Institutional Repository - Research Portal
Dépôt Institutionnel - Portail de la Recherche

researchportal.unamur.be

University of Namur

THESIS / THÈSE

MASTER EN SCIENCES INFORMATIQUES

Méthodes de communication inter-processus

Toussaint, Hubert

Award date:
2004

Link to publication

General rights
Copyright and moral rights for the publications made accessible in the public portal are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.

- Users may download and print one copy of any publication from the public portal for the purpose of private study or research.
- You may not further distribute the material or use it for any profit-making activity or commercial gain
- You may freely distribute the URL identifying the publication in the public portal ?

Take down policy
If you believe that this document breaches copyright please contact us providing details, and we will remove access to the work immediately and investigate your claim.

Download date: 10. Jul. 2024

Facultés Universitaires Notre-Dame de la Paix, Namur
Institut d'Informatique
Année Académique 2003 - 2004

Méthodes de communication inter-processus

Hubert Toussaint

Mémoire présenté en vue de l'obtention du grade de Maître en Informatique.

Résumé

Les diverses méthodes de communication inter-processus (IPC) couvrent des champs d'actions très variés. Ce document présente la plupart des méthodes employées sur les plateformes les plus couramment utilisées ainsi qu'un ensemble de critères de classification. Ces critères peuvent être utilisés afin de faciliter la sélection des méthodes IPC dans le cadre d'un projet concret d'application.

Le développement d'une architecture multi-processus pour un *proxy* HTTP simple sera utilisé pour illustrer l'emploi de ces critères lors du processus de sélection.

mots-clés : communication inter-processus, IPC, Unix, critères de sélection.

Abstract

The various inter-process communication methods (IPC) can be used in many contexts. This document presents most of the methods found on usual systems and a way to classify them all using several criteria . These criteria can be used to ease the choice of suitable methods for a concrete project.

Development of a multi-process architecture for a simple HTTP proxy will illustrate the selection process.

keywords : inter-process communication, IPC, Unix, selection criteria.

Remerciements :

Je tiens a remercier tous ceux sans qui ce travail n'aurait probablement pas pu être mené à bien.

En particulier mon promoteur, monsieur Jean Ramaekers ainsi que messieurs Gilles Fleury et Dimitri Tombroff pour leur direction attentive.

Table des matières

Glossaire xv

Introduction 1

1 Principes généraux 3

1.1 Définition IPC . . . 3

1.2 Nécessité de la synchronisation . . . 3

1.3 Le cas particulier des threads . . . 4

1.4 Modèles de communication . . . 6

2 IPC Unix 7

2.1 Les pipes . . . 7

2.1.1 Présentation . . . 7

2.1.2 Performances . . . 10

2.2 Les mémoires partagées . . . 11

2.3 Les sockets (Unix / network) . . . 15

2.3.1 Sockets réseaux . . . 15

2.3.2 Sockets Unix . . . 16

2.4 Les fichiers . . . 18

2.4.1 Les 'fifos' . . . 18

2.4.2 Les périphériques spéciaux (/dev, /proc) . . . 20

2.4.3 Les fichiers standards . . . 21

2.5 Les signaux . . . 22

2.6 L'interface SYSV . . . 26

2.6.1 Les mémoires partagées . . . 26

2.6.2 Les sémaphores . . . 28

2.6.3 Les files de messages . . . 29

2.7 Performance des différentes IPC . . . 32

3 IPC Windows 35

3.1 Le clipboard . . . 35

3.2 Data Copy . . . 37

3.3 DDE . . . 38
3.4 File . . . 39
3.5 MailSlot . . . 40
3.6 Pipes (anonymous / named) . . . 41
3.6.1 Anonymous Pipes . . . 41
3.6.2 Named Pipes . . . 42
3.7 Network sockets . . . 43

4 IPC indépendantes de la plateforme **45**
4.1 Présentation . . . 45
4.2 Exemple : Curry . . . 45
4.3 Distribution . . . 45

5 IPC Distribuées **47**
5.1 Définition . . . 47
5.2 Problèmes additionels engendrés par la distribution . . . 47
5.2.1 Hétérogénéité . . . 47
5.2.2 Le problème du temps . . . 48
5.2.3 La fiabilité . . . 49
5.2.4 La sécurité . . . 50
5.3 Exemples d'IPC distribuées . . . 50
5.3.1 Sockets réseau et protocole spécifique . . . 50
5.3.2 Architecture *middleware* . . . 51
5.4 Conclusion . . . 54

6 Classification des différentes méthodes **57**
6.1 Critères . . . 57
6.1.1 Portée . . . 57
6.1.2 Protection . . . 57
6.1.3 Parallélisme . . . 58
6.1.4 Propreté . . . 58
6.1.5 Préemption . . . 58
6.1.6 Restrictions . . . 58
6.1.7 Débit . . . 58
6.2 Comparaison . . . 58
6.2.1 Fichier . . . 59
6.2.2 Mémoire partagée . . . 60
6.2.3 Signaux . . . 61
6.2.4 Sockets réseau . . . 62
6.2.5 Tubes anonymes . . . 63
6.2.6 Tubes nommés . . . 64
6.2.7 Synthèse . . . 65

6.3 Conclusion 65

7 Application au stack **67**

7.1 Problématique 68

7.1.1 Entrées/sorties asynchrones 68

7.1.2 Appel système *poll(..)* 70

7.1.3 Mono-processus 71

7.2 Proposition d'architecture 72

7.2.1 Principe 72

7.2.2 Choix de la méthode de communication 73

7.2.3 Description 76

7.2.4 Communication Interne 77

7.2.5 Critique de l'architecture 79

7.3 Mesures de performances 82

7.3.1 Procédure de test 82

7.3.2 Résultats 84

8 Conclusion **93**

Annexes **95**

1 Protocole de mesure de performances des IPC Unix **97**

2 Bibliographie **99**

Table des figures

1.1 Méthode de communication inter-processus : principe général. 3
1.2 Méthode de communication inter-processus, intervention du noyau. 4
1.3 Processus et Threads - Mémoire. 5
1.4 Processus et Threads - Exécution. 5
2.1 *pipe* Unix. 7
2.2 *pipe* Unix après un *fork()*. 8
2.3 Appel système *pipe*. 8
2.4 Lecture/Ecriture dans un pipe. 8
2.5 La structure de données sous-jacente au *pipe*. 9
2.6 Débit moyen d'un pipe. 10
2.7 Principe de la mémoire partagée. 11
2.8 Appel système *mmap*. 11
2.9 La structure de données sous-jacente à la mémoire partagée. 12
2.10 Débit moyen d'un transfert par mémoire partagée. 13
2.11 Performances des fonctions de copies de blocs mémoire. 13
2.12 Débit moyen d'un transfert par mémoire partagée. 14
2.13 Sockets Réseaux. 15
2.14 Débit moyen d'un transfert par socket réseau (mode flux). 16
2.16 Sockets Unix 16
2.15 Débit moyen d'un transfert par socket réseau (mode datagramme). 17
2.17 Débit moyen d'un transfert par socket Unix. 18
2.18 Création d'un 'fifo'. 19
2.19 Principe de fonctionnement d'un fifo. 19
2.20 Accès au périphériques par */dev*. 20
2.21 Accès au noyau par */proc*. 21
2.22 IPC par fichier standard. 22
2.23 Signaux Linux x86. 23
2.24 Actions des signaux. 23
2.25 Appel système *signal(..)*. 24
2.26 Appel système *kill(..)*. 25
2.27 Appel système *shmget(..)*. 27

2.28 Appels système *shmatt(..)* et *shmdt(..)*. 27
2.29 Appel système *shmctl(..)*. 27
2.30 Appels système *semget(..)*,*semctl(..)* et *semop(..)*. 29
2.31 La structure de données sous-jacente à une file de messages. 30
2.32 Appels système *msgget(..)*,*msgctl(..)*, *msgsnd(..)* et *msgrcv(..)*. 30
2.33 Débit moyen d'un file de message 31
2.34 Débit des différentes IPC 32
2.35 Débit des différentes IPC - Vue rapprochée. 33
3.1 Opérations de base sur le clipboard. 36
3.2 Principe de fonctionnement du clipboard. 37
3.3 Appel *SendMessage()*. 37
3.4 Communication par fichier 39
3.5 Appel *CreateMailslot(..)*. 40
3.6 Tube anonyme sous Windows. 41
3.7 Tubes nommé sous Windows 42
4.1 Curry : unification à distance 46
5.1 Système et IPC distribué. 47
5.2 Le problème des délais incertains. 49
5.3 La fiabilité en contexte local. 49
5.4 La fiabilité en contexte distribué. 50
5.5 Communication directe par sockets 51
5.6 Middleware 52
5.7 RPC : principe 52
6.1 Critère de portée 57
7.1 Schéma d'un serveur proxy 67
7.2 Un proxy particulier : *HttpStack*. 68
7.3 Entrées/Sorties synchrones. 69
7.4 Entrées/Sorties asynchrones. 69
7.5 Fonction poll(..). 70
7.6 Proxy - vue rapprochée. 72
7.7 Solution : multiplication de l'appel *poll(..)*. 72
7.8 Solution : multiplication des processus. 73
7.9 Solution : combinaison. 73
7.10 Architecture du prototype 76
7.11 Emulation de *malloc(..)/free(..)* en mémoire partagée. 77
7.12 Communication composite : Message court. 78
7.13 Communication composite : Message long. 78
7.14 Création de fichier *core* incomplet. 80
7.15 Configuration de test. 82
7.16 Options du serveur *boa*. 83

7.17 Résultats hors charge. 86
7.18 Résultats en charge. 88
7.19 Résultats 90
9.1 Fonction *gettimeofday(..)* 97
9.2 Mesure du temps de transfert. 98

Liste des tableaux

6.1 Synthèse des éléments de comparaison. 65
7.1 Synthèse des éléments de choix. 74
7.2 Résultats hors charge. 86
7.3 Résultats en charge. 88
7.4 Résultats 91

Glossaire

Inode Une inode est une structure de données contenant toutes les informations sur les fichiers dans un système de fichiers Unix. Chaque fichier possède une inode et est identifié de manière unique par le système de fichiers sur lequel il réside et par le numéro d'inode sur ce système. Une inode contient les informations suivantes : le périphérique abritant le fichier (/dev/XXX), des informations de verrouillage, le mode et le type du fichier, le nombre de liens vers ce fichier, le numéro du propriétaire du fichier ainsi que de son groupe, les temps d'accès et de modification, le temps de dernière modification de l'inode, et les adresses sur le disque des blocs contenant les données du fichier.

IPC Une méthode de communication inter-processus (IPC) peut être définie comme un procédé volontaire et techniquement bien défini par lequel un processus transmet une information à un ou plusieurs autres processus.

IPC Distribuée Une IPC distribuée est une IPC permettant de faire communiquer deux ou plusieurs processus d'un système distribué.

Processus Un processus est un contexte d'exécution noyau (i.e. schédulé par le noyau) possédant un espace mémoire propre, espace dont la protection est assurée par le noyau.

Système Distribué Un système ditribué est une collection d'hôtes autonomes qui sont connectés par l'intermédiaire d'un réseau.

Thread Un thread est un contexte d'exécution utilisateur (i.e. schédulé par une librairie utilisateur) partageant son espace mémoire avec l'ensemble des autres thread constituant le processus.

Introduction

Le puissance de calcul des ordinateurs est en constante progression ces dernières années. Cette croissance répond aux besoins de plus en plus importants des utilisateurs de l'informatique : ils demandent des systèmes capables de traiter leurs données en des temps records tout en réagissant immédiatement aux demandes de l'utilisateur. Une des pistes suivies pour l'amélioration des performances matérielles est la multiplication des processeurs. L'augmentation du nombre de processeurs touche à la fois les systèmes professionels (machines à 16, 32, 64, ... processseurs) que les systèmes particuliers (par exemple l'architecture *hyper-threading* d'Intel qui permet de dédoubler les capacités du micro-processeur en simulant une architecture bi-processeurs).

Afin de tirer pleinement partie de ces architectures multi-processeurs, les programmes ont été adaptés : les applications monolithiques ont été éclatées en plusieurs entités de taille réduite. Ces entités, appelées processus, ont été créées de manière à être relativement indépendantes afin d'être capables de s'exécuter de manière parallèle sur les différents processeurs de la machine. On parle alors d'application multi-processus.

Le fait que les processus soient des objets indépendants n'ayant *a priori* aucuns contacts entre eux est nécessaire pour leur permettre d'exploiter pleinement les ressources de la machine sans risquer de s'interrompre l'un l'autre. Ce cloisonnement a contribué à créer le besoin de méthodes de communication bien structurées capables de mettre en relation deux ou plusieurs processus sans pour autant remettre en cause leur capacité d'exécution parallèle. Ces méthodes de communication inter-processus constituent le propos central de ce document.

Dans un premier temps, nous allons nous concentrer sur la communication entre processus locaux (i.e. situés sur la même machine). Après avoir défini de manière précise le terme *communication inter-processus* (chapitre 1), nous allons présenter les principales méthodes de communication inter-processus utilisées actuellement. Cette présentation se fera au travers de l'étude de différentes plateformes (chapitres 2 à 4). Le chapitre 5 présentera une extension des méthodes de communication inter-processus aux cas où les différents processus communiquants ne se trouvent pas tous sur une même machine physique. Une classification des différentes méthodes présentées sera réalisée afin de mettre en évidence les caractéristiques communes de certaines méthodes (chapitre 6). Ensuite nous allons utiliser un exemple concret d'utilisation de ces méthodes dans la construction d'une application spécifique afin de montrer la complémentarité existant entre les différentes méthodes (chapitre 7).

Remaques :

- La quasi-totalité des codes et exemples de ce documents sont écrits en langage C. Le lecteur qui ne serait par familier avec ce formalisme peut consulter [Willms, 1997] pour une introduction aux concepts et aux notations.

- Ce document est centré sur les architectures Unix et Windows. Ces deux architectures regroupent la plupart des systèmes informatiques actuellement en service. Le lecteur désireux d'approfondir les aspects de bas niveau de ces systèmes peut consulter [Stevens, 1993] pour les systèmes Unix et [Microsoft, 2004] pour les systèmes Windows.
- La documentation et les mesures présentées dans ce document ont été rassemblées au cours du stage effectué durant le premier semestre de cette année au sein de la société *Nextenso S.A.*. Le principal objet du stage concernait l'étude des IPC du monde Unix et de l'O.S. Linux en particulier. En conséquence, la majeure partie du propos de cet ouvrage portera sur les IPC Unix utilisables dans un contexte de communication locale.

Chapitre 1 : Principes généraux

1.1 Définition IPC

Une méthode de communication inter-procesus (IPC) peut être définie comme *un procédé volontaire et techniquement bien défini par lequel un processus transmet une information à un ou plusieurs autres processus*. Le terme information est ici utilisé au sens large : il peut tout aussi bien s'agir d'un transfert de données que d'une simple notification (ouverture d'une section critique, terminaison d'un processus, ...). Cette définition exclut d'emblée toutes les communications involontaires (dépassements de segments mémoires,...) ou frauduleuses (interceptions de communications réseau, *symlink* vers un fichier d'output,...).

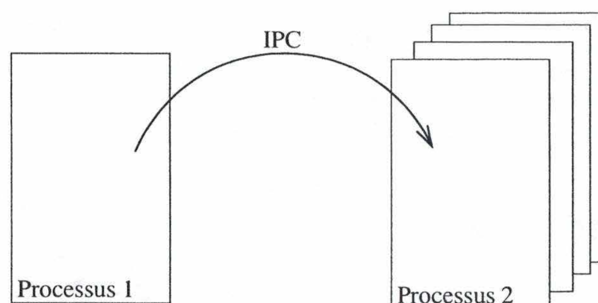

FIG. 1.1 – Méthode de communication inter-processus : principe général.

Cette étude se limitera à l'analyse des IPC dites *de bas niveau*[1] car elles sont les pierres angulaires des méthodes de communication plus évoluées et elles permettent une approche plus simple des notions sous-jacentes. La première partie présentera les IPC locales, c'est-à-dire les méthodes de communication entre deux ou plusieurs processus s'exécutant sur la même machine. Cette restriction du point de vue sera levée dans la seconde partie.

Les méthodes de communication inter-processus sont aujourd'hui présentes sur tous les systèmes d'exploitation, avec parfois quelques particularités intéressantes.

1.2 Nécessité de la synchronisation

Toute interaction entre deux (ou plusieurs) processus requiert la prise en charge des aléas engendrés par leurs exécutions parallèles : la plupart des ressources nécessaires à la communication inter-processus doivent être protégées contre les tentatives d'accès concurrents afin de garantir l'intégrité des informations transférées.

[1]Par IPC de bas niveau, nous entendons, les IPC fournies par le noyau de l'OS, par opposition aux IPC fournies par des librairies situées dans le domaine utilisateur.

C'est à ce niveau qu'interviennent les primitives de synchronisation que sont les sémaphores et autres verrous.

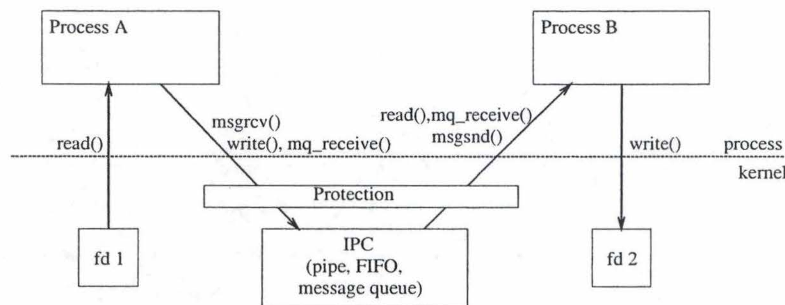

FIG. 1.2 – Méthode de communication inter-processus, intervention du noyau.

Dans la plupart des méthodes présentées, cette protection est effectuée par le noyau de l'OS de manière invisible pour l'utilisateur (figure 1.2). Mais il ne faut pas pour autant en ignorer l'existence, car cette opération provoque parfois des délais notables lors des transferts d'information.

1.3 Le cas particulier des threads

Avant toutes choses, il convient de clarifier les notions de *thread* et de *processus* : dans ce qui suit, un *thread* sera considéré comme étant *un contexte d'exécution utilisateur partageant son espace mémoire* tandis qu'un processus sera *un contexte d'exécution noyau possédant un espace mémoire propre*. Il s'agit là de la convention utilisée sur les systèmes Unix.

Une conséquence importante de ces définitions est que, du point de vue de l'O.S., l'ensemble des threads composant un programme est vu comme un seul et unique programme par le noyau ; ce qui signifie l'utilisation d'un seul espace d'adressage, mais aussi un seul gestionnaire de signaux et un seul token de scheduling (cf. figures 1.3 et 1.4) . La gestion de l'enchaînement de l'exécution des différents threads est laissée à la charge d'une librairie spécialisée. La puissance de ce mode de fonctionnement dépend grandement de la qualité de cette librairie, c'est elle qui doit gérer les problèmes habituellement dévolus à l'OS. Ainsi, si un seul espace d'adressage permet une communication facile entre les différents threads, il ne permet par contre aucune protection des zones mémoires utilisées ; de même, l'utilisation d'un seul gestionnaire de signaux va provoquer l'arrêt de tous les threads lors de la réception d'un signal non géré. Enfin, un seul token de scheduling signifie que la libraire doit absolument intercepter tous les appels systèmes potentiellement bloquants, sous peine de voir l'ensemble des threads bloqués lors d'une entrée/sortie par exemple.

Afin de ne pas devoir entrer dans la problématique de cette gestion, la suite de ce document se limitera à la communication entre processus, bien que la plupart des méthodes présentées ici soient aussi applicables dans le cas des threads.

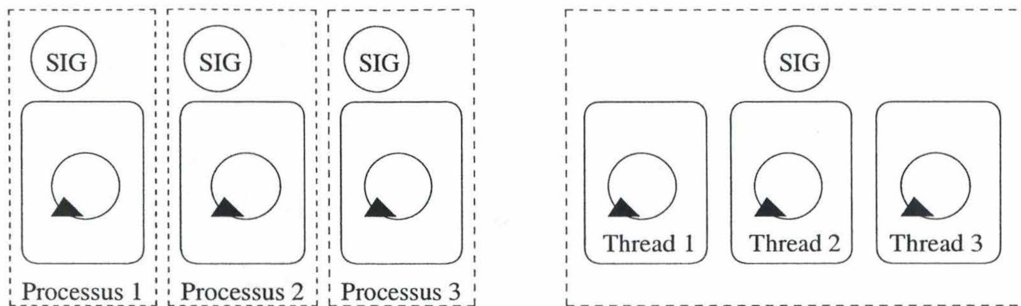

FIG. 1.3 – Processus et Threads - Mémoire.

