Dans le cas du transputer, la suspension d'un thread ne peut avoir lieu que si ce thread n'est plus dans un état actif [Inm88], d'où un retard possible pour la suspension d'un thread. Par ailleurs, dans le cas où il s'agit d'un thread de basse priorité s'exécutant sur l'unité flottante, il ne faut pas le suspendre par un thread de plus haute priorité, car certains registres non accessibles en lecture/écriture sont utilisés pour sauvegarder le contexte du thread. Nous utilisons dans ce cas aussi la solution retardant la suspension.

III.6 Conclusion

Ce chapitre présente les différents problèmes et contraintes liés à la suspension d'un processus, l'extraction de son image et sa restitution sur un nouveau processeur d'exécution. Ces actions doivent s'effectuer en préservant la transparence de la migration et la cohérence de l'exécution vis-à-vis du processus ayant migré et de son environnement.

La cohérence de la migration d'un processus a été formulée grâce à deux conditions suffisantes (i.e. C_1, C_2). Nous avons ainsi pris en considération différentes solutions pour respecter ces conditions. Le critère de choix d'une solution a été en tout premier lieu la minimisation des délais supplémentaires introduits sur le fonctionnement du système. Les systèmes auxquels nous nous intéressons devant minimiser leurs temps de réponse. Ce critère nous a amené à écarter les solutions qui nécessitent la gestion de l'historique des processus.

La solution que nous proposons est d'utiliser le mécanisme de désignation. Grâce à ce mécanisme, l'accès aux prérequis d'un processus peut se faire de manière locale ou distante selon les critères de cohérence. Ainsi, lors d'une migration d'un processus, certains prérequis sont transférés avec le processus, d'autres sont substitués par des entités équivalentes sur le processeur destination de la migration.

Notons enfin que cette solution ne permet pas de poursuivre l'exécution normale d'un processus ayant migré dans le cas où le processeur source tombe en panne (à cause des dépendances résiduelles). Néanmoins, le processus a la possibilité de réagir à cette situation et de suivre d'autres alternatives d'exécution que le programmeur peut prévoir.

Chapitre IV

Transfert de l'Image d'un Processus

Une fois l'image d'un processus à faire migrer est déterminée, l'étape suivante est de transférer cette image sur le processeur destination. Cette opération fait l'objet de ce chapitre.

Dans le cas où la migration d'un processus est effectuée entre deux processeurs ayant en commun le même espace d'adressage, le transfert du processus se réduit à changer le contexte d'exécution du processus d'une file d'exécution à une autre. En effet, l'espace mémoire adressable par le processus est accessible uniformément de la même manière à partir des deux processeurs, aucune opération de transfert n'est donc nécessaire. Néanmoins, tout en étant accessible à partir des différents processeurs, la mémoire physique peut être distribuée entre différents nœuds d'exécution. La migration d'un processus affecte alors les temps d'accès mémoire. Elle pourrait ainsi servir à favoriser les accès locaux à la mémoire, la stratégie de migration reste alors à déterminer.

En revanche, dans le cas où chaque nœud a son propre espace d'adressage, le transfert de l'image d'un processus à faire migrer reste à considérer. L'aspect le plus important à étudier au niveau de cette opération est la minimisation des coût induits. En effet, l'image d'un processus est souvent de taille importante, son transfert constitue ainsi une surcharge significative pour le réseau de communication. De plus, le temps de transfert qui pénalise le processus en migration est considérable.

Dans ce chapitre, nous analysons diverses techniques réduisant les coûts de transfert. En particulier nous cherchons à déterminer, selon le comportement des applications, les avantages ainsi que les limites de ces techniques. En tenant compte des caractéristiques des architectures parallèles visées et de l'analyse des techniques existantes, nous proposons un algorithme de transfert que nous appliquons au noyau ParX.

IV.1 Préliminaires

Du point de vue taille, la partie dominante de l'image d'un processus est l'espace mémoire adressable, le reste - conservé dans la mémoire système - est généralement de taille réduite. Il comporte, en particulier, les structures nécessaires pour que le processus puisse continuer à s'exécuter et à avoir accès aux entités qu'il manipule.

Nous consacrons ce chapitre principalement au transfert de l'espace d'adressage du processus à faire migrer. Le transfert des structures système pour la communication et l'accès aux objets est traité aux chapitres V et VI. Par ailleurs, nous supposons que la migration d'un processus n'engage pas le transfert de fichiers, l'accès aux fichiers étant assuré au moyen du mécanisme de désignation.

L'opération de transfert a été l'un des aspects de la migration de processus les plus traités dans la littérature. L'intérêt porté à cette opération est dû au fait que le transfert de l'espace d'adressage en entier induit un coût important au niveau du temps de transfert et de l'encombrement du réseau. Ce coût est prédominant par rapport à ceux des autres opérations de migration, particulièrement lorsque la taille du processus est importante. En effet, plusieurs évaluations de systèmes existants comme par exemple Accent [Zay87ab], Charlotte [ArF89] Sprite [DoO91], Emps [DGi92], Mach [MZD93] montrent que le coût de transfert varie de façon linéaire en fonction de la taille de l'espace d'adressage.

La plupart des techniques proposées ont eu pour premier objectif la réduction du temps de gel d'un processus à faire migrer ainsi que la réduction des volumes de données transférées. Les techniques de transfert diffèrent selon :

- que l'image d'un processus est entièrement transférée ou qu'elle est maintenue en partie sur le processeur source de la migration,

- le choix des instants de suspension et de reprise de l'exécution du processus par rapport à la progression du transfert : suspension du processus avant le début du transfert, suspension après le transfert d'une partie de l'image, reprise avant le transfert complet de l'image, etc.,

- le support de transfert, par exemple, à travers le réseau de communication, à travers un disque, etc. En particulier, les caractéristiques des diverses architectures parallèles peuvent être exploitées pour améliorer les temps de transfert.

Dans ce qui suit nous analysons les diverses techniques existantes. Nous élaborons, ensuite, la solution adoptée pour le noyau ParX.

IV.2 Copie directe de l'espace d'adressage et pré-copie

IV.2.1. Principe

La copie directe consiste à transférer l'espace d'adressage entier, d'un processus à faire migrer, sur un nouveau processeur d'exécution. La taille de l'espace mémoire transféré est donc maximale. Dans le cas où le processus n'accède qu'à une faible portion de cet espace, la copie directe induit un coût de transfert inutile d'autant plus grand que la taille de l'espace mémoire transféré est importante.

Plusieurs systèmes comme par exemple Charlotte, Demos/MP, Locus et V utilisent la copie directe. Mis à part le système V, les autres systèmes opèrent le transfert pendant le gel du processus, ce qui accroît le délai de suspension du processus. Pour réduire ce délai, Theimer propose une technique de pré-copie qu'il utilise pour le système V [The86].

Technique de pré-copie

La pré-copie consiste à entamer la copie de l'espace d'adressage d'un processus avant de le suspendre. Le transfert pourra donc se faire en parallèle avec l'exécution du processus. Une première copie est effectuée, elle est suivie de copies répétées des parties ayant été modifiées par le processus. Theimer suppose que les délais des copies successives sont décroissants et que l'on peut ainsi atteindre une faible taille qui reste à recopier, auquel cas, le processus est suspendu. La dernière copie est alors faite en un délai de temps réduit. Afin d'assurer la terminaison de l'algorithme Theimer suggère d'utiliser une valeur maximum du nombre de copies possibles.

IV.2.2. Analyse de la technique de pré-copie

La pré-copie permet de réduire le temps de gel, cependant elle augmente le délai compris entre l'instant où la migration est déclenchée et l'instant où le processus reprend son exécution sur le nouveau processeur. Cet inconvénient est majeur dans le cas où la migration d'un processus doit se faire le plus rapidement possible pour des raisons de pannes par exemple. De plus, la pré-copie induit un volume de transfert plus important. Notons que la technique de pré-copie nécessite un support matériel pour la détermination des zones mémoire modifiées au cours de la phase de pré-copie, sans quoi, l'implantation de cette technique au niveau logi-

ciel serait prohibitive. Afin de mieux apprécier les coût induits par la pré-copie, nous effectuons dans ce qui suit une étude analytique de cette technique.

Les accès mémoire en écriture sont modélisés par une fonction de la manière suivante : soit *P* un processus, nous définissons la fonction *Taille_modifiée(P,d)* comme étant la taille de l'espace mémoire modifié par le processus *P* durant un délai de temps *d*.

Notation

- T_P : la taille de l'espace d'adressage du processus *P*,
- *V* : la vitesse de transfert à travers le réseau que nous supposons constante,
- D_i : la durée de la i ème copie lors de la pré-copie de l'espace d'adressage de *P*,
- *C* : durée de transfert (i.e., durée de la pré-copie y compris la dernière copie),
- *G* : durée de gel due à l'opération de transfert,
- T_t : taille totale transférée.

Nous considérons un comportement linéaire pour la fonction *Taille_modifiée* borné par la taille T_P de l'espace d'adressage du processus :

$Taille\_modifiée(P,d) = Min\ (E_P * d, T_P)$ où

> E_P est le taux d'accès mémoire (en écriture) effectué par *P* à de nouvelles zones mémoire pendant une durée *d*.

Pour que la technique de pré-copie puisse apporter un gain, deux conditions sont nécessaires :

(C_1) $E_P < V$

(C_2) $\forall i,\ D_i < T_P / E_P$

Autrement, la technique de pré-copie est inutile puisque l'espace d'adressage du processus sera modifié en entier avant la terminaison de la première copie. Suivant la condition C_2, on a :

$\forall i,\ Taille\_modifiée(P,D_i) = E_P * D_i$

D'où :

$D_1 = T_P / V$

$D_2 = Taille\_modifiée(P, D_1)/V = T_P * E_P * V^{-2}$

$D3 = Taille\_modifiée(P, D_2)/V = T_P * E_P^2 * V^{-3}$

$$D_n = Taille\_modifiée(P, D_{n-1})/V = (T_P/V)*(E_P/V)^{n-1}$$

Supposons que la n ième copie soit la dernière, on a alors :

$$C = \sum_{1 \le i \le n} D_i = (T_P/V) * \frac{1-(E_P/V)^n}{1-(E_P/V)}$$

$$T_t = C*V = T_P * \frac{1-(E_P/V)^n}{1-(E_P/V)}$$

$$G = D_n = (T_P/V)*(E_P/V)^{n-1}$$

T_P/V représente le temps de transfert lorsqu'on applique la copie directe sans pré-copie. Ce temps est inclus dans le temps de gel.

Dans le cas où le processus accède le plus souvent à une même région de la mémoire, le taux d'accès E_P est alors faible, il est tout au moins inférieur à V (selon la condition C_1). Si l'on néglige $(E_P/V)^n$, on a :

$$C = T_P/(V-E_P) \qquad T_t = T_P*V/(V-E_P) \qquad G << T_P/V$$

Ainsi la technique de pré-copie est viable si la valeur de E_P est nettement inférieure à celle de V. Par exemple, pour E_P/V égale à 0.5, le temps de transfert de même que la taille transférée sont doubles par rapport à une copie directe. Néanmoins le temps de gel reste faible. Dans ces conditions, si la migration est effectuée pour réduire le temps d'exécution du processus, la technique de pré-copie est intéressante. En revanche, le temps de transfert est considérablement augmenté.

Considérons maintenant le cas où E_P est proche de V. Pour que le temps de gel soit faible par rapport à T_P/V- objectif de la pré-copie - il faut que n soit important. En effet, Pour réduire ce temps il faut réduire $(E_P/V)^n$, or E_P/V est proche de 1. On a alors :

$$C >> T_P/V \qquad T_t >> T_P \qquad G << T_P/V$$

Par exemple, pour E_P/V égale à 0.9 il faut plus de 6 itérations pour que l'on réduise de moitié le temps de gel (par rapport à la copie directe). Le temps de transfert, ainsi que la taille transférée, sont multipliés par 5. Ce coût est considérable et représente une surcharge importante pour le système.

Ainsi pour que la pré-copie puisse réduire le temps de gel, elle impose (dans bien de cas) une surcharge importante du réseau de communication. De plus, elle retarde l'achèvement de l'opération de migration.

Dans la section IV.8.3, nous appliquons la technique de pré-copie à des applications numériques où nous illustrons les avantages et les limites de cette technique.

IV.3 Transfert à la référence

IV.3.1. Principe

A la migration d'un processus *P*, la technique de transfert à la référence, encore appelée transfert à la demande, n'effectue pas le transfert complet de l'espace d'adressage associé à *P*. C'est après la reprise de l'exécution de *P* que les zones mémoire référencées par le processus sont chargées physiquement dans lamémoire locale. Cette méthode est dite paresseuse ("lazy transfer") puisqu'elle n'entreprend un transfert que lorsque c'est nécessaire. Ainsi, lors de la migration de *P*, l'espace d'adressage associé à *P* est gardé en grande partie sur le processeur source.

L'implantation de cette technique se fait au niveau du gestionnaire mémoire qui contrôle les accès mémoire. Il permet de virtualiser l'espace d'adressage en le rendant indépendant de sa localisation physique. L'espace mémoire est découpé en pages, la page étant l'unité élémentaire de transfert.

Différentes variantes peuvent être envisagées :

- au cours de la migration, on effectue la copie d'une partie de l'espace d'adressage,

- au cours de l'exécution du processus, et lorsque il y a un défaut de page (i.e., la page référencée n'est pas présente sur le processeur d'exécution) on peut ne ramener que la page référencée, mais aussi on peut précharger d'autres pages en prévision de nouveaux défauts de pages. Cette opération est mieux connue sous le terme "prefetch".

Le transfert à la référence a été proposé et utilisé par Zayas pour l'implantation de la migration dans le système Accent [Zay87ab]. Afin d'évaluer cette technique, Zayas a choisi un certain nombre d'applications sur lesquels il a appliqué la migration de processus. Les mesures qu'il a effectuées montrent que le délai de migration est réduit d'un facteur 1000 par rapport à la copie directe. La durée d'exécution des processus ayant été migrés ainsi que les volumes de données transférées induits par ces migrations sont réduits approximativement de moitié.

Nous effectuons dans ce qui suit une évaluation analytique de la technique de transfert à la référence.

IV.3.2. Analyse

La technique de transfert à la référence réduit la taille de l'espace mémoire copié lors de la migration, et en conséquence, la durée du transfert et du gel. Cette technique est d'autant plus justifiée lorsqu'un processus n'accède qu'à une faible partie de sa mémoire. Zayas considère que cette condition est généralement vérifiée.

Toutefois, la technique de transfert à la référence risque d'être pénalisante lorsque la migration est opérée au début de l'exécution du processus. Ceci apparaît même à travers certaines mesures effectuées par Zayas. En effet, et contrairement à l'hypothèse où l'on suppose qu'une application accède à une faible portion de son espace d'adressage, certaines applications accèdent à une grande partie de leurs espaces d'adressage durant leurs exécutions. Des exemples de telles applications sont illustrées dans la section IV.8.4.

Par ailleurs, en adoptant le transfert à la référence, une dépendance résiduelle vis-à-vis du processeur source est maintenue puisque les pages non encore transférées sont maintenues sur ce processeur jusqu'à ce qu'elles soient transférées ou que le processus ait terminé son exécution. Cette technique n'est donc pas applicable lorsque la migration est effectuée pour libérer le processeur source.

Une approximation de la durée de gel G est la suivante :

$G = T_{init}/V$ où

T_{init} est un paramètre de l'algorithme de transfert qui désigne la taille initialement transférée par défaut,

V la vitesse de transfert.

La taille totale transférée comprend celle transférée initialement T_{init} et les transferts à la demande effectuées à la suite de la migration. A chaque défaut de page, l'exécution du processus est retardée le temps que la page soit ramenée. A ce temps se rajoute le délai dû au fait que le processus ne peut pas reprendre directement son exécution puisque le processeur est probablement occupé par un autre processus. Le traitement des défauts de pages représente un surcoût supplémentaire qui se rajoute à l'opération de transfert. Nous traduisons ce surcoût par un délai R que nous supposons, pour simplifier, constant pour chaque défaut de page. Ainsi lors d'un défaut de page, le retard infligé au processus en exécution est :

$\Delta = T_d/V + R$ où

T_d représente la taille de l'espace mémoire ramené à chaque défaut de page.

Le retard global Δ_t infligé au processus et dû aux opérations de transfert peut se calculer de la manière suivante :

$\Delta_t = G + n*\Delta = T_{init}/V+n*(T_d/V+R)$ où

n est le nombre de défauts de pages avec $n \leq (T_P-T_{init})/T_d$.

Dans le cas général la vitesse de transfert V dépend de la taille transférée, elle est plus importante pour les gros messages. Ainsi, lors du traitement d'un défaut de page, la vitesse V est réduite par rapport à celle pour la copie directe. Néanmoins dans ce qui suit, nous supposons que la vitesse V est la même indépendamment de la taille du message, ce qui est à l'avantage du transfert à la référence.

Pour que le retard Δ_t reste inférieur au temps de transfert par une copie directe il faut que :

$\Delta_t < T_P/V$

Ce qui est équivalent à :

(C3) $n < (T_P-T_{init})/(T_d+R*V)$

Il existe donc une limite au nombre de défauts de page à partir de laquelle le temps de transfert devient plus important que pour la copie directe. Notons que plus le produit $R*V$ est important, moins il faut qu'il y ait de défauts de page (C3). Ce produit croit suivant la charge du processeur. En effet, le retard R comprend le temps où le processus est en attente pour reprendre l'exécution (après que le défaut de page ait été traité). Ainsi, plus il y a de processus qui partagent le processeur plus l'attente est longue. Si l'on suppose que :

- la taille ramenée à chaque défaut de page T_d est de l'odre du Kilo-octets[1],
- le retard R est de l'ordre de la milli-seconde,
- la vitesse V de l'ordre de la dizaine de Méga-octets/seconde,

alors le produit $R*V$ n'est pas négligeable par rapport à T_d .

[1] Comparable à la taille d'une page.

Néanmoins même si $\Delta_t \geq TP/V$, la technique de transfert à la référence garde un avantage puisque le temps de gel aura été réduit : la taille transférée pendant le gel est T_{init}, elle est inférieure à TP.

IV.4 Utilisation d'un système de stockage partagé

Une autre technique pour transférer l'espace d'adressage, d'un processus à faire migrer, est de sauvegarder les pages modifiées sur un système de stockage partagé par les processeurs source et destination de la migration. Cette opération est effectuée pendant le gel du processus. Les zones, ou plus communément les pages, mémoire utiles pour la poursuite de l'exécution du processus sont ensuite récupérées à partir du système de stockage sur le nouveau processeur d'exécution. Notons que la mémoire de stockage n'est pas forcément composée que de disques, elle peut aussi comporter de la mémoire cache répartie dans le réseau. L'opération de transfert peut donc, dans certains cas, s'effectuer sans passer par un disque.

Cette solution a été adoptée dans le système Sprite [DoO91] qui possédait déjà des mécanismes de gestion mémoire supportant en grande partie ce principe de transfert. Dans le système Mosix [BGW93], cette solution a été combinée avec la copie directe : les pages modifiées sont transférées directement sur le nouveau processeur d'exécution, les autres pages sont récupérées à partir de la mémoire de stockage.

Grâce à la technique de transfert à travers un système de stockage, la taille T_m à transférer pendant le gel se limite à la taille mémoire modifiée et non encore mise à jour sur la mémoire de stockage. Selon cette taille, la durée de gel dûe au transfert (G) sera donc plus ou moins importante.

$G = T_m/V$ où

V est la vitesse de transfert.

Comme pour le transfert à la référence, après la reprise de l'exécution du processus, les pages sont ramenées sur le nouveau processeur d'exécution suite à des défauts de page. Si les pages transférées sont ramenées à partir du disque, la vitesse de transfert va dépendre aussi des délais d'accès disque.

IV.5 Transfert en parallèle à travers plusieurs chemins

Les trois techniques que nous venons de présenter considèrent le transfert d'un processus indépendamment du chemin suivi. Dans le cas d'un réseau multipoint, où les processeurs partagent un même câble ou bus de transmission, le réseau est complètement partagée entre les processeurs. Il existe donc un seul chemin de transfert entre deux processeurs. En revanche, dans le cas des réseaux maillés, différents chemins peuvent exister entre deux processeurs. En outre, la charge n'est pas a priori uniformément répartie entre les liens d'interconnexions. Dans le cadre de telles architectures on pourrait envisager des protocoles dédiés au transfert de gros volumes de données à travers plusieurs chemins alternatifs. La détermination de tels chemins a été étudiée par Saad et Schultz pour différentes topologies régulières : anneau, grille, hypercube [SaS88, 89]. Nous proposons dans la section IV.9 un protocole de transfert à travers plusieurs chemins alternatifs pour des topologies quelconques.

Un autre aspect qui a été le sujet de nombreux travaux est celui de la migration d'un ensemble de processus à la fois à travers des réseaux d'interconnexion réguliers. On peut avoir besoin de faire migrer un ensemble de processus, par exemple, pour résoudre le problème de fragmentation de la machine (cf. §I.2.2). La solution se ramène alors à faire migrer un ensemble de processus d'une partie de la machine à une autre de façon à réorganiser les processeurs libres. L'objectif des algorithmes proposés est de minimiser le temps de transfert d'un ensemble de processus. Pour cela, ces algorithmes déterminent une correspondance entre un ensemble de processeurs source et un ensemble de processeurs destination ainsi que les chemins de transfert de façon à éviter les contentions. Des exemples de réseaux réguliers pour lesquels de tels algorithmes ont été proposés sont : les hypercubes [ChS89], les réseaux cube multi-étages [SSC89,90], les grilles (éventuellement rebouclées) [Bok93]. Notons que ces algorithmes sont du niveau décisionnel puisqu'ils déterminent quels processus migrer et vers quelles destinations.

IV.6 Transfert par reconfiguration du réseau

IV.6.1. Principe

Dans le cas des architectures reconfigurables (par exemple le Supernode [MuW90]), une technique de transfert serait d'établir une connexion directe entre les processeurs source et destination d'une migration. Ceci nécessite qu'il y ait, sur chacun des deux processeurs, au moins un lien que l'on peut connecter à l'autre sans avoir à défaire les autres connexions. Si le

réseau est non bloquant[1] cette condition est satisfaite. En revanche, si le réseau est seulement réarrangeabe[2] cette condition peut ne pas être satisfaite. Dans ce dernier cas, on peut envisager la possibilité d'établir un chemin de longueur supérieure à 1 entre les processeurs sources et destination. Les processeurs intermédiaires seront alors impliqués directement dans la reconfiguration.

En outre, pour mettre en œuvre une telle technique, il faut aussi assurer l'acheminement correct des messages en transit à travers les liens reconfigurés. Pour cela, une première solution est de bloquer la communication sur les liens (reconfigurés) en attente de leur rétablissement. Une deuxième solution consiste à emprunter un chemin alternatif ne passant pas par les liens reconfigurés. Ceci n'est pas toujours possible, par exemple lorsqu'un lien impliqué dans la reconfiguration est le seul permettant d'atteindre une certaine destination.

IV.6.2. Analyse

Dans [AKn90], une étude expérimentale comparant la communication par reconfiguration et la communication par routage de messages montre que la reconfiguration devient intéressante pour des messages de tailles importantes. Pour des tailles supérieures à 1 Ko, on obtient un gain en temps de transfert, de l'ordre de 8. Bien sûr ce résultat est relatif à une machine spécifique et un algorithme de routage particulier. Néanmoins, il montre que la possibilité de transfert par reconfiguration est à considérer, notamment dans le cas du transfert de l'espace d'adressage d'un processus dont la taille est généralement importante.

L'intérêt du transfert par reconfiguration dépend principalement d'une part du coût de la reconfiguration et d'autre part de la perturbation introduite dans le réseau.

Coût de la reconfiguration

Pour que la reconfiguration soit viable, il faut que le temps de reconfiguration et de transmission ne soit pas supérieur au temps de transfert par routage sans modification de la topologie du réseau. Le temps de reconfiguration dépend des caractéristiques de la machine. Par exemple dans le cas du Supernode [MuW91], la reconfiguration est faite grâce à un processeur contrôleur qui commande les commutateurs d'interconnexion. Les demandes de reconfiguration devront être envoyées à un gestionnaire de reconfiguration (sur le contrôleur). Le temps de reconfiguration comprend ainsi les temps d'envoi de la requête, d'attente dans une file pour être servi, de service (calcul des commandes des commutateurs, envoi des com-

[1] Un réseau est non bloquant s'il est capable d'établir ou de supprimer une connexion sans interrompre les autres liaisons existantes.
[2] Un réseau est réarrangeable s'il permet de réaliser n'importe quel graphe de connexion entre deux sous-ensembles de liens (les deux sous-ensembles ont la même cardinalité).

mandes, positionnement des commutateurs) et d'envoi de l'acquittement de la requête. Une étude plus détaillée sur l'évaluation des temps de reconfiguration et de transfert est faite dans [Gon91].

Perturbation introduite par la reconfiguration

Dans le cas où on utilise un lien persistant[1], la reconfiguration dynamique produit une perturbation. Soit on bloque les messages passant par ce lien jusqu'à ce qu'il soit rétabli, soit on modifie le routage pour prendre un chemin alternatif ne passant pas par ce lien. La perturbation introduite dans ce dernier cas est le temps nécessaire pour la correction du routage et l'éventuel allongement des routes. Par ailleurs, la reconfiguration favorise une seule communication qui se réserve les liens affectés par la reconfiguation. Plus la durée de reconfiguration est importante, plus la perturbation des autres communications est sensible.

Bien que cette technique puisse améliorer considérablement les temps de transfert, sa mise en œuvre est complexe et nécessite un noyau de communication adapté. De plus, les architectures reconfigurables posent des problèmes d'extensibilité [Wai91] [Mug93].

IV.7 Comparaison des techniques de transfert

Les techniques que nous venons d'étudier agissent sur deux facteurs pour optimiser le transfert de l'espace d'adressage. Le premier facteur est la répartition de l'opération de transfert dans le temps. Le second facteur concerne les moyens de transfert. La figure 1 montre les différentes répartitions dans le temps de l'opération de transfert pour les techniques de copie directe, de pré-copie et de transfert à la référence. La figure 2 illustre le second aspect à travers divers exemples de transfert qui diffèrent selon les supports matériels mis en œuvre.

Par ailleurs et comme on peut le remarquer, plusieurs solutions de transfert peuvent être élaborées en agissant sur ces deux facteurs. Par exemple la solution retenue dans Sprite combine le transfert à la référence et le transfert à travers une mémoire de stockage.

Le tableau de la figure 3 énumère brièvement les critiques qualitatives que l'on peut porter à chacune des techniques présentées dans les sections précédentes.

[1] Lien déjà utilisé pour l'acheminement des messages.

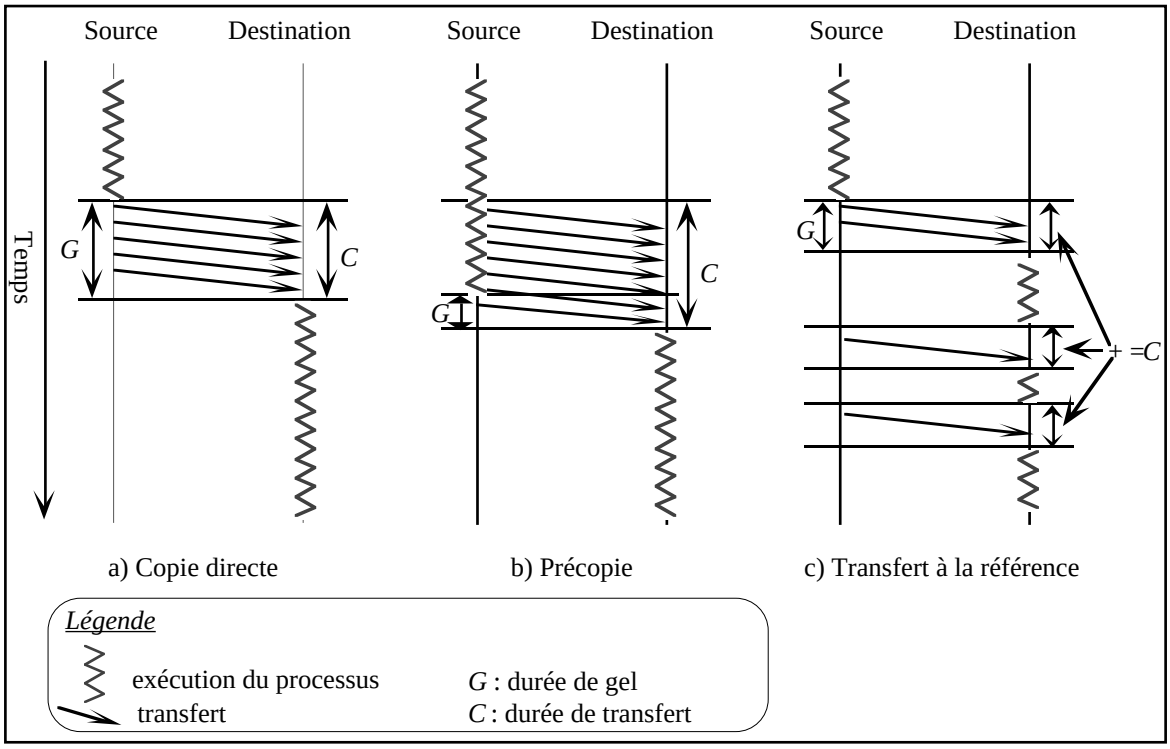


Figure 1 : Répartition de l'opération de transfert dans le temps

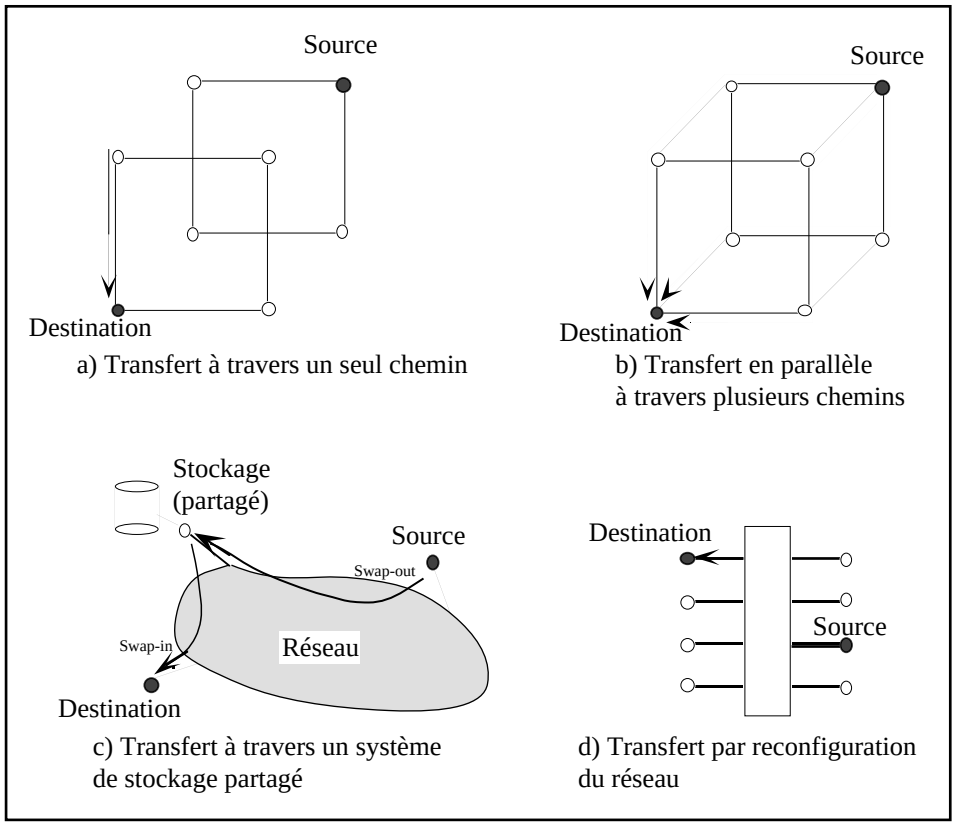


Figure 2 : Moyens de transfert

TECHNIQUES	AVANTAGES	INCONVÉNIENTS
Pré-copie	- indépendance vis-à-vis du processeur source - réduction du temps de gel	- la taille mémoire transférée peut devenir supérieure à celle de l'image du processus, le temps pour réaliser la migration s'accroît en conséquence
Transfert à la référence	- réduction du temps de gel - transfert uniquement de ce qui est utile pour le reste de l'exécution	- dépendance résiduelle vis-à-vis du processeur source, le processeur source n'est pas complètement libéré du point de vue charge - le traitement des défauts de page ralentit l'exécution du processus
Transfert à travers un système de stockage partagé	- indépendance vis-à-vis du processeur source - simplicité, aucun mécanisme spécifique de transfert	- nécessite un service fichier accessible à partir des deux processeurs - le serveur de fichiers devient un goulot d'étranglement : l'efficacité de ces méthodes dépend de celle du service fichier - le temps d'accès disque se rajoute au délai de transfert
Transfert à travers plusieurs chemins	- réduction du temps de transfert - meilleur distribution de la charge de communication	- détermination des chemins alternatifs
Transfert par reconfiguration	- réduction du temps de transfert (transfert entre voisins)	- complexe à mettre en œuvre - perturbe les communications en cours

Figure 3 : Principales caractéristiques des techniques de transfert

IV.8 Etudes de cas

Nous nous proposons dans cette section d'illustrer certains aspects, déjà évoqués, concernant les techniques de transfert, et ceci à travers deux exemples d'application dans le domaine du calcul numérique : la somme et le produit de deux matrices. Les techniques de transfert qui sont prises en compte, dans cette section, se distinguent par une répartition particulière de l'opération de transfert dans le temps. Les autres techniques, comme par exemple le transfert à travers plusieurs chemins ou le transfert par reconfiguration, ne dépendent pas du comportement de l'application.