Tous les processus disposent de leur espace mémoire et de leur gestionnaire de signaux (SIG) propres, là où les threads se partagent le même objet.

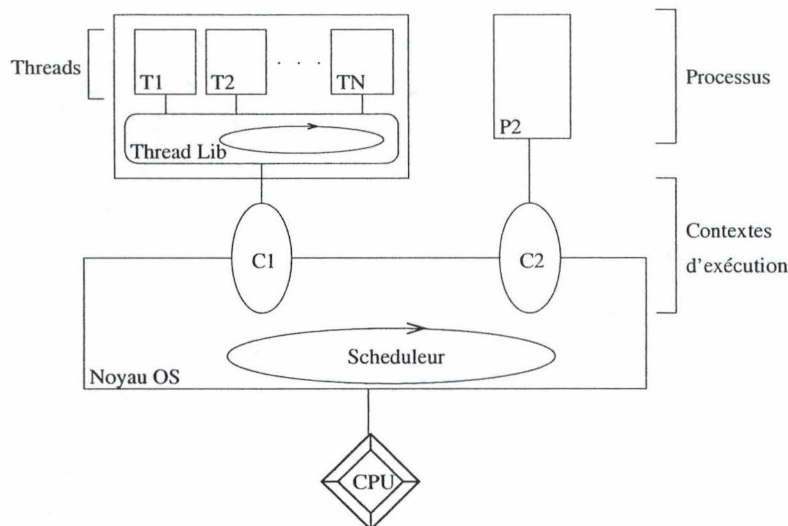

FIG. 1.4 – Processus et Threads - Exécution.

Un Processus (P2) dispose d'un contexte d'exécution dédié (architecture 1-1) tandis qu'un thread (T1) partage son contexte d'exécution avec tous les autres threads (T2...TN) composant le processus (architecture N-1).

Il est important de remarquer ici que les processus, bien qu'ayant été définis comme des éléments indépendants l'un de l'autre, peuvent entretenir certains contacts privilégiés entre eux. Il s'agit par exemple de la relation de filiation qui lie un processus père à tous les processus fils qu'il a créés. De même deux processus peuvent posséder un accès sur même zone mémoire gérée par le noyau (cas du descripteur de fichier partagé entre plusieurs processus).

1.4 Modèles de communication

Nous présentons ici les deux modèles de communication les plus courants : le modèle producteurs-consommateurs où un message est lu par un et un seul consommateur, et le modèle lecteurs-rédacteurs où un message peut être lu par un nombre quelconque de lecteurs.

Le modèle producteurs-consommateurs comporte deux classes de processus : les producteurs fabriquent des messages et les mettent à la disposition des consommateurs qui les utilisent. Les messages sont échangés via un tampon accessible par tous les processus. Un message déposé dans le tampon peut être utilisé par n'importe quel consommateur ; il n'est utilisable qu'une seule fois.

Le modèle lecteurs-rédacteurs comporte deux classes de processus en compétition pour accéder à une section critique. Un lecteur prélève la valeur des données ; le rédacteur modifier certaines données. Le problème à résoudre est celui de la cohérence de l'ensemble des données : tout lecteur doit pouvoir lire un ensemble de données cohérent[2]. Un même message peut être lu par plusieurs lecteurs ou ne pas être lu du tout (cas de deux rédacteurs modifiant successivement la donnée).

Les deux modèles effectuent une distinction entre les processus en charge de le génération des messages (les producteurs et les rédacteurs) et les processus en charge de l'utilisation de ces messages (les consommateurs et les lecteurs). La plupart des IPC présentées dans ce document supportent une fusion des rôles : un même processus peut bien souvent à la fois produire et consommer des messages dans une communication bi-directionnelle.

Dans la suite de ce document, le modèle de communication employé sera, sauf mention contraire, celui des producteurs-comsommateurs.

[2]Un ensemble de données est cohérent entre deux interventions de rédacteur.

Chapitre 2 : IPC Unix

Les exemples et mesures de performance de ce chapitre proviendront, sauf mention contraire, du système Linux qui est un OS libre de la famille Unix.

Remarque : Les mesures de performances effectuées sur les IPC de ce chapitre suivent le protocole défini dans l'annexe 1 page 97.

2.1 Les pipes

2.1.1 Présentation

Aussi appelés tubes ou de manière plus complète tubes de communication, les *pipes* sont la plus ancienne forme de communication inter-processus du monde Unix ; ils font partie intégrante de l'OS depuis ses débuts. Le nom *pipe* vient de l'analogie que l'on peut faire entre son mode de fonctionnement et un tuyau : les données placées à l'une des extrémités ressortent dans le même ordre à l'autre extrémité (figure 2.1). Il s'agit là d'un principe simple et fiable.

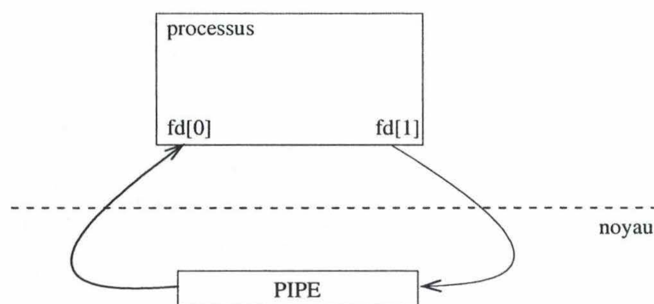

FIG. 2.1 – *pipe* Unix.

L'utilité d'une telle construction dans un cas mono-processus est quasi nulle ; par contre, lorsqu'il est utilisé en conjonction avec les appels de création de processus (*fork()*, *clone(..)*,...), elle révèle toute sa puissance (figure 2.2). Le pipe devient un canal de communication entre tous les processus partageant les descripteurs de fichier identifiant le pipe.

Un *pipe* est obtenu au moyen de l'appel système *pipe(..)* (figure 2.3). Toutes les opérations nécessaires à la création du *pipe* sont effectuées par le noyau de manière invisible à l'utilisateur.

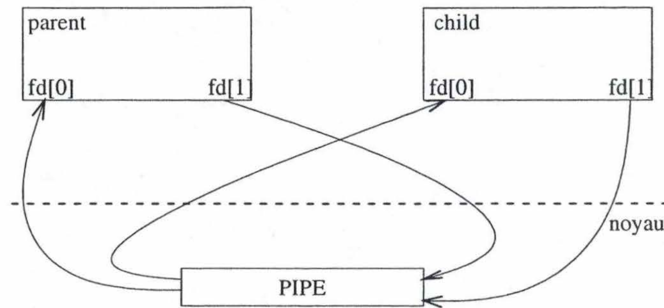

FIG. 2.2 – *pipe* Unix après un *fork()*.

```
#include <unistd.h>

int pipe(int filedes[2]);
```

FIG. 2.3 – Appel système *pipe*.

Le pipe, ainsi créé, devient un canal de communication entre tous les processus possédant les descripteurs de fichier correspondants (donc seuls les processus possédant une relation de filiation plus ou moins directe peuvent l'employer). L'envoi et la réception de messages depuis le pipe sont effectués au moyen des opérations de lecture/écriture dans un fichier (figure 2.4). Il s'agit là d'une application pratique de la philosophie sous-jacente à Unix : "*Tout peut être manipulé comme un fichier*". Cette particularité permet d'inclure la gestion des pipes dans les gestionnaires d'E/S asynchrones que sont *poll()* et *select()*. Malgré cette apparence de fichier classique, le pipe est une structure entièrement contenue dans la zone de mémoire réservée du noyau.

```
#include <unistd.h>

//ssize_t write(int fd, const void *buf, size_t count);
//ssize_t read(int fd, void *buf, size_t count);

// ecriture
int size = write(fd[1], message, sizeof(message));

// lecture
int size  = read(fd[0], message, sizeof(message));
```

FIG. 2.4 – Lecture/Ecriture dans un pipe.

Ce canal de communication est unidirectionnel : la lecture s'effectue toujours sur fd[0] et l'écriture sur fd[1]. En outre, si deux processus tentent de lire des données depuis le pipe, il ne leur est pas possible – en toute généralité – de savoir si les données présentes dans le pipe leur sont destinées ou bien si ils sont occupés à lire les données de l'autre processus. C'est pourquoi les pipes sont généralement dédoublés dans

les cas où une communication bidirectionnelle est nécessaire : chaque pipe est utilisé comme un canal de communication unidirectionnel. De cette manière, l'incertitude sur le destinataire du message est levée.

Il convient néanmoins de prendre en compte les risques de conflits entre deux accès concurrents ; afin d'éviter toute corruption des données transmises, chaque lecture/écriture devra être de taille inférieure à la constante PIPE_BUF, taille en dessous de laquelle le noyau garantit l'atomicité des écritures/lectures dans/depuis un pipe. (PIPE_BUF doit être supérieure à 512 octets selon la norme POSIX, et est habituellement de 4096 octets sous Linux). Si cette taille limite ne peut pas être garantie, il conviendra de protéger l'accès au pipe par d'autres moyens tels que des sémaphores, ou de gérer les éventuels entrelacements de messages.

La figure 2.5 illustre la manière dont le concept de pipe est implémenté : il s'agit d'une inode pointant sur une zone de mémoire gérée par le noyau (*Data Page* sur la figure 2.5). Une inode est une structure de données contenant toutes les informations sur les fichiers dans un système de fichiers Unix. Chaque fichier possède une inode et est identifié de manière unique par le système de fichiers sur lequel il réside et par le numéro d'inode sur ce système. Une inode contient les informations suivantes : le périphérique abritant le fichier (/dev/XXX), des informations de verrouillage, le mode et le type du fichier, le nombre de liens vers ce fichier, le numéro du propriétaire du fichier ainsi que de son groupe, les temps d'accès et de modification, le temps de dernière modification de l'inode, et les adresses sur le disque des blocs contenant les données du fichier.

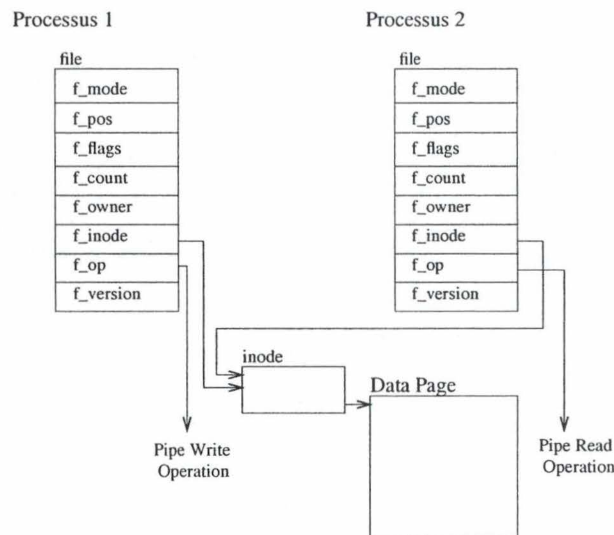

FIG. 2.5 – La structure de données sous-jacente au *pipe*.

Le pipe est représenté par une inode pointant vers une page de mémoire appartenant au noyau.

Les processus utilisant le pipe accèdent à cette zone au travers d'une structure *file* classique placée dans leurs tables de fichiers, comme pour tout autre fichier, et n'ont même pas conscience de la nature exacte de

l'inode utilisée. La seule différence avec un fichier classique est que l'inode ne pointe pas vers une zone de disque, mais bien vers la mémoire. Les transferts ne souffrent donc pas des pénalités dues aux accès disque.

2.1.2 Performances

Techniquement parlant, un pipe est un tampon d'une taille de 4096 octets[3] situés dans l'espace réservé au système. C'est le noyau qui gère l'accès à ce tampon et prévient les accès concurrents. Lorsque le tampon est plein, les appels d'écriture bloquent ou renvoient une erreur (E_AGAIN). En conséquence, les pipes sont particulièrement inadaptés aux transferts de gros volumes de données : par exemple, un message de 1Mo nécessitera que le processus émetteur et le processus receveur soient schédulés alternativement un minimum de 256 fois (chaque processus lisant/écrivant un bloc de 4Ko avant de bloquer en attendant la suite de l'opération d'entrée-sortie).

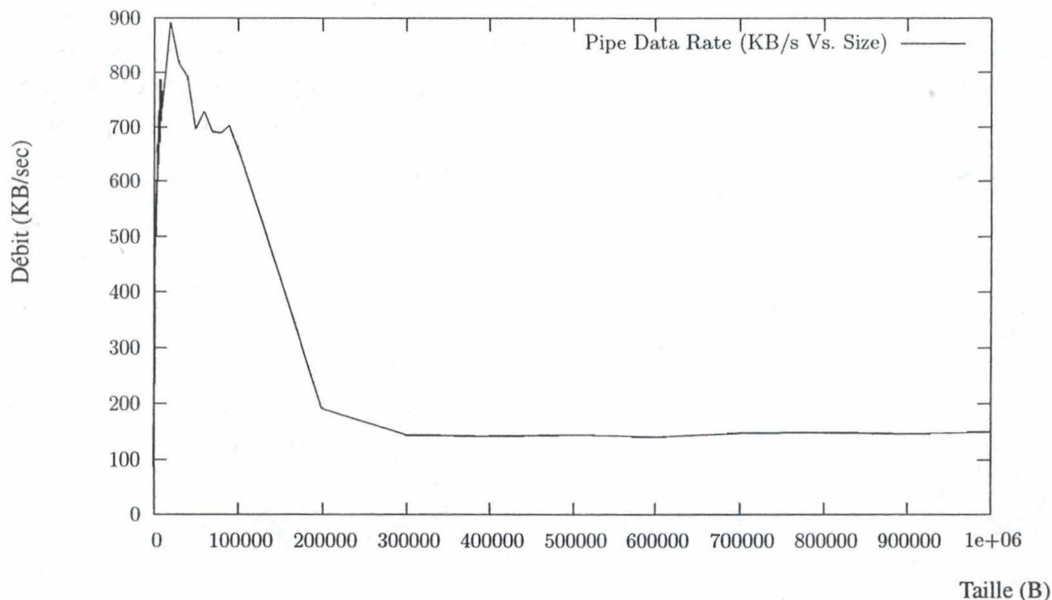

FIG. 2.6 – Débit moyen d'un pipe.

Débit moyen d'un pipe en fonction de la taille des blocs de données à transmettre.

Comme le montre la figure 2.6, les performances du transfert chutent rapidement de plus de 600kB/sec à moins de 200kB/sec dès que la quantité de données dépasse 150kB. L'utilisation des pipes est donc réservée pour le transfert de quantités relativement faibles de données, pour les transferts ponctuels ne nécessitant pas des contraintes de délai ou pour les situations où leur caractère d'*équivalent fichier* est un avantage.

[3] Cette valeur est configurable par le super-utilisateur, mais est le plus souvent fixée à la taille d'une page de mémoire (4096 octets sur les architectures x86).

2.2 Les mémoires partagées

L'idée sous-jacente à l'IPC par mémoire partagée est de fournir aux processus une zone de mémoire commune à laquelle ils peuvent librement accéder et de les laisser gérer eux-même les éventuels problèmes de concurrence (figure 2.7). Il s'agit là du strict minimum pour une IPC : le noyau donne accès à une zone utilisable par tous les processus participants et laisse le programmeur libre d'en réglementer ou non l'accès.

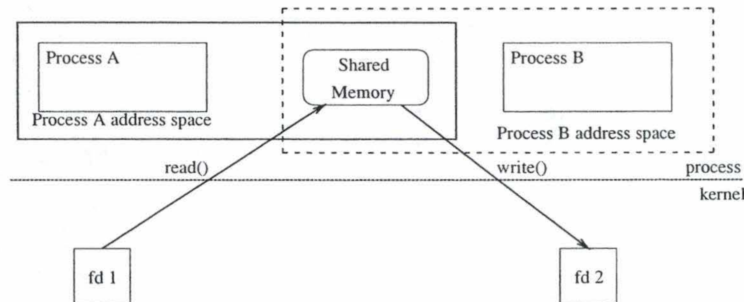

FIG. 2.7 – Principe de la mémoire partagée.

La forme la plus simple de mémoire partagée consiste à utiliser une particularité du périphérique spécial /dev/zero ; ce périphérique, aussi appelé *bit bucket* est une source infinie de 0 s'il est utilisé en lecture. Il accepte aussi les données en écriture mais ne fait que les ignorer.

```
#include <unistd.h>
#include <sys/mman.h>

void * mmap(void  *start,  size_t length, int prot,
             int flags, int fd, off_t offset);
int munmap(void *start, size_t length);
```

FIG. 2.8 – Appel système *mmap*.

Mappé en mémoire comme un simple fichier avec la fonction *mmap* (figure 2.8), ce périphérique révèle ses caractéristiques :

- Une zone de mémoire anonyme de taille *length* est crée (taille arrondie à la page mémoire près).
- Cette zone de mémoire est remplie de zéros.
- Plusieurs processus peuvent partager l'accès à cette zone si leur ancêtre commun qui a créé le mapping a spécifié MAP_SHARED dans les paramètres de création (*flags*).

Cette technique de communication est restreinte à la communication entre des processus possédant un ancêtre commun (pour l'héritage de la zone partagée).

La zone de mémoire partagée est implémentée au moyen des tables de mémoire virtuelle (figure 2.9) : les entrées de la table de mémoire virtuelle de chaque processus pointent sur une même zone de mémoire physique. Cette zone de mémoire physique est alors partagée de manière transparente aux différents processus. Le noyau se charge de maintenir un compteur du nombre de processus utilisant la zone partagée afin de ne la détruire que lorsque le dernier des processus utilisateurs se sera terminé ou lorsqu'une demande de destruction explicite aura été effectuée (appel *munmap*).

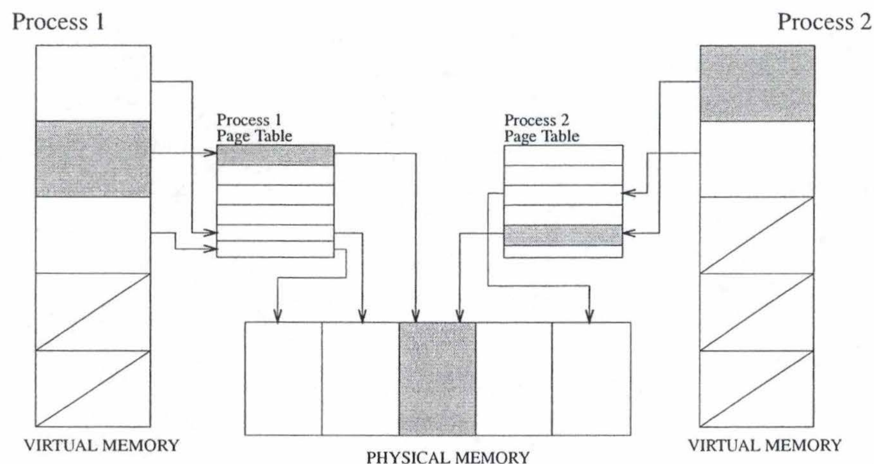

FIG. 2.9 – La structure de données sous-jacente à la mémoire partagée.

Les performances de cette IPC sont incomparables à celles des pipes : en effet, une fois la zone créée et une synchronisation efficace mise en place, les débits sont uniquement limités par la vitesse des méthodes de copie de blocs de mémoire (figures 2.10 et 2.11).

Mais, dans certaines situations, les communications se font sur de gros blocs de données (de l'ordre de plusieurs Mo) et de manière ponctuelle, de telle sorte que le processus receveur peut traiter les données directement depuis la mémoire partagée. Ce comportement rend toute mesure de performance difficile car il provoque des débits instantanés énormes (de l'ordre de 32Mo en moins d'une micro-seconde, soit un débit théorique de 32TB/sec) (figure 2.12).

Le fait que la gestion de la régulation des accès soit entièrement à la charge des processus participants est à la fois un avantage important (seules les synchronisations réellement nécessaires sont utilisées) et une grande responsabilité (le noyau ne fait aucune vérification sur les accès à cette zone et donc personne ne detectera les erreurs avant qu'elles ne produisent leurs effets).

Remarque : Une autre méthode permettant d'obtenir le même résultat fut introduite avec les systèmes BSD 4.3 : il n'est plus nécessaire d'utiliser le périphérique /dev/zero comme source si le flag MAP_ANON est spécifié. Le descripteur de fichier utilisé dans l'appel *mmap* peut alors être -1.

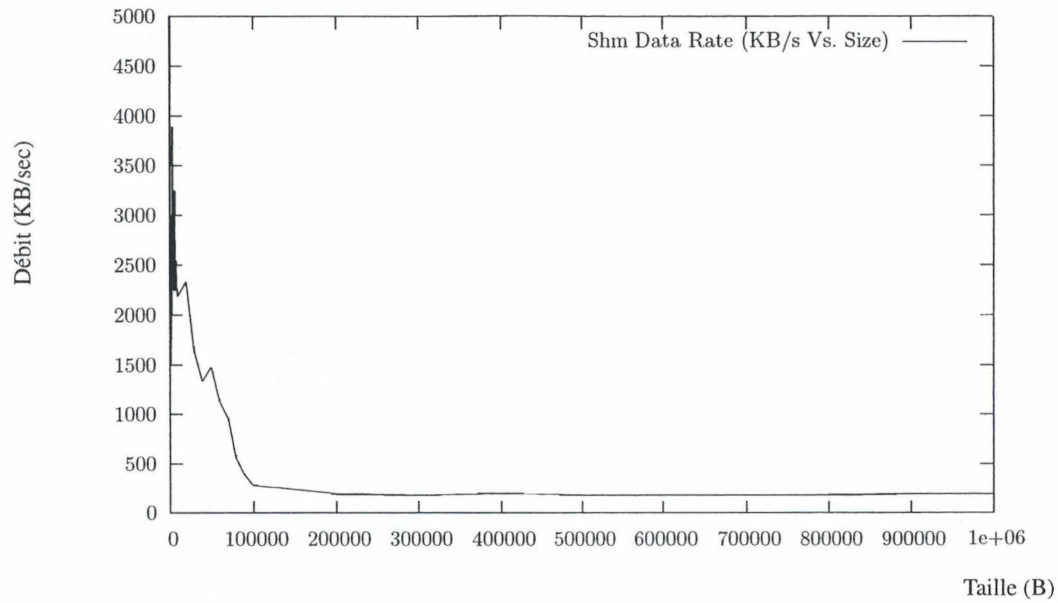

FIG. 2.10 – Débit moyen d'un transfert par mémoire partagée.

Débit moyen d'un transfert par mémoire partagée. Dans ce cas, le client copie les données dans son espace d'adressage local avant de les exploiter.

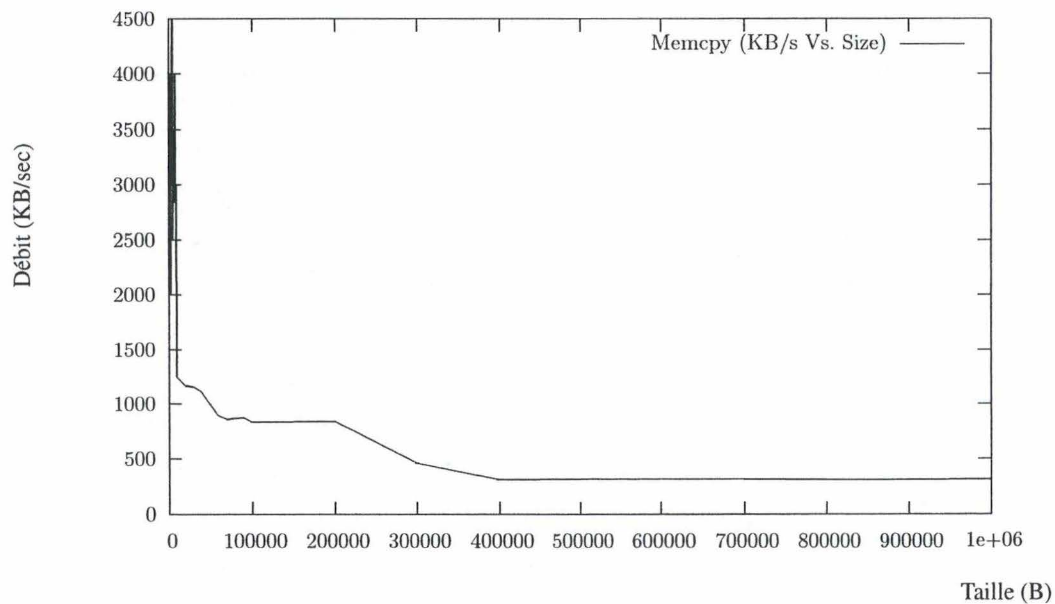

FIG. 2.11 – Performances des fonctions de copies de blocs mémoire.

Débit moyen d'une copie de bloc mémoire dans l'espace d'adressage d'un processus au moyen de la fonction memcpy(..).

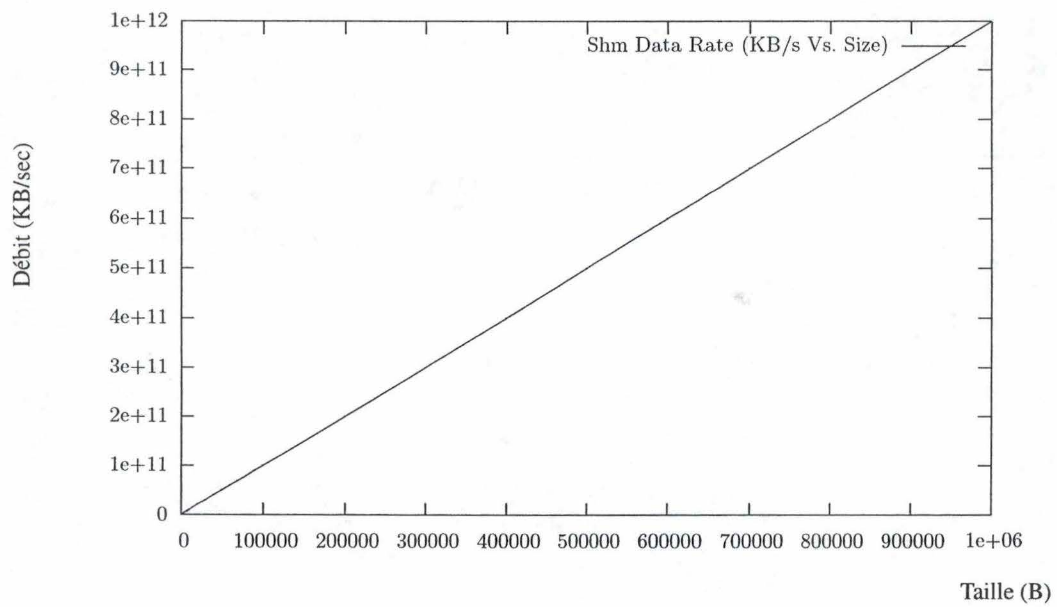

FIG. 2.12 – Débit moyen d'un transfert par mémoire partagée.

Débit moyen d'un transfert par mémoire partagée. Dans ce cas, le client exploite les données directement depuis la zone de mémoire partagée.

2.3 Les sockets (Unix / network)

Les systèmes Unix supportent deux types de sockets distincts : les sockets réseaux classiques et les sockets Unix.

2.3.1 Sockets réseaux

Les sockets réseaux sont des objets de l'OS qui permettent à un processus d'envoyer ou de recevoir des données au travers d'un réseau. Les deux modes principaux de fonctionnement des sockets sont le mode datagramme (utilisé par UDP) et le mode flux (ou *stream*, utilisé par TCP). Le mode datagramme fournit un service sans connexion et supporte le transfert de messages de taille limitée (inférieure à 64K) vers un nombre quelconque de destinataires (*broadcast*) mais ne garantit ni la séquence ni la fiabilité de la communication. Les paquets transmis peuvent donc être perdus ou altérés et leur ordre modifié lors du transfert via le réseau. Par opposition, le mode flux fournit un service orienté connexion entre deux sockets et garantit la fiabilité du flux de donnée transmis ainsi que le respect de la séquence.

```
#include <sys/types.h>
#include <sys/socket.h>

// type = SOCK_STREAM/SOCK_DGRAM
network_socket = socket(PF_INET, type, 0);

int bind(int sockfd, struct sockaddr *my_addr,
         socklen_t addrlen);
```

FIG. 2.13 – Sockets Réseaux.

Une fois créés, les sockets réseau doivent se voir attribuer une adresse (en particulier un numéro de port local) avant de pouvoir commencer à émettre ou recevoir des données ; c'est le rôle de l'appel *bind* (figure 2.13). Cette adresse sera utilisée par le noyau pour aiguiller les données reçues depuis le réseau vers le socket à qui elles sont destinées.

Les performances de cet IPC dépendent grandement de la qualité et de l'état de congestion du réseau reliant les processus en communication. En conséquence, il n'est pas possible de dégager de mesures de performances génériques en dehors du cas local présenté aux figures 2.14 et 2.15. On peut néanmoins constater un net avantage au niveau des performances pour le mode datagramme. Il s'agit là d'une conséquence de l'absence de tous les mécanismes de gestion des aléas réseau incorporés dans le mode flux (gestion des pertes de paquets et des altérations de contenu, maintient de la séquence, ...).

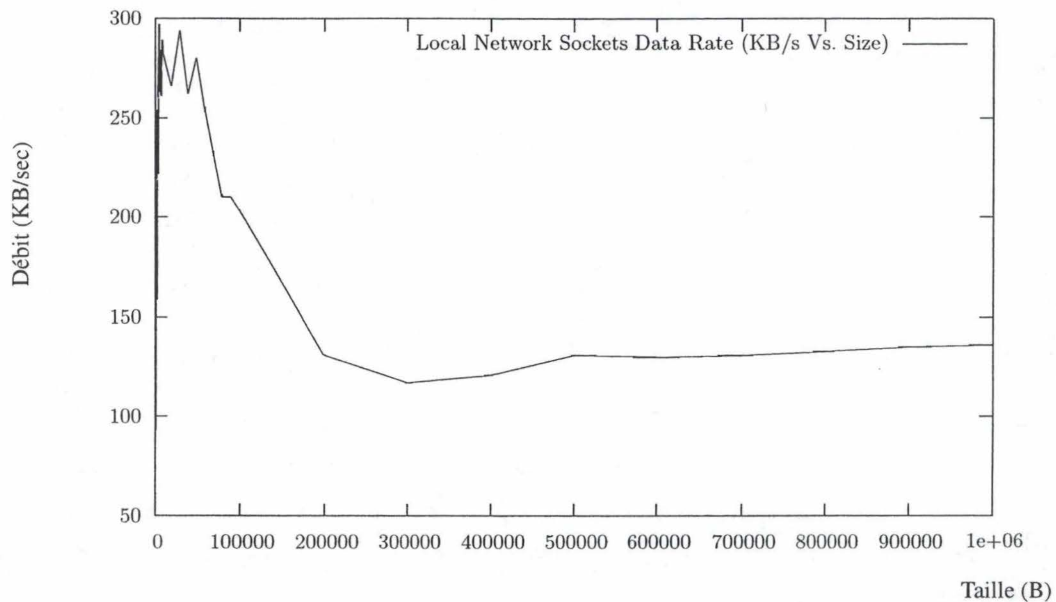

FIG. 2.14 – Débit moyen d'un transfert par socket réseau (mode flux).

Communication par sockets réseau classiques en mode flux. La communication s'effectue entièrement dans le contexte local et ne passe donc pas par le réseau.

2.3.2 Sockets Unix

Les sockets Unix sont des sockets utilisables uniquement dans un contexte local : leur portée est limitée à la machine locale. Leur principal usage est dans la communication efficace entre processus locaux.

```
#include <sys/socket.h>
#include <sys/un.h>

unix_socket = socket(PF_UNIX, type, 0);
error = socketpair(PF_UNIX, type, 0, int *sv);
```

FIG. 2.16 – Sockets Unix

Les sockets Unix peuvent être de deux types : anonymes ou nommés. Les sockets anonymes sont créés au moyen de la fonction *socketpair(..)* (figure 2.16) et se transmettent à la manière des pipes : tous les processus possédant les descripteurs de fichiers du socket peuvent l'utiliser. Par opposition, les sockets nommés sont identifiés par un nom virtuel indépendant du système de fichiers, ce nom se situe dans un espace de nommage uniquement utilisé pour les sockets nommés. Les processus peuvent alors se connecter à ce socket nommé avec la fonction *connect(..)* comme ils le feraient avec un socket réseau classique.

Les sockets Unix se manipulent comme des sockets normaux et supportent les communications en mode

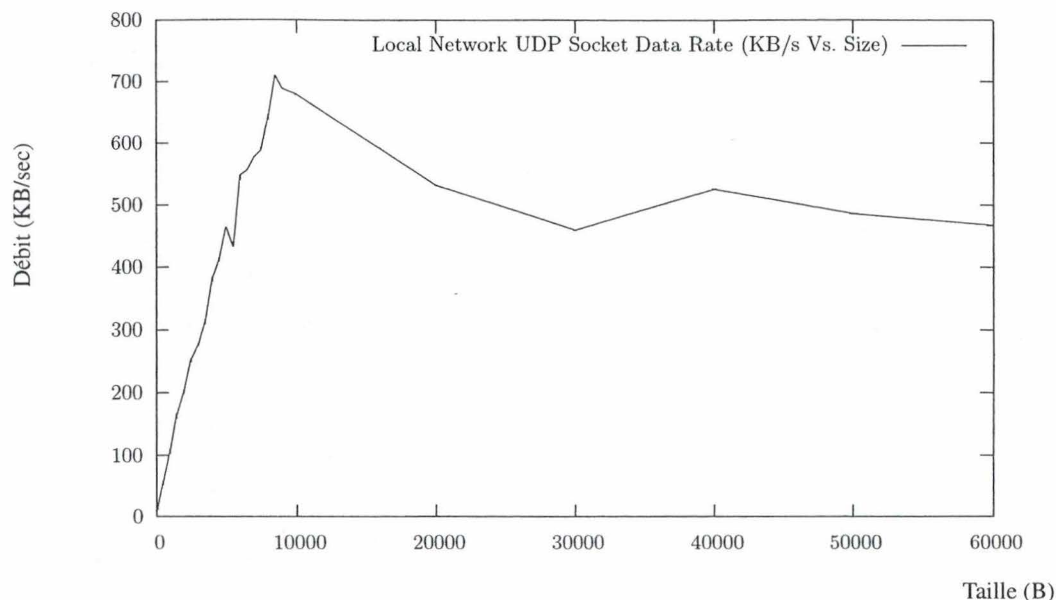

FIG. 2.15 – Débit moyen d'un transfert par socket réseau (mode datagramme).

Communication par sockets réseau classiques en mode datagramme. La communication s'effectue entièrement dans le contexte local et ne passe donc pas par le réseau. Le mode datagramme limite la quantité maximale de données transmises à 65507 bytes par paquet (64K moins la taille des différentes en-têtes nécessaires).

flux (SOCK_STREAM comme pour TCP) ou en mode datagramme (SOCK_DGRAM comme pour UDP). Néanmoins, en mode datagramme, les sockets Unix sont plus fiables que les sockets classiques, dans le sens où les sockets Unix garantissent le respect de la séquence ainsi que la fiabilité des transmissions.

En outre, les sockets Unix apportent plusieurs fonctionnalités absentes des sockets réseaux habituels :

- Une méthode d'authentification : un message spécial (`SCM_CREDENTIALS`) permet à un processus d'envoyer des informations d'identification. Tous les processus en communication étant localisés sur la même machine, ils partagent tous la même méthode d'authentification régulée par le noyau (sous Unix, le PID, l'identifiant du propriétaire ainsi que de son groupe). Le noyau garantit que seuls les processus du super-utilisateur puissent modifier les valeurs d'authentification qu'ils transmettent.
- Une méthode de transfert de descripteurs de fichier : un processus peut transmettre un ensembe de descripteurs de fichier à un autre processus, le noyau se charge alors de modifier en conséquence les tables de descripteurs des processus concernés.
- Une capacité de distribution quasi immédiate : en transformant le PF_UNIX de l'appel de création en PF_INET, le socket se transforme en socket réseau et l'application peut être distribuée. La seule contrainte est que les deux autres fonctionnalités additionelles ne sont pas disponibles dans le cas distribué.

Enfin, dans le but de faciliter les accès aux sockets ainsi créés, il est possible de lier les sockets nommés avec un nom disponible sur le système de fichiers local : la méthode *bind(..)* permet d'assigner un nom de

fichier au socket. Il est alors de la responsabilité du processus ayant créé ce socket d'effacer le fichier au moment de la destruction du socket.

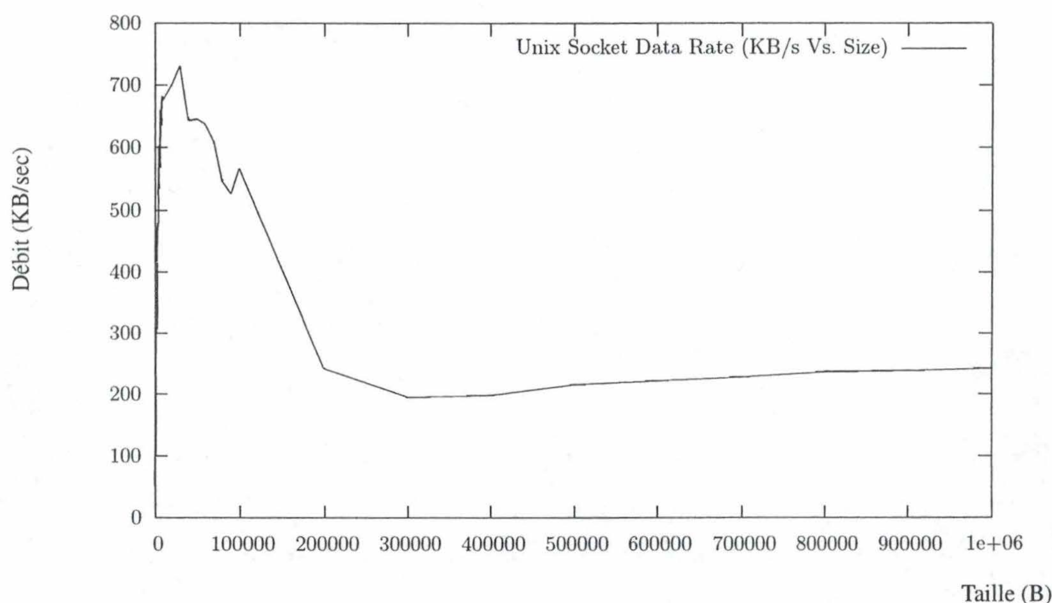

FIG. 2.17 – Débit moyen d'un transfert par socket Unix.

Le transfert est effectué au moyen de sockets Unix en mode flux (SOCK_STREAM).

Les performances de cette méthode de communication (figure 2.17) sont un peu supérieures à celles des pipes. A titre de comparaison, la figure 2.14 illustre les performances des sockets réseaux classiques lors d'une communication entre deux processus locaux. Les performances des sockets Unix sont légèrement supérieures à celles des sockets classiques. Cette différence est principalement due à la présence des mécanismes en charge de la gestion des aléas réseau dans le cas des communications par sockets réseaux.

2.4 Les fichiers

Cette partie présente quelques IPC utilisant comme support des fichiers (spéciaux ou non) situés sur le système de fichiers.

2.4.1 Les 'fifos'

Les 'fifos' [4], aussi appelés tubes nommés (named pipes) sont une extension des pipes classiques : là où les pipes ne permettent la communication qu'entre des processus liés par une relation de filiation plus ou moins

[4]First In First Out : file d'attente fournissant les éléments dans l'ordre de leur arrivée.

directe (pour le transfert des descripteurs de fichier associés aux extrémités du pipe), les fifos permettent à deux processus quelconques d'entrer en communication.