Les matrices que nous considérons sont des matrices carrées 512 x 512 de réels. Un réel étant représenté sur 8 octets, la taille d'une matrice est donc 2 Moctets. Un élément d'une matrice A est noté $A_{i,j}$, où i est le numéro de ligne et j le numéro de colonne. Nous supposons qu'une matrice est rangée par colonnes dans la mémoire. Ainsi, les éléments d'une matrice A sont disposés successivement dans la mémoire dans l'ordre : $A_{1,1}, A_{2,1}, ... A_{512,1}, A_{1,2}, A_{2,2}, ... A_{512,2}, ... A_{1,512}, A_{2,512}, ... A_{512,512}$.

L'application *Somme_matrice* effectue la somme de deux matrices A et B et range le résultat dans une matrice S. Il s'agit donc d'effectuer les opérations suivantes :

$S_{i,j} = A_{i,j} + B_{i,j}$ pour tout *i, j* variant de 1 à 512.

L'application *Produit_matrice* effectue le produit de deux matrices *A* et *B* et range le résultat dans une matrice *P*. Il s'agit d'effectuer les opérations suivantes :

$$P_{i,j} = \sum_{1 \le k \le 512} A_{i,k} * B_{k,j}$$ pour tout *i, j* variant de 1 à 512.

Le calcul des matrices *S* et *P* est effectué par colonnes.

Les deux applications, ainsi définies, ont des comportements d'accès mémoire différents :

- la valeur de $S_{i,j}$ est déterminée à la suite d'une opération d'addition, alors que la valeur de $P_{i,j}$ est déterminée à la suite d'un millier d'opérations d'addition et de multiplication. Ainsi la taille mémoire modifiée pendant une durée de temps est plus importante dans le cas de l'application *Somme_matrice*.

- la progression des accès mémoire dans le cas de *Somme_matrice*, par rapport à chaque matrice, est séquentielle. De plus, un élément d'une matrice n'est accédé qu'une seule fois. En revanche, dans le cas de *Produit_matrice*, pour le calcul d'un élément $P_{i,j}$ il faut accéder à toute la ligne *i* de *A* et à toute la colonne *j* de *B*. Notons que les éléments d'une même ligne ne sont pas rangés suivant des adresses consécutifs. Un peu moins de 2 Mo séparent le premier et le dernier élément d'une ligne.

Etant données ces deux applications, nous nous intéressons uniquement au transfert des deux matrices *A*, *B* opérandes et de la matrice résultat (*S* ou *P*) suite à la migration de chacune de ces deux applications. La taille du code et des autres structures de données associés à un processus application est négligeable par rapport à la taille de ces matrices.

IV.8.1. Utilisation de la technique de pré-copie

La technique de pré-copie présente des performances différentes selon l'application. En effet, dans le cas de *Produit_matrice*, le taux d'accès mémoire en écriture $E_{Produit}$ à de nouvelles zones mémoire est faible alors que celui de l'application *Somme_matrice* E_{Somme} est important. Nous avons mesuré ces taux d'accès en exécutant, sur des transputers T800 fonctionnant à 20 MHz [Inm89], les deux applications écrites en langage C. Nous avons obtenu les résultats suivants :

E_{Somme} = 2.15 Mo/s

$E_{Produit}$ = 2.25 Ko/s

Dans le cas de *Somme_matrice*, la vitesse de communication *V* doit être supérieure à 2.15 Mo/s (cf. §IV.2.2). Si l'on suppose que l'on a une vitesse de communication *V* de 3 Mo/s, pour réduire le temps de gel à moins d'une seconde il faut effectuer 6 pré-copies. Le temps *C* pour que le transfert soit complété est alors proche de 20 secondes, ce qui est 10 fois plus supérieur que pour une copie directe.

Si l'on considère maintenant l'application *Produit_matrice*, pour *V* égale à 1Mo/s et en effectuant une seule pré-copie, on a un temps de transfert d'environ 6 secondes et un temps de gel *G* de 13 ms. Si l'on applique la copie directe, le temps de gel (ou de transfert) est aussi de 6 secondes. La pré-copie permet ainsi de réduire le temps de gel sans augmenter le temps global de transfert. Rappelons que pour ces mesures, on ne considère que le transfert des matrices *A*, *B* et *P* dont la taille globale est de 6 Mo.

IV.8.2. Utilisation de la technique de transfert à la référence

L'une des caractéristiques de l'application *Produit_matrice* est qu'elle a besoin de toute la matrice *A* pour le calcul d'une colonne. Ainsi même s'il ne reste que la dernière colonne à calculer, il faut transférer la matrice *A* dont la taille, rappelons le, est de 2 Mo. Au transfert de cette matrice se rajoute celui des colonnes de *B* qui correspondent aux colonnes de *P* non encore calculées. Ainsi, si la migration intervient dans les premières phases de calcul, une grande partie de la matrice *B* sera transférée. La technique de transfert à la référence sera alors pénalisante dans la mesure où elle rajoute les surcoûts de traitement des défauts de page pour une taille transférée comparable à celle effectuée par une copie directe.

Dans le cas de *Somme_matrice*, la taille de l'espace mémoire utile à transférer est linéairement dépendante de l'avancement des calculs. Là aussi, nous remarquons que le transfert à la référence s'adapte mal si la migration est opérée au cours des premières phases de calcul.

En revanche, la technique de transfert à la référence se prête bien pour la réduction des temps de gel et de transfert lorsqu'il ne reste plus beaucoups d'éléments à calculer. Encore faut-il trouver l'utilité d'une migration en fin de calcul tout en gardant des dépendances résiduelles !

Ainsi le nombre de défauts de page dépend de l'âge du processus au moment de la migration. Par exemple, si la migration intervient après que la moitié des calculs ont été effectués, l'application *Somme_matrice* va devoir ramener 3 Mo (sur 6 Mo) par des défauts de pages, l'application *Produit_matrice*, quant à elle ramène 4 Mo.

IV.9 Proposition d'une solution

Dans cette section nous présentons la solution élaborée pour le transfert de l'image d'un processus dans le cas de ParX. Nous décrivons d'abord la répartition dans le temps de l'opération de transfert, nous abordons ensuite le problème de l'acheminement des gros messages qui représentent dans notre cas l'image d'un processus. La migration de fichiers est un autre exemple où le volume des données transférées est important. Les algorithmes d'acheminement sont proposées dans le cadre d'un protocole de transfert qui n'est pas propre à la migration de processus, mais que nous avons eu à le développer.

IV.9.1. Répartition de l'opération de transfert dans le temps

L'élaboration d'un algorithme de transfert de l'espace d'adressage d'un processus dépend en premier lieu du mécanisme de gestion mémoire. Par exemple, les techniques de transfert à la référence et de chargement à partir de la mémoire de stockage, ne sont applicables que si le mécanisme de gestion mémoire permet de gérer les défauts de page.

Dans la version actuelle de ParX, l'espace d'adressage d'une task n'est pas paginé et est entièrement chargé sur le processeur d'exécution. Des études sont en cours pour la mise en œuvre d'un mécanisme de cache mémoire qui permettrait à un processus de s'exécuter sans que tout son espace mémoire ne soit forcément chargé sur le processeur local. En attendant la mise en œuvre de ce mécanisme, la solution que nous proposons est de transférer entièrement l'espace mémoire résident avant de reprendre l'exécution du processus.

Mis à part le fait que l'on ne dispose pas de mécanisme de gestion des défauts de page, nous avons écarté le transfert à la référence à cause des dépendances résiduelles (vis-à-vis du processeur source de la migration) induits par cette technique. De plus, l'efficacité de cette solution est conditionnée par l'évolution des accès mémoire du processus et par le degré d'avancement du processus dans son exécution (cf. §IV.3.2 et §IV.8.4).

Par ailleurs, pour réduire le temps de gel, on pourrait envisager de combiner la solution retenue avec la technique de pré-copie. Cependant cette technique présente un comportement très différent selon l'application : la pré-copie risque de retarder la migration et de rajouter une surcharge de communication (cf. §IV.2.2 et §IV.8.3). C'est pourquoi nous avons limité la pré-copie aux zones mémoire non modifiées par une task en exécution. Plus précisément il s'agit des zones mémoire où sont maintenus le code exécutable et le contexte d'exécution des threads suspendus (appartenant à la task en migration). La figure 4 décrit le déroulement de l'opération de transfert dans le temps. Deux phases sont distinguées : la première est la phase

de pré-copie du code et du contexte des threads gelés, la seconde est la copie du reste de l'image de la task.

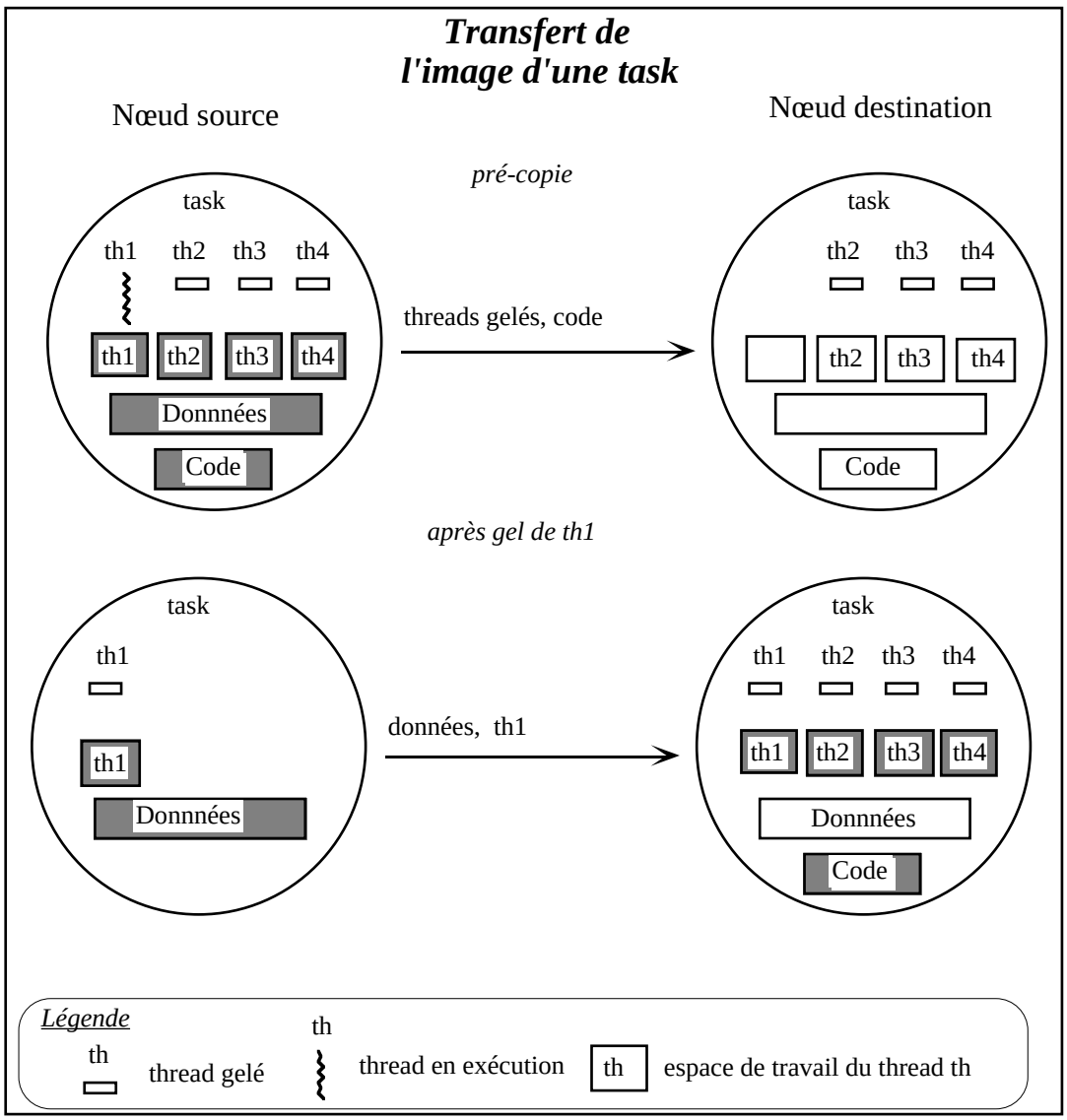


Figure 4 : Transfert d'une tâche

IV.9.2. Acheminement de l'image d'un processus

Dans le cas des architectures parallèles ayant des réseaux d'interconnexion maillés, il peut exister plus d'un chemin entre deux processeurs. Il serait alors utile de découper un message suivant le nombre de chemins alternatifs et d'envoyer les différents morceaux en parallèle à travers ces chemins. En effet, ceci permet :

(1) d'augmenter la bande passante pour l'envoi des messages entre les deux processeurs,

(2) de mieux répartir la charge de communication entre ces différents chemins.

Ces avantages sont d'autant plus significatifs lorsqu'il s'agit d'envoyer de gros volumes de données, ce qui est le cas pour la migration de processus.

Etant donné l'importance du coût de transfert, nous avons été amenés à modifier la couche routage de ParX pour la génération et l'utilisation de plusieurs chemins alternatifs entre deux processeurs. Un protocole de transfert mémoire, qui exploite ces chemins, a été aussi développé.

IV.9.2.1 Détermination des chemins alternatifs

ParX a été conçu de façon à ce qu'il puisse s'exécuter sur des machines parallèles où les processeurs sont interconnectés par un réseau de topologie quelconque. Pour cela, un algorithme de génération des chemins de routage a été développé [Gon91]. Le calcul des chemins se fait à partir de la description du réseau d'interconnexion.

A l'envoi d'un message celui-ci est découpé en paquets, les paquets sont alors acheminés suivant la technique *stocker-et-réexpédier* ("store-and-forward"). Sur chacun des processeurs intermédiaires d'un chemin de routage, cette technique consiste à recevoir un paquet et le stocker dans un tampon. Si le paquet n'est pas arrivé à destination, le routeur détermine le lien de sortie et renvoie le paquet à travers ce lien. La taille des tampons de routage sur chaque processeur est limitée (nous utilisons une taille égale à celle d'un paquet). De ce fait, il existe un risque d'interblocage du routage que l'on peut décrire de la manière suivante :

> Etant donné qu'un paquet, stocké dans un premier tampon de routage, ne peut être émis que si le tampon de routage sur le processeur récepteur est libre, une relation de dépendance existe entre ces deux tampons de routage. Un interblocage du routage se produit lorsque (1) les chemins de routage créent un cycle de relations de dépendance entre un ensemble de tampons et que (2) tous ces tampons soient occupés.

La résolution du problème d'interblocage se fait lors du calcul des chemins de routage en évitant de créer des cycles de dépendance entre les tampons de routage. Dans son principe de base, l'algorithme de calcul des chemins génère un arbre recouvrant tous les processeurs du réseau[1] et un sous-ensemble des liens de communication. Les messages sont alors acheminés entre les processeurs le long des branches de l'arbre. Cet algorithme garantit l'absence d'interblocage du fait qu'aucun cycle n'est créé. Une amélioration de cet algorithme a été introduite [Gon91] en rajoutant de nouvelles arêtes dans l'arbre de routage (ces arêtes doivent cor-

[1]Dans le cas de ParX il s'agit plutôt d'une partie de la machine (cluster) sur laquelle s'exécute une application.

respondre à des connexions existantes). Le rajout de ces arêtes se fait en respectant trois règles de routage qui évitent la création d'un cycle de dépendance. En disposant les processeurs de l'arbre par niveaux où le sommet est la racine de l'arbre, ces règles s'énonce de la manière suivante :

Règle 1 : éviter qu'un lien descendant ne soit suivi d'un lien ascendant,
Règle 2 : éviter qu'un lien horizontal ne soit suivi d'un lien ascendant,
Règle 3 : éviter la formation d'un cycle contenant seulement des liens horizontaux.

Ainsi, dans sa version déterministe l'algorithme de calcul des chemins de routage consiste à :

- Etape 1 - calculer l'arbre recouvrant,
- Etape 2 - déterminer le plus court chemin entre chaque paire de processeurs en tenant compte des restrictions imposées par les règles de routage.

Afin de pouvoir exploiter les chemins alternatifs entre chaque paire de processeurs, nous avons modifier l'étape 2 de façon à ce que l'algorithme puisse générer le maximum de chemins alternatifs respectant les trois règles de routage déjà énoncées. De plus, il est nécessaire que les chemins retenus soient disjoints, c'est-à-dire, qu'ils n'ont pas d'arête en commun. Ainsi après avoir déterminer tous les chemins alternatifs, nous choisissons les chemins non disjoints de façon à maximiser le nombre de chemins alternatifs. Notons que le nombre de chemins alternatifs est forcément inférieur au degrés d'interconnexion des processeurs (émetteur et récepteur).

Nous avons aussi développé un protocole de copie mémoire d'un processeur vers un autre. La copie d'une zone mémoire se fait en utilisant équitablement les chemins alternatifs existants entre les deux processeurs source et destination de la copie.

IV.9.2.2 Estimation des temps de transfert

Le débit de communication entre une paire de processeurs varie selon le nombre de chemins alternatifs générés par l'algorithme. Ce nombre dépend de la topologie du réseau et de l'arbre de recouvrement retenu à l'étape 1.

Dans le cas où la taille de la zone mémoire à copier est réduite (i.e. inférieure à la taille maximum d'un paquet), l'envoi d'un message à travers plusieurs chemins alternatifs ne réduit pas le temps de transfert du message et peut même le rallonger. En revanche, lorsque la taille de la zone mémoire est importante - cas qui nous intéresse - nous avons obtenu un gain considérable proportionnel au nombre de chemins alternatifs :

$V_a = V_1 * a$ où

V_a est la bande passante maximale lorsque il existe a chemins alternatifs
(V_1 = 680 Ko/s dans le cas des mesures effectuées)

La figure 4 donne le débit de communication maximal que l'on peut obtenir entre deux processeurs en fonction du nombre de chemins alternatifs. Les mesures ont été effectuées sur une machine SuperNode à base de transputers T800 et où le débit de communication entre deux processeurs est de 10 Mbits/s. Notons aussi que les valeurs reportées dans la figure 4 sont atteintes pour des messages de tailles importantes :

$t >> L * p$ où

t est la taille du message,
L la longueur du plus long chemin alternatif,
p la taille d'un paquet.

En fonction du débit de communication V_a, nous estimons la durée C du transfert de l'image d'une tâche de la manière suivante :

$C(t_r) = V_a * t_r$ où

t_r taille de l'espace mémoire résident sur le processeur.

La durée G de gel due à l'opération de transfert de la tâche est estimée par :

$G(t_l) = Va * t_{le}$ où

t_l correspond à la taille des zones mémoire accédées en lecture / écriture.

Nous avons négligé dans ces estimations la charge existante sur les chemins de transfert due à d'autres communications en cours.

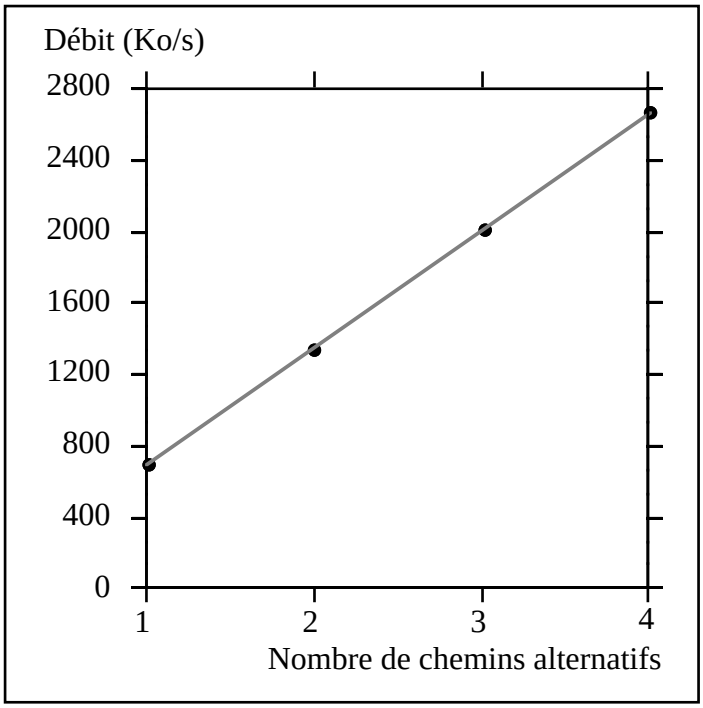


Figure 4 : Débit de communication entre deux processeurs en fonction du nombre de chemins alternatifs

IV.10 Conclusion

Les techniques de transfert de l'image d'un processus se distinguent, d'une part, selon la répartition de l'opération de transfert dans le temps, et d'autre part, selon les ressources mis en œuvre pour le transfert.

L'efficacité des différentes stratégies de répartition dans le temps de l'opération de transfert, dépend du comportement d'accès mémoire des processus composant l'application. Ces stratégies peuvent même pénalisantes pour certaines applications. Ainsi, la stratégie que nous avons retenue est celle de transférer entièrement l'image du processus résidente sur le processeur source lors de la migration. Seules les zones mémoires non accédées en écriture sont précopiées avant le gel du processus.

Par ailleurs, étant donné que la taille de l'image d'un processus est généralement importante, nous avons été conduits à élaborer un nouveau protocole de transfert mémoire qui tire parti de l'existence de plusieurs chemins alternatifs entre deux processeurs. Ceci nous a permis de diminuer considérablement le coût de migration.

Chapitre V

Acheminement des Messages

Lorsque les communications entre processus se font à travers une mémoire partagée, elles se traduisent par des accès mémoire contrôlés. La migration de processus ne va pas donc perturber le déroulement des communications : l'accès mémoire se faisant de la même manière à partir de n'importe quel processeur.

Dans le cas où la communication est effectuée par échange de messages sans utiliser une mémoire commune, il faut assurer un acheminement correct des messages entre des processus pouvant migrer. Les messages doivent être délivrés aux processus destinataires même si ces derniers se déplacent d'un processeur à un autre. Une particularité de ce problème est que les messages ainsi que les processus sont mobiles. En outre, le déclenchement d'une opération de migration d'un processus se fait a priori indépendamment des envois de messages destinés à ce processus.

L'objet de ce chapitre est la prise en compte de la migration de processus au niveau du noyau de communication afin d'assurer un échange correct des messages entre les processus. Dans un premier temps, nous examinons les différents concepts et paradigmes associés à la communication entre processus. Nous passons ensuite en revue les principales approches adoptées dans la littérature. Suivant la sémantique des communications, nous proposons trois protocoles assurant un acheminement correct des messages et tenant compte de la migration de processus. Ces protocoles répondent à certains critères, notamment le critère d'extensibilité caractérisant les systèmes massivement parallèles.

V.1. Préliminaires

V.1.1. Définitions

Nous distinguons dans un noyau de communication deux niveaux de contrôle pour l'acheminement des messages :

i) Le premier correspond au routage qui permet d'acheminer les messages d'un point du réseau à un autre. Il regroupe les trois premières couches du modèle ISO (i.e. physique, liaison et réseau).

ii) Le second niveau est relatif à la mise en œuvre d'objet de communication, que nous appelons aussi connecteurs, régis par des protocoles adéquats. Ces protocoles correspondent à ceux du reste des couches du modèle ISO (i.e. transport, session, présentation et application) et ont pour fonctions le multiplexage des communications, le contrôle de bout en bout, la désignation des points d'accès, la synchronisation, etc. On définit aussi à ce niveau, la sémantique des opérations d'émission et de réception des messages.

Il est à noter que ce parallèle avec le modèle ISO ne signifie pas qu'un noyau de communication est organisé en couche ni qu'il doit forcément assurer tous les types de protocoles prévus dans ce modèle.

Concernant le premier niveau, la migration de processus n'affecte pas la fonction de routage. En effet, celle-ci assure l'acheminement des messages entre les processeurs du réseau et non pas entre les processus qui s'exécutent sur ces processeurs.

Au second niveau, les communications entre processus se font grâce à des *points d'accès*. Avant d'être introduit dans le routage, un message est émis ou reçu à travers un point d'accès. A ce niveau, la migration de processus perturbe les échanges de messages. Les points d'accès, par lesquels le processus migrant communique, doivent être transférés sur le processeur destination de la migration. Il faut aussi assurer un acheminement correct des messages tout en respectant les protocoles du second niveau. Ceci nous amène à rajouter des protocoles de migration assurant un échange correct des messages. Ces protocoles vont dépendre des *schémas de communication* qui se caractérisent par :

- la nature des opérations que peuvent effectuer les intervenants dans une communication (par exemple, l'émission, la réception, l'invocation d'une procédure, etc.),

- le nombre des intervenants et le sens des communications (i.e. un émetteur vers un récepteur, plusieurs émetteurs vers un récepteur, etc.),
- la nécessité ou non d'établir une connexion avant d'échanger des messages,
- le type de synchronisation (synchrone ou asynchrone) appliqué à chacun des intervenants,
- la conservation de l'ordre d'envoi des messages à la réception, etc.

V.1.2. Modélisation de la communication

Le noyau de communication est décomposé en deux éléments : le *routeur* et le *gestionnaire de communication*. Ce dernier met en œuvre les protocoles de communication. En particulier, il permet d'intégrer les protocoles de migration (figure 1).

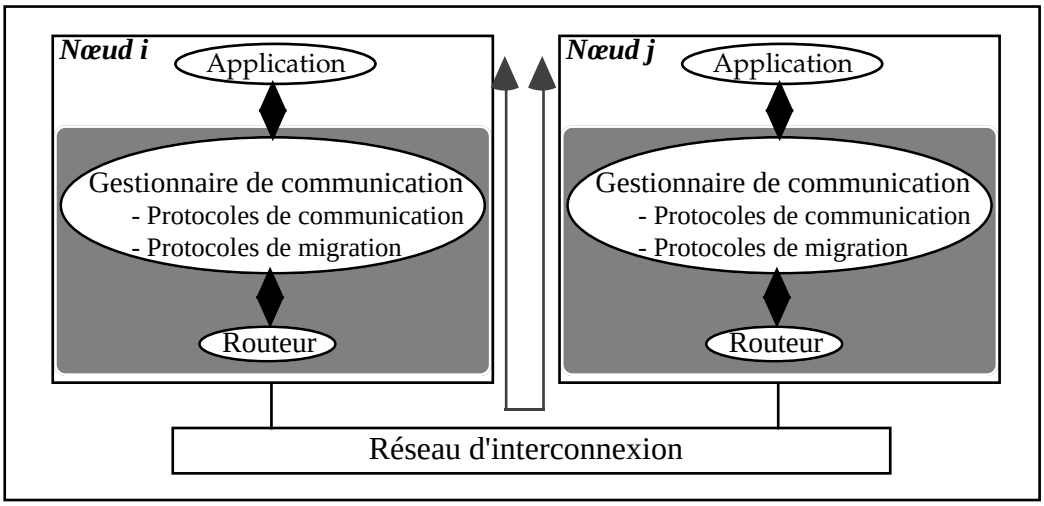


Figure 1 : Modèle de communication.

Comme nous l'avons déjà précisé, à un envoi ou une réception d'un message est associé un point d'accès maintenu par le gestionnaire de communication. La sémantique de la communication est spécifiée à travers ce point d'accès. Ce dernier dispose aussi de l'espace mémoire nécessaire pour la gestion des communications.

A l'émission, le gestionnaire de communication découpe le message en *paquets* avant de le passer au routeur. Les paquets délivrés par le routeur au gestionnaire de communication sont assemblés pour reformer le message émis.

Les processus qui communiquent avec un processus *P* sont dits : *processus adjacents* à *P*. Les gestionnaires de communication (respectivement les processeurs) correspondants sont appelés les *gestionnaires de communication adjacents* (respectivement les *processeurs adjacents*) à *P*. Une communication est définie par un ensemble de points d'accès associés à chacun des pro-

cessus adjacents qui interviennent dans cette communication. L'*état complet* relatif à une communication est donc distribué sur plusieurs processeurs.

V.1.3. Hypothèses et contraintes de travail

Nous supposons dans la suite que le routage assure un acheminement correct des messages entre les processeurs, en un délai de temps fini, sans interblocage ni famine. Aucun message n'est perdu ou dupliqué. En outre, les communications entre des processus applications ne se font qu'à travers les gestionnaires de communication. Nous supposons aussi qu'un processus ou un point d'accès est désigné de manière unique à travers un identificateur. Le mécanisme de désignation fait l'objet du chapitre suivant.

Par ailleurs, les protocoles de migration doivent assurer un acheminement correct des messages de manière transparente aux applications. De plus, nous devons tenir compte des différents schémas de communication (par exemple : synchrone ou asynchrone, un vers un, plusieurs vers un, etc.) auxquels il faut proposer des protocoles de migration adéquats.

Etant donné que le mécanisme de migration est destiné à être utilisé dans un système massivement parallèle, un certain nombre de contraintes de conception s'imposent. Ces contraintes se rattachent à l'extensibilité des protocoles proposés. La centralisation des traitements et des informations doit donc être évitée. De plus, la taille mémoire nécessaire (par processeur) pour la mise en œuvre de ces protocoles doit être indépendante du nombre de processeurs dans le réseau.

V.2. Les diverses approches

Il existe différents protocoles utilisés dans les systèmes distribués pour assurer l'échange correct des messages en supportant la migration de processus. Nous les classons suivant trois approches :

i) prévention de la perte des messages,
ii) retransmission des messages,
iii) redirection des messages.

Le reste de cette section est consacré à la présentation ainsi qu'à une analyse critique de ces trois approches.

V.2.1. Approche par prévention de la perte des messages

Dans cette approche, la migration effective d'un processus *P* ne peut avoir lieu qu'après que tous les gestionnaires de communication adjacents à *P* aient été informés. Ces derniers suspendent alors l'envoi des messages jusqu'à ce que le processus migrant reprenne son exécution. L'acheminement des messages se fera ensuite directement vers le processeur destination de la migration.

A moins de diffuser l'intention de migration d'un processus, cette approche n'est applicable que si les processeurs adjacents sont connus. Même ceux qui ne vont plus communiquer avec le processus migrant sont prévenus, ce qui représente un surcoût inutile. Par ailleurs, du fait que l'émission des messages vers le processus migrant est retardée, il faut soit bloquer les émetteurs, soit mettre ces messages dans des tampons locaux durant la migration. Cette dernière option a été adoptée dans [LCL87], l'inconvénient est que l'espace mémoire nécessaire n'est pas limité a priori. Dans [DGi92], l'émission des messages est bloquée s'il n'existe plus de tampons disponibles.

Le principal avantage de l'approche par prévention de la perte de message est l'indépendance du processus migrant vis-à-vis du processeur source. Même si ce processeur tombe en panne, après que la migration ait été achevée, les communications peuvent se poursuivre correctement. De plus, elle permet de libérer complètement le processeur source ce qui est important lorsque la migration est opérée pour des considérations de redistribution de la charge.

V.2.2. Approche par retransmission des messages

Cette approche ne prend aucune disposition particulière lors de la migration d'un processus. Celui-ci est migré sans tenir compte des messages qui lui sont destinés ; ils sont ainsi ignorés à leurs arrivées au processeur source de la migration. Les gestionnaires de communication sont chargés de reprendre l'émission de ces messages vers la destination de la migration. Des protocoles de retransmission de messages sont donc nécessaires, il faut aussi pouvoir retrouver la nouvelle adresse d'un processus ayant migré.

Le principal intérêt de cette approche est de pouvoir entamer la migration d'un processus rapidement sans avoir à prévenir les gestionnaires de communication adjacents. Ceci est particulièrement important pour que la décision de migration, qui dépend de l'état (dynamique) du système, reste judicieuse.

Les systèmes Amœba [MRT90] et V [TLC85] implantent cette approche en utilisant un mécanisme rejetant un message si le processus destinataire n'est plus présent. Les ressources mé-

moires nécessaires, pour l'envoi des rejets, ne sont cependant pas limitées. Un protocole de diffusion d'une demande de localisation d'un processus est utilisé, il permet de retrouver la nouvelle adresse d'un processus ayant migré.

V.2.3. Approche par redirection des messages

Il s'agit dans cette approche de faire suivre les messages destinés à un processus ayant migré. Ceci se fait grâce à des *redirections* (encore appelés *liens de poursuite*) maintenues sur les processeurs sources. L'ensemble des redirections sur un processeur peut être assimilé à une table de routage. Nous avons ainsi deux niveaux de routage :

- le routage de base utilisé pour atteindre un processeur donné,
- le routage pour atteindre un processus donné. C'est à ce niveau que s'insère les redirections.

Dans cette approche, il est nécessaire que les processeurs maintenant des redirections restent fiables. D'autre part, en migrant un processus, la charge imposée par les messages en arrivée à ce processus n'est pas supprimée sur le processeur source. Tout au contraire, elle est plus importante puisque les messages sont retransmis par ce processeur. De plus, le chemin suivi par un message redirigé est allongé. Il est d'autant plus long que le processus destinataire migre plusieurs fois. Pour y remédier, il faut mettre à jour les redirections en utilisant l'adresse la plus récente du processus (vers lequel les redirections sont effectuées). Cette mise à jour ne préserve pas l'ordre d'envoi des messages à la réception. Notons aussi que l'approche par redirection nécessite des tampons intermédiaires où les messages n'ayant pas encore atteint leurs destinations sont stockés.

Dans son principe de base, la redirection est simple à réaliser. Elle permet aussi de ne pas retarder l'émission d'un message. Ainsi, si le type de communication est asynchrone, l'émetteur pourra poursuivre son exécution dès que le message est émis.

Les systèmes Demos/MP [PoM83], Locus [WPR83], Mach [Bar91], Mosix [BGW93] et Sprite [DoO91] utilisent l'approche par redirection des messages. Différentes techniques de maintien des chemins de redirection sont utilisées dans ces systèmes. Dans le système Demos/MP, la mise à jour d'une redirection se fait lorsque le processeur auquel a été envoyé le message redirige lui même le message. Pour cela il transmet à l'émetteur initial du message l'adresse qu'il utilise. Locus, Mosix et Sprite gèrent la redirection à travers le processeur origine où a été créé le processus destinataire du message. Un message envoyé à ce processus est émis vers son processeur origine, celui-ci se charge de retransmettre le message vers le processeur adéquat. A chaque fois que le processus migre, l'adresse de redirection sur son processeur origine est mise à jour en conséquence. Dans Mach, la mise à jour des adresses de redi-

rection se fait grâce à un jeton qui est échangé périodiquement entre les différents processeurs du réseau. Lorsqu'un processeur reçoit le jeton et s'il maintient une adresse différente de celle du jeton, il met à jour soit celle du jeton soit sa propre adresse par la plus récente. Notons que dans les systèmes massivement parallèles, étant donné le nombre important de processeurs par lesquels transite le jeton, cette technique n'est pas applicable puisque les délais de mise à jour deviennent alors considérables.

Charlotte [ArF89] utilise aussi l'approche par redirection durant l'opération de migration d'un processus. De plus, du fait qu'une communication se fait à travers un lien représentant une connexion entre deux processus déterminés, la mise à jour de l'adresse du processus en migration (chez ses processus adjacents) est faite lors de la migration.

La redirection des messages a aussi été utilisée dans Emerald [JLH88] pour la localisation des objets. Le protocole adopté a été proposé par Fowler [Fow86] où il s'intéresse en particulier aux objets dont la mobilité est importante. Le problème étant que les chemins de redirection vers de tels objets sont trop longs. Pour cela il propose une technique de compression des chemins de redirection. Cette compression ne s'effectue que sur les chemins par lesquels un message est redirigé.

Ravi et Jefferson [RaJ88] ont aussi proposé un protocole de redirection où la compression des chemins ne se limite pas à ceux parcourus par les messages redirigés. Ils associent à chaque redirection vers un processus un ensemble de processeurs ayant une redirection vers le processeur où se trouve le processus. La mise à jour des redirections vers un processus s'étend alors à cet ensemble. L'avantage est que la mise à jour n'est plus passif mais elle s'effectue même si aucun message n'est redirigé.

Etant donné la granularité importante des processus, la fréquence de migration d'un processus doit être faible. L'optimisation des techniques de compression des chemins de redirection se justifie d'une part par le besoin de diminuer les temps de communication et l'encombrement du réseau, d'autre part, elle permet de réduire le nombre de redirections à effectuer, et par là même, le recours à des tampons de redirection. Dans le cas des systèmes massivement parallèles, il est nécessaire que la taille des tampons de redirection (par processeur) soit limitée, ce qui n'est pas assurée par les techniques que nous venons de présenter.

V.3. Protocoles pour l'acheminement correct des messages

Comme nous l'avons déjà souligné dans la section V.1.1, divers schémas de communication sont utilisés dans les systèmes distribués. En tenant compte des contraintes d'extensibilité, nous avons élaboré trois protocoles pour l'acheminement correct des messages en supportant la migration de processus[1]. Les conditions d'utilisation de ces protocoles, liées aux schémas de communication, sont spécifiées à la fin de cette section.

V.3.1. protocole 1 : gel des communications

V.3.1.1. Description

L'approche suivie par ce protocole est la prévention de perte des messages. La figure 2 illustre les trois phases qui composent ce protocole : la phase de suspension, la phase de transfert et la phase de reprise des communications. Avant que la migration d'un processus *P* n'ai lieu, une demande de suspension des communications (*migration_request*) est envoyée du processeur source vers tous les processeurs adjacents au processus *P*. Ce message indique aux gestionnaires adjacents à *P*, l'intention de migration. Ces derniers suspendent alors l'envoi de nouveaux messages vers le processus *P* et renvoient une confirmation (*migration_confirm*) au gestionnaire de communication du processeur source. Une fois ce message reçu de tous les processeurs adjacents, le processeur source transfère l'état de communication sur le processeur destination de la migration (*state_transfer*) et libère les ressources mémoire occupées par cet état. Le processeur destination crée un nouveau point d'accès selon cet état et envoie aux gestionnaires de communication adjacents à *P* un message autorisant la reprise des communications (*resumption_message*).

Lorsque l'ordre d'émission des paquets n'est pas forcément le même qu'à la réception, un paquet envoyé avant un message de confirmation de migration risque d'arriver après ce message comme le montre la figure 3. Si le paquet fait partie d'un message dont la transmission a été déjà entamée, le paquet ne pourra pas être traité puisque le point d'accès destination a déjà été transféré. Afin de résoudre ce problème, lorsqu'un message est en cours de réception, le message *state_transfer* n'est envoyé qu'après que tous les paquets de ce message n'ai été reçus.

[1] Ces protocoles sont décrits par ailleurs dans l'article [ElM94].

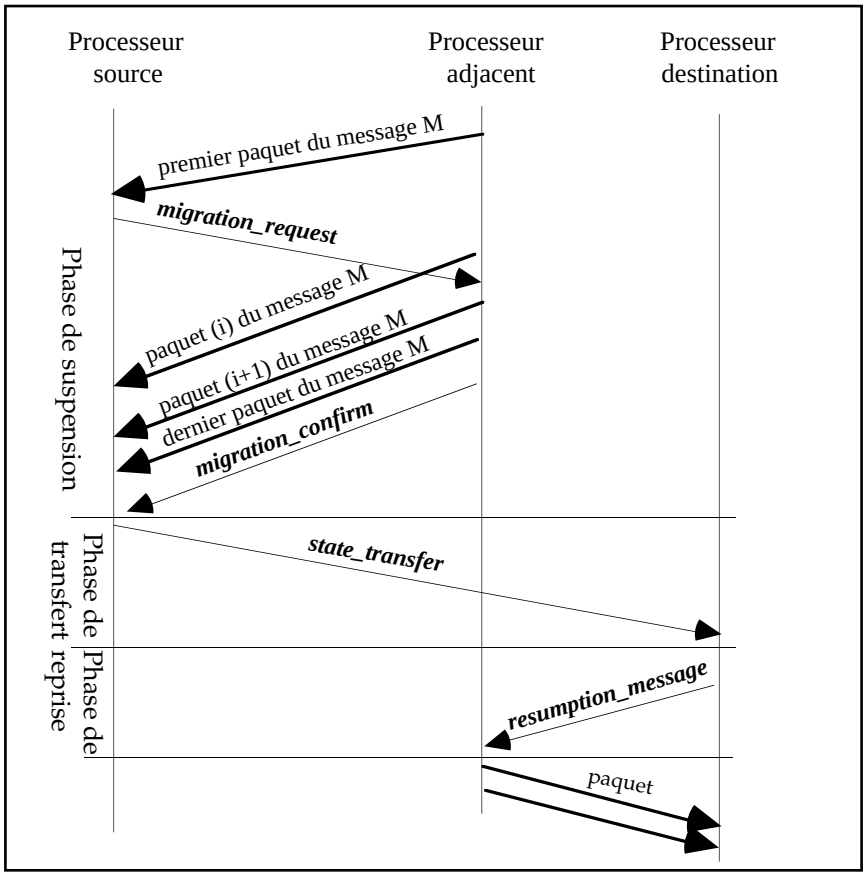


Figure 2 : Protocole de gel des communications

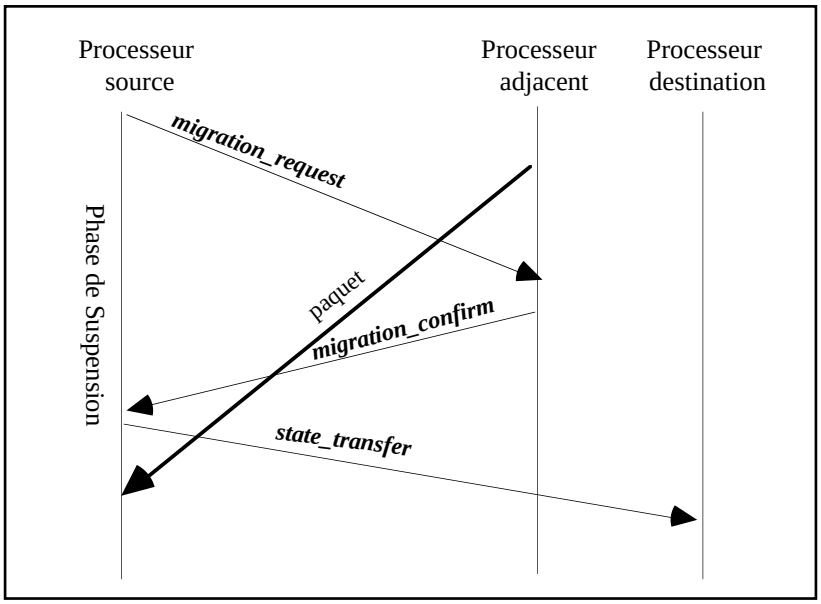


Figure 3 : Arrivée d'un paquet après la confirmation de migration

V.3.1.2. Migrations simultanées de processus adjacents

Le protocole de gel tel qu'il a été défini dans la section précédente ne tient pas compte de la possibilité de faire migrer deux processus simultanément. Le gestionnaire de communication de l'un, doit pouvoir confirmer la migration de l'autre. Cette situation introduit un interblocage potentiel décrit par la figure 4. L'interblocage est dû au fait que le processeur 2 a émis une requête de migration du processus T_2, suivie d'une confirmation de la migration du processus adjacent T_1, mais que ces messages de contrôle sont reçus dans l'ordre inverse. Le processus T_1 peut donc migrer avant que la requête de migration, émise par le processeur 2, ne soit reçue sur le processeur 1. A l'arrivée de celle-ci, le processus est ailleurs et donc la requête ne pourra pas être satisfaite.

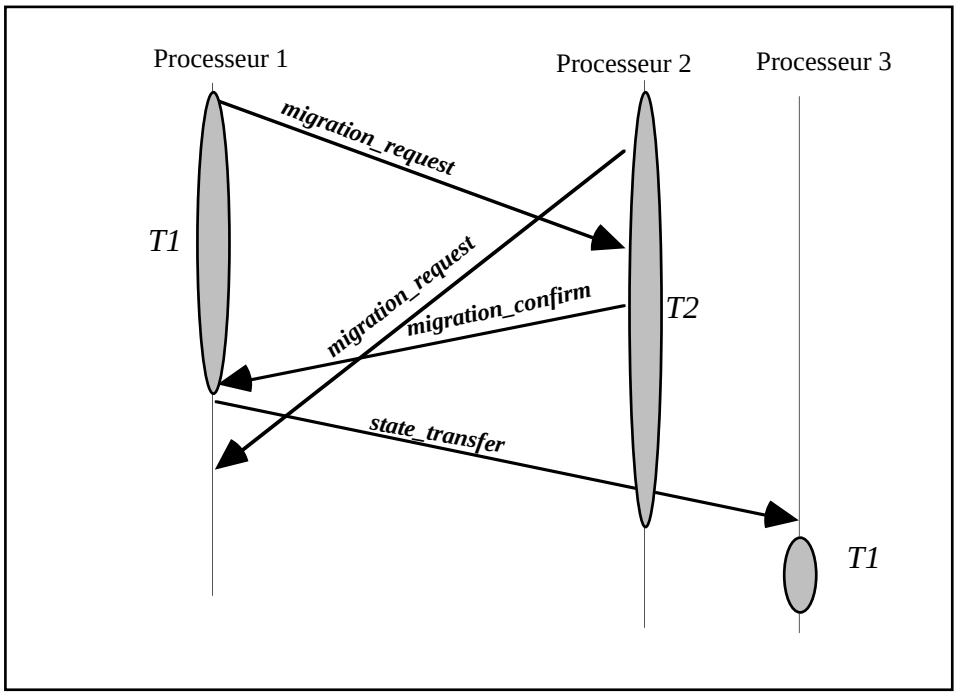


Figure 4 : interblocage dû à des migrations simultanées

Dans [LCL87], ce problème a été résolu en utilisant un délai de garde déclenché à la réception de la confirmation et au bout duquel la migration peut être entamée. Ce délai est calculé de façon à pouvoir s'assurer qu'aucun message émis avant la confirmation ne peut encore être en transit. Inévitablement la migration sera retardée, souvent inutilement car le problème ne risque de se poser que lorsque deux processus adjacents migrent simultanément.

Nous proposons de résoudre ce problème en ignorant la requête de migration. Ainsi, dans le cas de la figure 4, la requête de migration du processus T_2 est ignorée par le processeur 1 car le processus n'est plus présent. Lorsque le processeur 2 reçoit le message de reprise de la

communication (avec T_1), il déduit que la requête a été éventuellement ignorée. Il retransmet une seconde fois la requête de migration de T_2 vers le processeur 3. Remarquons que l'adresse du processeur 3 est connue sur le processeur 2 car elle est envoyée dans la requête de migration de T_1.

Enfin notons que lorsque deux processus adjacents migrent simultanément, les messages de reprise sont envoyées vers les processeurs destinations de chacun de ces processus. La communication est rétablie dès que les deux messages de reprises sont reçus sur les processeurs destinations.

V.3.2. protocole 2 : rejet des messages

V.3.2.1. Description

Le protocole de rejet suit l'approche de retransmission des messages. A la réception d'un message destiné à un processus ayant migré, le gestionnaire de communication ignore le message et renvoie un message de rejet (*reject)* en réponse au premier message. L'émetteur du message rejeté doit le retransmettre au processeur destination de la migration. Trois problèmes sont à résoudre :

- la conservation des messages non encore délivrés pour une éventuelle retransmission,
- l'envoi du message de rejet,
- la recherche de la nouvelle adresse d'un processus ayant migré.

Maintien des messages
Un message envoyé doit être mémorisé jusqu'à ce qu'il soit reçu, ceci afin de pouvoir le retransmettre. En conséquence, si les communications ne sont pas synchrones l'espace mémoire nécessaire n'est pas limité.

Envoi d'un rejet
Afin d'envoyer un rejet, il est nécessaire de disposer d'un certain espace mémoire pour la gestion du rejet. Le nombre de messages à rejeter dépend du nombre d'émetteurs possibles. Celui-ci peut ne pas être limité (cas d'un port plusieurs vers un) auquel cas la taille de l'espace mémoire nécessaire pour les structures d'envoi des rejets n'est pas limitée. En effet, le délai d'envoi d'un rejet n'est pas limité et par conséquent un nombre indéterminé de messages à rejeter peuvent être reçus sans pouvoir en rejeter un. Pour remédier à cet inconvénient, nous proposons de ne pas rejeter un message s'il y a un défaut de l'espace mémoire. L'émetteur du message, ne recevant pas un acquittement du message et après un délai de garde, doit alors

retransmettre le message. Ce protocole n'est ainsi applicable que pour des communications synchrones.

Recherche de l'adresse d'un processus

Etant donné l'identificateur d'un processus, l'adresse de celui-ci est déduite grâce à un mécanisme de désignation et de localisation qui fait l'objet du chapitre VI.

Remarquons enfin que durant la migration d'un processus, les messages qui lui sont destinés risquent d'être ignorés par les processeurs source et destination de la migration comme le montre la figure 5. Néanmoins, le protocole reste correct.

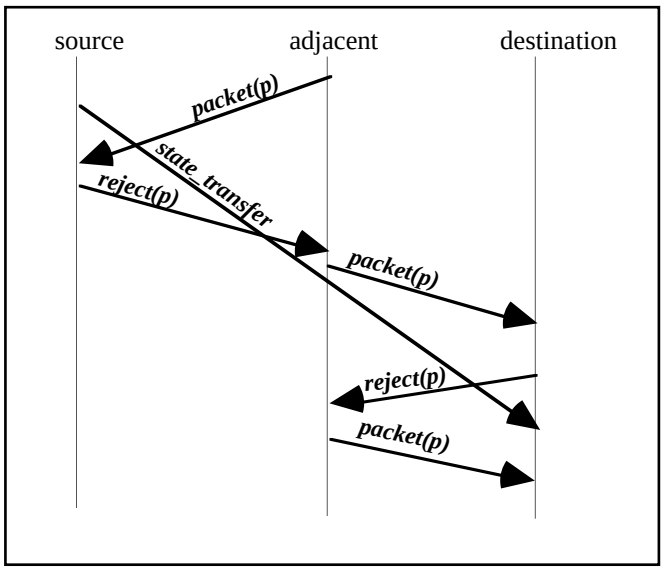


Figure 5 : rejet multiple d'un message

Par ailleurs, il est possible de faire migrer simultanément plusieurs processus adjacents. Un rejet est ignoré s'il arrive après que le processus ayant émis le message rejeté ait migré. Le mécanisme de délai de garde assure la reprise de l'émission du message.

V.3.3. Protocole 3 : redirection des messages

V.3.3.1. Description

Une redirection vers un processus *P* comporte les deux champs suivants :

- *identification(P)* : identificateur du processus *P* vers lequel se fait la redirection,
- *address(P)* : adresse présumée du processus *P*. Notons que cette adresse peut être périmée.

Dans le cas où il n'est pas nécessaire de conserver l'ordre d'envoi des messages à la réception, nous utilisons un protocole de mise à jour des redirections décrit ci-dessous.

Soit deux processus adjacents P_e et P_r. Supposons que le processus Pr migre un certain nombre de fois. A chaque migration du processus P_r une redirection vers ce processus est laissée sur le processeur source de la migration. Un chemin de redirection est ainsi constitué. Supposons que le processus P_e envoie alors un message *M* au processus P_r. Le principe de la compression de ce chemin est le suivant. Sur tous les processeurs ayant redirigé le message *M* les redirections sont mises à jour de la manière suivante. Un message de correction (*update_address*) est envoyé du processeur où réside le processus P_r vers le processeur de l'émetteur du message. La correction est ensuite propagée tout au long du chemin de redirection. Fowler s'est limité dans son protocole de compression des chemins de redirection à la description que nous venons de faire. Cette technique de compression proposée par Fowler est décrite et évaluée dans [Fow86]. La correction des redirections est passive puisqu'elle est déclenchée seulement par l'envoi d'un message. Dans le protocole que nous proposons, nous maintenons dans le contexte d'un processus l'adresse du processeur ayant créé ce processus (processeur origine). A chaque migration du processus, un message de correction est envoyé à ce processeur. La mise à jour des redirections est alors propagée sur tous les processeurs sur lesquels s'est exécuté le processus. De cette façon les mises à jour des redirections sont anticipées.

Du fait qu'un processus *P* peut migrer plusieurs fois, différents messages de correction peuvent arriver sur un processeur maintenant une redirection vers *P*. En particulier un message de correction peut arriver après un autre qui est plus récent comme le montre la figure 6. Dans ce cas, il ne faut pas modifier la valeur de *address(P)*. A cet effet, nous rajoutons dans une redirection un troisième champs (*time_stamp(P)*) à la structure d'une redirection. Ce champ est une estampille qui indique le nombre de migrations du processus *P*, avant qu'il n'arrive sur le processeur désigné par la redirection. Un message de correction doit ainsi contenir la valeur de l'estampille associée à l'adresse de correction. Ainsi il est possible de déterminer si cette adresse est plus récente que celle qui existe déjà dans la redirection.

De la même manière que pour la mise à jour des redirections, lors de la terminaison d'un processus, toutes les redirections vers ce processus sont éliminées.

Par ailleurs la migration simultanée de deux processus adjacents ne pose pas de problème d'interblocage. En effet, une fois qu'un message est envoyé, il n'est plus dépendant du processus émetteur. Lorsque le message arrive à un processeur, il est soit délivré au processus desti-

nataire, soit redirigé si le processus destinataire n'est plus présent sur le processeur. Le processus émetteur du message peut ainsi migrer entre temps.

La principale particularité de notre protocole de redirection est qu'il utilise une taille limitée des tampons de redirection par processeur. La section suivante est consacrée à cet aspect.

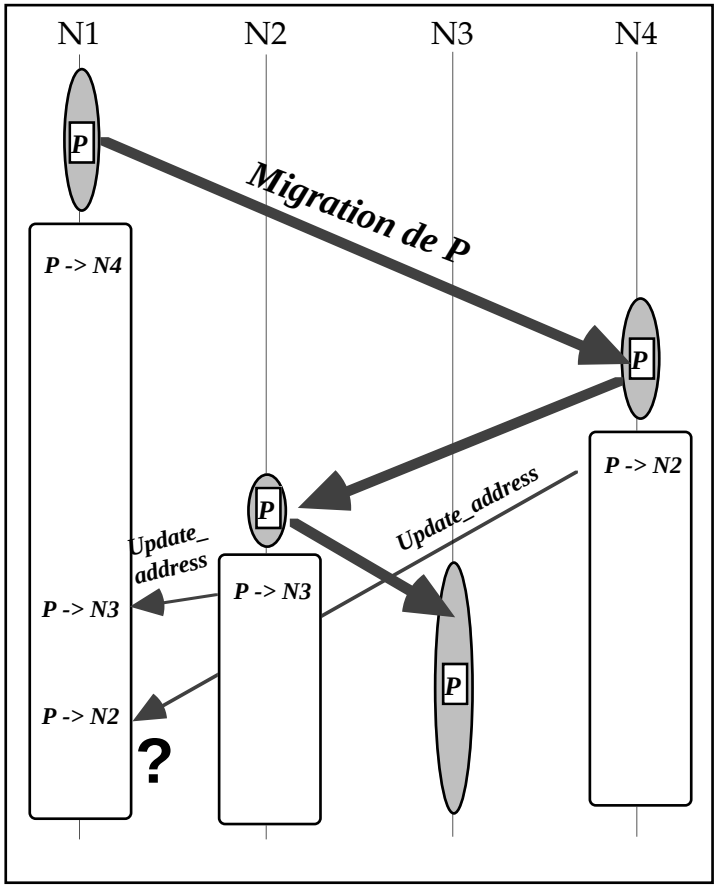


Figure 6 : Problème de mise à jour d'une adresse périmée

V.3.3.2. Gestion des tampons de redirection

Dans un premier temps nous associons à chaque redirection un tampon pour stocker les paquets à rediriger. La taille du tampon est égale à la taille maximum d'un paquet. Sur un processeur donné, le nombre de tampons nécessaires est donc égal au nombre de processus ayant migré à partir de ce processeur et sont encore en exécution. Pour limiter ce nombre, une nouvelle migration est interdite si l'on atteint un seuil fixé.

Cependant, la limitation du nombre de tampons introduit des problèmes d'interblocage. Lorsqu'un tampon de redirection est occupé, le paquet suivant qui est à rediriger à travers ce tampon, est retardé jusqu'à ce que le tampon soit libéré. De ce fait, on introduit une dépen-

dance entre les tampons de redirection vers un même processus. Un interblocage peut se produire comme le montre la figure 7, l'interblocage est dû aux trois faits suivants :

(1) un cycle de dépendance s'est constitué entre les tampons de redirection vers le processus destinataire,
(2) le processus destinataire n'est plus dans le cycle et
(3) tous les tampons sont occupés par des messages redirigés sur des processeurs du cycle[1].

Pour résoudre ce problème, un nouveau tampon est rajouté sur le processeur source de la migration courante. La redirection correspondante est mise à jour par la nouvelle adresse du processus. Ainsi, si le processus migre vers un processeur hors du cycle, le cycle est rompu. Ce qui élimine le problème d'interblocage. S'il n'existe pas un tampon disponible, la migration hors du cycle est refusée.

Afin de pouvoir associer plus d'un tampon à une redirection, nous maintenons un ensemble de tampons partagés entre les différentes redirections sur un processeur. Ceci permet de réduire les attentes sur les tampons occupés.

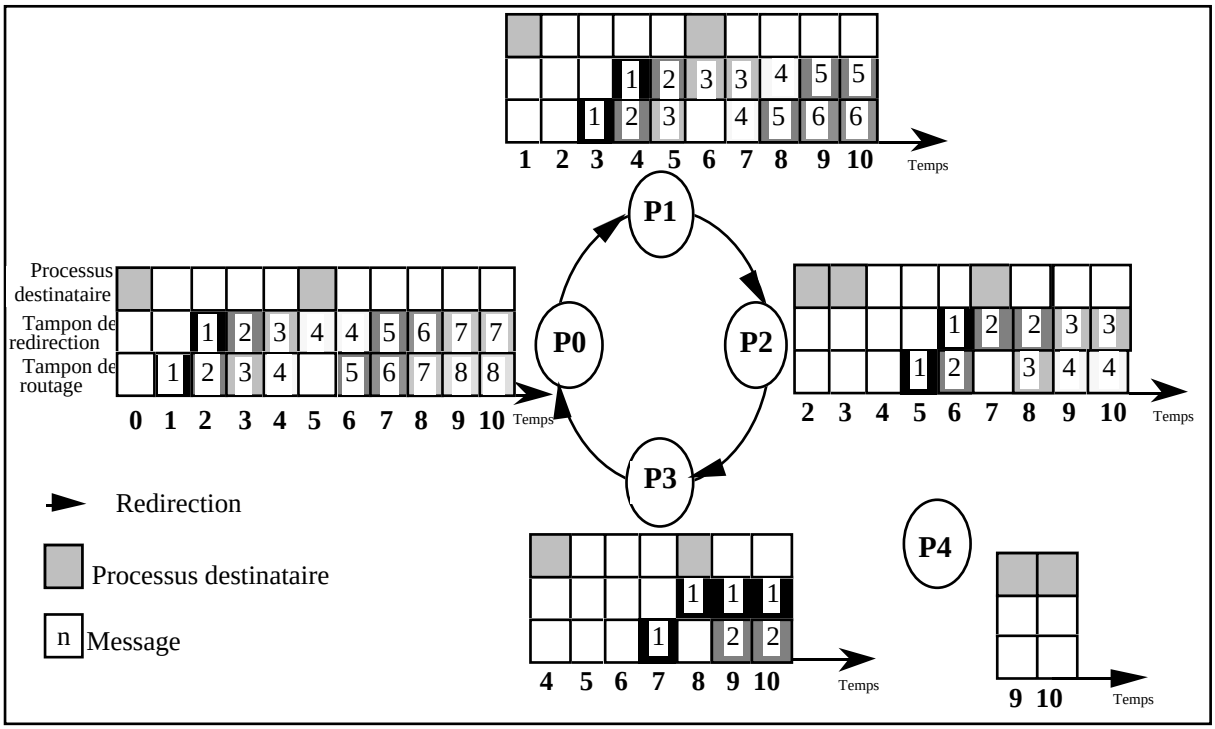


Figure 7 : Exemple d'interblocage

[1] Après avoir décidé de rediriger un message, il n'est plus possible d'annuler cette décision car le contrôle du message est donné au routeur.