La création d'un fifo est similaire à la création d'un fichier, toutes les opérations nécessaires sont encapsulées dans un appel système unique : *mkfifo(..)* (figure 2.18).

```
#include <sys/types.h>
#include <sys/stat.h>

int mkfifo(const char* pathname, mode_t mode);
    // renvoie 0 si OK, -1 si erreur
```

FIG. 2.18 – Création d'un 'fifo'.

Après un tel appel, un fichier particulier *pathname* existe dans le système de fichiers. Ce fichier spécial se manipule comme un fichier classique et supporte toutes les fonctions d'entrée-sortie standards. Tous les processus possédant le nom (*pathname*) du fifo peuvent l'utiliser en écriture ou en lecture (modulo les permissions d'accès).

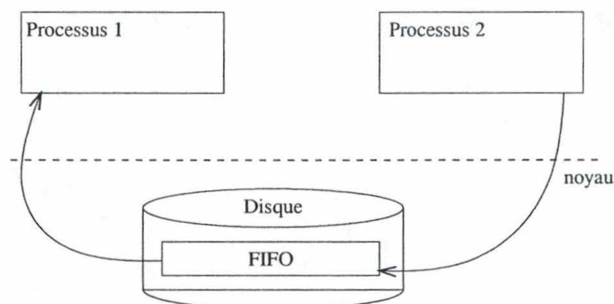

FIG. 2.19 – Principe de fonctionnement d'un fifo.

Afin d'éviter des débordements du fifo, l'écriture sur un fifo ne possédant pas de lecteur génèrera un signal SIG_PIPE. De même le retrait du dernier rédacteur provoquera l'envoi d'un caractère de fin du fichier aux lecteurs dans le but de leur éviter d'attendre indéfiniment de nouvelles données.

La manière d'utiliser un fifo dépend de la valeur du paramètre non-bloquant spécifié lors de l'ouverture :

1. Si les accès sont bloquants (mode par défaut ou *flag* O_NONBLOCK non spécifié), une ouverture du fifo en lecture sera bloquante jusqu'à l'arrivée d'un premier rédacteur. Symétriquement, une ouverture du fifo en écriture bloquera le processus jusqu'à l'apparition d'un lecteur.

2. Si les accès sont non-bloquants, une ouverture en lecture réussira immédiatement, tandis qu'une ouverture en écriture provoquera une erreur si aucun processus lecteur n'est connecté au fifo.

Le cas non-bloquant est très utilisé dans un contexte client-serveur local : il permet à un serveur d'ouvrir un fifo possédant un nom 'bien connu' et de s'y connecter en tant que lecteur afin d'attendre la connexion de 'clients' sur ce fifo.

Il convient néanmoins de prendre en compte les risques de collisions entre deux demandes de clients concurrents ; afin d'éviter toute corruption des messages transmis, ceux-ci devront être de taille inférieure à la constante PIPE_BUF, taille en dessous de laquelle le noyau garantit l'atomicité des écritures/lectures depuis un fifo. (PIPE_BUF doit être supérieure à 512 octets selon la norme POSIX, et est habituellement de 4096 octets sous Linux).

Les performances des fifos sont en tout points comparables à celles des pipes ; en effet ces deux IPC reposent sur les même structures internes. Seule l'interface d'accès diffère entre les fifos et les pipes.

2.4.2 Les périphériques spéciaux (/dev, /proc)

Sous Unix, la convention est de représenter tous les éléments du système par des fichiers. En particulier, les périphériques de l'ordinateur sont associés à des fichiers (figure 2.20) : par exemple, la souris est */dev/mice*, l'horloge interne est */dev/rtc* ou encore la mémoire du noyau est */dev/kmem*. Toute lecture ou écriture sur l'un de ces fichiers est immédiatement répercutée sur le périphérique cible : une lecture sur */dev/rtc* fournit l'heure (en notation binaire) ou une écriture sur */dev/kmem* permet de fixer une valeur précise dans le noyau (en cours d'exécution). Les accès à ces périphériques sont réglementés via le même système de permissions que les fichiers standards.

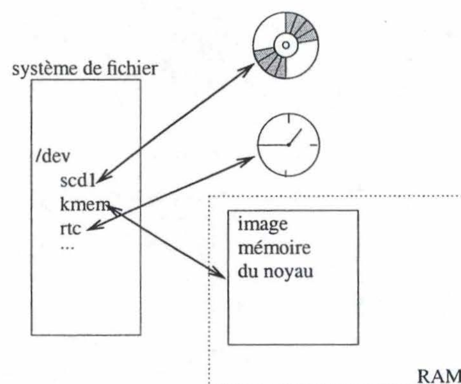

FIG. 2.20 – Accès au périphériques par */dev*.

Afin de fournir une interface plus simple et moins risquée pour la modification *à chaud* (c'est-à dire sans arrêt puis redémarrage) des valeurs à l'intérieur du noyau, le périphérique spécial */proc* a été développé : là où */dev/kmem* représente un énorme bloc binaire (l'image du noyau en RAM), */proc* fournit à l'utilisateur une arborescence de variables (figure 2.21). Il est ainsi beaucoup plus simple de consulter/modifier une valeur : au lieu de devoir tenter un accès absolu (lecture/ecriture de la zone mémoire à l'adresse X), le noyau

effectue lui-même la correspondance entre le nom de la variable demandée et la position de celle-ci dans son espace d'adressage. Les risques d'erreurs sont alors très fortement diminués (surtout en cas d'écriture). Pour l'instant, seul le noyau utilise cette technique de communication ; de surcroit, le super-utilisateur est bien souvent le seul à disposer des permissions nécessaires afin de lire ou d'écrire directement depuis/dans le noyau.

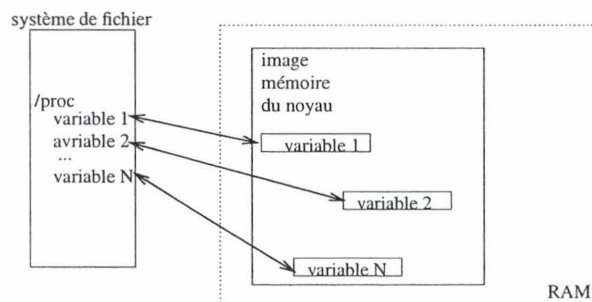

FIG. 2.21 – Accès au noyau par */proc*.

Seules les parties du noyau spécialement prévues pour autoriser la consultation/modification à chaud sont maintenant accessibles.

Ces deux techniques sont des IPC un peu particulières dans le sens où il ne s'agit plus vraiment d'une communication entre deux processus avec une intervention régulatrice du noyau, mais plutôt de la modification directe du processus cible par le noyau en réponse à la demande d'un autre processus. Le noyau est donc passé du rôle de simple surveillant à celui d'intervenant à part entière.

Ces IPC ne se prêtent pas bien aux mesures de performances dans la mesure où elles sont réservées à des applications très spécifiques (comme la communication avec les périphériques ou avec le système) et à des besoins très particuliers (comme la consultation/modification d'une valeur directement dans le noyau pour /proc).

2.4.3 Les fichiers standards

La communication par fichier repose sur la capacité qu'ont les différents processus d'accéder au système de fichiers local. Son principe de base est le suivant (figure 2.22) :

1. Le processus source copie les données à transférer dans un fichier au moyen des primitives standards d'accès aux fichiers.
2. Le processus cible récupère (par une autre méthode IPC ou par une convention commune) le nom et le chemin d'accès du fichier. Il peut ensuite lire et exploiter les données contenues dans le fichier. (ce principe est facilement étendu au cas de lecteurs multiples).

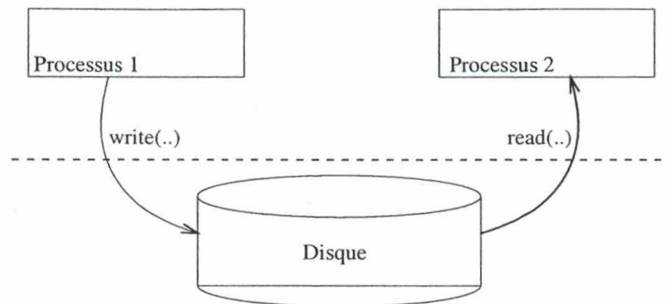

FIG. 2.22 – IPC par fichier standard.

Cette méthode à l'avantage d'être très simple à mettre en oeuvre, de supporter des transferts de très gros volumes de données (à priori, la taille des fichiers est illimitée) et d'être utilisable même dans le cas où les deux processus s'exécutent à des prériodes de temps disjointes. Néanmoins, cette méthode est rarement utilisée pour les transferts entre deux processus ; en effet, outre le fait que le passage par le disque introduit une lourde pénalité au niveau des performances, il revient aussi aux processus la tâche de synchroniser leurs accès au fichier.

Les mesures de performances sur cette IPC n'ont que peu de sens dans une approche générique. En effet, les performances obtenues dépendent en grande partie de la configuration matérielle et de la configuration logique du système de fichier. Ainsi un transfert via un système de fichier "simple" (comme le système de fichier FAT) implanté sur un disque dur rapide (par exemple un disque dur à 10000 rpm) obtiendra de meilleures performances que le même transfert via un système de fichier journalisé (comme ext3) implanté sur un disque dur meilleur marché (par exemple un disque dur à 5400 rpm).

2.5 Les signaux

Les signaux sont l'une des plus anciennes formes de communication inter-processus du monde Unix. Ils permettent d'envoyer des évènements asynchrones à un ou plusieurs autre processus. Les signaux peuvent par exemple être générés par des interruptions clavier (des combinaisons de touches comme Ctrl-C ou Crtl-Z), par le noyau (SIGSEV en cas de faute d'adressage mémoire, SIGILL pour une instruction invalide, SIGCHLD pour la terminaison d'un processus fils...) ou par un processus.

Le nombre minimal de signaux dans la norme POSIX est de 32, et chacun correspond à un message précis, à l'exception des signaux SIGUSR1 et SIGUSR2 qui sont laissés à la disposition de l'utilisateur. Nous ne détaillerons pas ici la sémantique de chacun d'entre eux ; celle-ci est disponible sur la *manpage* signal(7). Les numéros associés aux signaux peuvent varier d'une architecture à une autre.

1)	SIGHUP	2)	SIGINT	3)	SIGQUIT	4)	SIGILL
5)	SIGTRAP	6)	SIGABRT	7)	SIGBUS	8)	SIGFPE
9)	SIGKILL	10)	SIGUSR1	11)	SIGSEGV	12)	SIGUSR2
13)	SIGPIPE	14)	SIGALRM	15)	SIGTERM	17)	SIGCHLD
18)	SIGCONT	19)	SIGSTOP	20)	SIGTSTP	21)	SIGTTIN
22)	SIGTTOU	23)	SIGURG	24)	SIGXCPU	25)	SIGXFSZ
26)	SIGVTALRM	27)	SIGPROF	28)	SIGWINCH	29)	SIGIO
30)	SIGPWR	31)	SIGSYS	32)	SIGRTMIN		

FIG. 2.23 – Signaux Linux x86.

A chaque signal est associé un comportement par défaut, qui indiquera à l'OS comment se comporter si un processus reçoit le signal et ne le gère pas. Par exemple, le comportement par défaut pour un SIGSEV (faute de segmentation) est d'arrêter le processus en construisant un fichier core (fichier qui reprend l'image mémoire du processus au moment de la faute, ce fichier est ensuite exploitable au moyen de débuggeurs), tandis que le comportement par défaut pour un SIGCHLD (terminaison de processus fils) est de ne rien faire. Ces comportements peuvent être redéfinis par chaque processus, sauf pour les signaux SIGSTOP et SIGKILL qui indiquent au processus de se terminer et imposent cet arrêt si le processus ne se termine pas assez vite.

Les processus disposent de 3 manières de gérer les signaux reçus (figure 2.24) ; ils peuvent soit les ignorer (S1), soit les intercepter (S2), soit les bloquer (S3).

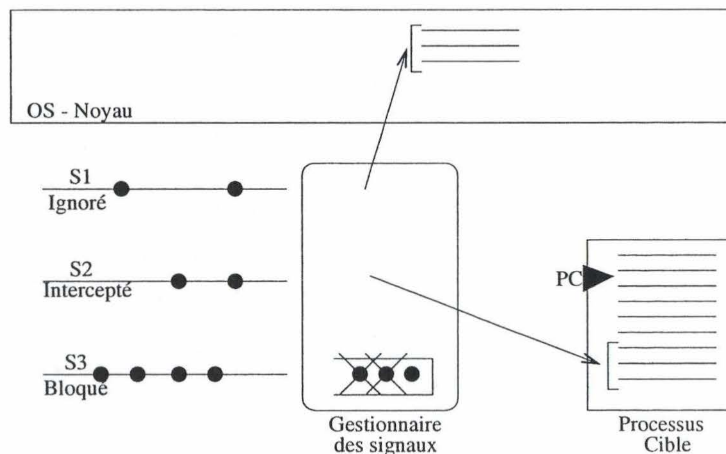

FIG. 2.24 – Actions des signaux.

Un signal est dit ignoré (S1) lorsque le processus informe le noyau qu'il ne désire pas gérer lui-même les actions à entreprendre lors de la réception du signal ; c'est donc le noyau qui va exécuter l'action par défaut. Ce comportement est celui par défaut pour tous les signaux au début d'un programme : sans indication contraire, tous les signaux sont affectés à leur comportement par défaut.

Lorsqu'un signal est intercepté (S2), le processus indique au noyau qu'il doit exécuter une fonction du programme lors de la réception d'un signal. Le processus est donc 'à l'écoute' du signal. A la réception de ce signal, l'exécution du processus sera suspendue pendant la durée de l'exécution de la fonction spécifiée dans la demande d'interception ; ensuite l'exécution du processus reprendra à l'endroit où elle a été interrompue (PC).

Enfin, un processus bloque un signal (S3) lorsqu'il indique au noyau que le signal en question ne doit pas être traité du tout : ni le processus ni le noyau ne doivent entreprendre la moindre action lors de la réception de ce signal. Le signal reste donc 'en attente' dans le gestionnaire de signaux et ne sera délivré au processus que lorsque celui-ci débloquera le signal. Ce procédé permet de protéger certaines sections de code contre les perturbations qui pourraient intervenir si l'exécution était interrompue par un signal.

Ces comportements sont spécifiés au moyen de l'appel système *signal(..)* (figure 2.25) : pour chaque signal, le programmeur peut choisir entre les trois actions : en positionnant *handler* à SIG_IGN, le signal sera bloqué, avec SIG_DFL le signal sera ignoré et si *handler* est la référence d'une fonction du programme, alors cette dernière sera appelée lors de la réception du signal correspondant.

```
#include <signal.h>

typedef void (*sighandler_t)(int);

sighandler_t signal(int signum, sighandler_t handler);
```

FIG. 2.25 – Appel système *signal(..)*.

Il faut signaler ici que les signaux sont un moyen de communication très primitif et souffrent de deux défauts majeurs : d'abord le gestionnaire de signaux ne gère pas de priorité relative entre les signaux, ce qui signifie que deux signaux générés en même temps seront traités par le processus dans un ordre aléatoire, et ce même si un des deux signaux indique un comportement considéré comme prioritaire (une faute de segmentation par exemple). Le second défaut est que le gestionnaire de signaux ne possède pas de mécanisme pour gérer les signaux multiples : si un même signal se présente à trois reprises avant que le processus ne puisse le traiter, ce dernier ne recevra qu'une seule notification de signal, ce qui, dans l'optique d'une communication inter-processus, peut conduire à une perte d'informations.

Un autre défaut, plus technique, provient des premières implémentations des fonctions de gestion de signaux : ces premières implémentations repositionnaient le gestionnaire de signaux sur le comportement par défaut après chaque interception de signal. Il était donc nécessaire que la fonction utilisateur de gestion des signaux soit ré-armée à chaque appel sous peine de ne plus recevoir les notifications ultérieures pour ce signal. Le problème majeur de ce comportement était que les arrivées rapprochées de deux signaux identiques pouvaient conduire à ce que le premier déclenche la fonction utilisateur d'interception, et que le second soit

géré par le comportement par défaut du noyau car la fonction utilisateur n'avait pas encore eu le temps de se ré-armer. La plupart des implémentations actuelles corrigent ce problème en bloquant tous les signaux pendant la durée de l'exécution de la fonction utilisateur d'interception. Certains systèmes implémentent même le ré-armement automatique de la fonction utilisateur.

Tous les processus ne peuvent pas émettre de signaux vers tous les autres processus : seuls les processus du super-utilisateur et ceux du propriétaire du processus cible peuvent émettre des signaux vers ce dernier.

Les signaux sont un moyen de communication asynchrone dans le sens où le moment où un processus émet un signal ne correspond pas toujours au moment où le processus cible reçoit ce signal. Outre le cas des processus ayant délibérement choisi de bloquer certains signaux, les processus occupés dans l'exécution d'un appel système ne reçoivent les signaux qu'à la sortie de celui-ci. Par exemple, un processus bloqué dans un appel système de lecture/écriture ne recevra les signaux qui lui sont envoyés que lorsque cet appel se sera terminé et que le processus sera de nouveau schédulé.

Emettre un signal vers un processus est relativement simple : il suffit de connaître l'identifiant du processus cible (son PID) et d'appeler la fonction *kill(..)* (figure 2.26). Une telle communication d'information est très pratique car toujours non-bloquante (les signaux s'accumulent dans le masque du receveur) et très rapide (le receveur est notifié du signal dès qu'il redevient exécutable). Néanmoins, seules des informations très simples peuvent transiter de cette manière : un signal permet d'envoyer une information telle que "*il est maintenant temps de ...*". De surcroit, seuls deux signaux (SIGUSR1 et SIGUSR2) sont disponibles pour l'utilisateur. L'utilisation d'autres signaux implique qu'il faut en conserver la sémantique (par exemple envoyer un SIGTERM à un processus pour lui demander de s'arrêter proprement) ou prendre le risque de mal réagir si ce signal est produit suite à une erreur. Par exemple, même si la réutilisation de SIGFPE (qui indique une faute de calcul en virgule flottante) sans conservation de la sémantique associée peut être envisagée dans un programme n'effectuant pas de calculs en virgule flottante, ce choix met le programme sous la menace d'un signal 'extérieur'. En effet, ce signal peut parfois être déclenché par une librairie utilisée par le programme (actuellement ou dans une évolution future de cette librairie[5]), ou encore par le noyau en cas de problème matériel dans l'unité de calcul en virgule flottante.

```
#include <sys/types.h>
#include <signal.h>

int kill(pid_t pid, int sig);
```

FIG. 2.26 – Appel système *kill(..)*.

Cette IPC ne se prête pas aux mesures de performances dans la mesure où les informations véhiculées sont très simples (équivalentes à un changement d'état binaire) et instantanées (du point de vue de l'émetteur et

[5]Et donc le problème de 'faux message' peut parfois n'apparaître que très longtemps après la réalisation du programme

du receveur, même si le temps entre l'émission et la réception est non-nul).

2.6 L'interface SYSV

L'interface Système V comporte trois méthodes IPC et est apparue avec le système Unix System V en 1983. Elle est maintenant généralisée à la plupart des autres versions Unix. La philosophie de cette interface est d'unifier les méthodes d'accès aux IPC et de permettre à des processus ne possédant pas de relation de filiation de communiquer entre eux de manière simple et fiable.

Ainsi, les objets des trois méthodes ne sont plus accédés au moyen de pointeurs, descripteurs de fichier ou autre, mais au moyen d'une clé identifiante unique et d'un appel permettant de 'traduire' cette clé dans la structure de plus bas niveau nécessaire à l'utilisation de l'objet.

Les méthodes de communication Sys V étant globales à tous les processus du système, il convenait de construire un système de protection afin que tout un chacun ne puisse pas accéder aux données transmises. Le choix qui fut effectué fut de reprendre celui des permissions de fichier : tout comme les fichiers, les objets de communication Sys V possèdent des droits (lecture, écriture et exécution pour l'utilisateur, les membres de son groupe et pour le monde entier) qui sont fixés à la création de l'objet.

Ce choix garantit une sécurité importante, mais pas parfaite : en effet, un processus externe ne peut pas connaître le contenu de la communication entre deux processus, mais il peut savoir qu'il y a communication, ce qui, dans certains cas, peut représenter un risque de sécurité. Ce comportement est une conséquence de la gestion d'un compteur de références dans les structures de données sous-jacentes : chaque processus utilisant ces ressources est comptabilisé.

2.6.1 Les mémoires partagées

Les zones de mémoire partagée du Sys V fonctionnent de manière analogue aux zones de mémoire partagée 'classique' (construites par l'appel *mmap(..)*). Les principales différences proviennent de la manière dont ces zones sont créées et surtout détruites.

Lors de la création des zones de mémoire partagée, il n'est plus nécessaire de fournir un fichier (comme /dev/zero) à mapper en mémoire ; la fonction *shmget(..)* (figure 2.27) permet de créer à la demande une zone de mémoire de la taille demandée[6] et renvoyer au processus la clé numérique identifiant la zone nouvellement créée.

[6]La taille d'un bloc de mémoire partagée doit toujours être inférieure à 32Mo. Il s'agit là d'une conséquence des choix de design posés en 1983, époque à laquelle 32Mo paraissait être une '*limite inaccessible*' [Stevens, 1993].

```
#include <sys/ipc.h>
#include <sys/shm.h>

int shmget(key_t key, int size, int shmflg);
```

FIG. 2.27 – Appel système *shmget(..)*.

Tous les processus désirant utiliser la zone ainsi créée doivent alors s'y attacher (figure 2.28), c'est-à dire indiquer au noyau que la zone de mémoire partagée doit être insérée dans l'espace d'adressage virtuel du processus. Il est très important de noter ici que la même zone de mémoire partagée peut être placée à deux adresses différentes dans les espaces d'adressage de deux processus différents ; en conséquence, le programmeur doit porter une attention toute particulière à la gestion des références mémoires vers cette zone. Par exemple, si le processus 1 copie une liste chaînée dans la zone partagée, le processus 2 ne peut pas être sûr que les pointeurs entre les différents éléments de la liste soient corrects. Il convient donc que tous les objets placés dans cette zone utilisent un adressage relatif de la forme *base + offset*, de manière à ce que chacun puisse adapter les références à son espace particulier.

```
#include <sys/types.h>
#include <sys/shm.h>

void *shmat ( int shmid, const void *shmaddr, int shmflg )

int shmdt ( const void *shmaddr)
```

FIG. 2.28 – Appels système *shmatt(..)* et *shmdt(..)*.

Les ressources utilisées pour créer la zone de mémoire partagée sont des ressources de portée globale ; c'est à dire que ces ressources sont allouées par le noyau dans un zone mémoire indépendante de celles de processus. Il convient donc de porter un soin tout particulier à leur destruction après utilisation car si aucun des processus utilisateurs ne détruit la zone de mémoire partagée avant de se terminer, la zone continuera à exister sur le système jusqu'à sa destruction manuelle (au moyen de la commande *ipcrm*) ou jusqu'au redémarrage de la machine.

```
#include <sys/ipc.h>
#include <sys/shm.h>

int shmctl(int shmid, int cmd, struct shmid_ds *buf);
```

FIG. 2.29 – Appel système *shmctl(..)*.

La fonction *shmctl(..)* (figure 2.29) est en quelques sorte le 'couteau suisse' des zones de mémoire partagée Sys V : cette fonction autorise toutes les opérations de gestion (modification des permissions, verrouillage,

déverouillage, ...) et de destruction (immédiate, à l'abandon, ...) sur la zone de mémoire. Ce genre de fonction à usages multiples est une caractéristique de l'interface Sys V où une seule fonction à sémantique et paramètres variables contrôle tous les paramètres d'un objet.

Les performances des mémoires partagées créées au moyen de l'interface Sys V sont identiques à celles des mémoires partagées 'classiques'. En effet, les deux méthodes sont implémentées de manière identique au moyen des tables de mémoire virtuelle. Les différences entre les deux approches se situent principalement au niveau des fonctions utilisées pour la création/gestion/destruction de la zone partagée.

2.6.2 Les sémaphores

Un sémaphore peut être vu comme une barrière autorisant un certain nombre de processus à la traverser avant de devenir bloquante. Il s'agit donc d'un dispositif utilisé pour réguler les accès concurrents à des ressources partagées. L'utilisation la plus fréquente du sémaphore est l'implémentation de sections critiques, qui sont des sections de code ou de données auxquelles au plus un processus peut accéder à un instant donné. L'interface Sys V est un peu particulière dans le sens où elle ne donne pas accès directement aux sémaphores, mais plutôt à des tableaux de sémaphores.

Le fonctionnement d'un sémaphore est très simple : une fois créé et initialisé, le sémaphore s'apparente à un compteur fixant le nombre maximum de processus pouvant résider dans la section critique ; chaque processus désirant y entrer effectue un appel au sémaphore, qui autorise ou non l'entrée. L'opération d'autorisation d'entrée doit être implémentée de manière atomique (i.e. doit pouvoir s'exécuter entièrement ou pas du tout sans aucune interférence avec d'autres processus) ; il s'agit là du rôle du noyau que de garantir cette atomicité. L'entrée d'un processus en section critique correspond donc à décrémenter d'une unité la valeur du compteur, tandis que la sortie est l'ajout d'une unité au compteur. Les processus qui n'ont pas été autorisés à entrer en section critique (car d'autres processus s'y trouvent déjà) sont bloqués et placés en attente. Ils seront débloqués dès que de nouvelles opportunités d'entrée se présenteront. Il faut noter ici que la spécification est muette sur la manière dont les processus en attente doivent être ordonnés ; rien ne garantit que cette gestion soit équitable.[7]

Comme pour les mémoires partagées, l'utilisation des sémaphores repose sur trois fonctions (figure 2.30) : une fonction de création (*semget(..)*), une fonction de contrôle (*semctl(..)*) et une fonction de manipulation (*semop(..)*). Les opérations de création et de contrôle sont très similaires à celles sur les mémoires partagées.

[7] Et donc un processus placé en attente peut y rester indéfiniment si la file d'attente n'est jamais vide. Il s'agit d'un cas de famine.

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>

union semun {
  int val;                      // value for SETVAL
  struct semid_ds *buf;         // buffer for IPC_STAT/SET
  unsigned short int *array;    // array for GETALL,SETALL
  struct seminfo *__buf;        // buffer for IPC_INFO
};

int semget(key_t key, int nsems, int semflg )
int semop(int semid, struct sembuf *sops, unsigned nsops)
int semctl(int semid, int semnum, int cmd, union semunarg)
```

FIG. 2.30 – Appels système *semget(..),semctl(..)* et *semop(..)*.

Les sémantiques données à ces différentes fonctions ont provoqué ce que certains auteurs appelent "le défaut fatal des sémaphores" ("fatal flaw of semaphores" [Stevens, 1993]) : il n'est pas possible de créer et d'initialiser un sémaphore en une seule opération, ce qui peut conduire à des entrées non autorisées en section critique. En effet, lors de sa création, le compteur interne du sémaphore est, en toute généralité, non-initialisé[8].

Remarque : Comme avec toutes les primitives de synchronisation, il convient de prendre en compte les risques d'interblocage entre processus : deux processus possédant chacun un sémaphore et attendant celui possédé par l'autre resteront bloqués dans cet état jusqu'à une éventuelle intervention extérieure.

2.6.3 Les files de messages

Comme leur nom l'indique, les files de messages sont des files contenant un ensemble de messages formatés. Plusieurs processus peuvent à la fois lire ou écrire dans ces files des messages à condition de synchroniser leurs accès (au moyen de sémaphores par exemple).

Chaque message de la file contient plusieurs informations (figure 2.31) : un *type* qui permet au receveur de connaître la structure interne du message transmis, une information de temps qui indique l'heure à laquelle le message à été placé dans la file et le message en lui-même. La taille du message est limitée à 8192 octets (constante définie lors de la compilation du noyau). En outre, la taille totale de la file est limitée par certains systèmes (dont Linux) afin de ne pas surcharger les ressources globales avec des messages en attente. En conséquence, un processus plaçant un message dans une file de message peut être bloqué le temps que la file se vide suffisament pour inclure le nouveau message.

[8] La plupart des OS l'initialisent à 0 afin d'éviter le problème, mais brisent par là la spécification des fonctions de gestion des sémaphores.

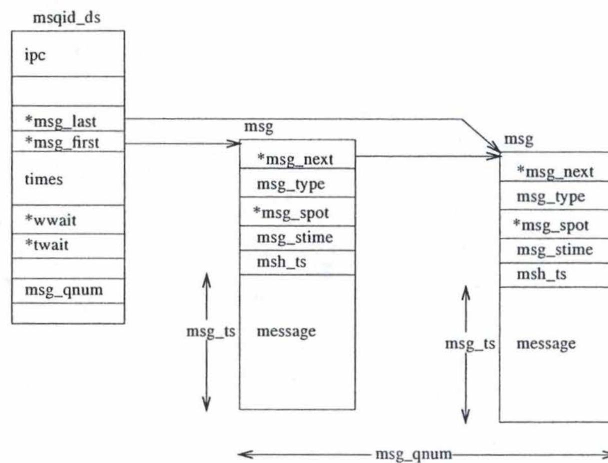

FIG. 2.31 – La structure de données sous-jacente à une file de messages.

La gestion des files de messages est, elle aussi, effectuée au moyen de fonctions 'de style Sys V' (figure 2.32). La fonction de manipulation a ici été séparée en fonction de ses deux rôles (émission et réception).

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>

int msgget ( key_t key, int msgflg )
int msgsnd ( int msqid, struct msgbuf *msgp,
             size_t msgsz, int msgflg )

ssize_t msgrcv ( int msqid, struct  msgbuf  *msgp,
                 size_t msgsz, long msgtyp, int msgflg )
int msgctl ( int msqid, int  cmd, struct msqid_ds *buf )
```

FIG. 2.32 – Appels système *msgget(..),msgctl(..), msgsnd(..)* et *msgrcv(..)*.

Les performances des files de messages (figure 2.33) sont correctes, bien qu'inférieures à celles des pipes. Mais les files de messages souffrent surtout de la limitation sur la taille des messages transmis : 8192 octets s'avèrent bien souvent trop peu pour la plupart des besoins de communication.

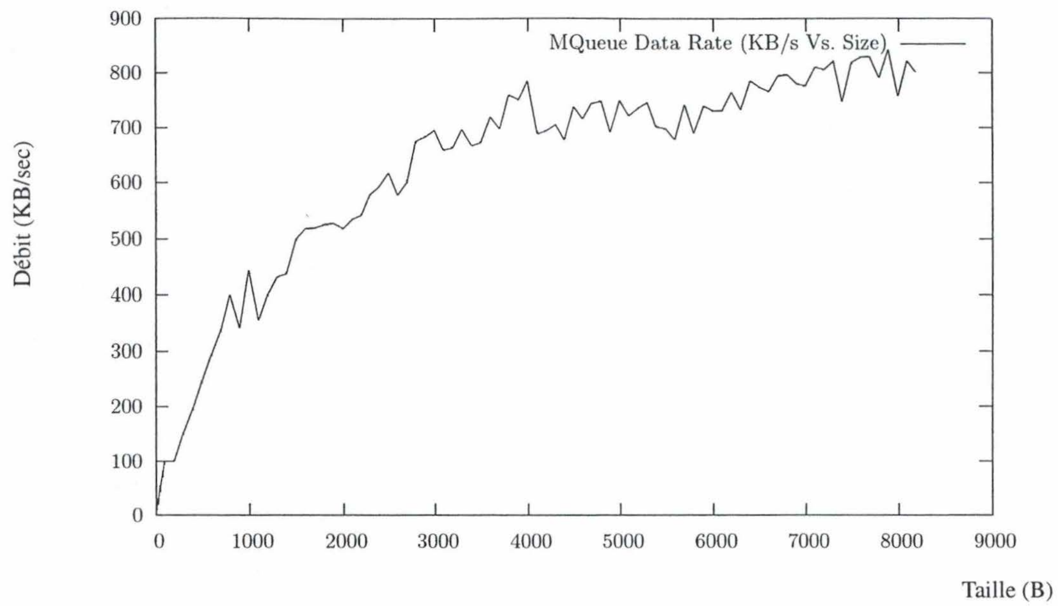

FIG. 2.33 – Débit moyen d'un file de message

Débit moyen d'une file de messages en fonction de la taille des blocs de données à transmettre.

2.7 Performance des différentes IPC

Remarques :

1. L'IPC par mémoire partagée sans copie dans l'espace d'adressage local (figure 2.12, page 14) n'est pas reprise dans cette comparaison ; en effet, ses performances sont trop atypiques pour pouvoir être comparées à celles des autres IPC présentées.
2. Les IPC[9] pour lesquelles les mesures de performances ne peuvent pas être exécutées de manière fiable et relativement indépendante de la machine de ne sont pas reprises dans cette comparaison. En conséquence, les IPC utilisant les fichiers n'apparaissent pas dans la suite.

La comparaison des performances des IPC présentées laisse apparaitres quelques phénomènes intéressants (figure 2.34). D'abord, on peut voir que les performances des différentes IPC sont relativement équivalentes dès que le transfert concerne des blocs de grande taille (taille supérieure à 200 KB). Toutes les méthodes mesurés supportent des débits de l'ordre de 250 KB/sec pour les blocs de plus de 200 KB.

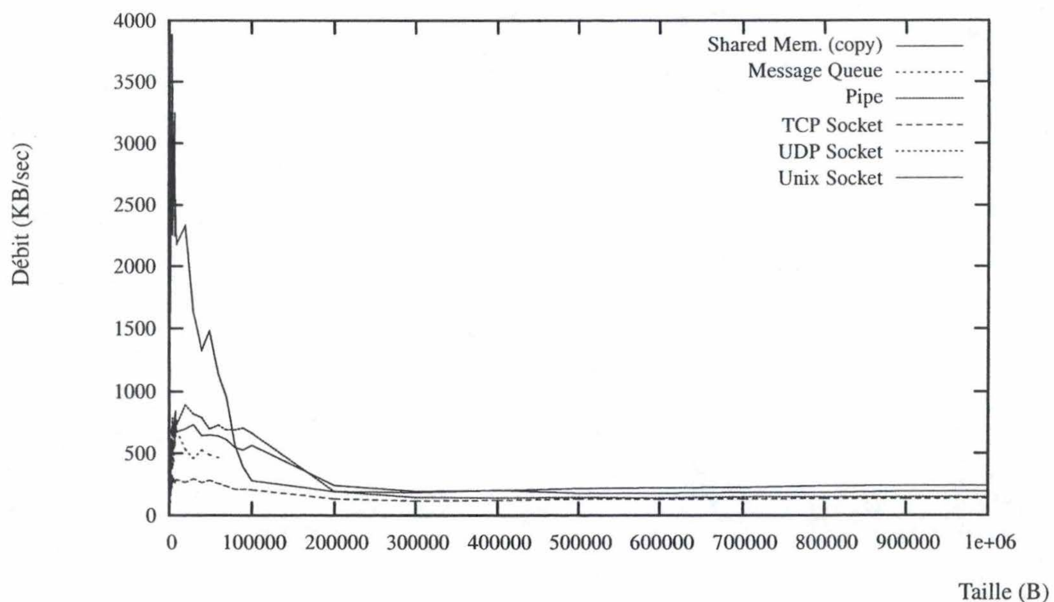

FIG. 2.34 – Débit des différentes IPC

Les débits sont mesurés en KB/sec en fonction de la taille des blocs d'origine.

En ce qui concerne les transferts de blocs de taille inférieure à cette limite de 200 KB, les IPC présentent toutes les même performances à deux exceptions près (figure 2.35) : la première concerne les sockets réseaux utilisés en mode flux, dont les performances sont moitié moindres que celles des autres IPC. Il convient de remarquer que les performances des sockets réseaux utilisés en mode flux sont stables quelque

[9] Il s'agit de la communication par signaux ainsi que des IPC utilisant le disque comme support.

soit la taille des données à transmettre ; le débit moyen reste aux alentours de 200KB/sec. L'autre exception concerne les transferts par mémoire partagée (avec copie dans l'espace d'adressage local) donc les performances pour les blocs de taille inférieure à 75 KB sont loin au dessus de celles des autres IPC. Néanmoins, pour des blocs de taille supérieure à 75 KB les performances chutent rapidement avant de se stabiliser au niveau des performances des sockets réseaux en mode flux.

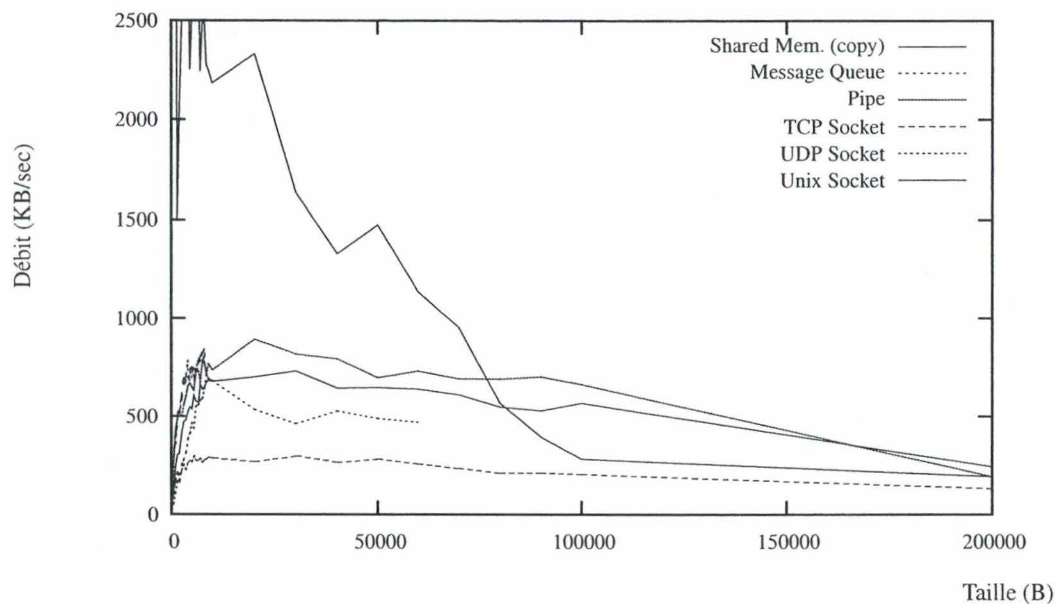

FIG. 2.35 – Débit des différentes IPC - Vue rapprochée.

Les débits sont mesurés en KB/sec en fonction de la taille des blocs d'origine.

La taille des blocs est ici limitée à 200 KB.

Chapitre 3 : IPC Windows

Ce chapitre présente de manière succinte les principales méthodes de communication inter-processus utilisées sur les plateformes Windows. L'étude de ces IPC n'étant pas le but de ce document, nous nous limiterons ici à l'approche de leur principe de fonctionnement, en évitant autant que possible les détails d'implémentation. En particulier, aucune mesure de performances ne sera effectuée sur les IPC présentées.

Remarques :

1. Les données à la base de ce chapitre proviennent quasi exclusivement de la même source d'informations : `http://msdn.microsoft.com/`. Ce site est le portail officiel donnant accès à toute la documentation originale concernant les produits de la société Microsoft.
2. Certaines IPC Windows reposent sur le concept de boucle de messages. Les boucles de messages naissent de la différence fondamentale existant entre les application Windows (qui sont dirigées par des évènements) et les processus classiques. Par exemple, une application Windows ne fait pas d'appel de fonction explicite pour vérifier si des données ont été entrées aux clavier, elle attend que le système d'exploitation lui envoie un message lui indiquant la présence de telles données. Le rôle de la boucle de messages est d'attendre l'arrivée de ces messages et de réagir en conséquence. Les messages reçus par une application peuvent aussi être générés (sous certaines conditions) par d'autres applications.
3. Le vocabulaire utilisé sur les systèmes Windows diffère parfois de celui des systèmes Unix ; voici un tableau permettant de mettre en correspondance les concepts qui le supportent.

Unix	Windows	Commentaire
Pointeur	Handle	/
Processus	Application	La plupart des méthodes de communication Windows supposent l'utilisation d'une boucle de messages ; cette dernière est uniquement disponible pour les applications. De surcroit, Windows ne gère pas la filiation de processus.

3.1 Le clipboard

Le clipboard ou presse-papier est probablement la méthode de communication inter-processus la plus simple et la plus intuitive : tout un chacun l'a déja utilisée pour transférer un bout de texte ou un graphique d'une application à une autre. Le clipboard peut être vu comme un récipient pouvant accueillir au plus une information à un instant donné ; chaque écriture efface définitivement le contenu précédent.