V.3.4 Condition d'utilisation des protocoles 1, 2 et 3

Le protocole de gel est applicable lorsque les processus adjacents au processus migrant sont connus. De plus les communications doivent être synchrones. En effet, si l'émetteur d'un message n'est pas bloqué jusqu'à ce que le message ait été envoyé, il faut stocker ce message durant toute la phase de suspension des communications. La taille de l'espace mémoire de stockage nécessaire n'est pas limitée puisque l'émetteur peut envoyer plusieurs messages pendant la phase de suspension.

Quant au protocole de rejet, il peut s'utiliser pour des communications synchrones, les processus adjacents au processus migrant ne doivent pas forcément être connus. Notons aussi que ce protocole nécessite un mécanisme de désignation et de localisation des processus.

Enfin en ce qui concerne le protocole de redirection, celui-ci garde des dépendances résiduelles sur le processeur source d'une migration. Ce protocole n'est donc pas applicable lorsque la migration est opérée pour libérer complètement le processeur source d'une migration. A la différence des deux autres protocoles de gel et de rejet, le protocole de redirection s'applique aussi bien pour les communications synchrones qu'asynchrones. De plus il n'est pas nécessaire de connaître les processus adjacents.

V.4. Mise en œuvre dans ParX

La mise en œuvre des protocoles de communication, dans ParX, se fait suivant un modèle générique de construction de protocoles au dessus de la couche routage. Nous commençons dans cette section par présenter ce modèle. Nous décrivons ensuite des exemples de protocoles de communication auxquels nous appliquons les protocoles de migration proposés. Une évaluation de ces protocoles est effectuée à la fin de la section.

V.4.1. Modèle générique de construction de protocoles

L'intérêt du modèle générique de construction de protocoles est de faciliter l'implantation correcte et le multiplexage de divers protocoles de synchronisation et de communication [Mun89]. La réalisation d'un protocole se fait en spécifiant un certain nombre d'événements et d'actions régissant le protocole. Trois types d'événements sont définis :

- arrivée d'une requête provenant d'un processus application,
- arrivée d'un paquet délivré par la couche de routage,
- émission d'un paquet.

A chaque événement il est possible d'associer un traitement particulier que le programmeur d'un protocole doit spécifier. Différents protocoles sont ainsi construits. La figure 8 décrit le modèle de construction des protocoles et son interaction avec le reste du système. Un protocole est construit au dessus de la couche de routage, il est mis en œuvre grâce à trois serveurs, chacun traitant un type d'événement. Le serveur *protocol_server* traite les requêtes en provenance de la couche supérieure. Le serveur *protocol_sender* traite les requêtes déposées par les serveurs *protocol_server* et *protocol_receiver*. Il a la possibilité de déposer des requêtes d'envoi de paquets à travers la couche de routage. Ces requêtes sont servies par le processus *link_sender*. Le serveur *protocol_receiver* traite les paquets en provenance de la couche de routage. Ceci se fait par l'intermédiaire de requêtes déposées par les processus *link_receiver* assurant la réception des paquets.

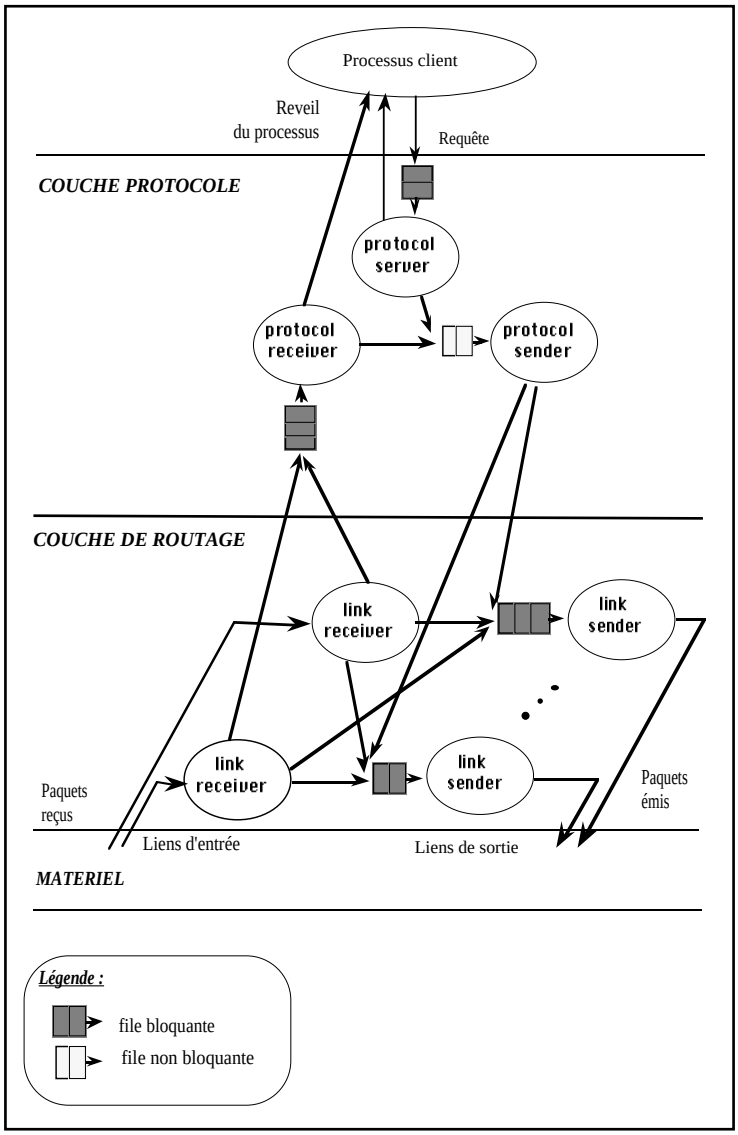


Figure 8: modèle générique de construction de protocoles

Les traitements effectués par les serveurs *protocol_server* et *protocol_receiver* ne doivent pas bloquer ces derniers en attente d'événements autres que celles auxquels ils sont dédiés. Ceci garantit que les requêtes déposées par les processus clients et la couche de routage soient traitées en un délai de temps fini. C'est la raison pour laquelle la mise en file d'une requête destinée au *protocol_sender* n'est pas bloquante.

La couche de routage utilise une taille limitée pour les tampons de routage. Ainsi, dès qu'un paquet est reçu par le *link_receiver*, le *protocol_receiver* doit être capable de le consommer. Pour cela, il doit disposer de l'espace mémoire nécessaire au traitement d'un paquet délivré par la couche de routage.

Le comportement des trois serveurs est spécifié en définissant les fonctions à exécuter à la réception d'une requête. A chaque requête est associé un numéro de fonction à exécuter lors du traitement de la requête. Les protocoles de migration sont implantés en rajoutant de nouvelles fonctions aux protocoles de communication.

V.4.2. Protocoles de communication

ParX offre deux protocoles de communication de base qui correspondent aux canaux et aux ports. D'autres protocoles peuvent être rajoutés en se basant sur le modèle générique que nous venons de décrire.

Un canal est un objet de communication synchrone point à point. Il peut être unidirectionnel ou bidirectionnel. Les primitives d'émission et de réception sont :

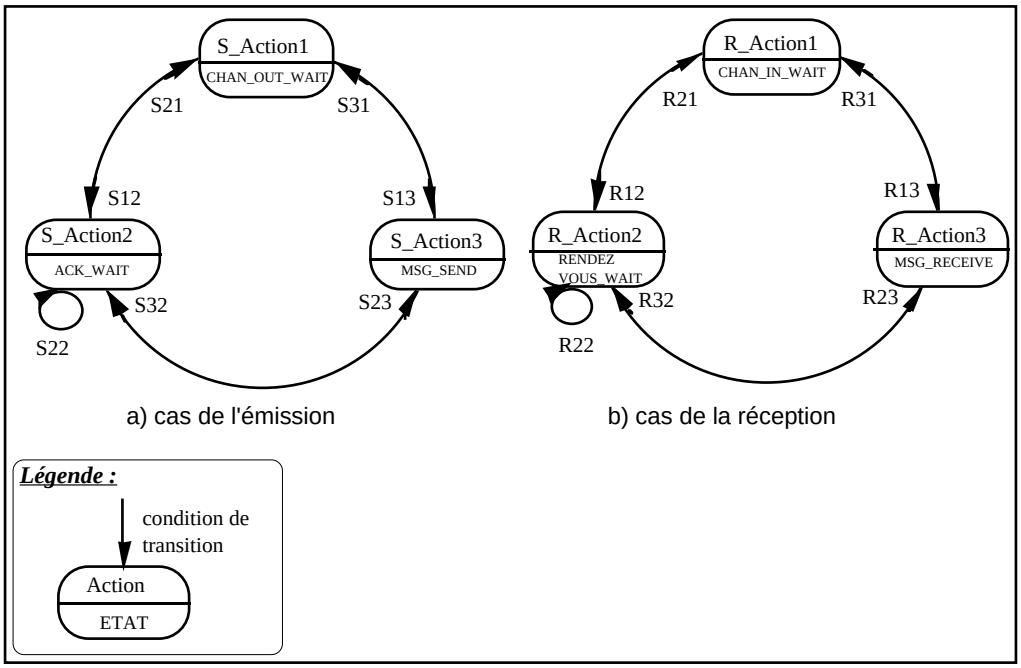
- channel_out(*chan, buf, size*) : émission sur le canal *chan* à partir du tableau *buf* de taille *size.*

- channel_in(*chan, buf, size*) : réception sur le canal *chan* de données de taille *size* dans le tableau *buf.*

La figure 9 donne les diagrammes de transitions d'états d'un canal. Deux diagrammes sont représentés, ils correspondent à deux états associés à une extrémité d'un canal : l'état d'émission (figure 9.a) et l'état de réception (figure 9.b).

Dans le cas de l'émission (resp. réception), si aucun message n'est à envoyer (resp. recevoir) l'état est CHAN_OUT_WAIT (resp. CHAN_IN_WAIT). Lorsque la primitive d'émission *channel_out* (resp. de réception *channel_in*) est exécutée, un premier message de rendez-vous est envoyé du côté émetteur vers le côté récepteur. L'état est alors ACK_WAIT (resp. RENDEZVOUS_WAIT). Lorsque le côté récepteur reçoit ce message, il envoie un acquittement à

l'émetteur et passe à l'état MSG_RECEIVE. L'émetteur passe à l'état MSG_SEND à la réception de l'acquittement.



S12	channel_out(chan,buffer_s, taille_e)
S21	acquittement reçu & (taille_e ≤ BUF_SIZE)
S22	acquittement reçu & (taille_e > BUF_SIZE) & taille_r ≤ BUF_SIZE
S13	taille émise par le précédent channel_out<taille_r
S31	taille_e ≤ taille_r
S23	acquittement reçu & (taille_e > BUF_SIZE) & (taille_r > BUF_SIZE)
S32	taille_r < taille_e

R12	channel_in(chan,buffer_r,taille_r)
R21	premier paquet reçu & (taille_r≤BUF_SIZE)
R22	premier paquet reçu & taille_e ≤ BUF_SIZE & (taille_r > BUF_SIZE)
R13	taille reçu par le précédent channel_in < taille_e
R31	taille_r ≤ taille_e
R23	premier paquet reçu & (taille_e > BUF_SIZE) & (taille_r > BUF_SIZE)
R32	taille_r > taille_e

S_Action1	réveiller le client
S_Action2	envoyer le premier paquet
S_Action3	envoyer le reste du message

R_Action1	réveiller le client
R_Action2	recevoir le premier paquet
R_Action3	envoyer aquittement ; recevoir le reste du message

BUF_SIZE est la taille des données envoyée dans le premier paquet de rendez-vous

Figure 9 : Diagrammes de transtions d'états pour les canaux

Le port est un objet de communication synchrone plusieurs vers un, il est unidirectionnel. L'identité des émetteurs ainsi que leur nombre sont inconnus. Les diagrammes d'états pour le port sont similaires à ceux du canal unidirectionnel, sauf qu'il faut tenir compte du fait qu'il existe plusieurs émetteurs en même temps.

Pour un canal l'unité de migration est l'extrémité du canal alors que pour le port, il existe une extrémité réceptrice et un nombre indéterminé d'extrémités émettrices.

Dans ce qui suit nous allons considérer la migration d'objets de communication. Pour cela, deux types de primitives sont réalisées : *communication_object_migrate_out* exécutée sur le processeur source de la migration et *communication_object_migrate_in* exécutée sur le processeur destination de la migration. Ces primitives permettent de transférer un objet de communication d'un processeur à un autre. Elles peuvent être appelées dans n'importe quel ordre de précédence, de façon asynchrone. Elles sont utilisées lors de la migration d'un processus en les appliquant sur chaque objet de communication associé au processus migrant. L'ensemble des appels peuvent s'effectuer en parallèle et de manière indépendante de la progression des autres tâches effectuées à la migration du processus.

V.4.3. Cas des canaux

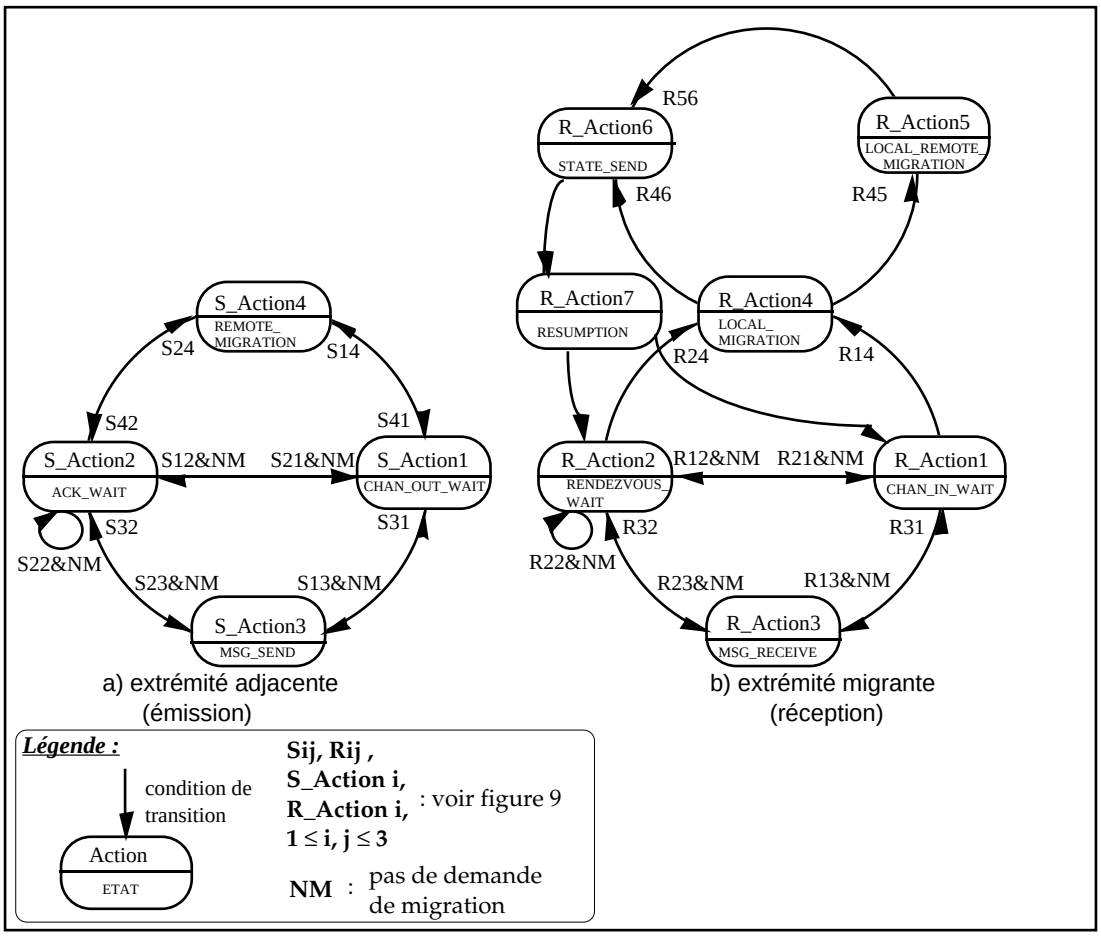
La communication à travers un canal est de type synchrone, elle est établie entre deux processus bien déterminés. Les trois protocoles proposés sont ainsi applicables. Cependant le protocole de redirection nécessite une gestion des tampons de redirection, qui de plus, laisse des dépendances résiduelles sur les processeurs sources de migration. D'autre part, ce protocole peut refuser une demande de migration. En raison de ces inconvénients, ce protocole n'a pas été retenu. Nous avons plutôt utilisé les deux autres protocoles, c'est à dire, le protocole de gel des communications et le protocole de rejet. Une amélioration a été introduite au protocole de rejet : la nouvelle adresse du processus migrant est envoyée avec le rejet, on évite ainsi le recours au mécanisme de localisation. Etant donné que le canal est établi entre deux processus adjacents et que la communication est synchrone, on a au plus un message qui est rejeté à la fois. L'espace mémoire nécessaire pour la gestion du rejet est par conséquent limité. Reste le fait que si le processeur source d'une migration tombe en panne, le rejet ne pourra plus être utilisé pour déterminer l'adresse du processus migrant. Le recours au mécanisme de localisation est alors inévitable.

Application du protocole de gel

La figure 10 donne les nouveaux diagrammes d'états associés à l'extrémité d'un canal, où la migration a été supportée en appliquant le protocole de gel des communications.

Considérons la migration d'une extrémité d'un canal associée à un processus à faire migrer. Sur le processeur source de la migration, aucun message ne peut être envoyé à travers l'extrémité du canal puisque le processus émetteur est suspendu. C'est pourquoi sur le processeur source nous ne considérons que le diagramme d'états correspondant à la réception de messages. Quatre états ont été rajoutés au diagramme de base. L'extrémité du canal est à l'état LOCAL_MIGRATION lorsque une migration est initiée sur le processeur local. S'il existe un message en cours de réception, la réception est achevée avant d'envoyer la requête de migra-

tion (*migration_request*). Lorsque la confirmation de cette requête (*migration_confirm*) arrive, l'état est alors STATE_SEND. Mais si une requête de migration de l'extrémité adjacente arrive avant la confirmation, l'état passe alors à LOCAL_REMOTE_MIGRATION. Une confirmation en réponse à cette requête est envoyée, l'état devient alors STATE_SEND. A ce stade, la valeur des structures associées à l'extrémité du canal est envoyée à la destination de la migration à travers le message *state_transfer*. Une fois que cette valeur est reçue sur le processeur destination, l'état passe alors à RESUMPTION. Un message de reprise (*resumption_message*) est envoyé au processus adjacent. L'état revient à la valeur avant l'opération de migration. L'extrémité du canal se trouve alors sur la destination de la migration.



S24, S14	reception de *migration_request*
S42, S41	réception de *resumption_message*

R24, R14	*faire migrer l'*extémité du canal
R45	réception de *migration_request*
R46, R56	réception de *migration_confirm*

S_Action4	achever l'envoie du message courant

R_Action4	achever la reception du message courant envoyer *migration_request*
R_Action5	envoyer *migration_confirm*
R_Action6	envoyer *state_transfer*
R_Action7	envoyer *resumption_message*

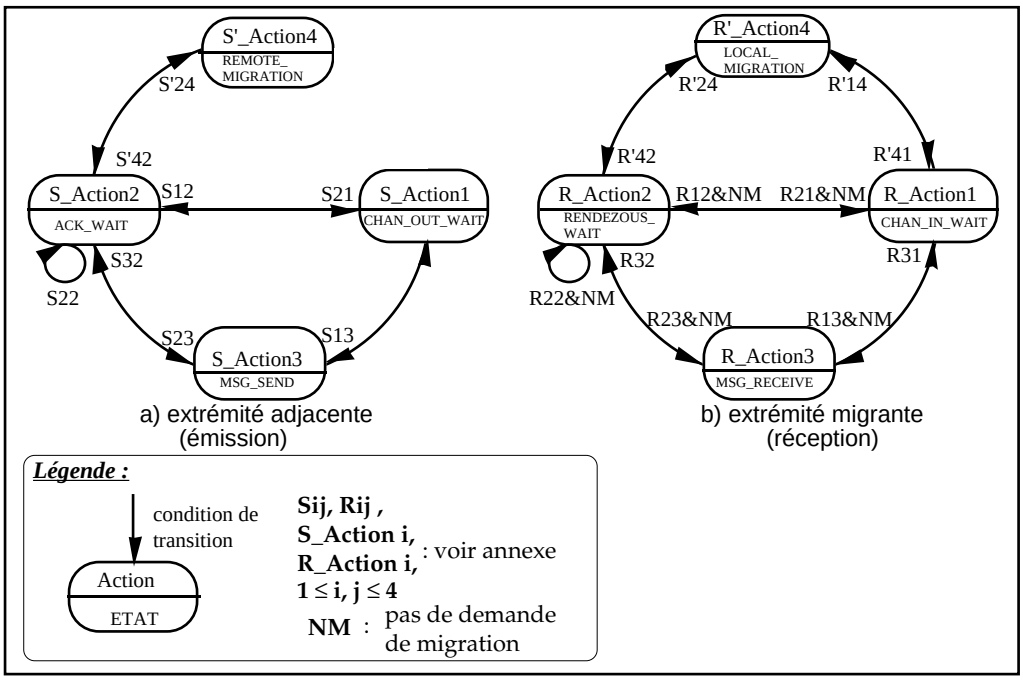
Figure 10 : Diagrammes de transitions d'états modifiés
- application du potocole de gel -

Sur le processeur adjacent à l'extrémité du canal en migration, seul le cas d'émission est à considérer, l'objectif étant de geler l'émission des messages. Lorsqu'une requête de migration est reçue, l'état de l'extrémité du canal est REMOTE_MIGRATION, la communication courante est achevée, ensuite, une confirmation de la migration est envoyée. L'émission des messages reste gelée jusqu'à ce que le message de reprise soit reçu.

Remarquons que dans le cas où les deux extrémités d'un canal sont simultanément en migration, nous assurons l'acheminement correct des deux messages de reprise vers les nouvelles destinations.

Application du protocole de rejet

La figure 11 représente les diagrammes de transitions d'états associés à l'extrémité d'un canal, où la migration a été supportée en appliquant le protocole de rejet des messages.



S'24	réception d'un *reject* ou après *time_out*
S'42	retransmission du dernier paquet

S'_Action4	retourner à l'état pour retransmettre le dernier paquet

R'14, R'24	faire migrer l'extémité du canal
R'41, R'24	fin de la migration

R'_Action4	durant la migration, rejeter les messages après la migration, revenir à l'état précédent

Figure 11 : diagrammes de transitions d'états modifiés
- application du protocole de rejet -

Un nouvel état LOCAL_MIGRATION est rajouté au diagramme d'états de réception des messages. Lorsque l'extrémité d'un canal est en migration, elle passe à cet état. Durant la migra-

tion, les messages destinés à cette extrémité sont rejetés. Après que la migration soit achevée, l'extrémité du canal revient à l'état dans lequel elle a été avant la migration.

Sur le processeur adjacent à l'extrémité migrante, si un message déjà émis est rejeté ou après l'écoulement du délai de garde (déclenché à l'envoi du message), l'état passe à REMOTE_MIGRATION et le message est alors retransmis.

V.4.4. Cas des ports

Etant donné que les émetteurs vers un port sont a priori inconnus, le protocole de gel n'est pas applicable. Le protocole de redirection pourrait être utilisé, mais pour les mêmes raisons que nous avons données dans le cas des canaux, nous ne retenons pas cette solution. Nous avons ainsi adopté le protocole de rejet, la description des diagrammes d'états pour supporter ce protocole sont semblables à ceux relatifs aux canaux. Dans le cas des ports, des messages peuvent être ignorés à la réception sans envoyer un rejet. Ceci est dû au fait que le nombre d'émetteurs n'est pas limité, et par conséquent, le nombre de rejets à envoyer ne l'est pas non plus.

V.4.5. Evaluation

L'implantation d'un protocole de migration introduit un surcoût au niveau des besoins en espace mémoire ainsi qu'au niveau des délais de communication. Une évaluation de ces surcoûts est faite dans cette section. Les résultats reportés sont relatifs au noyau ParX s'exécutant sur une machine SuperNode [MuW90] à base de transputers.

V.4.5.1. Besoins en espace mémoire

Les protocoles que nous avons élaborés utilisent un espace mémoire de taille limitée indépendamment du nombre de processeurs dans le réseau. Nous distinguons trois types d'actions nécessitant de la mémoire :

- la construction et la mise en file d'une requête,
- la conservation d'un message reçu ou à émettre,
- le maintien de l'état d'un objet de communication.

Protocole de gel

Sur chacun des processeurs source et destination d'une migration, le protocole de gel (appliqué aux canaux) utilise le même espace mémoire pour la construction des différentes

[1] L'espace mémoire nécessaire pour charger le code exécutable n'est pas considéré.

requêtes. Rappelons que l'interaction entre les trois serveurs d'un protocole se fait au moyen de requêtes. La taille d'une requête est constante et est fixée à 25 octets dans l'implémentation actuelle de ParX.

La figure 12 donne les différentes tailles des messages de contrôle échangés dans le protocole de gel. Ainsi sur un processeur donné et pour une opération de migration d'une extémité du canal, un tampon de 76 octets suffit pour la gestion de ces messages car l'envoi de ces messages ne se recouvrent pas dans le temps (sur un même processeur).

Messages de contrôle	**Taille (Octets)**
channel_end_migration_request	20
channel_end_migration_confirm	16
state_channel_end_transfer	76
channel_end_resumption	16

Figure 12 : tailles des messages de contrôles

Le rajout de nouveaux états dans le diagramme de transitions d'états ne nécessite pas d'espace mémoire supplémentaire.

Protocole de rejet

L'envoi d'un rejet se fait à travers une requête déposée par le serveur *protocol_receiver* et servie par le serveur *protocol_sender*. Avant de pouvoir envoyer un rejet, un nouveau message à rejeter peut arriver. Il faut alors disposer d'un nouvel espace mémoire pour la construction d'une nouvelle requête permettant de rejeter ce dernier message. Dans le cas du canal, ceci ne peut pas se produire puisque la communication est synchrone et point à point. Par conséquent, un nouveau message ne peut être envoyé qu'après que le précédent ait été traité. Par contre, dans le cas du port, le nombre d'émetteurs est indéterminé et donc plusieurs requêtes de rejet peuvent être déposées au *protocol_sender* avant qu'il ne puisse les satisfaire. Cette situation se produit lorsque le taux d'arrivée des messages à rejeter est plus important que le taux d'émission des rejets. La taille de l'espace mémoire réservé pour la construction des rejets est donc un paramètre qui dépend de ces taux d'arrivée et d'émission mais aussi de la disponibilité de la mémoire. La limitation de la taille mémoire nécessaire est assurée grâce au mécanisme de délai de garde.

Le transfert de l'état d'un canal se fait à travers le même type de message que dans le cas du protocole de gel.

Les besoins en espace mémoire que nous venons d'énumérer sont relatifs à la migration d'un seul objet de communication. Lorsqu'il s'agit de faire migrer un processus, il faut considérer l'ensemble des objets de communication qui lui sont associés. Les migrations de ces objets

peuvent se faire en parallèle, mais les besoins en espace mémoire se multiplieront alors par le nombre de migrations effectuées en parallèle. Le degré de parallélisme va donc dépendre de la disponibilité de l'espace mémoire.

V.4.5.2. Surcoût au niveau des délais de communication

L'intégration des protocoles de migration aux protocoles de communication introduit des délais supplémentaires au niveau de la communication :

- délai dû à la prise en compte de la migration au niveau de la couche protocole,
- délai dû à la correction de l'acheminement d'un message destiné à un processus ayant migré.

Nous avons mesurer ces délais dans le cas des protocoles de gel et de rejet. Les mesures sont effectuées sans imposer une charge supplémentaire sur les processeurs, ni sur le réseau de communication.

Protocole de gel

Après avoir intégrer le protocole de gel, un surcoût d'au plus 20 µs est rajouté à une communication à travers un canal, ce qui reste négligeable par rapport aux délais de communication qui est de l'ordre de la milli-seconde (figure 13).

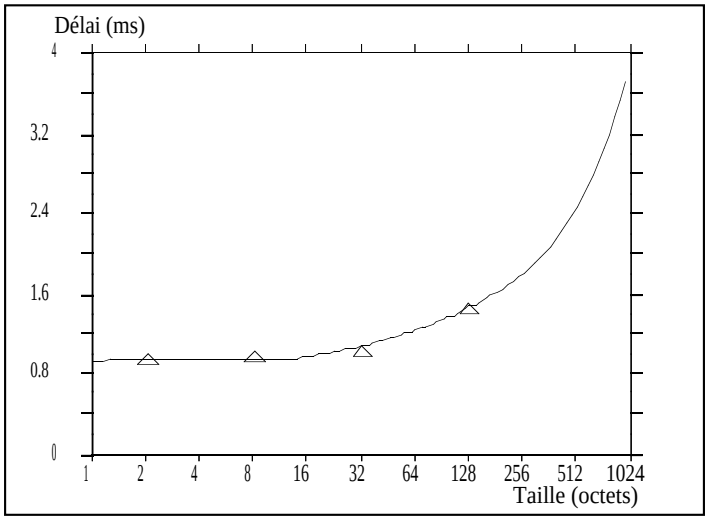


Figure 13 : délai de communication en fonction de la taille des messages[1]

[1] Ces mesures ont été faites pour une distance égale à deux.

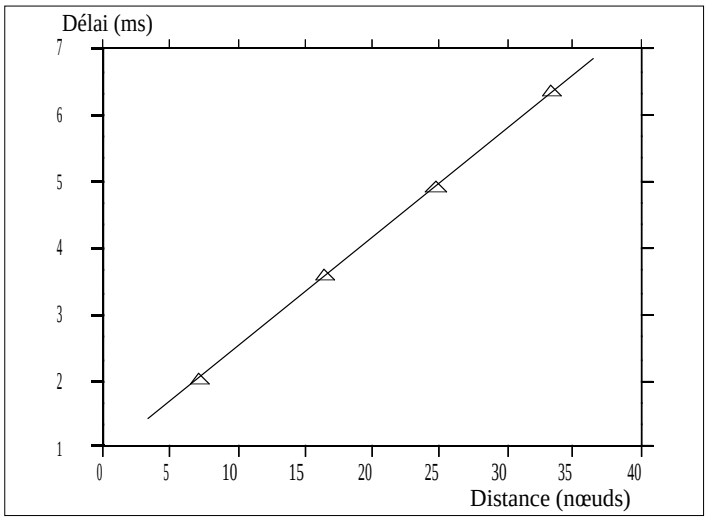


Figure 14 : Durée de gel en fonction du nombre de nœuds parcourus par l'ensemble des messages de contrôle

La durée de gel des communications constitue le surcoût qui peut se rajouter aux délais d'envoi des messages vers le processus ayant migré. Si pendant la période de gel aucune requête d'envoi de message n'est effectuée, le surcoût est alors nul.

Etant donné que le délai d'envoi des différents messages de contrôle (mettant en œuvre le protocole de gel) est sensiblement le même pour une même distance, la durée de gel dépend plutôt du nombre de nœuds parcourus par les différents messages de contrôle. La figure 14 montre que ce délai croit de façon linéaire en fonction du nombre de nœuds parcourus.