L'utilisation du presse-papier pour la communication inter-processus est assez limitée dans la mesure où la spécification même du clipboard stipule que "Le presse papier est commandé par l'utilisateur (user-driven).

Une application ne devrait transférer une information depuis/vers le presse-papier qu'en réponse à une requête de l'utilisateur. Une application ne doit pas utiliser le presse-papier pour un transfert d'informations sans en avoir averti l'utilisateur." [Microsoft, 2004]. Il n'est donc utilisable que pour des transferts ponctuels et sans contrainte de réactivité puisqu'il fait explicitement intervenir l'utilisateur.

Le presse-papier supporte cinq opérations de base (figure 3.1 et 3.2) encadrant les principales fonctions attendues dans l'optique de la communication : protection des accès concurrents au moyen de barrières à l'entrée (fonctions *Ouverture* et *Fermeture*) et gestion du contenu (fonctions *Exportation*, *Importation* et *Effacement*).

Opération	Description
Ouverture	Verrouille l'accès au clipboard pour empêcher les accès concurrents.
Fermeture	Déverrouille l'accès au clipboard.
Exportation	Place une information dans le clipboard (en écrasant le contenu précédent).
Importation	Lit l'information contenue dans le clipboard. Cette information reste disponible jusqu'à la prochaine opération d'exportation ou d'effacement.
Effacement	Efface le contenu du clipboard.

FIG. 3.1 – Opérations de base sur le clipboard.

Le presse-papier est implémenté en utilisant une allocation globale : l'application qui souhaite émettre une information place cette dernière dans un bloc de mémoire possédant une portée globale (i.e. alloué avec *GlobalAlloc(..)*) et transmet le handle de cette dernière, accompagné de quelques informations de type, vers le clipboard. Lors de la lecture des données, le receveur doit, après avoir identifié le type des données, copier le contenu de la zone de mémoire globale dans son propre espace d'adressage. La désallocation de la zone globale est effectuée automatiquement par les routines de gestion du presse-papier lors de l'effacement ou du remplacement de son contenu.

La protection contre les accès concurrents est effectuée de manière transparente par les appels *OpenClipboard(..)* et *CloseClipboard(..)* ; leur rôle est d'empêcher qu'une application n'écrive dans le presse-papier alors qu'une autre est en phase de lecture ou d'écriture.

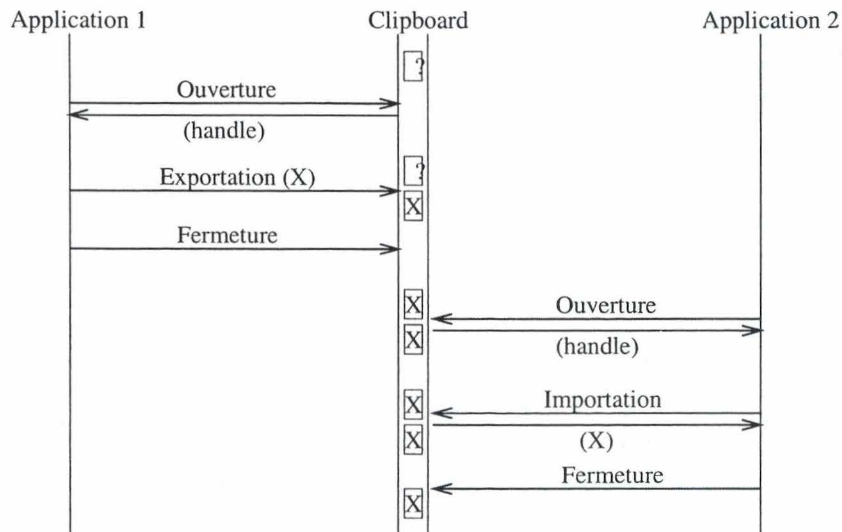

FIG. 3.2 – Principe de fonctionnement du clipboard.

Transfert d'une donnée (X) depuis l'application 1 vers l'application 2 au moyen du presse-papier.

3.2 Data Copy

La méthode *DataCopy* permet à une application d'autoriser une autre application à lire une partie de son propre espace d'adressage ; en envoyant un message *WM_COPYDATA* dans la boucle de messages de l'application cible, l'application source autorise cette dernière à lire le contenu de la zone mémoire pointée par la structure *COPYDATASTRUCT* passée en paramètre au message initial.

```
#include <winuser.h>

typedef struct tagCOPYDATASTRUCT {
  ULONG_PTR dwData;   // données à transférer
  DWORD cbData;       // taille de la cible de lpData
  PVOID lpData;       // pointeur vers les données
} COPYDATASTRUCT, *PCOPYDATASTRUCT

lresult = SendMessage (
    (HWND) hWndControl,   // handle de la destination
    (UINT) WM_COPYDATA,   // message ID
    (WPARAM) wParam,      // handle de la source
    (LPARAM) lParam);     // pointeur vers COPYDATASTRUCT
```

FIG. 3.3 – Appel *SendMessage()*.

Cette méthode requiert une grande précision de la part de l'application source : elle doit pouvoir garantir que :

1. Les données contenues dans la zone de mémoire transmise ne contiennent pas de références à d'autres parties de son espace d'adressage. (sinon l'application cible risque de déclencher des fautes de segmentation en accédant aux données).
2. Aucune partie de l'application source (thread ou autre) ne doit modifier le contenu de la zone transmise tant que l'appel à *SendMessage(..)* (figure 3.3) est en cours[10].

Si l'application cible souhaite réutiliser les données, elle doit les copier dans son espace d'adressage local, car leur intégrité n'est garantie que pour la durée de traitement du message *WM_COPYDATA*.

3.3 DDE

L'architecture de communication Windows, centrée sur le passage de messages, favorise leur utilisation pour les transferts des données inter-applications (cfr. *DataCopy* section 3.2). Cependant, les messages n'acceptent que deux paramètres : *wParam* et *lParam*. En conséquence, ces paramètres doivent souvent référencer de manière indirecte d'autres structures de données contenant les véritables informations à transférer. Le protocole *Dynamic Data Exchange* (DDE) définit une manière précise d'utilisation des deux champs paramètres *wParam* et *lParam* lors du transfert de volumes de données importants au moyen de pointeurs vers des espaces de mémoire partagée. DDE peut alors être vu comme une extension du modèle *Data Copy* dans le sens où il permet des échanges plus réguliers car plus règlementés.

La gestion de la mémoire partagée est soumise à des règles très spécifiques définies dans le protocole DDE afin de garantir que les handles reçus par toutes les entités participant à la communication ne puissent pas être corrompus. En outre, les zones de mémoire partagée transmises par DDE doivent toujours être considérées comme des zones en lecture seulement par le receveur ; elles ne peuvent donc pas servir au retour du message par exemple.

Un autre intérêt de DDE est sa capacité à gérer aussi bien les communications ponctuelles (envoi unique d'un bloc de données entre deux applications) que les communications persistantes (par exemple l'inclusion d'un graphique issu d'une feuille de calcul dans un document texte, la feuille de calcul devant être mise à jour à chaque mise à jour du document source).

Enfin, une communication DDE est identifiée par le couple des handles des applications source et cible. Une même application ne peut donc pas être engagée dans plusieurs discussions parallèles avec la même cible. DDE repose sur le transfert de messages entre les boucles de messages des deux applications. En conséquence, une application qui serait engagée dans plusieurs conversations parallèles avec le même destinataire devrait utiliser un moyen externe à DDE pour identifier à quelle communication appartiennent les

[10] L'appel *SendMessage(..)* est bloquant pendant la durée de traitement du message par l'application cible, mais ce caractère bloquant ne s'applique qu'au thread effectuant l'appel, les autres threads de l'application continuent à être exécutés.

messages reçus. Ce qui est clairement en porte-à-faux avec la volonté de simplification à la base de DDE.

Remarque : Le protocole DDE fut initialement développé pour les architectures 16bits (windows 3.X) et ne fut jamais porté vers les architectures suivantes ; néanmoins, son principe fut largement réutilisé dans le protocole OLE (*Object Linking and Embeeding*) utilisé sur les architectures 32 et 64 bits.

3.4 File

La communication par fichier repose sur la capacité qu'ont les différentes applications d'accéder au contenu du disque local. Son principe de base est le suivant :

1. L'application source copie les données à transférer dans un fichier au moyen des primitives standards d'accès aux fichiers.
2. L'application cible récupère (par une autre méthode IPC ou par une convention commune de placement) le nom et le chemin d'accès du fichier. Elle peut ensuite lire et exploiter les données contenues dans le fichier.

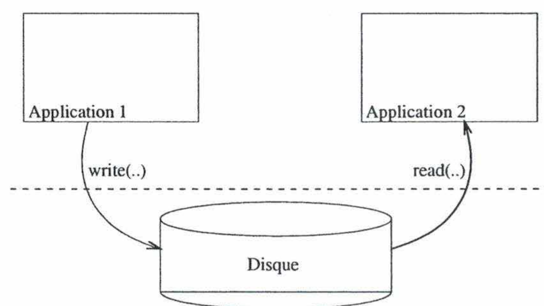

FIG. 3.4 – Communication par fichier

Cette méthode a l'avantage d'être très simple à mettre en oeuvre et de supporter des transferts de très gros volumes de données (à priori, la taille des fichiers est illimitée). Néanmoins, cette méthode est rarement applicable pour les transferts entre deux applications ; en effet, outre le fait que le passage par le disque introduit une lourde pénalité au niveau des performances, il revient aux applications à synchroniser leurs accès au fichier.

3.5 MailSlot

Un mailslot est un pseudo-fichier : il réside entièrement en mémoire centrale mais est accessible au moyen des fonctions d'accès aux fichiers standards. Les données placées dans un mailslot peuvent prendre n'importe quelle forme tant que la taille totale ne dépasse pas 64K. Contrairement aux fichiers standards, les mailslots sont temporaires : lorsque tous les handles vers un mailslot sont fermés, ce dernier est détruit et les données qu'il contenait sont perdues.

Les mailslots ont la particularité de pouvoir être utilisés comme moyens de communication distibués : si un client distant obtient un handle sur un mailslot, les données qu'il va y lire / écrire sont transmises via le réseau. Les messages de moins de 425 bytes sont transmis au moyen d'un datagramme UDP, les messages de taille supérieure sont transmis via une connexion TCP dans le cadre d'une session SMB. Ce comportement engendre deux conséquences notables :

1. Les accès à un mailslot sont limités au sous-réseau courant (les paquets UDP ne peuvent généralement pas franchir les limites du sous-réseau local en raison de la configuration des firewalls).
2. La communication est 1-N pour les messages de taille inférieure à 425 octets et 1-1 pour les messages de taille supérieure. La capacité de broadcast de cet IPC est donc limitée par la taille des messages ; il convient donc que celle-ci soit très strictement garantie dans le cas où un broadcast est requis.

Le principe de fonctionnement d'un mailslot est celui d'une file FIFO : chaque application possédant le handle du mailslot peut y ajouter un message (tout en respectant la limite de taille globale de 64K), lire un message ou retirer le premier message de la liste.

```
#include <winbase.h>

HANDLE CreateMailslot (
   LPCTSTR lpName,          // nom identifiant du mailslot
   DWORD nMaxMessageSize,   //
   DWORD lReadTimeout,      // durée de blocage si
                            // lecture sur un mailslot vide
   LPSECURITY_ATTRIBUTES lpSecurityAttributes);
```

FIG. 3.5 – Appel *CreateMailslot(..)*.

Le nom identifiant d'un mailslot est de la forme : \\. \mailslot\[path]nom ; il permet donc de spécifier un nom qui sera accessible à distance en plaçant dans la partie *path* un chemin *NetBios* comme \\machine \.

La faible capacité (un maximum de 64K) de cette IPC la relègue à des tâches très spécifiques où sa capacité de distribution sur l'ensemble du domaine local et sa relative simplicité d'utilisation sont des avantages par rapport aux autres méthodes, généralement beaucoup plus lourdes à mettre en oeuvre dans le cas distribué.

3.6 Pipes (anonymous / named)

Deux types de pipes coexistent sur les systèmes Windows : les *anonymous pipes* (tubes anonymes) et les *named pipes* (tubes nommés). Les tubes anonymes sont plus simples à mettre en oeuvre, mais n'autorisent pas toutes les possibilités des tubes nommés.

Dans les deux cas, l'idée sous-jacente est toujours de construire un canal de communication entre deux (ou plusieurs) applications, canal qui soit accessible comme un simple fichier local.

3.6.1 Anonymous Pipes

Un tube anonyme est un conduit d'information unidirectionnel utilisable entre deux applications possédant un lien de parenté. Il n'est donc employable que dans un contexte local et ne peut pas être utilisé à travers un réseau. Le lien de parenté est requis pour le transfert des handles identifiant le pipe (un pour l'écriture et un pour la lecture).

L'accès au pipe s'èffectue au moyen de fonctions de lecture/écriture similaires à celles utilisées pour les fichiers. (*ReadFile(..)* et *WriteFile(..)*, figure 3.6). Néanmoins, les I/O asynchrones ne sont pas supportées par les tubes anonymes. L'utilisation des variantes asynchrones des fonctions de lecture/écriture se transforme en un appel caché vers les méthodes synchrones, ce qui peut provoquer des blocages imprévus de l'application.

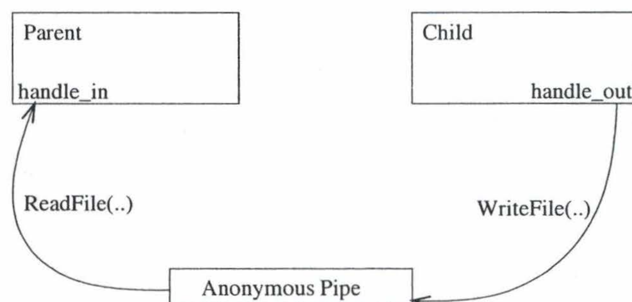

FIG. 3.6 – Tube anonyme sous Windows.

Le pipe est automatiquement détruit lorsque la dernière application utilisatrice ferme le handle correspondant.

Remarque : Sur les systèmes 'non-familliaux' (toutes les versions de Windows 32 et 64 bits sauf Windows 95, 98 et Me), les tubes anonymes sont implémentés au moyen des tubes nommés : le système génère lui-même un nom privé unique pour le pipe. En conséquence, les tubes anonymes ne sont que très rarement

employés car la faible simplification qu'ils apportent au niveau de l'écriture du code, ne contrebalance pas les restrictions qu'ils imposent.

3.6.2 Named Pipes

Un tube nommé est un canal de communication bi-directionnel entre une application (le 'pipe server') et un certain nombre d'applications clientes (les 'pipe clients'). Toutes les instances d'un tube nommé partagent le même nom, mais possèdent chacune leur propre buffer et leur propre handle, ce qui leur permet de fournir un canal de communication client-serveur séparé à chaque client (figure 3.7).Cette particularité permet au serveur d'identifier aisément l'émetteur d'un bloc de données et évite de devoir multiplier les pipes sur le système.

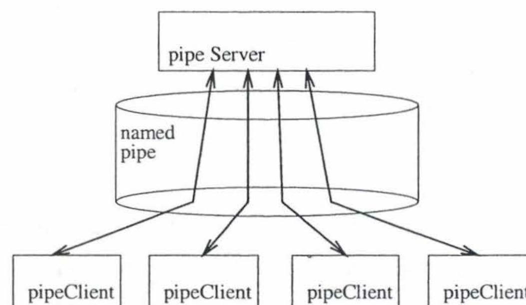

FIG. 3.7 – Tubes nommé sous Windows

Touts les clients partagent le même tube nommé, mais chacun dispose de son propre canal de communication avec le serveur.

Les tubes nommés peuvent aussi être employés au dessus d'un réseau pour faire dialoguer deux applications distantes (sous la contrainte que les deux applications utilisent des versions de Windows compatibles). En effet, les conventions de nommage utilisées pour la création des tubes nommés permettent leur identification à distance : le nom est de la forme

ServerName \pipe **PipeName**

Néanmoins, l'appel de création de tubes ne permet que la création de ceux-ci sur la machine locale, il n'est pas possible de créer des pipes à distance.

Toutes les applications possédant le nom du tube nommé peuvent y accéder (sous réserve des paramètres de sécurité additionels utilisables dans l'appel de création). Comme pour les tubes anonymes, les opérations de lecture/écriture de fichier sont utilisées pour les accès au pipe ; mais dans ce cas, les I/O asynchrones sont supportées. Cette caractéristique est obligatoire dans l'optique d'un déploiement sur un réseau, où les latences de communication peuvent bloquer l'application pendant de longues périodes.

3.7 Network sockets

Le principe des communications par sockets réseau est similaire au cas Unix, le lecteur est donc invité à se reporter à la section 2.3.1 page 15.

Chapitre 4 : IPC indépendantes de la plateforme

Dans un souci de complétude, ce chapitre présente de manière rapide un exemple d'IPC entièrement indépendante de la plateforme.

4.1 Présentation

Les IPC complètement indépendantes de la plateforme sont en général des méthodes très lourdes/complexes à mettre en oeuvre. En effet, celles-ci ne peuvent à aucun moment reposer sur des mécanismes de bas niveau (matériel ou OS) pour assurer leur synchronisation ou leur protection.

Ces IPC se rencontrent principalement dans les langages de script et dans les langages entièrement interprétés. Il est alors du rôle de l'interpréteur de permettre au processus de communiquer sans que ce dernier ne connaîsse quoi que ce soit de la machine ou l'OS.

4.2 Exemple : Curry

Curry[11] est un langage interprété de dernière génération combinant à la fois des aspects de programmation fonctionnelle, des aspects de programmation logique et des aspects de programmation concurrente.

Ce langage fonctionne un peu à la manière de *Prolog* : il permet de résoudre des contraintes en unifiant des variables. L'une des originalités est ici que, grâce au concept de port de communication, les variables à unifier peuvent être réparties sur plusieurs processus, éventuellement distribués (figure 4.1). L'interpréteur se chargeant de la communication nécessaire de manière invisible au programme.

Du point de vue du programme curry, ce type de communication est une IPC indépendante de la plateforme.

4.3 Distribution

Cette classe d'IPC est très faiblement peuplée pour une raison simple : construire une IPC qui soit complètement indépendante de la plateforme sur laquelle elle s'exécute est techniquement très proche de la réalisation d'un mécanisme d'IPC distribuée[12], mais ne fournit pas les avantages de la distribution physique (distribution de la charge, ...). Les IPC entièrement indépendantes de la plateforme ne sont donc développés que dans des cas très particuliers et afin de répondre à des besoins très spécifiques.

[11] Pour des plus amples informations sur Curry, le lecteur peut consulter [Hanus, 2004].

[12] Les mécanismes IPC distribués seront décrits au chapitre 5 (page 47 et suivantes).

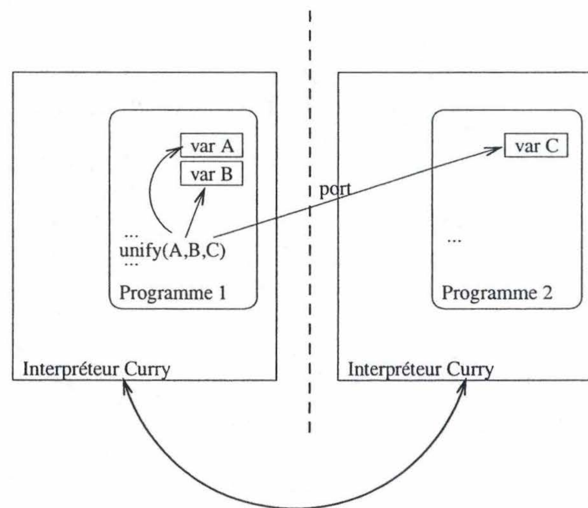

FIG. 4.1 – Curry : unification à distance

Chapitre 5 : IPC Distribuées

Ce chapitre lève la restriction faite au chapitre 1 (étude des IPC locales uniquement) et présente une introduction aux problèmes et enjeux relatifs aux méthodes de communication capables de mettre en relations des processus ne se trouvant plus forcément sur la même machine physique.