Protocole de rejet

Nous avons appliqué ce protocole aux canaux et aux ports, le surcoût introduit par le rajout de ce protocole est inférieur à 3 µs et est donc négligeable. Nous avons évalué le surcoût rajouté au délai de communication en mesurant la durée entre l'instant d'envoi d'un message et la réception du rejet correspondant, les résultats sont présentés dans la figure 15. Cette durée est limitée par le délai de garde au bout duquel un message est retransmis (si aucune réponse n'est reçue : acquittement ou rejet). Elle dépend de la distance qui sépare le processeur émetteur du message et sa destination. Le délai de rejet croit de façon linéaire en fonction de la distance entre les processeurs émetteur et destination du message. La valeur du délai de garde est choisie en estimant le délai d'envoi d'un paquet sur une distance égale au diamètre du réseau, lorsque celui-ci est chargé. Nous avons ainsi choisi une grande valeur pour le délai de garde car nous supposons que la probabilité qu'un message soit ignoré sans être rejeté est faible. En effet, pour que cela se produise, il faudrait que le processus récepteur migre et qu'il n'y ait plus d'espace mémoire pour l'élaboration d'une requête de rejet.

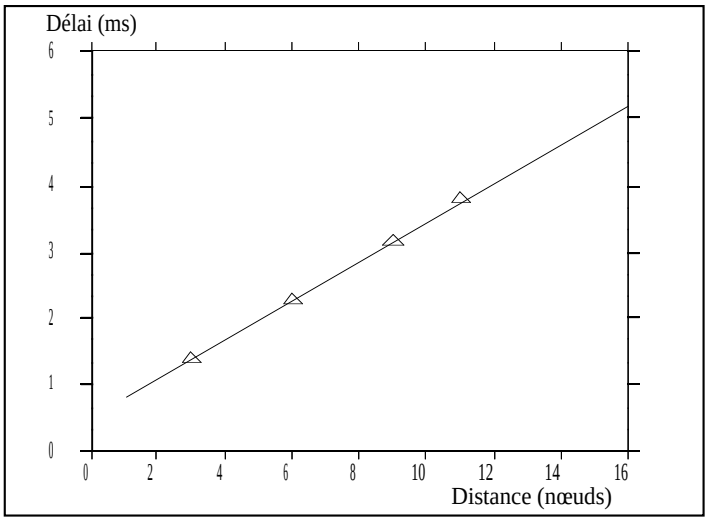


Figure 15 : Délai de rejet

A la différence du protocole de gel, le surcoût au niveau des délais de transmission est inévitable. Il pénalise seulement le premier message envoyé d'un processus adjacent vers le processus migrant. Le second message n'est pas a priori pénalisé, étant donné qu'à la réception d'un rejet la localisation du processus migrant est mise à jour.

Au délai de rejet s'ajoute le délai de détermination de la nouvelle adresse. Ceci est effectué au moyen du mécanisme de désignation et de localisation.

V.5. Conclusion

Nous avons considéré divers sémantiques de communication pour lesquels nous avons proposés des protocoles de migration adéquats. Ces protocoles sont applicables sur des architectures extensibles. Pour cela, nous avons évité les traitements centralisés, et sur chaque processeur du réseau, les besoins en espace mémoire sont indépendants de la taille du réseau.

Le protocole de redirection est particulièrement utile lorsque la communication est de type asynchrone. Les deux autres protocoles proposés ne s'appliquent que dans le cas de communications synchrones. L'inconvénient majeur du protocole de redirection est qu'il maintient des dépendances résiduelles sur le processeur source d'une migration.

L'utilisation du protocole de gel suppose que l'identité de chaque processus adjacent est connue sur le processeur source d'une migration. Lorsque cette condition est satisfaite et pour des communications synchrones, le protocole de gel est applicable.

Le protocole de rejet est applicable lorsque les communications sont synchrones. Son utilisation nécessite le recours aux mécanismes de désignation et de localisation.

Les communications asynchrones peuvent être construites au dessus des communications synchrones. Une primitive de communication est appelée par un processus lancé en parallèle avec le processus père (voulant effectuer la communication). Le processus fils se bloque sur la communication, le processus père poursuit son exécution de manière asynchrone. Dans ces conditions, les protocoles de gel et de rejet sont applicables.

Notons enfin que l'intégration des protocoles de migration introduit un surcoût négligeable au niveau des délais de communication. Ce résultat est important puisque ces protocoles sont intégrés dans un micro-noyau où les temps de réponse doivent être minimisés.

Chapitre VI

Cohérence et transparence de la Désignation

Le chapitre précédent ne traite que de la correction des échanges de messages au niveau du noyau de communication. Nous avons proposé des protocoles de migration assurant un acheminement correct des messages, tout en respectant la sémantique des protocoles de communication. Cependant, le problème de la désignation des processus n'a pas été encore traité. Pour pouvoir communiquer avec un processus, nous avons supposé qu'il est possible de le désigner par un identificateur unique dans le domaine de communication.

Les entités passives (fichier, canal, etc.) peuvent être accédées à travers des processus (serveurs) qui effectuent l'accès à ces entités localement (comme par exemple le cas de Mach [SJR86] ou Charlotte [ArF89]). Ainsi, en supportant la migration au niveau de la communication inter-processus, le problème de la localisation de ces entités est résolu. En revanche, les mécanismes de communication peuvent ne pas supporter directement la migration de processus. Un service de localisation tenant compte de la migration est alors nécessaire. Ce service, qui n'est pas propre à la communication, permet de retrouver la nouvelle adresse d'un processus ayant migré. Cette approche a été adoptée par exemple pour le système V.

Il apparaît donc que la communication d'une part, et la désignation et l'accès aux entités d'autre part, sont très liés. Plus précisément, et en ce qui nous concerne, la transparence de la migration doit être assurée conjointement par ces deux mécanismes.

Les techniques de désignation et d'accès aux entités dépendent des propriétés des identificateurs. Dans ce chapitre, nous commençons par étudier les conséquences de ces propriétés sur la migration. Nous décrivons ensuite les techniques de localisation des entités pouvant migrer. Nous présentons enfin les solutions retenues dans le cas du le noyau ParX. Ces solutions sont évaluées à la fin du chapitre.

VI.1. Concepts de base

VI.1.1 Définitions

* Un *nom* (ou *identificateur*) d'une entité est un moyen pour désigner cette entité. Selon le contexte d'utilisation, différents noms peuvent être associés à une même entité. On distingue ainsi deux types de noms : les noms externes et les noms internes.

* Un nom *externe* (ou *symbolique*) est un nom manipulé par l'utilisateur, il est généralement représenté par une chaîne de caractères.

* Un nom *interne* est interprétable par le système qui peut ainsi retrouver l'entité désignée. Une capacité est un exemple de nom interne qui en plus est utilisée pour la protection de l'entité [TMV86].

* Dans un système distribué, il existe des entités équivalentes (par exemple des processus fournissant un même service, des bibliothèques, etc.) qu'il est utile de désigner par un même identificateur. Pour cela, on définit des *classes* ou *groupes d'entités,* chaque groupe est désigné par *un identificateur* permettant d'accéder à l'une des entités disponibles du groupe.

* La *liaison* est une opération qui associe à un identificateur une adresse physique localisant l'entité désignée.

* Un *service de désignation* est un mécanisme qui permet d'associer des noms à des entités manipulées dans le système, et par là même l'accès à ces entités.

* Un nom est qualifié de référence *logique* s'il ne renferme pas l'adresse de l'entité désignée. Dans le cas contraire, le nom représente une référence *physique* indiquant l'adresse de l'entité.

* Un nom est *global* s'il permet de référencer une même entité à partir de n'importe quel processeur du réseau. Pour qu'un nom soit global il faut qu'il soit unique dans ce domaine.

* Un nom est *local* s'il dépend du contexte d'utilisation. Il peut référencer une entité globalement accessible, mais il n'est pas forcément le même d'un processeur à un autre. Un même nom local peut désigner différentes entités selon le processeur où il est utilisé.

* Une liaison est dite *statique* si elle est effectuée avant l'exécution du programme : à la compilation, à l'édition de liens ou au moment du chargement.

* Une liaison est dite *dynamique* si elle est effectuée au cours de l'exécution du programme, soit au premier accès à une entité, soit à chaque accès. Cette opération est appelée aussi *résolution* d'un identificateur.

VI.1.2 Modèle d'accès

Nous présentons, maintenant, un modèle général décrivant l'opération d'accès à une entité par un processus. Ce modèle s'applique aussi bien pour un accès local que distant.

Afin d'accéder à une entité, un processus doit formuler une requête d'accès spécifiant le nom de l'entité et l'opération d'accès. Cette requête est traitée par le service de désignation qui déduit l'adresse physique de l'objet et exécute la fonction associée à l'opération d'accès à cette entité. Lorsque l'objet se trouve sur un processeur distant, la requête est relayée au service de désignation du processeur distant où se trouve l'entité. La figure 1 schématise le modèle d'accès.

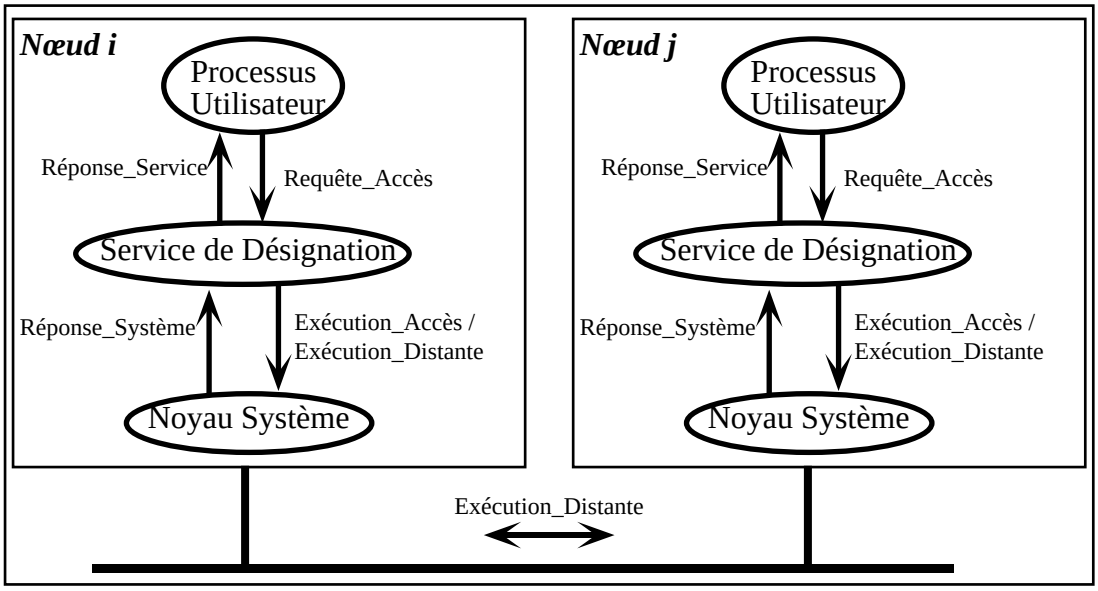


Figure 1 : Modèle d'accès

Suivant à ce modèle, différentes opérations d'accès peuvent être traitées :

- lecture / écriture d'un segment mémoire,
- opérations d'entrée / sortie sur un fichier,
- émission / réception d'un message à travers un objet de communication,
- appels de procédure.

VI.2. Problèmes posés

Dans diverses situations et lorsque la migration de processus n'est pas prise en compte, la désignation devient incohérente. Nous illustrons ces situations à travers les exemples suivants.

Cas 1 : chaque processus est désigné par un nom interne au système. Si ce nom est un nom physique, à chaque migration du processus ce nom change. L'ancien nom utilisé par d'autres processus devient alors périmé.

Cas 2 : dans le cas où le nom interne d'un processus est local, la migration du processus risque de produire des conflits de noms : le même nom du processus peut être déjà utilisé sur le processeur destination de la migration pour désigner une autre entité. De plus, si un processus ayant migré a accès à d'autres entités et si les noms utilisés sont locaux, l'utilisation de ces noms sur le processeur destination peut ne plus correctement référencer les entités désignées avant la migration.

Cas 3 : il existe un certain nombre de ressources système qui sont équivalentes et qui sont dupliquées sur chaque processeur du réseau. Il serait coûteux qu'un processus ayant migré continue à accéder à ces ressources sur le processeur source de la migration.

Cas 4 : le service de désignation maintient la localisation des entités désignées, la migration d'un processus nécessite la mise à jour des tables de localisation. De plus, l'opération de migration d'un processus n'est pas atomique ce qui pose un problème, puisqu'à tout moment, il faut pouvoir attribuer une adresse à un nom.

Dans ce qui suit, nous étudions les moyens pour résoudre ces problèmes. Différentes organisations du service de désignation sont considérées, notamment, au niveau de l'identification des entités et de la technique de liaison.

VI.3. Choix d'un identificateur

Nous avons déjà énuméré un certain nombre de propriétés qui peuvent se rattacher à un identificateur selon sa structure et sa sémantique. Dans cette section, nous analysons les différentes possibilités d'identification dans le contexte de la migration de processus. Les mécanismes supplémentaires pour supporter la migration ainsi que les choix de conception qui s'imposent sont ainsi dégagés.

Reprenons le problème posé au cas 1 (cf. §VI.2), lorsque le nom utilisé est une référence physique, la migration de l'entité désignée implique le changement du nom. Ainsi, l'ancienne

valeur de ce nom, utilisée dans différents endroits du système, sera incohérente. Si des processus utilisent directement ce nom, il serait difficile de mettre à jour toutes les variables gardant la valeur du nom. En effet, ces variables ne sont pas connues a priori, puisque la valeur d'un nom peut s'échanger entre les processus. Il faut donc éviter d'utiliser des références physiques et recourir plutôt à des références logiques qui ne sont pas modifiées à la suite de la migration des entités désignées. Néanmoins, un nom peut contenir l'adresse de création qui reste immuable même à la suite d'une migration. Par exemple c'est le cas de la désignation des processus dans les systèmes Demos/MP [PoM83], Locus [PWC81], Mosix [BGW93], et Sprite [DoO91].

Le recours à une désignation logique introduit cependant des délais à chaque accès. Ce surcoût correspond à l'opération de liaison qui se rajoute pour retrouver l'adresse physique d'une entité à accéder. Les entités conservées dans l'espace d'adressage du processus ne sont pas concernées par ce surcoût, puisqu'ils vont migrer avec l'image du processus. Il est donc important d'identifier les entités pouvant être représentées dans l'espace d'adressage d'un processus et de les garder dans cet espace.

Considérons le problème posé par le cas 2, lorsqu'un nom est local, il peut ne pas être unique sur l'ensemble des processeurs du réseau. De plus, son interprétation doit se faire, a priori, sur le processeur où il a été utilisé initialement. Or, pour que l'on puisse migrer un processus, il faut que son nom de même que tous les noms utilisés par ce processus ne soient pas déjà réservés par le système du processeur destination. Dans ces conditions, les entrées correspondant à ces différents noms pourront être créées et utilisées sur le processeur destination. Ces noms seront alors locaux par rapport au nouveau processeur.

Lorsque les noms utilisés sont globaux, les accès se font correctement sans mécanismes supplémentaires. En effet, les entités peuvent être désignées de la même manière à partir de n'importe quel processeur du réseau.

Amœba et système V ont adopté une désignation globale. L'identificateur d'une entité comporte un identificateur unique du serveur qui gère l'entité et un numéro local au serveur[1]. Remarquons que les identificateurs de processus dans les systèmes Sprite et Locus, décrits plus haut, sont aussi globaux.

Par ailleurs, les ressources répliquées sur les différents processeurs doivent être accédées de façon locale au processeur d'exécution (cas 3). L'utilisation d'identificateurs de groupes per-

[1] Dans le cas de Amœba, à ces champs s'ajoutent un champ spécifiant les droits d'accès et un champ pour la protection. L'ensemble est appelé capacité.

met de résoudre le problème. La tendance dans Sprite est de rendre les appels systèmes indépendant du processeur d'exécution [DoO91].

VI.4. Maintien de la cohérence de l'opération de liaison

On distingue différentes techniques de liaison. L'utilisation de ces techniques dépend des propriétés des identificateurs adoptés. Nous nous intéressons à la liaison dynamique, la liaison statique concerne les entités qui doivent être transférées avec le processus, essentiellement les entités faisant partie de l'espace d'adressage associé au processus. Nous présentons dans cette section des techniques de liaison dynamique supportant la migration de processus.

VI.4.1 Utilisation de tables de correspondances

L'opération de liaison peut s'effectuer au moyen de tables de correspondance entre les noms logiques et les nom physiques. Il existe différentes organisations de ces tables selon qu'elles sont centralisées ou réparties, complètes ou partielles, dupliquées ou non.

Lors de la migration d'une entité et dans le cas où des liens de poursuite ne sont pas utilisés, les entrées dans les tables de correspondance associées à cette entité doivent être mises à jour (problème posé dans la section VI.2, cas 4). Si les adresses de ces tables sont connues, les mises à jour pourront s'effectuer directement. Sinon, nous envisageons les deux solutions suivantes :

- soit de diffuser une requête de mise à jour. Pour être viable, cette technique devrait être supportée par le matériel.

- soit de définir un processeur de référence qui est mis à jour à chaque migration. Les autres processeurs le consultent à la suite des accès ayant échoué. Cette solution est développée, par la suite, dans le cas de ParX.

En effectuant une opération de liaison, des incohérences peuvent se produire entre l'adresse obtenue et l'adresse réelle. Par exemple, si à la suite de la migration d'un processus, les opérations de mise à jour sont effectuées après une opération de liaison à cette entité, l'accès sera alors incorrect. Ceci est dû, de façon générale, à des *aléas* sur les instants de mise à jour des tables de correspondance, d'envoi des requêtes d'accès, de début et de fin de migration [JMJ89]. Des mécanismes supplémentaires sont nécessaires pour remédier aux problèmes dus à ces aléas.

Les solutions consistant à éviter les problèmes posés par les aléas, sont complexes. En effet, les opérations de liaison et d'accès à une entité ne peuvent pas s'effectuer de façon atomique, d'autant plus qu'à une opération de liaison peuvent correspondre plusieurs accès et que la migration de l'entité peut se produire entre une liaison et un accès.

Ainsi, lorsqu'une requête d'accès à une entité parvient à un processeur où l'entité n'est plus présente, la requête doit être reprise. Pour cela deux solutions sont généralement utilisées :

i) L'émetteur initial de la requête doit reprendre l'opération d'accès et effectuer une nouvelle opération de liaison. La reprise peut être déclenchée suite à un délai de garde ou un acquittement négatif envoyé par le processeur où la requête a échoué. Les systèmes V et Amœba utilisent un tel mécanisme.

ii) On maintient un lien de poursuite (une redirection) sur le processeur source d'une migration pour pouvoir rediriger les requêtes destinées à l'entité ayant migré. Cette solution est utilisée dans plusieurs systèmes, par exemple : Mach, Accent, Demos/MP, Charlotte, Peace.

Ces deux solutions correspondent, respectivement, aux protocoles de rejet et de redirection proposés dans le chapitre précédent et appliqués à l'acheminement des messages.

Nous étudions dans ce qui suit une organisation particulière des tables de correspondance. Nous décrivons un exemple d'une autre organisation, dite hiérarchique, lors de l'étude du cas de ParX (cf. §VI.5). Une telle organisation est aussi utilisée dans Peace [SSS89].

Désignation suivant le processeur origine

On définit le processeur *origine* ("*home node*") d'une entité comme étant le processeur où l'entité a été créée. Plusieurs systèmes (comme Locus, Sprite et Mosix) assurent la transparence de la migration grâce à une désignation qui reste relative au processeur origine. Chaque processeur maintient une table de localisation des entités créées sur ce processeur. L'opération de liaison se fait en utilisant ces tables. Pour cela, dans le contexte d'un processus est maintenu l'adresse du processeur origine de ce processus. Les requêtes d'accès, effectuées par le processus, sont redirigées vers ce processeur. Les requêtes d'accès à un processus ayant migré arrivent d'abord sur son processeur origine, elles sont alors redirigées vers le processeur où se trouve le processus.

Virtuellement un processus ayant migré, s'exécute toujours sur le processeur origine. Même à la suite d'une seconde migration, le processeur origine d'un processus reste le même. La taille du chemin de redirection peut être réduite à un en utilisant le protocole de redirection proposé au chapitre V.

Cette technique de désignation permet une bonne transparence de la migration, mais a tous les inconvénients du protocole de redirection. Elle implique notamment, une dépendance résiduelle vis-à-vis du processeur origine. De plus, elle introduit un surcoût au niveau des accès, puisqu'ils doivent passer par le processeur origine.

VI.4.2 Diffusion d'une requête de localisation

L'adresse physique d'une entité est obtenue en diffusant une requête de localisation contenant le nom de l'entité. Ce nom doit être global. Le processeur, où réside l'entité, renvoie alors une réponse indiquant l'adresse de l'entité.

Cette technique, à elle seule, serait encombrante pour le réseau. Pour y remédier, les Systèmes V et Amœba l'ont combiné avec l'utilisation de tables de correspondance locaux. Ces tables sont gérées en tant que caches afin d'accélérer les accès aux entités déjà référencées. Notons que dans le cas des systèmes répartis, la communication entre les processeurs du réseau est moins importante que dans les systèmes massivement parallèles, ce qui rend la technique de diffusion tout à fait viable.

Les mêmes aléas, décrits dans la section précédente, peuvent se produire lors d'un accès à une entité. Les solutions déjà proposées s'appliquent aussi dans le cas de cette technique.

VI.5. Mise en œuvre dans ParX

Le service de désignation dans ParX est un mécanisme de base, il permet l'accès aux services systèmes sans avoir à se préoccuper de la localisation des entités manipulées. Cette propriété de transparence de la désignation est d'autant plus indispensable lorsqu'il s'agit de l'exécution de programmes sur des machines massivement parallèles, où la configuration offerte diffère d'une exécution à une autre.

VI.5.1 Organisation des domaines de désignation

Les entités gérées dans ParX sont organisées selon une hiérarchie particulière qui découle du modèle d'exécution. Cette hiérarchie, décrite par la figure 2, indique le rattachement de chaque type d'entité à un type d'entité englobant. Le système représente la racine de la hiérarchie et englobe toutes les entités existantes. Dans le système s'exécutent un certain nombre de tâches parallèles (Ptask). L'exécution d'une Ptask se fait sur une partie de l'ensemble des processeurs du réseau (cluster). Une Ptask est composée de tâches *(*task), on peut aussi rattacher des ports (*port*) ou d'autres entités à une Ptask. Une task est l'unité d'allocation,

elle s'exécute sur un seul processeur et se décompose en flots de contrôle (thread). A une task peuvent être associées différentes entités : segments mémoires, canaux, ports, etc.

La désignation dans ParX a été traitée dans la thèse de F. Menneteau [Men93]. Il distingue les domaines de désignation suivants :

- le *domaine public* composé des noms globaux visibles dans tout le système,
- le *domaine privé* associé à chaque Ptask en exécution. Un tel domaine représente l'ensemble des noms visibles par les différentes tasks d'une Ptask.

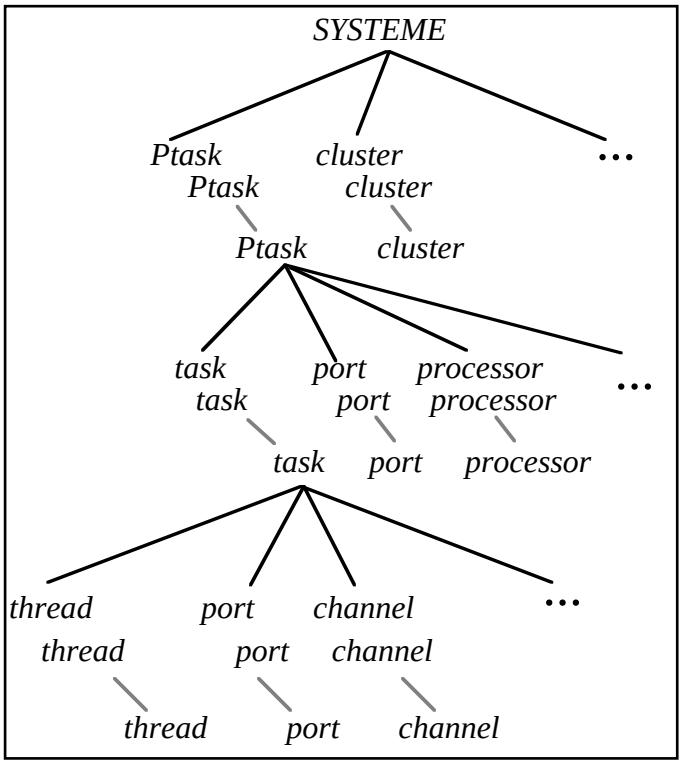


Figure 2 : Hiérarchie des entités

VI.5.2 Structure d'une capacité

Une entité est désignée grâce à une capacité composée des champs suivants :

- un identificateur unique de l'entité (*EID*),
- un champ indiquant les droits d'accès à l'entité (*EAR*),
- un champ pour la protection (*KEY*).

Notre intérêt se porte particulièrement à l'*EID*. En tenant compte de l'organisation hiérarchique des différentes entités, la gestion des identificateurs est effectuée de manière décentra-

lisée. Un *EID* est généré de façon à ce qu'il soit unique dans le domaine définie par l'entité englobante. Remarquons qu'un identificateur privé peut s'échanger entre différents processeurs du cluster alloué à la *Ptask*, il est donc utile de réduire la taille de cet *EID*. Pour cela selon le domaine de désignation, un *EID* comporte ou non le numéro de la Ptask englobante, un bit est prévu à cet effet. Dans sa version complète un *EID* comporte les champs suivants :

- le type de l'entité : Ptask, task, thread, *port*, *channel*, cluster, *processor*, etc,
- un numéro de la Ptask englobante si l'entité désignée appartient au domaine public,
- un numéro de la task englobante ou de l'entité si celle-ci est directement rattachée à la Ptask englobante,
- un numéro de l'entité si elle est rattachée à une task englobante.

Ainsi nous pouvons considérer qu'un *EID* est une référence logique et unique dans son domaine d'utilisation. Nous avons spécialement insisté sur cet aspect lors de l'élaboration du service de désignation car, comme nous l'avons déjà souligné (cf. §VI.3), cette condition est primordiale pour la migration d'entités.

Enfin, et en ce qui concerne l'accès aux services système dupliqués sur les différents processeurs, nous utilisons un même identificateur pour chaque type de service. C'est un identificateur de groupe qui est physique.

VI.5.3 Serveurs de désignation

Les opérations de liaison se font grâce à des tables de correspondance réparties sur les différents processeurs. Ces tables sont gérées par des serveurs de noms se trouvant sur chaque processeur du système, l'ensemble assurant le service de désignation [Men93].

Suivant la structure des identificateurs, une hiérarchie de serveurs est définie de façon à pouvoir décentraliser, le plus possible, le traitement des requêtes destinées au service de désignation. Ainsi, il existe 3 types de serveurs de noms :

- le serveur des noms publics qui gère les noms globaux à tout le système,
- les serveurs des noms privés (il en existe un par cluster),
- les serveurs des noms locaux qui se trouvent sur chaque processeur du réseau.

Lorsqu'un serveur de noms ne peut pas résoudre une opération de liaison, il a recours au serveur du niveau supérieur. Il maintient ensuite localement la correspondance entre le nom et l'adresse obtenue. Ceci a pour avantage de favoriser le traitement des requêtes par les serveurs des noms locaux, évitant ainsi le recours à des processeurs distants. De plus, les serveurs des

noms publics et privés, qui centralisent la gestion de certains espaces de désignation, seront moins sollicités et donc plus disponibles.

La migration d'une entité dont l'accès est géré par les serveurs de désignation implique la mise à jour des tables de correspondance. Ce problème est traité dans la section suivante.

VI.5.4 Mise à jour des tables de correspondance

La mise à jour des tables de correspondance pourrait être évitée en utilisant des liens de poursuite. Nous n'avons pas retenu cette solution au vu des critiques que nous avons porté à de telles techniques (cf. §V.2.3).

En reprenant l'organisation de la désignation dans ParX, décrite plus haut, nous constatons que la correspondance entre le nom d'une entité et son adresse peut être maintenue par plusieurs serveurs. Ces serveurs ne sont pas connus a priori. Afin de mettre à jour ces correspondances à la suite de la migration de l'entité, la solution consistant à diffuser une requête de mise à jour peut s'appliquer. Le coût de cette solution, c'est à dire l'encombrement du réseau et le traitement des requêtes diffusées, dépend de l'architecture du réseau et du support matériel pour la diffusion. Dans le cas défavorable, ce coût risque d'être important, cette solution n'a pas non plus été retenue.

La technique que nous avons adoptée consiste à différer les mises à jour et à les effectuer à la suite des opérations d'accès ayant échoué. Seule la correspondance maintenue par le serveur ayant créé initialement le nom de l'entité ayant migré est mise à jour directement lors de la migration. Ainsi, si le nom de l'entité est global, la mise à jour se fera au niveau du serveur des nom publics. Si l'entité est privée à une Ptask, la mise à jour se fera au niveau du serveur de noms du cluster où s'exécute la Ptask. Enfin, si le nom de l'entité est local au processeur source de la migration, il est rendu alors local au processeur destination de la migration.

Une fois que la mise à jour effectuée et à la suite d'un échec d'accès à une entité, le serveur ayant opéré la liaison consulte le serveur de noms du niveau supérieur pour obtenir la nouvelle adresse. Il met alors à jour la correspondance locale associée à l'entité. Ensuite, il exécute une nouvelle fois l'opération d'accès en utilisant la nouvelle adresse.

Les aléas de mise à jour ne posent pas de problèmes particuliers puisqu'à terme le serveur de noms du niveau supérieur effectue forcément la mise à jour. En effet, un échec d'accès entraîne systématiquement la reprise de l'opération d'accès, après avoir consulté le serveur de noms du niveau supérieur.

Dans la présentation de notre solution, nous avons implicitement supposé que l'exécution d'une opération d'accès rend toujours une réponse d'erreur. Ceci n'est pas vrai dans le cas général. L'écriture dans un mot mémoire à une adresse syntaxiquement correcte, peut se faire même si l'adresse ne correspond pas à l'entité à laquelle on veut accéder (figure 3).

Pour pouvoir générer un retour d'erreur à chaque accès, nous associons à toute entité un descripteur. Au niveau d'un serveur de noms, la correspondance maintenue est celle entre le nom logique d'une entité et l'adresse physique de son descripteur. Celui-ci comporte le nom et l'adresse physique de l'entité. A chaque accès, on vérifie si le nom utilisé pour la liaison correspond à celui indiqué par le descripteur (figure 4). Dans le cas favorable, l'accès pourra se faire en utilisant l'adresse contenue dans le descripteur. Sinon, le message de réponse indiquera l'échec de l'accès.

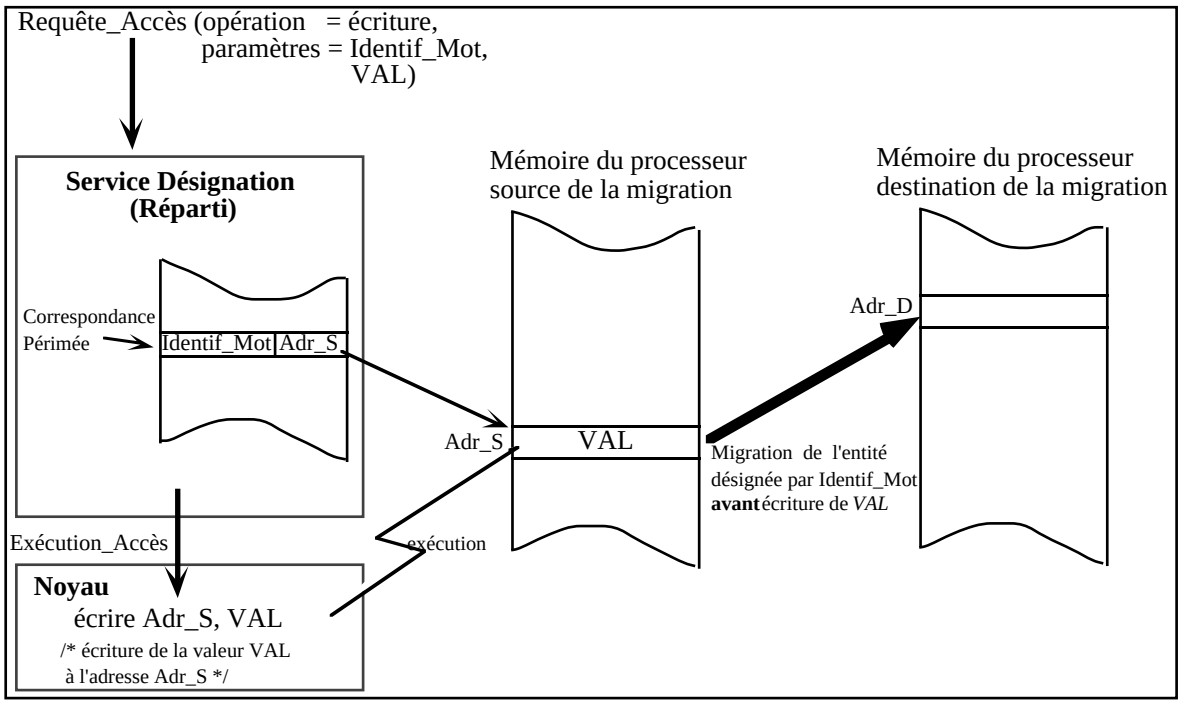


Figure 3 : Exécution d'une ecriture suivant une adresse incorrecte

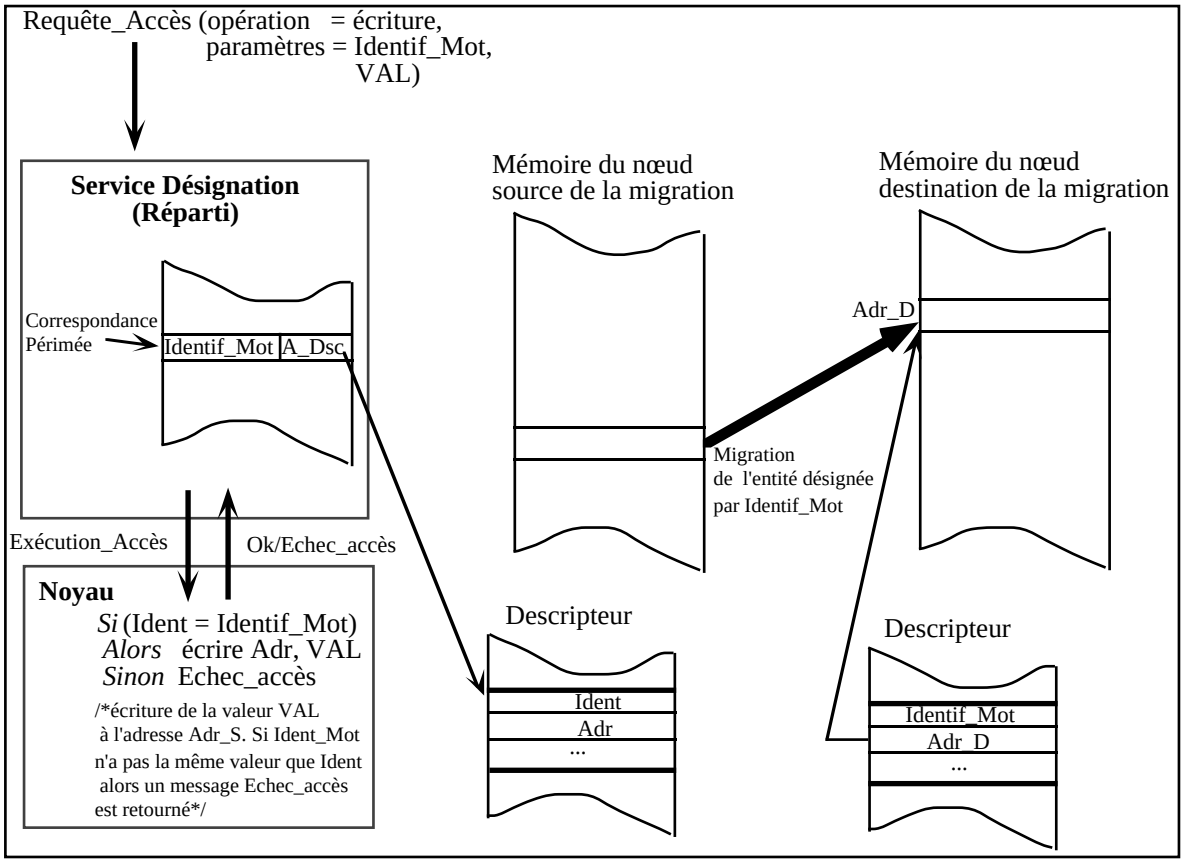


Figure 4 : Accès à travers un descripteur

L'inconvénient de la solution proposée est qu'elle n'anticipe pas la mise à jour des tables de correspondance. Cependant, elle évite d'encombrer le réseau par des requêtes de mise à jour, dont un certain nombre sont inutiles si aucune liaison n'est effectuée à partir de ces processeurs.

Par ailleurs, étant donné la granularité importante d'une task, la fréquence de migration d'une telle entité est faible. Par conséquent, le nombre de requêtes de liaison destinées au serveur du niveau supérieur ne constitue pas une charge énorme pour ce serveur. En effet, l'organisation hiérarchique des serveurs, décentralise le traitement de ces requêtes.

Enfin, notons que la consultation d'un descripteur de l'entité avant tout accès introduit un surcoût en temps. Néanmoins, ce descripteur est aussi nécessaire pour la technique de diffusion afin de résoudre les problèmes posés par les aléas (cf. §VI.4.1).

VI.5.5 Evaluation des délais de liaison

Nous avons mesuré le délai nécessaire pour reprendre un accès ayant échoué. Ce délai comprend le temps nécessaire pour effectuer la nouvelle opération de liaison et la mise à jour de la table de correspondance locale.

L'entité qui a été utilisée pour réaliser les mesures est un port privé à une Ptask. Ce port effectue une migration d'un processeur à un autre, de façon à ce qu'un autre processeur émettant vers ce port ait une adresse (du port) non à jour. La figure 5 trace le délai de laison en fonction de la distance séparant le processeur où est exécuté l'opération d'accès et le processeur où réside le serveur privé associé au port. Aucune charge supplémentaire n'est imposée ni sur les processeurs, ni sur le réseau de communication.

Lorsque le serveur privé se trouve sur le même processeur où est exécuté l'accès, le délai est faible (inférieur à 0.5 ms). Dans le cas où il se trouve sur un autre processeur, le délai est proportionnel à la distance séparant les 2 processeurs. Il risque donc d'être important par rapport à une opération de liaison exécutée localement. Néanmoins, il n'est infligé qu'au premier accès. Les accès suivants opèrent des liaisons traitées localement sans recourir au serveur privé.

Notons enfin que lors d'un accès à une entité, le délai rajouté par la consultation du descripteur de l'entité (pour le contrôle de la validité d'adresse) est négligeable par rapport au délai de l'opération de liaison.

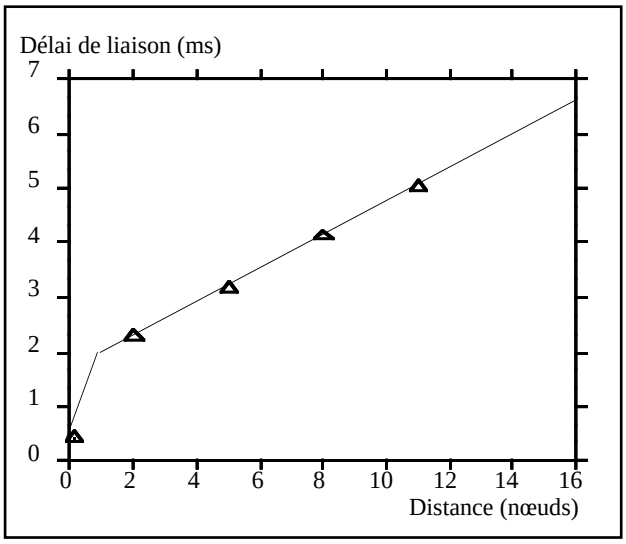


Figure 5 : Délai de liaison en fonction de la distance entre le processeur *local et le serveur privé - pour localiser une entité privée ayant migré -*

VI.6. Conclusion

Afin d'assurer la transparence de la migration, le service de désignation est un moyen permettant aux processus en exécution de s'abstraire de la localisation des entités manipulées. Dans ce chapitre nous avons étudié les moyens nécessaires pour que la désignation puisse supporter la migration de processus.

Deux volets ont fait l'objet de ce chapitre. Le premier concerne la structure des identificateurs, le second volet est relatif à la localisation des entités. Les solutions que nous avons adoptées ne conservent pas des dépendances résiduelles vis-à-vis d'un processeur source d'une migration.

Grâce à un choix approprié de la structure des identificateurs, nous avons évité de rajouter une couche supplémentaire d'interprétation de ces identificateurs. Pour cela, les identificateurs utilisés sont logiques et uniques dans leurs espaces de désignation.

S'agissant du second volet, nous avons proposé une technique de liaison qui profite de l'organisation hiérarchique des espaces de désignation. Cette technique nous permet d'éviter le recours à la diffusion de requêtes de mise à jour ou de localisation, tout en conservant une décentralisation du traitement des requêtes de localisation.

Chapitre VII

Répartition de Charge par Migration de Processus

Nous consacrons ce chapitre à un exemple d'application de la migration de processus qui concerne la répartition de charge dans un système distribué. La migration de processus est par ailleurs utilisée pour traiter d'autres problèmes tels que la tolérance aux pannes et l'accès à des ressources.

Dans le cas des systèmes massivement parallèles, le placement des processus sur les processeurs est devenu une tâche trop complexe pour être gérée par le programmeur. Il est donc nécessaire que le système puisse assurer cette tâche à travers des mécanismes de répartition de charge de façon à réduire les temps de réponse. Ces mécanismes doivent être efficaces dans la mesure où le principal intérêt des machines massivement parallèles est l'augmentation de la puissance de calcul. La gestion du placement de processus par le système a aussi pour avantage de permettre aux applications parallèles d'être moins dépendantes de l'architecture des machines.

La répartition de charge se fait tout d'abord lors du placement initial des processus. Elle peut aussi se faire de manière préemptive en faisant migrer des processus au cours de leurs exécution. Cette dernière approche semble être plus efficace puisqu'elle permet une meilleure adaptation de la charge suivant l'évolution dynamique de l'état du système. Cependant, le coût de migration conditionne le recours à une telle approche. Dans ce contexte nous proposons un algorithme qui améliore la qualité de la répartition de charge grâce à la migration de processus, l'objectif étant de réduire les temps de réponse.

Dans une première partie nous décrivons le cadre de travail. Nous présentons ensuite l'algorithme de répartition de charge. Une évaluation de l'algorithme est effectuée où nous mettons en évidence les conditions dans lesquelles la migration de processus peut améliorer les temps de réponse du système. Ce chapitre présente aussi une estimation des coûts de migration suivant les solutions retenues dans les chapitres III, IV, V et VI.

VII.1 Préliminaires

La répartition de charge consiste à allouer les ressources aux processus afin de pouvoir exécuter ces derniers. Même dans le cas des systèmes centralisés, le problème d'allocation des ressources est assez complexe, il l'est davantage pour les systèmes distribués. En effet cette dispersion des ressources rend difficile la détermination de leur état à un instant donné. Ceci est dû principalement au fait que la collecte des informations concernant l'état des ressources ne peut se faire de manière atomique (étant donné les délais de communication). De plus, dans le cas des systèmes massivement parallèles où le nombre de processeurs est important, le maintien d'un état complet du système de façon centralisée n'est plus envisageable.

Dans ce chapitre nous nous interessons en particulier à l'allocation des processeurs. Cette allocation se fait tout d'abord lors du placement initial des processus sur les processeurs d'exécution. Ce placement peut être *statique* c'est à dire calculé à l'avance sans prendre en compte l'état du système, ou au contraire *dynamique* en tenant compte de l'état de charge du système lors de l'exécution. En outre, l'allocation des processeurs peut être ajustée en modifiant le placement de certains processus de manière *préemptive* (en les faisant migrer).

L'objectif que nous nous fixons à travers la répartition de charge est la réduction du temps de réponse du système en déplaçant certains processus qui s'exécutent sur des processeurs trop chargés vers des processeurs qui le sont moins. Pour cela, une première approche serait d'*équilibrer la charge* entre les différents processeurs. Ceci permet d'une part d'éviter qu'il y ait des processeurs sous utilisés et d'autre part d'assurer une équité entre les processus. Une seconde approche, appelée *partage de charge*, évite que certains processeurs dépassent un seuil de charge alors que d'autres ont une faible charge.

L'équilibrage de charge n'implique pas systématiquement une amélioration de la moyenne du temps de réponse des processus mais une meilleure équité entre ces processus. Les temps de réponse sont améliorés en évitant que des processeurs restent oisifs. De plus, dans le cas où il existe des synchronisations entre les processus, l'équilibrage de charge améliore les temps de réponse en réduisant les délais de synchronisation. En effet, en l'absence d'équilibrage de charge, certains processus qui s'exécutent sur des processeurs faiblement chargés sont bloqués en attente d'autres processus, dont l'exécution est pénalisée par une forte charge sur leurs processeurs. Ce problème est d'autant plus critique pour les machines sans mémoire commune où l'interaction entre processus se fait par des communications synchrones.

Un aperçu plus détaillé sur les différentes techniques de répartition de charge existe dans [BSS91] [CaK88] [SKS92]. Ces travaux font une classification et une comparaison des approches et des techniques existantes.

VII.2 Placement des processus

Le présent travail est une continuation des travaux déjà entamés dans notre équipe par Talbi pour le développement d'un outil de placement de processus dans le noyau ParX [Tal91] [Tal93]. Afin d'équilibrer la charge, l'algorithme d'allocation ayant été proposé se base uniquement sur le placement initial des processus. Etant donné la complexité et le coût de la migration de processus, celle-ci n'a pas été utilisée dans l'algorithme, les processus ne sont transférés que s'ils n'ont pas déjà commencés leurs exécutions.

Dans cette section, nous poursuivons la description de l'algorithme de placement dynamique des processus. Une amélioration de cet algorithme est proposée dans la section suivante. Les solutions relatives à un certain nombre de problèmes fondamentaux concernant la répartition de charge - qui ont été traités dans la thèse de Talbi [Tal93] - sont conservées les mêmes dans le nouveau algorithme. Ces solutions se rapportent à l'estimation de la charge, le maintient de l'état du système et le placement initial des processus.

VII.2.1 Description de l'outil

Une fois qu'un sous réseau de processeurs (i.e., *cluster*) a été allouée à une application (i.e., Ptask), le placement des tâches composant la Ptask sur le *cluster* est déterminé grâce à un module d'allocation statique utilisant un algorithme génétique parallèle [MuT91]. A la suite de ce placement initial la charge est supposée équilibrée entre les différents processeurs. Cependant puisque la durée d'exécution peut différer d'une tâche à une autre, un déséquilibre de charge risque de se produire, d'autant plus, s'il y a création dynamique de nouvelles tâches placées sur les processeurs où elles sont créées. Pour remédier à ce déséquilibre, un algorithme de placement dynamique des nouvelles tâches est utilisé. Cet algorithme se décompose en deux éléments : un élément d'information et un élément de contrôle. Le rôle de l'élément d'information est d'évaluer et de maintenir l'état de charge du système. Quant à l'élément de contrôle, il a pour charge de mettre en œuvre une politique de répartition de charge. Ces deux éléments sont distribués de manière à ce que tous les processeurs ont un même rôle au niveau de la répartition de la charge, aucune gestion centralisée n'est opérée. La figure 1 décrit les deux niveaux de placement statique et dynamique.

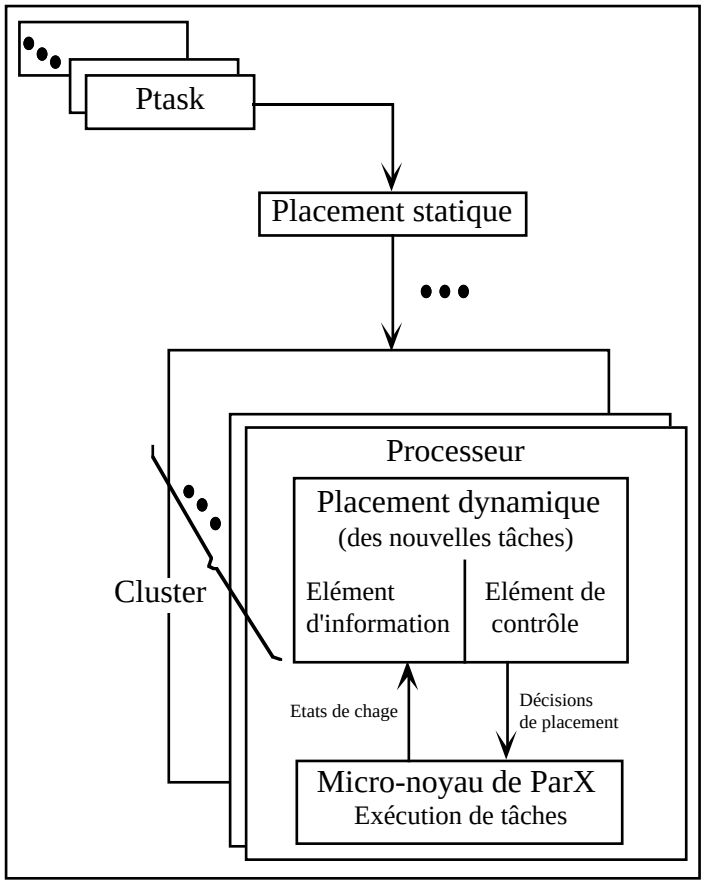


Figure 1 : Mécanismes de placement des processus

VII.2.1.1 Elément d'information

L'évaluation de la charge nécessite le choix d'un ou plusieurs indicateurs de charge. La difficulté étant de trouver un indicateur qui soit à la fois fortement corrélé avec l'utilisation du processeur et que l'on peut évaluer avec un coût réduit. L'indicateur qui a été ainsi choisi dans [Tal93] est le nombre de tâches en exécution. Afin d'éviter le problème de fluctuations rapides de l'état d'un processeur, trois états sont définis. Un processeur à l'état "Petite_charge" indique qu'il peut accepter l'exécution des tâches créées sur d'autres processeurs. Un processeur à l'état "Grande_charge" indique que les tâches créées doivent être transférées vers d'autres processeurs. Enfin, l'état "Charge_normale" indique qu'aucun effort n'est utile pour le transfert des tâches créées. Deux paramètres système sont utilisés : les seuils de charge T_{min} et T_{max} Ces paramètres fixent la marge de chacun des trois états comme le montre la figure 2.

Chaque processeur maintient sa charge locale et la valeur de la charge des processeurs voisins. Ainsi, lorsqu'une transition de l'état d'un processeur se produit, celui-ci diffuse à tous ses voisins son nouvel état. Notons enfin que la restriction de l'état de charge à trois états permet de réduire le flot de communication entre les processeurs voisins pour la mise à jour des états.

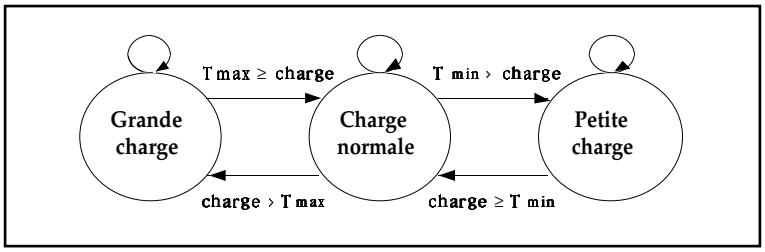


Figure 2 : Transitions d'états de la charge d'un processeur

VII.2.1.2 Elément de contrôle

A la création d'un processus, l'élément de contrôle détermine le processeur sur lequel va s'exécuter le processus. La politique de placement est décrite par l'algorithme suivant :

si (EtatLocal <> Grande_charge) alors
Exécuter localement la tâche
sinon si (∃ voisin/ Etat(voisin)=Petite_charge) alors
Exécuter la tâche sur le voisin
sinon
Saturation locale
finsi
finsi

La saturation locale correspond au cas où aucun des voisins n'est à l'état "Petite_charge". Dans cette situation, un des processeurs voisins est choisi aléatoirement ; une priorité est donnée aux processeurs qui sont à l'état "Charge_ normale". En effet, la probabilité de trouver un processeur en petite charge au voisinage d'un processeur en charge normale est plus grande que celle de le trouver au voisinage d'un processeur en grande charge. Afin d'éviter le problème d'instabilité de l'algorithme où la tâche circule dans le réseau sans jamais être exécutée, une distance maximale D_{max} est définie et que la tâche ne peut pas dépasser. Cette distance permet aussi de limiter le coût du transfert de la tâche qui croît avec la distance. Ainsi l'algorithme exécuté en cas de saturation est le suivant :

si (DistanceParcourue= D_{max}) ou (Etat=Petite_charge) alors
Exécuter la tâche
sinon
Choisir un nouveau voisin
finsi

Notons enfin que sur chacun des processeurs, la complexité de l'algorithme de placement est de l'ordre du nombre de voisins (elle donc indépendante de la taille du réseau).

VII.2.2 Critiques

L'outil de placement de tâches ainsi décrit est à la base de la répartition de charge. Il permet, grâce à l'algorithme de placement statique, de déterminer le placement initial d'une application parallèle sur un ensemble de processeurs alloués à l'application. Ce placement tend à équilibrer la charge sur toute la partition grâce une connaissance globale de l'application. L'algorithme de placement dynamique est utilisé pour réduire le déséquilibre de charge qui se crée au cours de l'exécution. Contrairement au placement statique la décision de placement dynamique est prise suivant une connaissance partielle de l'état du système. En effet, chaque processeur décide selon son état et l'état de ses voisins. De plus, un processus nouvellement créé ne peut être placé qu'à une distance inférieure ou égale à D_{max}, ce qui réduit l'espace de recherche. En fait, cette propriété de localité, sur la vision de l'état du système et sur l'espace de recherche, s'impose à cause du caractère d'extensibilité que tout mécanisme système doit respecter. Nous avons ainsi gardé le même élément d'information pour l'algorithme décrit dans la section suivante.

Par ailleurs, l'algorithme de placement dynamique s'adapte mal à la situation où on a une grande variabilité des durées d'inter-arrivées et d'exécution des tâches. En effet, d'une part, une grande variabilité de la durée d'exécution des tâches induit forcement, après un certain temps, un déséquilibre de charge à la suite du placement initial. D'autre part, la variabilité de la durée d'inter-arrivées des tâches ne permet pas d'effectuer régulièrement un placement dynamique des processus pour rééquilibrer la charge. Des résultats de simulation, décrits plus loin, aboutissent à cette même constatation.

En outre, en se limitant à un placement initial des processus, la dispersion de la charge est pénalisée par le fait qu'un processus - une fois placé - ne peut plus être transféré ailleurs. Etant donné que l'information utilisée pour la prise de décision se réduit à un certain voisinage, dans lequel le placement est effectué, cette pénalisation est rendue encore plus importante.

Pour remédier à ces problèmes, nous proposons d'utiliser la migration de processus. Celle-ci permet, d'une part, de ne pas limiter la dispersion de la charge à un certain voisinage, et d'autre part, de redistribuer la charge lorsque le placement des processus nouvellement créés ne compense pas le déséquilibre de charge qui existe.

VII.3 Nouvel algorithme de répartition de charge

VII.3.1 Description

L'algorithme de répartition de charge que nous proposons est une amélioration de l'algorithme de placement décrit ci-dessus[1]. L'objectif du nouveau algorithme est de remédier à la situation où l'algorithme de placement dynamique des tâches est incapable d'éviter qu'il y ait deux voisins respectivement en grande charge et en petite charge. Par exemple, cette situation se produit lorsque les tâches nouvellement créées sont en nombre insuffisant, ou qu'elles ne peuvent atteindre un processeur en petite charge (à cause de la propriété de localité de l'algorithme). Afin de résoudre ce problème, nous proposons d'utiliser la migration de processus qui nous permet de transférer des tâches même si elles ont déjà entamé leurs exécution. Etant donné que la migration de tâches est plus coûteuse que le placement, nous gardons aussi l'algorithme de placement dynamique des tâches nouvellement créées de façon à réduire le recours à la migration de processus (figure 3). Cette structuration du mécanisme de répartition de charge nous permet ainsi de bénéficier des avantages des deux techniques : répartition de charge par placement et par migration.

A la différence de l'algorithme de placement dynamique, c'est le processeur à l'état "Petite_charge" qui prend l'initiative de chercher un autre processeur à l'état "Grande_charge". Ceci a pour avantage de ne pas pénaliser davantage un processeur en "Grande_charge" par une telle recherche. Une fois ce couple établi, une ou plusieurs tâches sont faites migrer du processeur en "Grande_charge" vers le processeur en "Petite_charge". La recherche se fait parmi les processeurs distants d'au plus D_{max}. Afin de favoriser les processeurs les plus proches, l'algorithme de recherche est en largeur d'abord. Ceci a pour avantage de réduire le coût de la migration (étant donné que le temps de transfert augmente avec la distance entre la source et la destination).

Alors que l'algorithme de placement dynamique est activé à la création d'une tâche, l'algorithme de migration est activé à la terminaison d'une tâche. En effet seul cet événement fait passer un processeur à un état "Petite_charge". L'algorithme exécuté est le suivant :

[1] Décrit par ailleurs dans l'article [EKM94].

si (EtatLocal =Petite_charge) alors
Trouvé <- Chercher le plus proche Processeur en Grande_charge à une distance inférieure à D_{max}
si (Trouvé) alors
faire migrer un (ou plusieurs) processus du Processeur en Grande_charge vers le processeur local
finsi
finsi

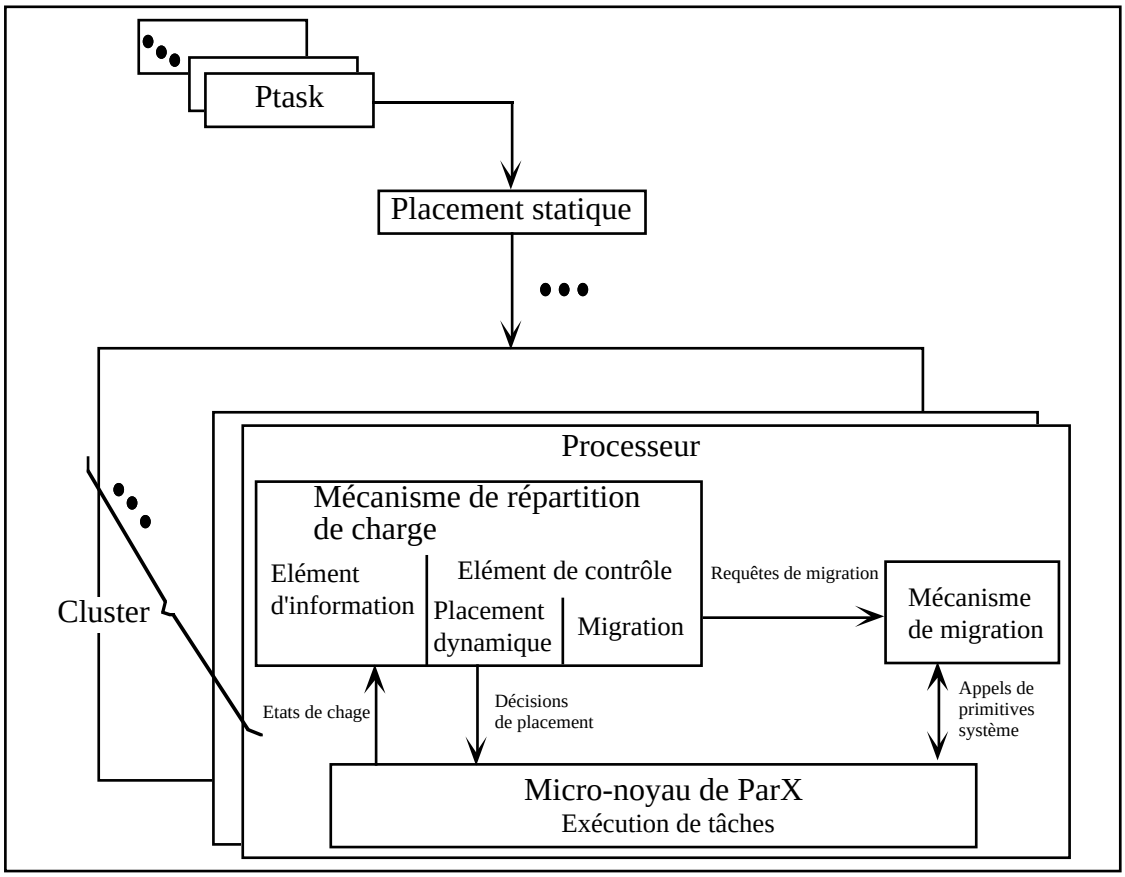


Figure 3 : Mécanismes de répartition de la charge

La phase de recherche d'un processeur en "Grande_charge" met en œuvre le protocole de négociation suivant. Le processeur en "Petite_charge" envoie une requête de migration au processeur en "Grande_charge" (ayant été déjà sélectionné) ; ce message indique que l'émetteur de la requête est en "Petite_charge" et qu'il demande de lui faire migrer des tâches. Après avoir reçu ce message, le processeur en "Grande_charge" peut soit acquitter la requête ou la rejeter. Un rejet est envoyé notamment lorsque le processeur n'est plus en haute charge ou qu'il n'existe pas une tâche que l'on peut faire migrer. Durant la phase de négociation, le processeur en "Petite_charge" réserve la charge proposée. Dans le cas où la requête de migration est rejetée, le processus de recherche se poursuit avec les autres voisins.

Par ailleurs, le temps de migration d'une tâche ne doit pas être supérieur au temps nécessaire pour achever l'exécution de la tâche sur le processeur courant. Seuls les tâches ayant des durées d'exécution importantes peuvent permettre un gain substantiel en les faisant migrer [Cab86]. Pour pouvoir prendre une décision de migration d'une tâche, il est donc utile de pouvoir estimer la durée de migration (cf. §VII.3.2) ainsi que la durée de vie restante de cette tâche. Afin d'estimer la durée d'exécution d'un processus, Bryant et Finkel ont envisagé trois approches différentes [BrF81] :

(1) tous les processus ont une durée de vie équivalente,
(2) la durée de vie restante est estimée par sa durée de vie au moment de l'estimation, et
(3) la durée de vie d'un processus est supposée connue.

Sans entrer dans les détails de ces approches, le choix des processus à faire migrer représente en lui même un problème non trivial. Dans ce travail nous nous limitons à un choix aléatoire des processus à faire migrer. Un module de filtrage des processus candidats à la migration est envisagé dans une perspective avenir.