5.1 Définition

Avant toute chose, il convient de préciser la notion d'*IPC distribuée* : nous prendrons ici comme convention qu'*une IPC distribuée est une IPC permettant de faire communiquer deux ou plusieurs processus d'un système distribué*. Par système ditribué, nous entendons *une collection d'hôtes autonomes qui sont connectés par l'intermédiaire d'un réseau* [Englebert, 2003].

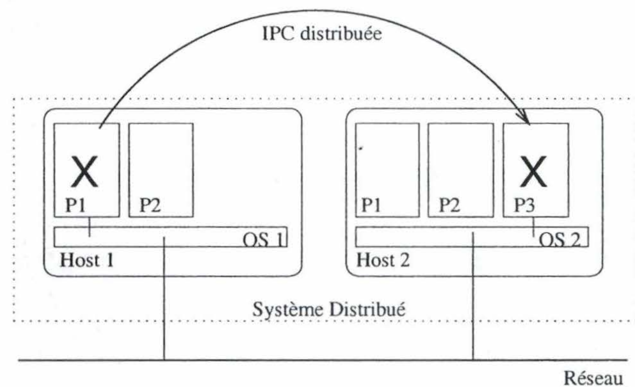

FIG. 5.1 – Système et IPC distribué.

5.2 Problèmes additionels engendrés par la distribution

Parmi les nombreux problèmes engendrés par la distribution de la communication, les plus importants sont : l'hétérogénéité des architectures, le problème de la gestion du temps et les problèmes de sécurité et de fiabilité.

5.2.1 Hétérogénéité

Les machines composant un système distribué ne présentent plus la même unité d'architecture que dans le cas non-distribué, les conventions de représentation des données peuvent varier (big endian ou little endian, méthode de représentation des chaînes de caractères, ...), ou plus simplement, les conventions des langages peuvent varier (cas de la communication entre un processus écrit en C et un processus Java par exemple).

En conséquence, il ne suffit plus de transmettre le contenu d'une variable ou d'une zone mémoire : il faut aussi en adapter le contenu – à l'émission et/ou à la réception – afin que les données soient exploitables par le receveur.

Ce travail d'adaptation est le plus souvent laissé à la charge d'une couche de communication spécialisée (par exemple le middleware CORBA). Néanmoins, dans certains cas simples, cette traduction de format peut être effectuée par le programmeur ; il lui incombe alors de prévoir toutes les possibilités d'architecture existantes ou de se diriger vers un format standard et bien documenté.

5.2.2 Le problème du temps

Cette section s'inspire de [Ramaekers, 1999b].

La chronologie des évènements dans un système distribué pose problème : les relations chronologiques entre deux évènements ne peuvent être affirmées que pour des évènements passés sur un même hôte. Pour établir des relations chronologiques entre des évènements datés par des hôtes différents, les temps fournis par les horloges ne sont d'aucune utilité, car il n'y a aucune garantie de synchronisation des différentes horloges. Ce problème impose la mise en place de protocoles complexes afin d'imposer des contraintes de séquence ou de synchronisation entre divers évènements (par exemple : le protocole de *2 phase commit*).

Un autre aspect du problème temporel provient du délai parfois important nécessaire à un message pour atteindre sa cible. Le temps s'écoulant entre l'envoi et la réception d'un message entre deux processus situés sur la meme machine est au plus de quelques milli-secondes, alors que dans le cas de deux processus distribués (figure 5.2), ce temps peut atteindre plusieurs secondes en fonction de l'état de charge des machines et/ou de la congestion du réseau [13]. Cela rend la plupart des IPC traditionnelles inadaptées ; en effet, celles-ci utilisent des appels bloquant le processus émetteur jusqu'a ce que le receveur reçoive le message. Ce comportement n'est pas admissible dans le cas de bloquages de plusieurs secondes, en particulier dans le cas de serveurs.

[13]Le coeur du problème est le caractère souvent imprévisible des délais engendrés par une connexion réseau (dûs à une perte de paquets par exemple).

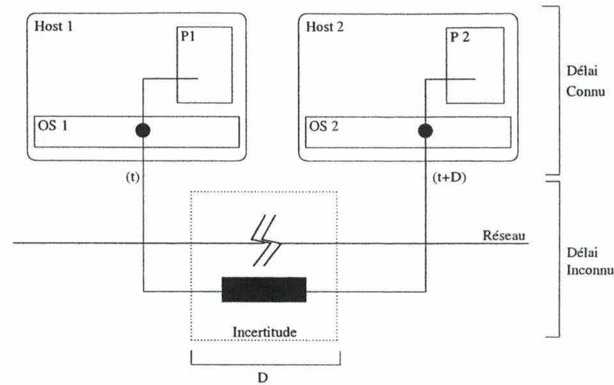

FIG. 5.2 – Le problème des délais incertains.

Le délai supplémentaire (D) introduit par le réseau est difficilement estimable car il dépend de l'état courant du réseau

5.2.3 La fiabilité

L'approche de la fiabilité des transferts dans le cas de communications distribuées est très différente du cas non distribué. En effet, dans le cas non distribué (figure 5.3), la fiabilité des communications est directement liée à l'état de la machine locale : il est raisonnable de penser que les communications sont fiables si la machine fonctionne correctement et non fiables en cas de panne ou de mauvaise configuration. Dans les deux cas, la gestion de la fiabilité est du ressort de l'administrateur système car il peut maîtriser tous les paramètres influençant la fiabilité de la communication.

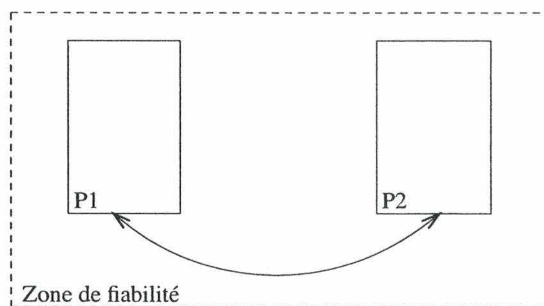

FIG. 5.3 – La fiabilité en contexte local.

La communication est entièrement dans le contexte local, sa fiabilité ne dépend donc que de la fiabilité de la machine locale.

A l'inverse, dans les problèmes de communication distribuée (figure 5.4), la communication sort de cette "zone de confiance" qu'est le contexte local. La fiabilité de la communication ne dépend plus uniquement de la fiabilité des éléments locaux, mais aussi de la fiabilité des éléments intermédiaires. Le problème est que, bien souvent, ces éléments intermédiaires sont inconnus ou changeants (par exemple les routes suivies

par les données entre les deux processus peuvent varier au cours du temps en fonction de la charge du traffic présent sur le canal de communication). Le seul moyen de garantir la fiabilité des données transmises est d'incorporer, au coeur même de l'IPC utilisée, des mécanismes capables de détecter d'éventuelles pertes ou altérations de données et, le cas échéant, de ré-émettre ces données.

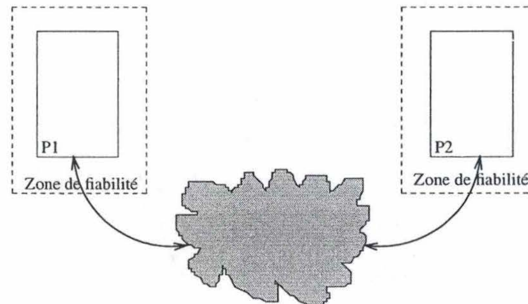

FIG. 5.4 – La fiabilité en contexte distribué.

La communication sort du contexte local, elle est donc à la merci des éventuels problèmes de fiabilité des éléments intermédiaires. Or ces éléments sont – en toute généralité – inconnus.

5.2.4 La sécurité

Le problème de la sécurité des transferts entre deux processus distribués est très similaire à celui de la fiabilité : la distribution provoque l'émission de données potentiellement sensibles au travers d'un réseau. Or ce réseau est un élément sur lequel le programmeur et l'utilisateur ont peu ou aucun contrôle.

Afin de garantir une certaine protection vis-à-vis des tentatives d'écoute ou d'altération des informations transmises, il conviendra souvent de mettre en place des procédures de cryptages des données transmises.

5.3 Exemples d'IPC distribuées

Cette section présente quelques méthodes de communication inter-processus développées dans l'optique d'une communication distribuée ; nous ne traiterons donc pas ici des méthodes IPC classiques qui ont été étendues afin de prendre en compte certains aspects distribués (comme les tubes nommés, les mailslots, ...).

5.3.1 Sockets réseau et protocole spécifique

La manière la plus simple de construire un système de communication distribué est d'utiliser les fonctions réseau de bas niveau : création de sockets, émission de données, réception de données,... (figure 5.5). Cependant, cette approche force le programmeur à gérer lui-même tous les problèmes, comme la traduction

des représentations de données, la spécification du protocole, la gestion des transferts, la gestion des aléas du réseau, la protection des données en transit (fiabilité et sécurité), ... Il s'agit là d'un grand pouvoir et d'un grand risque : un pouvoir car le programmeur maîtrise et peut optimiser tous les aspects de la communication ; et un risque car tous les cas de figure non pris en considération[14] par le programmeur peuvent avoir des conséquences désastreuses.

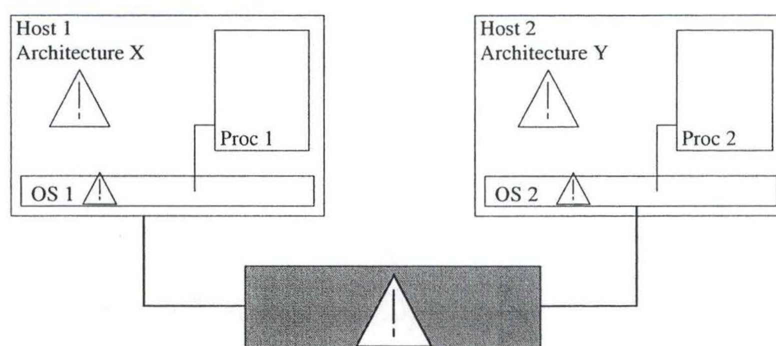

FIG. 5.5 – Communication directe par sockets

Les éléments à prendre en compte sont multiples et dispersés tout au long du canal de communication.

Cette IPC est donc souvent dédiée à des besoins de communication très particuliers : il peut s'agir par exemple de communication nécessitant un degré de sécurité non rencontré dans les solutions existantes ou plus simplement d'une volonté du programmeur de rendre la communication plus difficile à reproduire ou à intercepter qu'avec une méthode standard.

5.3.2 Architecture *middleware*

Afin de permettre au développeur de s'affranchir de la gestion des multiples aléas possibles dans le cas de communications distribuées, le concept de *middleware* a été développé.

Un *middleware* est un "ensemble de services situés entre les services réseau du système d'exploitation et les applications distribuées " [Emmerich, 2000] (figure 5.6). Son rôle est de fournir aux développeurs un plus haut degré d'abstraction en ce qui concerne la communication.

Par exemple, le *middleware* va gérer la traduction des représentations de données de manière invisible à l'utilisateur, ou encore il peut gérer les problèmes inhérents à l'utilisation d'un réseau (interruption de communication, délais, sécurité, fiabilité, ...).

[14]Les éléments à prendre en compte au moment du design de ce type d'IPC sont multiples et en constante évolution, augmentant par là le risque d'omission.

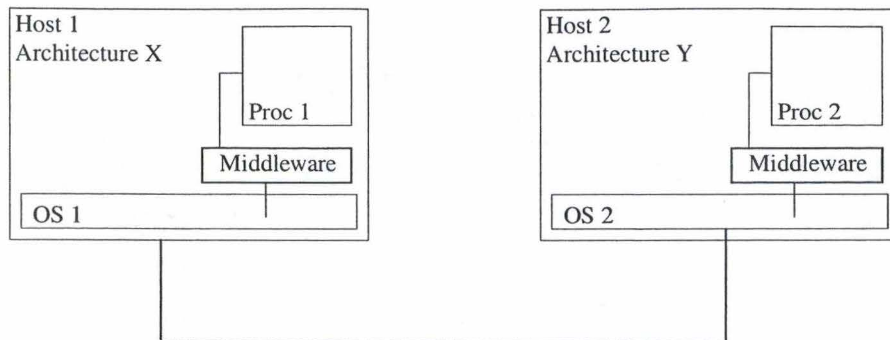

FIG. 5.6 – Middleware

RPC

Les *Remote Procedure Calls* (RPC) permettent à un client d'appeler une procédure située sur un serveur afin de l'exécuter sur ce dernier. Le client et le serveur possèdent chacun leur espace d'adressage propre. Le transit de l'appel entre le client et le serveur est séparé en phases distinctes (figure 5.7). Dans la suite, nous nous focaliserons principalement sur les concepts mis en oeuvre plutôt que sur la description des phases successives.

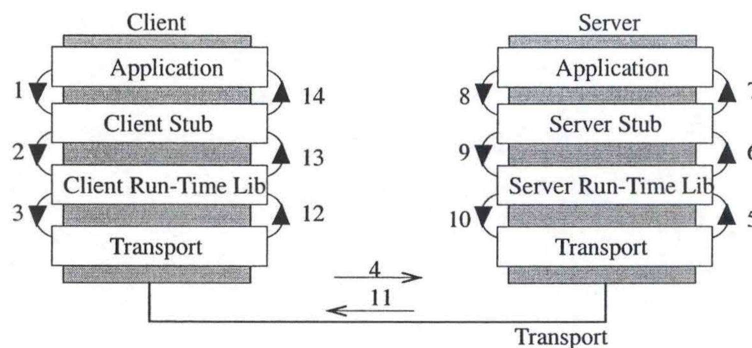

FIG. 5.7 – RPC : principe

Au lieu d'appeler directement la procédure distante, le client appelle une pseudo-procédure locale située dans le *Client Stub* ; ce composant a pour rôle de récupérer les paramètres dans l'espace d'adressage du client, de traduire ce paramètres dans un format standardisé en vue de leur transmission via le réseau et enfin d'appeler les fonctions RPC de la librairie cliente afin de transmettre la demande et les paramètres associés.

Le fonctionnement au niveau du serveur est symétrique : la librairie reçoit une demande d'exécution RPC, la transmet au *Server Stub* ; celui-ci traduit les paramètres en suivant les conventions de la machine et exécute la procédure demandée. Le chemin suivit par les résultats est ensuite symétrique au chemin emprunté par la requête.

CORBA

Corba (*Common Object Request Broker Architecture*) est une spécification définie par l'*Object Management Group*, une organisation sans but lucratif qui regroupe plus de 800 entreprises et organisations. L'un des buts poursuivis lors de la définition de Corba fut de résoudre les problèmes d'hétérogénéité des composants. Cette spécification définit un langage de description d'interfaces ainsi qu'un ensemble de règles de liaisons avec les langages de programmation. Ces liaisons déterminent comment les clients et les serveurs peuvent être implémentés dans différents langages de programmation et continuer à pouvoir interagir entre eux.

Il est important de noter que Corba n'est qu'une spécification, chaque implémentation pouvant résoudre de manière plus ou moins originale les problèmes non explicitement réglés par la définition. Par exemple, la portée du bus implémenté peut varier du bus processus (le client et les objets sont dans le même espace d'adressage local) au bus réseau (les objets sont distribués sur différents hôtes répartis dans un réseau). Néanmoins, la spécification prévoit des mécanismes d'interconnexion entre ces différentes solutions à priori incompatibles. En particulier, Corba définit de manière très précise les techniques d'encodage des données.

Le *middleware* Corba est composé de trois grands composants : l'*Object Request Broker* (ORB) qui est un bus permettant d'échanger des objets de manière transparente, les *Object Services* qui sont un ensemble de services généraux au-dessus de l'ORB (comme un service de nommage, de transaction, de sécurité, ...) et les *Common Facilities* qui sont des modèles d'applications résolvant des problèmes communs à de nombreuses architectures distribuées (administration, gestion des utilisateurs, ...).

Corba est un *middleware* générique dans le sens où il peut être utilisé avec tous les langages pour lesquels une définition précise de la liaison entre les représentations internes et les représentations Corba existe.

RMI

RMI (*Remote Method Invocation*) a été créé dans le but d'offrir une architecture objet distribuée à la fois simple et bien intégrée dans la sémantique Java en cachant les aspects techniques de bas niveau (comme les sockets, les appels de procédures à distance, ...). RMI est donc restreint au langage Java, ce qui limite *de facto* les besoins de traduction entre les formats et les représentations à leur plus simple expression.

RMI peut être vu comme une version de Java généralisée aux appels distants : les mécanismes de base du langage (comme le garbage collector) ont été étendus afin de prendre en compte les objets distribués.

La principale restriction introduite par RMI est que les objets distants ne peuvent être manipulés que via leur interface. Il n'est pas permis de manipuler directement le contenu des variables publiques d'un objet distant, contrairement aux objets locaux.

COM

Le Common Object Model (COM) de Microsoft est né à la suite de l'observation d'une lacune des modèles orientés objets : les contraintes imposées sur les langages sont trop faibles pour permettre une bonne réutilisation et/ou remplacement des composants. En particulier, les modèles ne supportent pas l'encapsulation du code et la compatibilité binaire.

L'encapsulation du code signifie que le code machine, représentant un objet serveur par exemple, peut évoluer de manière distincte de celle de ses clients. Les langages de programmation orientés-objets supportent l'encapsulation syntaxique (au niveau du code source) ; ainsi le client n'a accès qu'aux parties de l'objet serveur ayant été déclarées publiques. La conséquence de cette encapsulation limitée est que si l'objet serveur est modifié, alors l'objet client doit être recompilé. Ce comportement est peu souhaitable dans le cas de composants devant être dispersés sur un grand nombre de machines différentes.

La compatibilité binaire est la capacité d'utiliser un code objet, compilé avec un outil de développement particulier, dans le développement d'un autre objet ; le tout compilé avec un second compilateur. Cette caractéristique nécessite une spécification précise de la manière dont un code devra être compilé. [15]

La principale technique utilisée pour obtenir à la fois l'encapsulation du code et la compatibilité binaire est la séparation de l'interface et du code des différents objets : les clients voient les objets au travers de leur interface, décrite dans un langage standardisé et ne contenant aucune information sur l'implémentation. Tandis que les serveurs exécutent les codes objets correspondant aux spécifications.

5.4 Conclusion

Les communications entre processus distribués sont d'un autre ordre de complexité que les communications locales. En effet, les IPC distribuées doivent prendre en compte une série de facteurs externes commes les aléas dûs au réseau ou la possibilité de la déconnexion de l'hôte distant.

Afin de gérer de manière la plus efficace possible cet ensemble de contraintes, les architectures du type *middleware* s'intercalent entre les processus et la couche réseau de l'OS. Leur rôle est de rendre le plus transparent possible à l'utilisateur les mécanismes mis en place pour fiabiliser et simplifier la communication.

Néanmoins, le *middleware* ne résoud pas tous les problèmes : une partie de la gestion des aléas de communication reste à la charge des processus communiquants. Par exemple, les processus doivent pouvoir faire face à la disparition plus ou moins longue de leur interlocuteur.

[15] COM définit de manière très précise la disposition des objets en mémoire, afin de forcer les différents compilateurs à produire du code inter-opérable

Notre étude des IPC distribuées est ici limitée à une approche superficielle. Ces IPC représentent pourtant un domaine en plein développement avec l'apparition des solutions *.NET*, *Webservices* et de toutes les technologies autorisant le calcul distribué (RPC et autres).

Les IPC distribuées représentent plus que probablement la prochaine évolution permettant d'augmenter la puissance de calcul disponible sans pour autant faire exploser les coûts. Les technologies permettant de combiner les puissances de calcul de différentes machines (comme les technologies de *clustering*) en sont les prémisses.

Chapitre 6 : Classification des différentes méthodes

Ce chapitre va tenter une classification des différentes IPC présentées dans les chapitres 2 et 3 ; l'idée sous-jacente étant de montrer qu'il n'existe pas de meilleure IPC dans l'absolu, mais simplement des IPC plus ou moins adaptées à certains contextes de communication.

6.1 Critères

Chacune des IPC entrant dans la comparaison sera évaluée sur base des critères suivants : portée, protection, parallélisme, propreté, préemption, restrictions et débit. Le détail de ces critères est présenté ci-après.

6.1.1 Portée

Par portée d'une IPC, nous entendons l'ensemble des processus capables d'entrer en communication avec un processus donné au moyen de cet IPC. Les deux cas de figure les plus courants sont la portée globale, où tous les processus du système peuvent communiquer avec le processus de référence (avec parfois des limites dues aux permissions d'accès), et la portée locale, où seuls les processus possédant une relation de filiation plus ou moins directe avec le processus de référence sont en mesure de communiquer avec lui (figure 6.1).