VII.3.2 Analyse des coûts de migration

Les coûts induits par la migration d'une tâche sont répartis dans l'espace - i.e., l'ensemble des entités affectées par la migration - et dans le temps, comme le montre la figure 4. Suivant cette classification, nous analysons, dans ce qui suit, ces différents coûts dans le cas spécifique des solutions retenues lors de l'élaboration du mécanisme de migration.

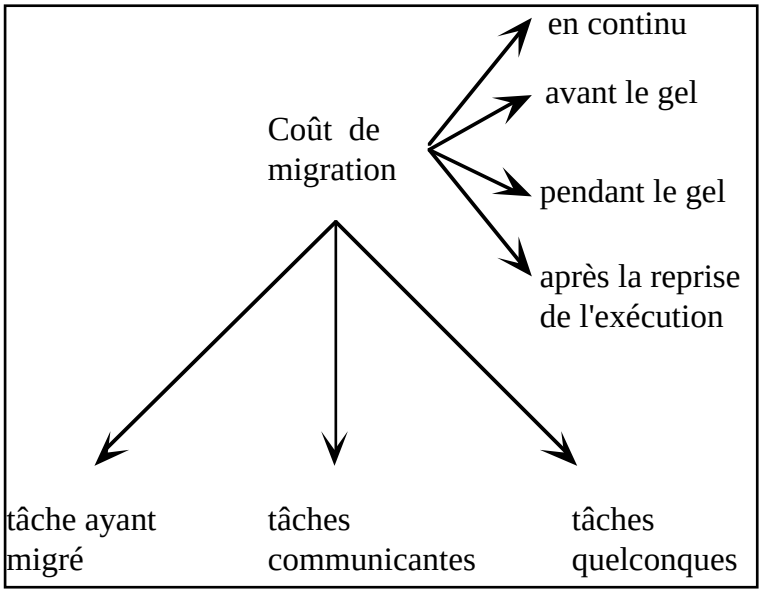


Figure 4 : Répartition du coût de la migration

L'objectif de cette analyse est de déterminer un moyen d'estimation du coût de migration. Cette estimation doit pouvoir se faire avec des temps de calcul faibles. Les coûts relatifs à la prise de décision, comprenant le maintien des informations de l'état de charge et le contrôle de la charge, ne sont pas inclus dans cette présentation.

VII.3.2.1 En continu

Il s'agit des coûts introduits de façon continue et qui ne sont pas dus directement à la migration d'une tâche donnée. Ces coûts administratifs concerne d'abord (1) l'intégration des mécanismes nécessaires pour supporter la migration de tâches et ensuite (2) l'ensemble des actions entreprises par le système en prévision d'éventuelles migrations de tâches (gestion d'un historique de l'exécution des tâches).

Dans le micro-noyau de ParX, nous avons eu principalement à intégrer des protocoles de migration pour gérer correctement les communications, le délai introduit reste négligeable par rapport aux délais de communication (cf. §V.4.5). En outre, au niveau du service de désignation et lors d'un accès à une entité, un test supplémentaire de validité de l'adresse d'accès est effectué (cf. §VI.5.4), le délai correspondant à ce test est négligeable par rapport au délai de liaison qui fournit l'adresse d'accès (cf. §VI.5.5).

S'agissant des actions entreprises en prévision d'éventuelles migrations de tâches (2), notamment pour assurer les conditions de cohérence de la migration, les solutions que nous avons retenues ne recourent pas à de telles actions (cf. §III.5.3).

En conclusion les coûts administratifs sont négligeables, ce qui est d'ailleurs l'un des critères principaux que le mécanisme de migration doit respecter.

VII.3.2.2 Avant le gel

Avant de suspendre une tâche à faire migrer et afin de réduire le temps de gel, nous avons prévu de précopier la partie de l'image (de la tâche) accédée uniquement en lecture (cf. §IV.9.1). Cette opération introduit une surcharge de calcul et de communication proportionnelle à la taille pré-copiée Tl. Ce coût pénalise l'ensemble des processus qui partagent les ressources mis en œuvre par l'opération de pré-copie. Nous représentons ce coût par le délai de pré-copie D_{pc} :

$D_{pc} = B_p * Tl$ où

B_p est la bande passante lors du transfert ; notons que cette bande passante diffère selon le couple de processeurs source / destination (cf. §IV.9.2).

VII.3.2.3 durant le gel

Durant le gel d'une tâche, l'exécution de celle-ci est retardée. La durée de gel D_g pénalise ainsi la tâche à faire migrer mais aussi les autres processus en exécution. En effet, en plus de la charge introduite dans le système (pour transferer de l'image), les tâches qui ont des points de synchronisation avec la tâche en migration vont devoir attendre l'accomplissement de la migration.

La durée de gel dépend principalement de la taille Tl_e de la partie de l'image transférée pendant le gel. Celle-ci correspond aux zones mémoire accédées en lecture / écriture. Nous l'estimons par :

$$D_g = B_p * T_e$$

Remarque :
Nous négligons les coûts d'extraction et de rétablissement de l'image d'une tâche.

VII.3.2.4 Après la reprise de l'exécution

Les opérations de rétablissement des communications et des accès aux entités se font pendant et à la suite de la reprise de l'exécution. Ces opérations introduisent un délai qui se rajoute au niveau des communications et des accès. Cette pénalisation concerne tout au plus les premiers accès ou envois de message (cf. §V.3, §VI.5.4). Suivant les mesures faites des délais de rétablissement des communications et des accès (cf. §V.4.5, §VI.5.5), celles-ci sont de l'ordre de la milliseconde. Ce délai est dû principalement au fait que la vitesse d'envoi, des messages courts, est faible. Une estimation des délais induits par ces opérations est difficile car les traitements de différents accès et communications peuvent se faire en parallèle, de plus, leurs déclenchement dépend de l'exécution.

Néanmoins, lorsque le temps nécessaire pour achever l'exécution de la tâche est important, on peut négliger les délais rajoutés pour le rétablissement des accès et des communications. Autrement, si l'exécution de la tâche est proche de sa fin, le coût de migration de la tâche sera de toute façon pénalisant.

La surcharge du système induite par les opérations de rétablissement des accès et des communications est négligeable. En effet, d'une part les traitements effectués sont simples et d'autre part la taille des messages échangés est réduite.

VII.3.2.5 Une estimation du coût de migration

Si l'on s'interesse à la surcharge introduite dans le système par une opération de migration d'une tâche, le transfert de l'image représente le principal coût. La durée de transfert D_t est estimée par :

$$D_t = D_{pc} + D_g = B_p * (T_l + T_e)$$

En ce qui concerne le retard introduit au niveau de l'exécution de la tâche ayant migré, celui-ci comprend le délai de gel D_g et le délai nécessaire pour le rétablissement des communications et des accès aux entités. Ce dernier délai est négligé puisqu'on les processus sujets à la migration doivent avoir une durée d'exécution importante.

VII.4 Evaluation de l'algorithme de répartition de charge

L'analyse des algorithmes d'équilibrage dynamique de charge est assez complexe, nous avons choisi d'évaluer les algorithmes proposés par des simulations basées sur les techniques de réseaux de files d'attente. Les simulations sont faites en utilisant l'outil QNAP "Queueing Network Analysis Package" [Sim86].

VII.4.1 Modélisation

Nous avons considéré un réseau de processeurs (i.e., *cluster*) dont la topologie est une grille torique 4x4. Chaque processeur a ainsi 4 voisins. Un processeur est modélisé par un serveur unique et une file où sont gardées les tâches en attente d'exécution. L'ordonnancement des tâches est de type processeur partagé. Une source d'arrivée de tâches est associée à chaque processeur. La modélisation d'un processeur est décrite par la figure 5, où μ_e désigne le temps moyen d'exécution d'une tâche (ou encore dit temps moyen de service), μ_p désigne le temps moyen pour effectuer un placement d'une nouvelle tâche sur un autre processeur, alors que μ_m désigne le temps moyen nécessaire pour faire migrer une tâche. Les temps de placement et de migration sont simulés sur les processeurs source et destination lors du transfert. Le taux de création (ou encore d'arrivée) de tâches est désigné par λ. Les valeurs de ces paramètres (i.e., λ, μ_e, μ_p, μ_m) sont choisis les mêmes pour tous les processeurs.

Etant donné qu'un processeur est partagé entre les tâches en exécution et les tâches assurant la répartition de charge, la simulation des opérations de répartition de charge induit à la fois (1) le coût pénalisant la tâche en migration et (2) la surcharge des ressources mis en œuvre (une

seule tâche s'exécute à la fois). Nous nous restreignons à simuler l'élément de contrôle du mécanisme de répartition de charge. Nous négligeons la surcharge induite par l'élément d'information. Reste à déterminer la loi de distribution régissant les temps de placement et de migration au niveau de l'élément de contrôle. Pour une tâche donnée, ces temps sont a priori indépendants de la durée d'exécution de la tâche. Nous utiliserons une distribution exponentielle pour le temps de placement et le temps de migration, dont les moyennes sont respectivement : μ_p, μ_m.

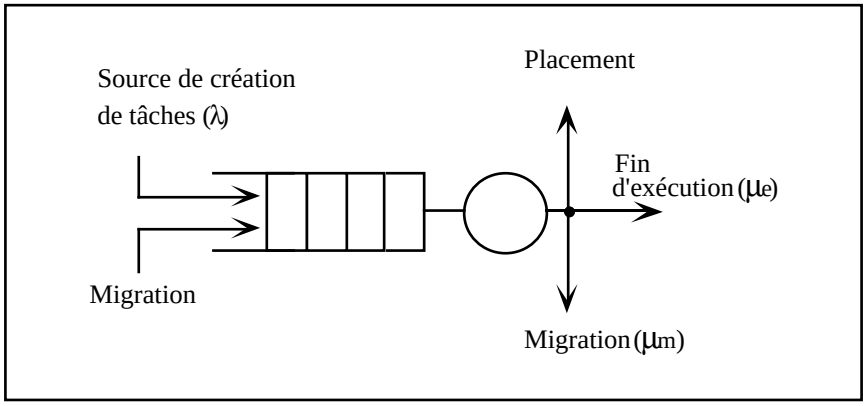


Figure 5 : Modélisation d'un processeur

Pour des raisons de simplification, les communications entre tâches ne sont pas pris en compte explicitement dans ce modèle de simulation. En conséquence, lors de la migration d'une tâche, l'attente de l'achèvement de la migration par d'autres tâches (voulons communiquer avec celle en migration) n'est pas considérée. Notons néanmoins que les temps d'attentes entre tâches communicantes sont différentes selon que les charges sur les processeurs d'exécution sont équilibrées ou non. En effet, l'équilibrage de charge permet de réduire ces temps en évitant que des tâches, s'exécutant sur des processeurs en "Petite_charge", soient en attente de tâches pénalisées par une forte charge sur leurs processeurs d'exécution. Ainsi l'amélioration des temps de réponse, en utilisant l'équilibrage de charge, est plus importante si l'on tenait compte des communications.

VII.4.2 Etudes des algorithmes suivant la variance des lois d'arrivée et de service

L'équilibrage de charge est altéré lorsque les temps d'exécution des tâches sont variables. Pour remédier à cette situation, la première stratégie adoptée est de placer les tâches nouvellement créées sur les processeurs en "Petite_charge". Si le taux de création des tâches est faible, le placement dynamique s'avère inefficace. C'est notamment dans cette situation où la stratégie de migration serait avantageuse. Il est donc intéressant d'étudier le comportement de

l'algorithme en fonction de la variance de la durée de service et celle d'inter-arrivée des tâches. Nous caractérisons la variance grâce au coefficient carré de la variance (il représente le rapport de la variance sur le carré de la moyenne). Celui de la loi de création des tâches est noté par CCV_a et celui de la loi de service par CCV_s. Les valeurs de ces coefficients seront les mêmes pour tous les processeurs du *cluster*. Quatre types de loi de distribution sont utilisés :
(1) une loi constante pour un coefficient carré de la variance nul,
(2) une loi d'erlang pour des coefficients carré de la variance entre 0 et 1,
(3) une loi exponentielle pour un coefficient carré de la variance égale à 1, et
(4) une loi hyperexponentielle pour des coefficient carré de la variance supérieurs à 1.

Le rapport temps moyen d'exécution sur temps moyen entre deux créations de tâches est pris inférieur à un ($\lambda * \mu_e = 0.8$). La moyenne du temps de placement μ_p (respectivement de migration μ_m) est fixée à 1% (respectivement à 1.5%) du temps moyen d'exécution μ_e. Les coûts de placement et de migration ainsi choisis sont assez proches ; le comportement de l'algorithme de répartition de charge en fonction de ces coûts est considéré dans une section ultérieure (VII.4.4).

Nous évaluons la qualité de la répartition de charge par la mesure des paramètres suivants :

- *W : le temps moyen d'attente sur un processeur*. Ce paramètre traduit le coût dû au partage du *cluster* par l'ensemble des tâches. La répartition de la charge doit réduire ce coût en transférant des tâches des processeurs en "Grande_charge" vers les processeurs libres.

$$W = \left(\sum_{1 \le i \le n_p} W_i \right) / n_p \qquad \text{où}$$

W_i : temps moyen d'attente sur le processeur *i*,
n_p : nombre de processeurs.

- *VW : la variance, entre les processeurs, du temps moyen d'attente*. Ce paramètre est réduit par l'équilibrage de la charge.

$$VW = \left(\left(\sum_{1 \le i \le n_p} W_i^2 \right) / n_p \right) - W^2$$

- *N : le nombre moyen de tâches sur un processeur*. Ce paramètre traduit la charge moyenne sur un processeur.

$$N = \left(\sum_{1 \le i \le n_p} N_i \right) / n_p \qquad \text{où}$$

N_i : nombre moyen de tâches sur le processeur *i*.

- VN : la variance, entre les processeurs, du nombre moyen de tâches.

$$VN = \left(\left(\sum_{1 \le i \le n_p} N_i^2\right) / n_p\right) - N^2$$

CCV_a	CCV_s	Pas de répartition de charge		Placement Dynamique		Placement Dynamique Migration	
		W	*VW*	*W*	*VW*	*W*	*VW*
0	0	0	0	0	0	0	0
0.05	0.05	1.97	0.00	2.02	0.00	2.05	0.00
0.10	0.10	5.01	0.01	4.78	0.00	4.07	0.00
0.25	0.25	13.80	0.24	9.23	0.02	4.94	0.00
1	1	42.14	16.45	13.28	0.01	4.95	0.00
10	10	86.50	315.62	13.61	0.05	5.44	0.00
20	20	97.73	772.8	13.68	0.28	5.81	0.01
0	10	38.08	41.80	13.56	0.25	6.18	0.00
10	0	355.5	2074	24.06	0.64	9.14	0.07

Table 1. Temps moyen d'attente W

($1/\lambda$=12.5, μ_e=10, $\mu_p=\mu_e/100$, $\mu_m=1.5\mu_e/100$, T_{min}=2, T_{max}=3, D_{max}=1)

La table 1 donne les valeurs de *W* et celles de *VW* obtenues pour différentes simulations où les paramètres CCV_a et CCV_s sont faits variés. Pour de très faibles valeurs de CCV_a et CCV_s (inférieures à 0.10), les temps d'attente sont sensiblement les mêmes que l'on applique ou non une répartition dynamique de la charge. Ce résultat est prévisible puisque les différents processeurs suivent les mêmes lois d'arrivée et de service, et que les variances des temps d'inter-arrivée des tâches et de service sont très faibles par rapport à leurs moyennes. Néanmoins, dans le cas de la répartition de la charge non préemptive, la valeur de *W* croit rapidement pour des valeurs de CCV_a et CCV_s assez faibles (CCV_a=CCV_e=0.25). En revanche, l'utilisation de la migration permet une amélioration de la valeur de *W* : elle est réduite pratiquement de moitié comparé à sa valeur obtenue par l'algorithme de placement dynamique uniquement. Par ailleurs, la variance *VW* reste pratiquement nulle lorsque la répartition de charge - avec ou sans migration - est effectuée. Notons aussi que la migration de processus permet de réduire la valeur de *W* même si un seul des deux coefficients CCV_a et CCV_s est non nul (CCV_a=0 et CCV_s=10, CCV_a=10 et CCV_s=0). Enfin, la table 1 montre que le placement dynamique sans migration de processus permet de réduire en grande partie la valeur de *W*, ce qui justifie notre recours au placement dynamique lors de l'élaboration du mécanisme de répartition de charge.

Une telle comparaison entre les algorithmes de partage de charge avec et sans migration de processus a été effectuée par Eager, Lazowska et Zahorjan [ELZ88]. Les résultats de leurs travaux restreignent l'amélioration des temps de réponse, en utilisant la migration de processus, au cas où les variances à la fois des lois d'arrivée et de service sont importantes. Ces ré-

sultats sont différents des notre car ils ne s'appliquent que sous certaines hypothèses. En particulier, l'état de tous les processeurs est supposé connu lors d'une prise de décision, ce qui n'est pas possible dans le cas des systèmes massivement parallèles. Par ailleurs, la comparaison faite concerne les algorithmes de partage de charge dont l'objectif est d'éviter qu'il y ait en même temps un processeur libre et un autre sur lequel s'exécutent plus d'un processus. En conséquence, lorsque les processus peuvent être amenés à se synchroniser, hypothèse non prise en compte dans [ELZ88], les résultats énoncé dans [ELZ88] ne sont plus valables. En effet, même les processus s'exécutant sur des processeurs faiblement chargés peuvent être retardés en attente d'autres processus s'exécutant sur des processeurs trop chargés.

CCV_a	CCV_s	Pas de répartition de charge		Placement Dynamique		Placement Dynamique Migration	
		N	***VN***	***N***	***VN***	***N***	***VN***
0	0	0.80	0	0.80	0	0.80	0.00
0.05	0.05	0.96	0.00	0.97	0.00	0.97	0.00
0.10	0.10	1.20	0.00	1.19	0.00	1.23	0.00
0.25	0.25	1.90	0.00	1.58	0.00	1.60	0.00
1	1	4.16	0.12	2.05	0.00	1.99	0.00
10	10	7.70	2.15	2.32	0.00	2.09	0.00
20	20	8.61	4.31	2.32	0.01	2.14	0.00
0	10	3.87	0.28	2.02	0.00	1.67	0.00
10	0	28.78	14.26	3.83	0.01	4.08	0.01

Table 2. Nombre moyen de tâches
($1/\lambda$=12.5, μ_e=10, $\mu_p=\mu_e/100$, $\mu_m=1.5\mu_e/100$, T_{min}= 2, T_{max}=3, D_{max}=1)

La table 2 reporte les valeurs du nombre moyen de tâches par processeur (*N*) en fonction de CCV_a et CCV_S. En appliquant l'un des deux algorithmes de répartition de charge, avec ou sans migration, nous remarquons que la valeur de *N* correspond à une "Charge_normale" (T_{min}=2, T_{max}=3) et ceci dans la plus part des cas. De plus, la variance *VN* devient pratiquement nulle grâce à la répartition de charge. Notons enfin que le paramètre *N* ne permet pas de faire ressortir une différence notable entre la répartition de charge avec et sans migration.

VII.4.3 Etude des algorithmes suivant la charge imposée

Les mesures précédentes ont été obtenues pour des moyennes du temps d'inter-arrivée, ainsi que de service, fixées ($1/\lambda$= 12.5 et μ_e=10). Ces paramètres déterminent la charge imposée au système. Nous nous interessons dans cette section à faire varier le rapport temps moyen de service sur temps moyen d'inter-arrivée, noté *U*. Ce rapport représente l'utilisation des proces-

seurs pour les besoins d'exécution des tâches et ne comprend pas l'utilisation des processeurs pour la répartition de charge.

$U = \lambda * \mu_e$

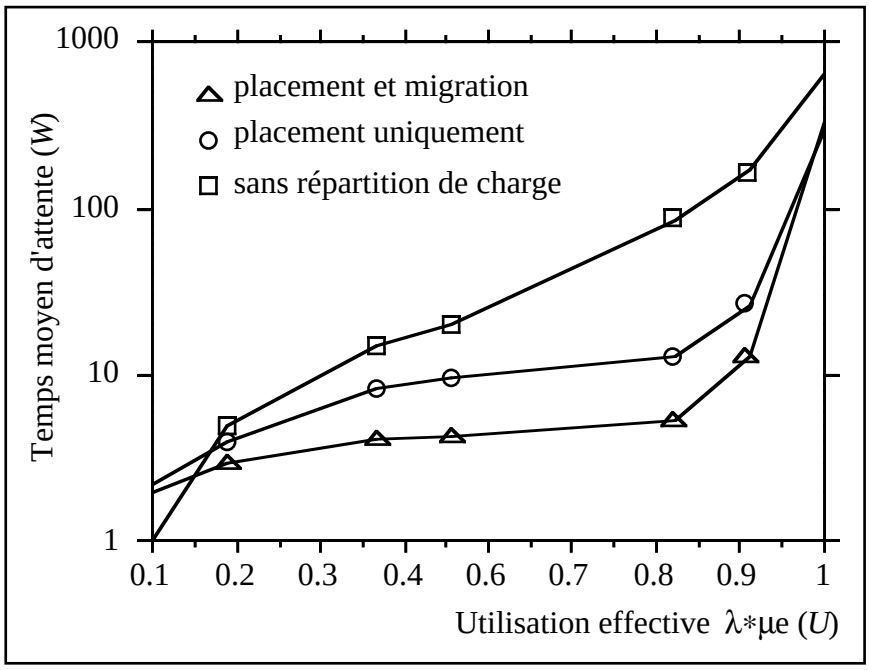


Figure 6 : Temps moyen d'attente W en fonction de l'utilisation U
(μ_e=10, $\mu_p=\mu_e/100$, $\mu_m=1.5\mu_e/100$, T_{min}=2, T_{max}=3, D_{max}=1, $CCV_a=CCV_s$=10)

La figure 6 trace l'évolution de *W* en fonction de *U*. Pour de très faibles valeurs de *U* (inférieur à 0.1), les algorithmes de répartition de charge n'apportent pas un gain substantiel. Cependant, lorsqu'aucune politique de répartition de charge n'est utilisée, la valeur de *W* tend à croître rapidement avec celle de *U*. En appliquant l'algorithme de placement dynamique des tâches, la valeur de *W* est réduite considérablement. Le recours, en plus, à la migration de processus permet de réduire encore de moitié les temps d'attente (par rapport aux temps obtenus par placement dynamique). Lorsque la valeur de *U* est très proches de 1, l'utilisation de la migration n'apporte pas un gain supplémentaire par rapport au placement dynamique des tâches. Ceci est dû au fait que le taux de création des tâches est suffisamment important pour que l'algorithme de placement dynamique à lui seul peut maintenir les processeurs en "Charge_normale" ou en "Grande_charge".

VII.4.4 Influence du coût de migration

Comparé à un placement dynamique d'un processus, la migration du processus présente des coûts supplémentaires qui sont :

- le coût de transfert du contenu des zones mémoires initialisées à l'exécution et des structures système pour la gestion de l'exécution du processus,
- le coût correspondant aux différents traitements et mises à jour nécessaires pour le rétablissement des communications et des accès altérés par la migration.

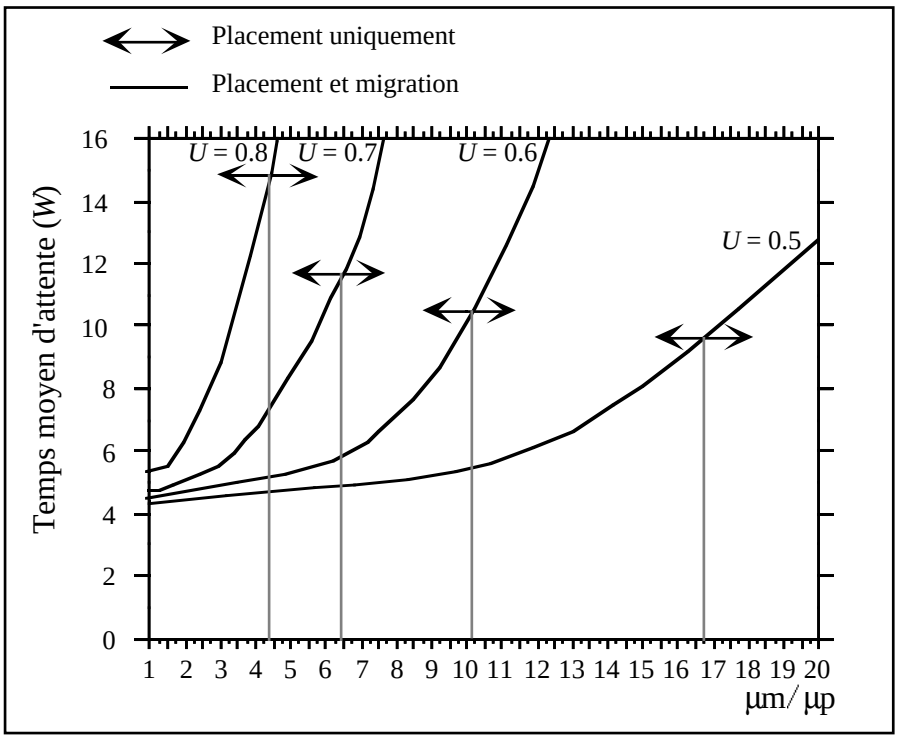


Figure 7 : Temps moyen d'attente W en fonction de μ_m/μ_p
(μ_e=10, $\mu_p=\mu_e/100$,T_{min}=2 ,T_{max}=3, D_{max}=1, CCV_a=CCV_s=10)

Dans cette section nous nous intéressons au comportement de l'algorithme de répartition de charge en fonction du coût moyen de migration μ_m. Dans une première série de mesures, nous faisons varier le coût de migration μ_m de façon à déterminer le seuil au delà duquel le recours à la migration est non viable. Pour différentes valeurs du facteur d'utilisation *U*, la figure 7 trace le temps moyen d'attente *W* en fonction de μ_m/μ_p avec μ_p égale 0.1. Nous constatons à travers cette figure que pour un taux d'utilisation moyen (*U*=0.5), la migration de processus ne devient non viable que pour un coût de migration élevé : plus de 16 fois le coût du placement, soit 16% du temps moyen d'exécution des processus μ_e. En revanche, plus le taux d'utilisation *U* est important, plus le coût de migration est pénalisant : pour *U* égale à 0.8

le coût de migration ne doit pas dépasser 4 fois le coût du placement, soit 4% du temps moyen d'exécution des processus μ_e.

Tout en gardant un rapport constant entre le temps moyen de placement μ_p et le temps de migration moyen μ_m, la figure 8 trace l'évolution du temps moyen d'attente *W* en fonction du rapport temps moyen d'exécution sur temps moyen de placement d'un processus (μ_e/μ_p). Nous constatons à travers cette figure que lorsque les coûts de placement et de migration sont négligeables comparé au temps moyen d'exécution μ_e et même si le coût de migration est important par rapport à celui du placement, la migration de processus permet une réduction importante du temps moyen d'attente *W*.

La figure 8 montre qu'en utilisant le placement dynamique uniquement, le temps moyen d'attente *W* n'est pas sensible au coût du placement μ_p, sauf pour des valeurs importantes de ce coût qui dépassent $\mu_e/50$. Cependant, lorsque la migration de processus est utilisée, le coût de migration devient un paramètre déterminant. Comme le montrent les figures 7 et 8, le recours à la migration peut être pénalisant, d'où la nécessité de filtrer les processus sujets à la migration.

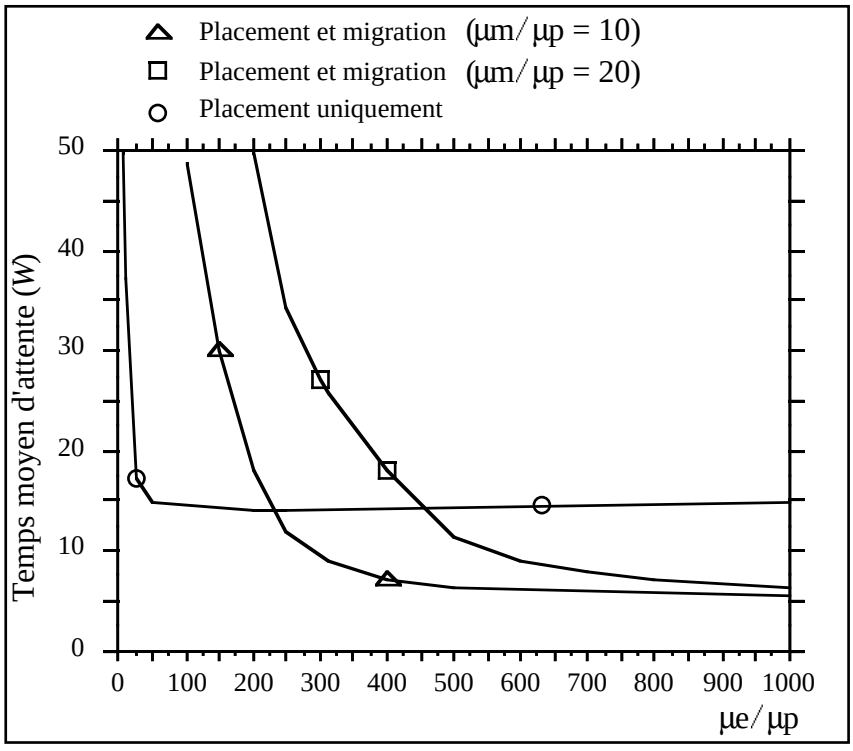


Figure 8 : Temps moyen d'attente W en fonction de μ_e/μ_p

($1/\lambda$=12.5, μ_e=10, T_{min}=2, T_{max}=3, D_{max}=1, CCV_a=CCV_s=10)

Afin que le coût de migration reste négligeable comparé au coût d'exécution des processus, nous rajoutons une nouvelle condition à la migration d'une tâche : une tâche ne peut migrer que si le temps qu'elle a dépensé en exécution est supérieur *I* fois le temps qu'elle a dépensé en migration. La figure 9 trace le temps moyen d'attente *W* en fonction de *I*. Les mesures sont effectuées dans le cas où le coût moyen de migration μ_m est suffisament important de façon à pénaliser la stratégie préemptive ($\mu_m=20*\mu_p$, $\mu_m=\mu_e/5$). Nous remarquons que pour une valeur assez faible de *I* (inférieure à 10), le recours à la migration de processus n'est plus pénalisant, bien au contraire, il permet d'améliorer davantage les temps de réponse du système, et ceci comparé à l'algorithme de placement dynamique. Notons aussi, comme on pouvait le prévoir, qu'en faisant croître la valeur de *I*, le temps d'attente *W* obtenu par l'algorithme préemptif converge vers la valeur obtenue par l'algorithme de placement.

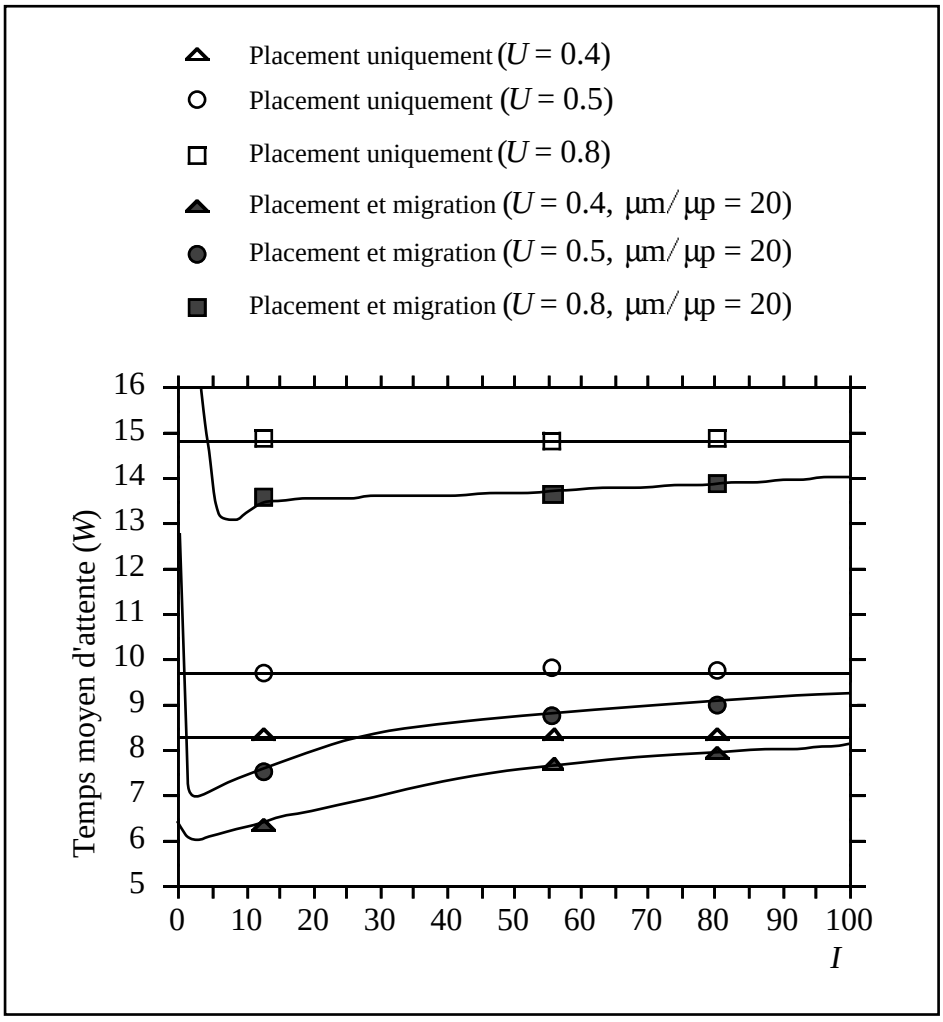


Figure 9: Temps moyen d'attente W en fonction de I

(μ_e=10, $\mu_p=\mu_e/100$, T_{min}=2, T_{max}=3, D_{max}=1, $CCV_a=CCV_s$=10)

VII.5 Conclusion

L'objectif de ce chapitre a été de montrer l'apport que la migration de processus peut avoir pour l'amélioration de la répartition de charge. Les expérimentations que nous avons effectuées montrent que la répartition de charge grâce au placement dynamique des processus permet une réduction considérable des temps de réponse. Ces expérimentations montrent aussi que la migration de processus permet d'améliorer davantage les temps de réponse du système. Cette amélioration est obtenue lorsque les processeurs sont sous utilisés, le temps d'exécution des tâches est variable et la durée d'inter-création de tâches est importante. Le recours à la migration de tâches n'est pénalisé par le coût de migration que si ce dernier n'est pas négligeable par rapport à la durée d'exécution du processus.

Chapitre VIII

Conclusion

Jusqu'à présent les machines massivement parallèles n'ont pas encore connu l'essor escompté. Leur coût et leur complexité d'exploitation en sont les principaux obstacles. Surmonter ces difficultés requiert un véritable système d'exploitation capable de gérer efficacement et simultanément l'exécution de plusieurs applications. La répartition automatique des programmes compte parmi les services de base qu'un tel système doit offrir. La réalisation de ce service est complexe de par le caractère dynamique de l'exécution des applications. Le service de migration de processus, que nous venons d'étudier, rentre dans cette problématique générale.

Disposer d'un service de migration ne doit en aucun cas altérer les temps de réponse du système. Bien plus, puisque la migration est utilisée pour des besoins d'optimisation de l'utilisation des ressources, la réduction des coûts de migration est un critère essentiel. Ainsi avons nous été amenés à proposer, pour les diverses actions à entreprendre lors de la migration d'un processus, des techniques qui répondent aux critères de coût. Deux autres critères de conception ont été également considérés : la transparence et l'extensibilité. La transparence de la migration est nécessaire pour pouvoir effectuer une répartition automatique des processus. L'extensibilité des mécanismes de migration permet de supporter les architectures massivement parallèles.

Même réduit, le coût de migration d'un processus reste à considérer. Il se pose alors la question de l'utilité effective de la migration. Pour répondre à cette question nous avons cherché à déterminer les conditions dans lesquelles la migration de processus permet une amélioration des temps de réponse du système.

La section suivante dresse un bilan de notre contribution où nous faisons ressortir les solutions que nous avons élaborées pour le développement d'un mécanisme de migration. La viabilité de ce mécanisme pour la répartition de charge est ensuite dégagée. Nous présentons enfin les perspectives de ce travail de thèse.

Bilan

La migration d'un processus nécessite plusieurs actions à entreprendre : l'extraction, le transfert et la restauration du processus de façon transparente.

L'extraction de l'image d'un processus et la restauration de son exécution sur un nouveau processeur doivent s'effectuer de manière à assurer la cohérence de l'exécution vis-à-vis du processus qui a migré et de son environnement. L'image du processus et la connaissance qu'a le processus sur l'état du système doivent ainsi rester les mêmes avant et après la migration. Les techniques de maintient de la cohérence, qui s'appuient sur une gestion de l'historique des processus, ont été écarté en raison de leur coût. Nous avons plutôt utilisé le mécanisme de désignation grâce auquel l'accès aux prérequis d'un processus se fait de manière locale ou distante selon les critères de cohérence.

La transparence de la migration est assurée en grande partie par le service de désignation qui permet aux processus en exécution de s'abstraire de la localisation des entités manipulées. Grâce à un choix approprié de la structure des identificateurs (logiques et uniques dans leurs espaces de désignation) aucune couche supplémentaire d'interprétation des identificateurs n'est nécessaire. La localisation des entités profite de l'organisation hiérarchique des espaces de désignation dans ParX. Ceci évite le recours à la diffusion de requêtes de mise à jour ou de localisation, tout en conservant une décentralisation du traitement des requêtes de localisation.

Lors de la migration d'un processus, le transfert de son image est la phase qui induit le coût le plus important. Plusieurs techniques ont été proposées pour réduire ce coût, celles-ci se distinguent notamment au niveau de la répartition dans le temps de l'opération de transfert. Cependant, comme nous l'avons montré, l'efficacité de ces techniques dépend du comportement d'accès mémoire du processus, elles peuvent même être pénalisantes. La stratégie que nous avons adoptée consiste à transférer entièrement l'image du processus. Seules les zones mémoire non accédées en écriture sont pré-copiées avant le gel du processus. Cette solution a l'avantage de ne pas avoir à modifier les mécanismes de gestion mémoire, ce qui évite des surcoûts supplémentaires.

Afin d'assurer l'échange correct des messages, nous avons considéré diverses sémantiques de communication pour lesquelles nous avons développé des protocoles de migration adéquats. Nos protocoles sont applicables sur des architectures extensibles. Pour cela nous avons évité les traitements centralisés ; sur chaque processeur du réseau les besoins en espace mémoire ont été limités indépendamment de la taille du réseau. De plus l'intégration des protocoles de

migration dans le noyau de communication de ParX n'introduit qu'un surcoût négligeable pour les délais de communication.

Ainsi pour les différentes actions à entreprendre lors de la migration d'un processus, nous avons élaboré des solutions qui ne pénalisent pas les temps de réponse du système et qui respectent les critères de transparence de la migration et d'extensibilité des architectures massivement parallèles. La construction du service de migration s'est faite en utilisant l'interface offerte par le micro-noyau de ParX. Seuls les protocoles de communication ont du être adaptés pour supporter la migration, sans pour autant modifier le micro-noyau. Ceci a pu se faire grâce au mécanisme générique de construction de protocoles qu'offre ce micro-noyau. Ces résultats montrent que les caractéristiques des systèmes massivement parallèles ne s'opposent pas à la réalisation d'un mécanisme de migration de processus dans ces systèmes.

Par ailleurs, nous avons proposé un algorithme de répartition de charge qui recourt à la fois au placement dynamique et à la migration de processus. Comparé à un algorithme uniquement fondé sur le placement dynamique, les expérimentations que nous avons effectuées montrent que la migration de processus améliore (davantage) les temps de réponse du système. Cette amélioration est obtenue lorsque les temps d'exécution et d'inter-création des processus sont variables et le coût de migration négligeable par rapport à la durée d'exécution des processus.

Perspectives

En ayant montré la faisabilité et la viabilité d'un mécanisme de migration dans un système massivement parallèle, nous pouvons engager à ce stade la validation du mécanisme de migration et de l'algorithme de répartition de charge à travers des applications réelles. Mais avant, pour que la migration de processus puisse se faire sans trop de contraintes, il est nécessaire de porter le système Paros sur une nouvelle machine où les processeurs intègrent une gestion virtuelle de la mémoire ; ce que n'offre pas les machines Supernodes à base de transputers. En outre, nous nous sommes limités à proposer un algorithme d'équilibrage de charge à l'intérieur d'un cluster, il reste à étendre cet algorithme pour répartir la charge entre les clusters. La migration de processus est alors particulièrement utile pour libérer des processeurs et les rattacher à de nouveaux clusters.

Un autre axe de travail qu'il serait intéressant d'étudier est la migration de processus entre des processeurs ayant des configurations matérielles et/ou logicielles différentes. Ceci permettrait notamment à un processus d'exploiter des ressources qui s'adaptent mieux aux traitements qu'il a à effectuer.

Bibliographie

[ACF86] Y. Artsy, H-Y Chang, R. Finkel. Processes Migrate in Charlotte. Computer Sciences Technical Report #655, Aug. 1986.

[AKn90] N. N. Avramov, A. E. Knowles. Evaluation of Two Systems for Distributed Message Passing in Transputer Networks. Real-Time Systems with Transputers, IOS Press, edited by H.S.M. Zedan, 1990.

[ArF89] Y. Artsy and R. Finkel. Designing a Process Migration Facility - The Charlotte Experience. IEEE Computer, 22(9), Septembre 1989, 47-56.

[BaL85] A. Barak, A. Litman. MOS: A Multicomputer Distributed Operating System. Software Practice and Experience 15(8), Aug. 1985.

[BaM91] S. A. Baker, K. R. Milner. A Process Migration Harness for Dynamic Load Balancing. OCCAM and the Transputer, J. Edwards Ed, IOS PRESS, 1991.

[BaM94] A. Balaniuk, T. Muntean. Programming with Shared Data in Parallel Loosely Coupled Machines: The Shared Virtual Memory Approach. Proceedings of the IEEE / USP International Workshop on High Performance Computing, Compilers and Tools for Parallel Processing, Mar. 1994.

[Bar91] J. S. Barrera. A Fast Network IPC Implementation. Proceedings of the Usenix Mach Symposium Nov. 1991.

[BBK91] R. Balter, J.-P. Banâtre, S. Krakowiak. Construction des systèmes d’exploitation répartis. Chapitre7 : Gestion Répartie d'objets, INRIA, 1991.

[BeO87] B. Beck, D. Olen. A parallel Process Model. Proceedings USENIX, sept. 1987.

[BGW93] A. Barak, S. Guday, R.G. Wheeler. The Mosix Distributed Operating System - Load Balancing for UNIX. LNCS N° 672, 1993.

[Bla92] D. L. Black et al. Microkernel Operating System Architecture and Mach. Proceedings of the Workshop on Micro-kernels and Other Kernel Architectures, seattle, Washington, Apr. 1992.

[BlS89] D. L. Black, D. Sleator. Competitive Algorithmes for Replication and Migration in Multiprocessors with Distributed Global Memory. Proceedings The 8th International Conference on Distributed Computing Systems, San Jose, California, june 1988, 161-169.

[Bok93] S. H. Bokhari. A Network Flow Model for Load Balancing in Circuit-Switched Multicomputers. IEEE Transactions on Parallel Distributed Systems, 4(6), Jun 1993.

[BrF81] R. M. Bryant, R. A. Finkel. Analysis of three dynamic distributed strategies with varying global information requirements. Proceedings of 2th International Conference on Distributed Computing Systems, Paris, Mar. 1981, IEEE Computer Society.

[BSS91] G. Bernard, D. Stève, M. Simatic. Placement et Migration de Processus dans les Systèmes Répartis Faiblement Couplés. TSI, 10(5), 1991.

[Cab86] L-F. Cabrera. The Influence of Work Load Balancing Strategies. Proceedings Usenix summer'86, Attlanta, Georgia, Jun 1986, 446-458.

[CaK88] T. L. Casavant, J. G. Kuhl. A Taxonomy of Scheduling in General-Purpose Distributed Computing Systems. IEEE Transaction on Software Engineering, 14(2), Feb. 1988.

[Cas91] H. Castro. Gestion de Ressources Virtuelles Partagées dans les Machines Parallèles. mémoire de D.E.A., Institut National Polytechnique de Grenoble, Grenoble, Sept 1991.

[CEM93] H. Castro, A. Elleuch, T. Muntean, P. Waille. Generic Microkernel Architecture for the PAROS PARallel Operating System. WTC-Transputer Applications and Systems, R. Grebe et al., IOS Press (Eds), Vol. 2, 1993.

[CGT85] W. Crowther, J. Goodhue, R. Thomas. W. Milliken, T. Blackadar. Perfomance measurements on 128-node butterfly parallel processor. International Conference Parallel Processing, Aug. 1985.

[Che84] D. R. Cheriton. The V Kernel: A softaware base for Distributed System. IEEE Software, 1(2), Apr 984, 19-42.

[Che88] D. R. Cheriton. The V Distributed System. Communication of ACM, 31(3), Mar. 1988, 314-333.

[ChS89] M-S Chen and K. G. Shin. Task Migration in Hypercube MultiProcessors. 16th annual International Symposium on computer Architectures, 1989.

[DeM93] R. Despons, T. Muntean. Constructing Correct Protocols for a Diffusion Virtual Machine in a Message Passing Parallel Architectures. Proceedings of the World Transputer Congress, Aachen, Sept. 1993.

[DGi92] G. J. W. Van Dijk, M. J. van. Gils. Efficient Process Migration in the EMPS Multiprocessor System. International Parallel Processing Symposium, Beverly Hills, California, Mar. 1992.

[DoO87] F. Douglis and J. Ousterhout. Process Migration in the Sprite Operating System. Proceedings 7th International Conference on Distributed Computing Systems, Berlin, Sept. 1987. 18-25.

[DoO91] F. Douglis and J. Ousterhout. Transparent Process Migration in the Sprite: Design Alternatives and Sprite Implementation. Software-Practice qnd Experience, 21(8), Aug. 1991.

[Dou89] F. Douglis. Experience with Process Migration in Sprite. Proceedings First Workshop on Experience with Building Distributed and Multiprocessor System, Ft. Lauderdade, Florida, October 1989.

[EKM94] A. Elleuch, R. Kanawati, T. Muntean, G. Talbi. Dynamic Load Balancing Mechanisms for a Parallel Operating System Kernel. Proceedings of CONPAR - VAPP VI, LNCS-854, Linz (Austria), Sept. 1994.

[ElM93a] A. Elleuch, T. Muntean. Process Migration in Distributed Memory Parallel Machines - Application to the PARX Kernel for PAROS. Deliverable report Esprit Project 2528-SUPERNODE II, Jan. 1993.

[ElM94] A. Elleuch, T. Muntean. Process Migration Protocols for Massively Parallel Systems. International Conference on Massively Parallel Computing Systems, IEEE Computer Society, Ischia, May 1994.

[ElT93b] A. Elleuch, E-G Talbi. Mécanisme de migration pour la gestion de processus. Journées des Jeunes Chercheurs en Systèmes Informatiques Répartis, Grenoble, Avr. 1993.

[ELZ88] D. L. Eager E. D. Lazowska et J. Zahorjan. The limited Performance Benefits of Migrating Active Processes for Load Sharing. Proceedings of the 1988 ACM, SIGMETRICS Conference on the Measurement and Modelling of Computers Systems Performance Evaluation, Jan. 1988.

[EMT93] A. Elleuch, T. Muntean, E-G Talbi. Migration et allocation dynamique de processus dans le système PAROS. 5ème Rencontres sur le parallélisme, Brest, Mai 1993.

[FBC91] M. J. Feeley, B. N. Bershad, J. S. Chase, H. M. Ley. Dynamic Node Reconfiguration in a Parallel-Distributed Environment. ACM SIGPLAN

Symposium on Principles and Practice of Parallel Programming, Jul. 1991, 114-121.

[Fow86] R. J. Fowler. The Complexity of Using Forwarding Addrsses for Decentralized Object Finding. Proceedings 5th ACM Sympsium on the Principles of Distributed Computation, Calgary, Canada, Aug. 1986.

[Gai90] J. Gait. Scheduling and Process Migration in Partitioned Multiprocessors. Journal of Parallel and Distributed Computing, 8(3), 1990.

[GLM91] N. A. Gonzalez Valenzuela, Y. Langue Tsobgny, T. Muntean. Paros Kernel Approach for the Supernode II Project. Deliverable report Esprit Project 2528, Sep. 1991.

[Gon91] N. A. Gonzalez Valenzuela. Parx : Noyau de système Parallèle pour les Ordinateurs Massivement Parallèles - Contrôle de la communication entre processus. Thèse de doctorat de l'Institut National Polytechnique de Grenoble, 1991.

[GuG90] R. Gupta, P. Gopinath. A Hierarchical Approach to Load Balancing in Distributed Systems, The 5th Distributed Memory Computing Conference, Charleston, Avr. 1990.

[Hab89] S. Habert. Gestion d'Objets et Migration dans les Systèmes Répartis. Thèse de doctorat de L'université Paris VI, 1989.

[Hac89] A. Hac'. A Distributed Algorithm for Performance Improvement Through File Replication, File Migration and Process Migration. IEEE transactions on Software Engineering 15(11), Nov. 1989.

[Hil92] D. Hildebrand. An Architectural Overview of QNX. Proceedings of the Workshop on Micro-kernels and Other Kernel Architectures, seattle, Washington, Apr. 1992.

[HMD86] J. P. hayes. T. N. Mudge, Q. F. Dtout, S. Colley, Architecture of a Hypercube supercomputer. 1986 Int'l Conference Parallel Processing, Aug. 1986.

[Hoa85] C.A.R. Hoare. Communicating Sequential Processes. Prentice-Hall, Englewoods Cliffs, 1985.

[Inm89] INMOS Ltd. Transputer Reference Manual. Prentice Hall International, 1988.

[JLH88] E. Jul, h. Levy, N. Hutchinson, A. Black. Fine-Grained Mobility in the Emerald System. Transaction On Computer Systems, 6(1), Feb 1988, 109-133.

[JMJ89] C. Jacqumot, E. Milgron, W. Joosen, Y. Berbers. Naming and Network Transparent Process Migration in Loosely Coupled Distributed Systems. Working Conference on Decentralized Systems, Lyon, Dec. 1989, 77-91.

[JoC90] P. Jones and H. Cha. On the Feasibility of Run-time Process Migration in Multi-Transputer Machines. Real-Time System with Transputers, H. Zeden, Ed IOS Press, 1990.

[JoV89] W. Joosen, P. Verbaeten, K.U. Leuven. On the Use of Process Migration in Distributed Systems. Microprocessing and Microprogramming Vol 28, North-Holland, 1989.

[Jul88] E. Jul. Object Mobility in a Distribued Object-Oriented System. Technical Report 88-12-06, University of Washington Seattle, Dec. 1988.

[KBC94] D. C. Kulkarni, A. B. Banerji, D. L. Cohn. Operating Systems and Cost Management. Operating Systems Review, 28(1), ACM Press, Jan. 1994.

[LaN79] H. C. Lauer, R. M. Needham. On the Duality of OperatingSystem Structures. ACM, SIGOPS, 13(2), Apr. 1979.

[Lan91] Y. Langue Tsobgny. ParX : Architecture de Noyau de système d'Exploitation Parallèle. Thèse de doctorat de l'Institut National Polytechnique de Grenoble, Grenoble 1991.

[LCL87] C. Lu, A. Chen, J. W. S. Liu. Protocols for Reliable Process Migration, IEEE INFOCOM'87, Proceedings of the Sixth Annual Conference on Global Network: Concept to Realization, Mar. 1987, 804-813.

[LeO86] W. E. Leland and T. J. Ott. Load Balancing Heuristics and Process Behaviour. Proceedings Performance 86, ACM SIGMETRICS, May 1986, 54-69.

[LHK93] W. Lux, H. Härtig, W. E. Künhauser. Migrating Multi-Threaded, Shared Objects. Proceedings of the Hawaii International Conference on System Sciences, Vol. 2, Wailea, Jan. 1993.

[LiS92] M. Litzkow, M. Solomon. Supporting Checkpointing and Process Migration Outside the Unix Kernel. Usenix Winter 1992 Technical Conference, San Fransisco, Jan. 1992.

[Lu89] C. Lu. Process Migration in Distributed Systems. PhD thesis, University of Illinois at Urbana - Champaign 1989.

[MaS88a] G. Q. Maguire. J. M. Smith. Process Migration : Effects on Scientific Computation. ACM Sigplan notices 23(3), Mar. 1988.

[MaS88b] K. I. Mandelberg. V. S. Sunderam. Process Migration in UNIX Networks. USENIX Winter Conference, Dallas, Texas, Feb. 1988, 357-363.

[Men93] F. Menneteau. Un Noyau de Système Parallèle à Objet. Thèse de doctorat de l'Institut National Polytechnique de Grenoble, 1993.

[MGZ93] D. Milojicic. P. Giese, W. Zint. Load Distribution on Microkernels. IEEE Workshop Future Trends in Distributed Computing Systems, Lisboa, Portugal, Sep. 1993.

[MMS90] L. Mugwaneza, T. Muntean, I. Sakho. A Deadlock-free Routing Algorithm with Network Size Independent Buffering Space, CONPAR90-VAPPIV, Zurich, Sep. 1990.

[MRT90] S. J. Mullender, G. van Rossum, A.S. Tanenbaum, R. van Renesse H. van Staveren. A Distibuted Operating System for the 1990's. IEEE computer Magazine, May 1990.

[Mug93] L. Mugwaneza. Contrôle des Communications dans les Machines Parallèles à Mémoire Distribuée. Thèse de doctorat de l'Institut National Polytechnique de Grenoble , Grenoble 1993.

[Mun89] T. Muntean et al. PARX: a Parallel Operating System for Transputer Based Machines. Applying transputers based parallel machines, OUG-10, Ed. , by A. Bakkes, Enschede, Netherland, Apr 1989, 115-141.

[Mun94] T. Muntean et al. PAROS: A generic multi virtual machines parallel operating system. ParCo'93 Conference, Parallel Computing: Trends and Applications, G. R. Joubert, et al., Elsevier Science (Eds), 1994.

[MuT91] T. Muntean, E-G Talbi. New approach for the mapping problem - A parallel genetic algorithm. 2nd Symposium on High Perfomance Computing, Montpellier, France, Oct. 1991.

[MuW90] T. Muntean, P. Waille. L'architecture des Machines Supernode. La lettre du Transputer, Sept. 1990, 11-40.

[MZD93] D. Milojicic, W. Zint, A. Dangel, P. Giese. Task Migration on the Top of the Mach Microkernel. Proceedings of the Third USENIX Mach Symposium, Jun 1993.

[Nug88] S. F. Nugent. The iPCS/2 direct-connect communications technology. Third Conference Hypercube Computers and Applications, Jan. 1988.

[NXG85] L. M. Ni, C-W. Xu, T. B. Gendreau. A Distributed Drafting Algorithm for Load Balancing. IEEE Transaction on Software Engineering , 11(10), Oct. 1985.

[OCD88] J. K. Outerhout, A. R. Cherenson, F. Duoglis. M. N. Nelson and B. B. Welch. The Sprite Network Operating System. IEEE Computer, Feb. 1988, 23-36.

[OSS92] T. Okamoto, H. Segawa, S. H. Shin. A Microkernel Architecture for Next Generation Processors Usenix Association. Micro-kernels and Other Kernel Architectures, Apr. 1992.

[OTC94] M. O'Connor, B. Tangney, V. Cahill, N. Harris. Micro-kernel support migration. Distributed Systems Engineering N° 1, 1994.

[PBG85] G. F. Pfister, W. C. Brantley, D. A. George, S. L. Melton, V. A. Norton, J. Weiss. The IBM Research Parallel Processor Prototype (RP3): introduction and architecture. 1985 Int'l Conference Parallel Processing, Aug. 1985.

[Phi93] L. Philippe. Contribution à l'Etude et la Réalisation d'un système d'Exploitation à Image Unique pour Multicalculateur. Rapport Technique 308, Université de Franche-comté, 1993.

[PoM83] M.L. Powel, B. P. Miller. Proccess Migration in DEMOS/MP. Proceedings 9th ACM Symposium Operating Systems Principles, ACM Operating systems Review, 17(5), 1983.

[PPT92] D. Presotto, K. Thompson, H. Trickey. Plan 9, A Distributed System. Proceedings of the Workshop on Micro-kernels and Other Kernel Architectures, seattle, Washington, Apr. 1992.

[PWC81] G. Popek, B. Walker, J. Chow, D. Edwards, C. Kline, G. Rudisin, G. Thiel. A Network Transparent, High Reliability Distributed System. Proceedings of the 8th Symposium on Operating Systems Principles, Pacific Grove, California, Dec. 1981.

[RaJ88] T. M. Ravi and D. Jefferson. A basic protocol for routing messages to migrating processes. Proceedings of the International Conference on Parallel Processing, The Penn State, Aug. 1988.

[Ras86] R. F. Rashid. Experiences with Accent Network Distributed Operating System. Networking in Open Systems. LNCS N° 248, Springer-verlag, 1986.

[Ros91] Rozier et al. Overview of the Chorus Distributed Operating Systems. Technical Report CS/TR-90-25.1, Chorus système, 1991.

[SaS88] Y. Saad, M. H. Schultz. Topological Propreties of Hypercubes. IEEE Transaction on Computers, 37(7), Jul. 1988.

[SaS89] Y. Saad, M. H. Schultz. Data communication in parallel architectures. Parallel Computing 11, North-Holland, 1989.

[ScD88] C. Scheurich, M. Dubois. Dynamic Page Migration in Multiprocessors with Distributed Global Memory. The 8th International Conference on Distributed Computing Systems, June 1988.

[Sch88] W. Schröder-Preiksschat. Peace - A Distributed Operating System for MIMD-Passing Architecture. Third International Conference on Super computing, Boston MA, May 988, 302-312.

[Sch90] W. Schröder-Preiksschat. Peace - A Distributed Operating System for High Perfomance Multicomputer Systems. LNCS N° 443. Progress in Distributed Operating Systems and Distributed System Management, Scröder-Preiksschat and W. Zimener (Eds.), 1990.

[Sim86] Simulog. QNAP2 reference manuel version 4.0. Paris, 1986.

[SJR86] R. D. Sansom, D. P. Julin, R. F. Rashid. Extending a Capability Based System into a Network Environment. Technical Report CMU-CS-86-115, Department of Computer Science Carnegie-Mellon University, 1986.

[SKS92] N. G. Shivaratri, P. Krueger, M. Singhal. Load Distributing for Locally Distributed Systems. IEEE computer Dec. 1992.

[Smi88] J. M. Smith. A Survey of Process Migration Mechanism. ACM Operating systems Review, 22(3), Jul. 1988, 28-40.

[SSC89] T. Shwerski, H. J. Siegel, T. L. Casavant. Task Migration Transfers in Multistage Cube Based Parallel Systems. Proceedings of the 1989 International Conference Parallel Processing, Vol I Aug. 1989, 296-305.

[SSC90] T. Shwerski, H. J. Siegel, T. L. Casavant. Optimizing Task Migration Transfers Using Multistage Cube Networks. Proceedings of the 1990 International Conference Parallel Processing, Vol I Aug. 1990.

[SSF92] G. Saghi, H. J. Siegel, J. A. B. Fortes. On The Viability of a Quantitative Model of System Reconfiguration Due to a Fault. International Conference on Parallel Processing, University of Michigan, Aug. 1992.

[SSS89] M. Sander, H. Schmidt, W. Schröder-Preikschat. Naming in the Peace Distributed Operating System. Proceedings of the International Workshop on Communication Network and Distributed Operating Systems Within the Space Environment, Noordwijk, The Netherlands, Oct. 1989.

[Sup91] Supernode II Workpackage 2. SNOS Kernel Specification. ESPRIT Project 2528, Operating Systems and Programming Environments for Parallel Computer, Final draft, Jun 1991.

[SWo92] T. T. Y. Suen, J. S. K. Wong. Efficient Task Migration Algorithm for Distributed Systems. IEEE Transaction on Parallel and Distributed Systems, 3(4), Jul. 1992.

[Tal91] E.-G. Talbi. Un algorithme d'allocation dynamique de processus sur un réseau de transputers. La lettre du transputer et des calculateurs distribués, N° 11, Sep. 1991.

[Tal93] E.-G. Talbi. Allocation de procesus sur les architectures parallèles à mémoire distribuée. Thèse de doctorat de l'Institut National Polytechnique de Grenoble, Grenoble 1993.

[Tev87] A. Tevanian. Jr. et al. Mach Threads and the Unix Kernel: The Battle for Control. Technical Report, CMU-CS-87-49, Carnegie Mellon University, 1987.

[The86] M. M. Theimer. Preemptable Remote Execution Facilities for Loosely-Coupled Distributed Systems. Technical Report STAN-CS-86-1128, Standford University, Jun 1986.

[ThH92] M. M. Theimer, B. Hayes. Heterogeneous Process Migration by Recompilation. Technical Report CSL-92-03, Palo Alto Research Center, 1992.

[TLC85] M. M. Theimer, K. A. Lantz, D. R. Cheriton. Preemptable Execution Facilities for the V-System. Proceeding of the 10th ACM Symposium on Operating Systems Principles, ACM SIGOPS, 1985,2-12.

[TMV86] A. S. Tanenbaum, S. J. Mullender, R. Van Renesse. Using Sparse Capabilities in a Distributed Operating System. Proceedings of 6th International Conference on Ditributed Computing Systems, Cambridge, MA, May 1986.

[TRo88] L. W. Tucker, G. G. Robertson. Architectures and applications of the Connection Machine. Computer, Aug. 1988.

[UyR88] M. U. Uyar, A. P. Reeves. Dynamic fault reconfiguration un a mesh-connected MIMD environment. IEEE Transaction on Computer, C-37, Oct. 1988.

[vRT92] R. van Renesse, A. S. Tanenbaum. Short Overview of Amoeba. Proceedings of the Workshop on Micro-kernels and Other Kernel Architectures, seattle, Washington, Apr. 1992.

[Zaj93] R. Zajcew et al. An OSF/1 UNIX for Massively Parallel Multicomputers. Proceedings of the Winter USENIX Conference, San Diego, Jan. 1993.

[Zay87a] E. R. Zayas. The Use of Copy-On-Reference in a Process Migration System. PhD in Computer Science at Carnegie Melon University, CMU-CS-87-121, Apr. 1987.

[Zay87b] E. R. Zayas. Attacking the Process Migration Bottleneck. Proceedings 11th ACM Symposium Operating systems Principles, Austin, Texas, 8(11), Nov. 1987, 13-24.

[ZGG90] W. Zhu, A. Gscinski, G. W. Gerrity. Process Migration in RHODOS. Technical Reasearch Report Australian Defence Force Academy North cott Drive, Canberra, ACT, 1990.