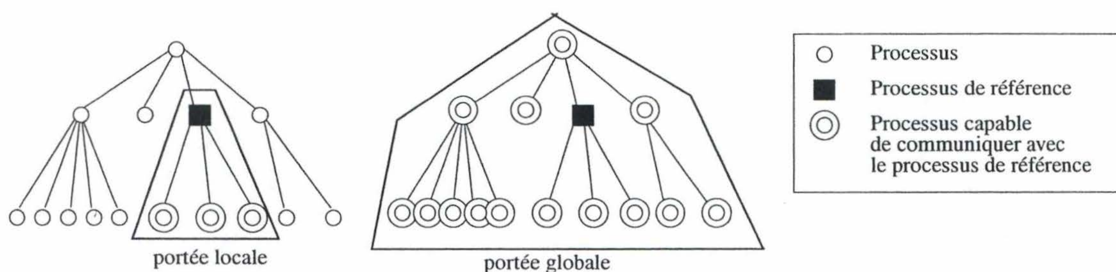

FIG. 6.1 – Critère de portée

6.1.2 Protection

Le critère de protection consistera à évaluer dans quelle mesure les circonstances exceptionelles survenant lors de la communication sont à la charge des processus participants ou du noyau. Il conviendra en particulier de prendre en compte la protection contre les accès concurrents ainsi que la protection contre les accès non-autorisés.

6.1.3 Parallélisme

Ce critère est double : dans un premier temps, il s'agira de voir si l'IPC analysée supporte que la communication contienne plusieurs lecteurs/rédacteurs en lieu et place d'un seul couple. Ensuite, il s'agira de voir si, dans la situation de lecteurs/rédacteurs multiples, chaque processus participant peut clairement identifier les messages qui lui sont destinés sans pour autant devoir recourir à une méthode externe où à l'analyse de tous les messages.

6.1.4 Propreté

Ce critère prend en compte le fait que les ressources utilisées par la méthode IPC soient éliminées automatiquement après usage par le noyau ou que ce travail soit à charge des processus participants.

6.1.5 Préemption

Ce critère tente d'identifier les IPC capables de notifier immédiatement le processus receveur de l'arrivée d'un message, par opposition aux IPC nécessitant que le processus vérifie par lui-même la présence ou l'absence de messages.

6.1.6 Restrictions

Prend en compte les éventuelles restrictions d'usage ou les conséquences sur l'architecture des applications résultant de l'utilisation de cet IPC.

6.1.7 Débit

Ce critère prend en compte la performance des transferts. Par performance, nous entendons à la fois le délai minimum nécessaire pour le transfert d'une information simple et la capacité de l'IPC à soutenir des transferts de gros volumes d'information. Ce critère tient principalement compte des mesures de performances présentées au cours du chapitre 2.

6.2 Comparaison

Cette section tente d'évaluer quelques IPC courantes sur base des critères définis dans la section précédente. Cette évaluation ne se veut pas une évaluation absolue, mais plutôt une base de réflexion permettant de dégager de grandes 'familles' d'IPC.

6.2.1 Fichier

(cf. sections 2.4.3 et 3.4 pour une présentation détaillée)

Portée Globale étendue. La portée de cette IPC est globale au système ; tous les processus capables d'accéder au système de fichiers local (et partageant les conventions de communication) sont en mesure de communiquer entre eux. Cette IPC permet en outre de faire communiquer deux processus s'exécutant à des périodes de temps disjointes.

Protection La protection des accès est quasi inexistante : cette IPC n'inclut aucun mécanisme permettant de garantir que le fichier contenant le message envoyé du processus A au processus B ne soit pas détruit/altéré par un processus C (exception faite des permissions de fichier).

Parallélisme Moyen. Ce modèle supporte bien la présence simultanée de plusieurs lecteurs pour un même rédacteur. Par contre, il ne permet pas la situation inverse (plusieurs rédacteurs et un/plusieurs lecteur(s)) car un fichier reste une ressource utilisable en écriture par un seul processus à la fois.

Propreté Faible. L'élimination des fichiers ayant permis le transit de messages est entièrement à la charge des processus participants.

Préemption Faible. Le processus receveur doit lui-même vérifier le contenu du fichier ; il n'existe aucun moyen automatique de notification en cas de réception d'un message prioritaire par exemple.

Restrictions Moyenne. L'utilisation des fichiers pour la communication ne constitue une contrainte sur l'architecture que dans la mesure où les processus doivent préférer une communication par messages aggrégés (la communication par fichier prend du temps) et sans réelle contrainte de délai maximum avant délivrance (le processus cible doit effectuer une action explicite pour vérifier la présence ou non de messages).

Débit Faible. Les transferts de données souffrent des pénalités encourues lors des accès au disque.

6.2.2 Mémoire partagée

(cf. sections 2.2 et 3.3 pour une présentation détaillée)

Portée Globale étendue (dans le cas des interfaces Sys V). Tous les processus du système peuvent accéder aux données contenues dans la zone de mémoire partagée (modulo d'éventuelles restrictions concernant les droits d'accès). En outre, deux processus dont les exécutions sont disjointes dans le temps peuvent communiquer au moyen de la mémoire partagée, même si ce comportement n'est pas à encourager (en raison de l'utilisation inutile des ressources mémoire du système pendant la période entre l'exécution des deux processus).

Protection Moyenne à faible. La clé d'accès à la mémoire partagée est publiquement disponible et seules les droits d'accès spécifiés à la création de la zone de mémoire partagée permettent d'empêcher tous les processus d'y accéder librement.

Parallélisme Complet. Tous les processus participant à la communication peuvent utiliser de manière parallèle la zone de mémoire partagée ; le noyau ne gère en rien une quelconque protection des données écrites par un processus vis-à-vis des écritures d'un autre processus. Il revient donc aux différents processus à mettre en place des conventions communes afin de ne pas se parasiter mutuellement. Ce modèle supporte les lecteurs/rédacteurs multiples.

Propreté Faible. Les processus participants doivent détruire eux-même la zone de mémoire partagée une fois celle-ci inutile. Dans le cas contraire, la zone continuera d'occuper les ressources du système jusqu'à une intervention de l'administrateur ou un redémarrage de la machine.

Préemption Faible. Comme pour les fichiers, le processus receveur doit lui-même vérifier la présence/absence de messages dans la zone de mémoire partagée.

Restrictions Moyenne. La création et l'accès à une zone de mémoire partagée sont des opérations relativement simples. La difficulté réside plutôt dans le respect des conventions communes d'accès afin de ne pas corrompre les transferts des autres processus participants.

Débit Important. Comme présenté dans le section 2.2, les transferts par mémoire partagée sont uniquement limités par la vitesse des opérations de copie de bloc mémoire.

6.2.3 Signaux

(cf. section 2.5 pour une présentation détaillée)

Portée Globale (pour le super-utilisateur) ou locale (pour un utilisateur normal). Un utilisateur ne peut envoyer de signaux qu'à des processus lui appartenant (exception faite du super-utilisateur qui peut envoyer des signaux à tous les processus).

Protection Forte. Seuls les processus appartenant au même utilisateur peuvent s'échanger des signaux, ce qui élimine le cas des accès non-autorisés. Les accès concurrents sont eux 'factorisés' : un signal reçu plusieurs fois avant que le processus cible ne puisse le traiter sera perçu que comme un seul et unique signal. L'accès concurrent ne provoque donc pas de corruption des données transmises, mais plutôt un risque de perte de données (dans le cas où la sémantique des signaux reçus ne serait pas équivalente à celle des signaux perçus après la "factorisation").

Parallélisme Faible. Les signaux ne permettent pas d'identifier leur source. Ils s'appliquent sans peine au modèle plusieurs lecteurs / un seul rédacteur. Par contre, pour le modèle avec plusieurs rédacteurs, les signaux ne sont applicables que si tous les rédacteurs sont équivalents (i.e. le lecteur n'a pas besoin d'informations de provenance concernant le message).

Propreté Complète. Les signaux sont entièrement gérés par le noyau, de leur création à leur destruction.

Préemption Importante. Les signaux que le processus cible ne bloque pas lui seront notifiés dès leur réception, interrompant par là le cours normal d'exécution du processus.

Restrictions Forte. L'utilisation des signaux comme méthode de communication impose que tous les processus potentiellement receveurs définissent des fonctions d'interception des signaux choisis. Dans le cas contraire, ces processus seront bien souvent arrêtés par l'OS à la réception de signaux non gérés. En outre, le faible nombre de signaux dont la sémantique a été laissée à la disposition de l'utilisateur fait que seules des informations très simples peuvent transiter via cette IPC.

Débit Faible. Bien que le temps entre l'émission et la réception du signal soit très court, les signaux ne peuvent véhiculer que des informations très simples ; leur débit est donc très faible.

6.2.4 Sockets réseau

(cf. sections 2.3 et 3.7 pour une présentation détaillée)

Portée Globale étendue. Les sockets réseau permettent une communication entre deux ou plusieurs processus quelconques, que les processus soient locaux ou non n'intervient que peu dans les techniques mises en oeuvre. En outre, les sockets permettent de passer outre les habituelles restrictions de la communication aux processus de l'utilisateur courant.

Protection Forte. Les couches réseau de l'OS gèrent la plupart des aléas pouvant provenir de l'usage du réseau. Des techniques avancées permettent même de se prémunir contre d'éventuelles tentatives d'interception/altération de la communication (comme SSL par exemple).

Parallélisme Complet. Les sockets supportent les lecteurs et les rédacteurs multiples au travers des mécanismes de *multicast* et de *broadcast*.

Propreté Forte. Durant l'exécution du processus, les sockets ne sont détruits que sur une demande explicite ; par contre, à la terminaison du processus, le noyau ferme tous les sockets encore ouverts.

Préemption Faible à moyenne. Les sockets ne supportent pas nativement de mécanisme permettant de notifier immédiatement le processus en cas de réception de message. Néanmoins, les appels systèmes de multiplexage (comme *poll(..)* ou *select(..)*) permettent une préemption limitée : le processus doit avoir explicitement demandé à être notifié.

Restrictions Faible. L'utilisation de sockets réseau impose seulement aux processus communicants de devoir supporter les éventuelles ruptures de connexion et/ou les délais de communication importants et variables.

Débit Moyen. La vitesse d'un réseau (même local) provoque une limitation des performances vis-à-vis des IPC purement locales.

6.2.5 Tubes anonymes

(cf. sections 2.1 et 3.6.1 pour une présentation détaillée)

Portée Locale. Seuls les processus possédant une relation de filiation relativement directe entre eux peuvent entrer en communication.

Protection Moyenne. Pour les transferts de faible taille, les OS garantissent en général une écriture/lecture atomique. Par contre, les transferts de données plus importantes nécessitent la mise en place par les processus de mécanismes de protection plus avancés afin de ne pas corrompre leurs écritures réciproques.

Parallélisme Faible. Un tube anonyme est avant tout un canal de communication entre deux processus. Rien n'a été prévu pour la gestion de plus de deux processus accédant au pipe de manière concurrente.

Propreté Complète. Les pipes sont des ressources gérées par le noyau ; ce dernier détruit le pipe lorsque le dernier processus communiquant ferme son descripteur. Dans tous les cas, les pipes sont détruits lors de la terminaison du dernier processus possédant un descripteur du pipe.

Préemption Faible. Le processus receveur doit effectuer une action explicite (*read(..)*, *poll(..)*, ...) pour être notifié de la présence d'éventuels messages.

Restrictions Faible. La seule véritable contrainte est la relation de filiation imposée pour le transfert des descripteurs de fichiers identifiant le pipe.

Débit Moyen. Entièrement contenus en mémoire, les pipes présentent des performances légèrement inférieures aux mémoires partagées. Cette différence est due à la présence de mécanismes noyau de protection contre les accès concurrents ainsi qu'à la faible capacité d'un pipe.

6.2.6 Tubes nommés

(cf. sections 2.4.1 et 3.6.2 pour une présentation détaillée)

Les tubes nommés sont une extension des tubes anonymes créée dans le but de permettre à des processus de communiquer sans avoir la contrainte de la relation de filiation. Ils souffrent donc presque des même avantages et défauts que les tubes anonymes.

Portée Globale étendue. Les tubes nommés permettent une communication entre tous les processus possédant le nom du tube. Certaines implémentations (Win32 en particulier) permettent en outre de faire communiquer deux processus distribués au moyen d'un tube nommé.

Protection Moyenne.

Parallélisme Faible/Forte. Certaines plateformes (Win32 en particulier) permettent d'utiliser un même tube nommé pour la communication entre différents processus, chacun possédant un canal de communication 'privé' au sein du pipe partagé.

Propreté Complète.

Préemption Faible.

Restrictions Très Faible. La principale restriction des pipes (la relation de filiation obligatoire) est ici levée.

Débit Moyen. Le débit d'un tube nommé (dans le contexte local) est équivalent à celui d'un tube anonyme.

6.2.7 Synthèse

IPC	Portée	Protection	Parallélisme	Propreté	Préemption	Restrictions	Débit
Fichier	[****]	*	[***]	[*]	[*]	**	*
Mémoire partagée	[****]	**	[****]	[*]	[*]	**	****
Signaux	[****]	[****]	*	[****]	****	****	*
Sockets réseau	[****]	[****]	****	[****]	***	***	**
Tubes anonymes	**	**	*	[****]	*	[*]	[***]
Tubes nommés	****	**	*/****	[****]	*	[*]	[***]

TAB. 6.1 – Synthèse des éléments de comparaison.

6.3 Conclusion

Trois 'familles' d'IPC se dégagent de cette comparaison (tableau 6.1) :

- La première catégorie contient les IPC entièrement gérées par l'utilisateur, comme les fichiers et les zones de mémoire partagée.
- La seconde englobe les IPC gérées par le noyau dont les caractéristiques principales sont la sécurité vis-à-vis des accès concurrents et la portée globale. (comme les signaux, les sockets, ...)
- Enfin, la troisième catégorie comprend les IPC gérées par le noyau pour lesquelles l'accent est surtout mis sur la facilité d'utilisation, comme les tubes (anonymes ou nommés).

La première classe rassemble les IPC laissant le programmeur gérer tous les aspects de la communication ; seul le programmeur est responsable de la gestion/protection/destruction de ces IPC. La seule intervention du noyau se situe au moment de la création de l'IPC. Cette catégorie contient donc les IPC les plus difficiles à gérer (car tous les cas de figures doivent être envisagés) mais aussi les plus puissantes en termes de performances et de parallélisme.

La seconde catégorie contient les IPC permettant (au prix d'une certaine complexité d'emploi) de mettre en place des communications tout en respectant des contraintes de délais (pour les signaux) ou de portée (pour les sockets) précises. La gestion des ces méthodes n'est plus entièrement à la charge du noyau ; les processus participants doivent aussi gérer une partie de la communication (pour les intercepteurs de signaux par exemple).

Enfin, les IPC du troisième groupe sont avant tout utilisées pour leur simplicité d'emploi : quelques lignes de code suffisent à mettre en place une communication entre deux processus et cette communication supporte des transferts de données de taille moyenne sans provoquer de trop grandes pénalités au niveau des

performances. La simplicité de ces IPC provient de leur gestion par le noyau : les processus sont de simples utilisateurs de moyens de communication gérés ailleurs.

Chapitre 7 : Application au stack

Cette section présente un cas d'utilisation combinée de différentes méthodes IPC afin de permettre une transmission rapide et efficace de l'information. Le développement de l'architecture présentée ici a été effectué lors d'un stage d'un semestre au sein de la société *Nextenso S.A.* sous la supervision de messieurs Gilles Fleury et Dimitri Tombroff. L'implémentation de l'architecture a été réalisée pour le système Linux et une architecture x86 32 bits[16].

L'application concernée est un *proxy*, c'est à dire un composant se plaçant entre un ensemble de clients et un ensemble de serveurs sur un réseau de communication. Son rôle est de filtrer les messages transitant sur le moyen de communication et de réagir de manière adéquate à leur contenu, par exemple en censurant certains paquets, en jouant le rôle de serveur cache ou encore en mesurant les quantités de données transmises afin de pouvoir établir une facturation. Un proxy peut être visible ou non de la part des clients/serveurs (par exemple, un proxy HTTP de cache comme *squid* peut être rendu totalement invisible pour le client).

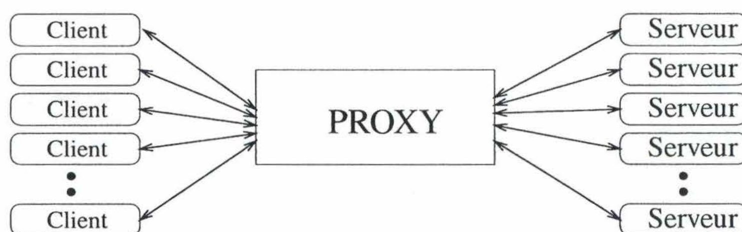

FIG. 7.1 – Schéma d'un serveur proxy

Les principales caractéristiques attendues d'un proxy sont la rapidité (l'étape supplémentaire de filtrage des requêtes doit être la plus invisible possible pour le client et le serveur), la stabilité (le proxy doit pouvoir supporter des requêtes erronées ou malveillantes sans pour autant bloquer tous les autres clients) ainsi qu'une bonne capacité à monter en charge (le temps de traitement d'une requête doit rester le plus stable possible, même en cas d'augmentation de la charge).

Conceptuellement, un proxy peut être vu comme un couple client-serveur, avec la partie client dialoguant avec les serveurs et la partie serveur 'simulant' un serveur aux différents clients.

Le cas traité ici sera celui d'un proxy pour le protocole HTTP. Ce protocole est utilisé par les navigateurs Internet lors du téléchargement d'objets depuis le réseau Internet. Néanmoins, les concepts développés peuvent sans peine être étendus à la plupart des autres protocoles de communication.

[16]Le caractère d'architecture 32 bits est uniquement utilisé en vue du calcul des tailles des différentes structures, il ne représente en rien une limitation du prototype.

7.1 Problématique

Le problème à la base de la rédaction de ce document et du stage associé était un problème de montée en charge d'un serveur proxy particulier : *HttpStack*. Il s'agit d'un proxy un peu particulier dans le sens où il ne joue qu'un rôle d'intermédiaire entre les clients, les serveurs et la *Proxy Platform Nextenso* : le cheminement d'une requête est le suivant (cf figure 7.2) :

1. Le client envoie sa requête vers le proxy.
2. Le proxy relaie la requête vers la *Proxy Platform*.
3. La plateforme traite le message (filtrage, facturation,...) puis le retransmet vers le proxy.
4. Le proxy transmet la requête modifiée vers le serveur.
5. La réponse du serveur est reçue par le proxy.
6. Cette réponse est transmise à la plateforme.
7. Le plateforme envoie la réponse (modifiée) vers le proxy.
8. Le proxy émet la réponse modifiée vers le client.

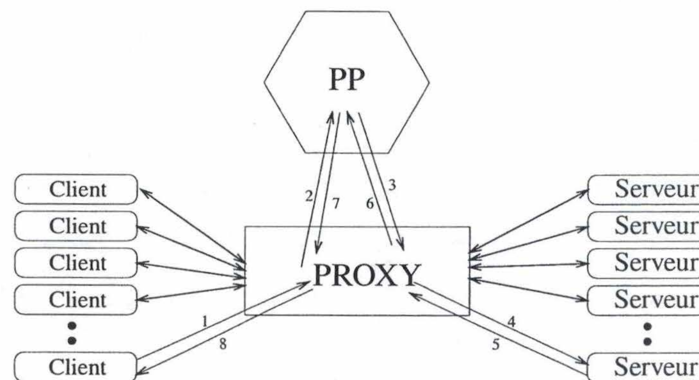

FIG. 7.2 – Un proxy particulier : *HttpStack*.

Ce comportement est un peu plus complexe que celui d'un proxy 'normal' puisque, dans le cas présent, un intermédiaire supplémentaire est inséré entre le client et le serveur. Cette architecture est une conséquence directe de l'absence de mécanisme de gestion d'entrées-sorties asynchrones en Java 1.3.

7.1.1 Entrées/sorties asynchrones

Afin de préciser le concept d'entrées-sorties asynchrones, il convient d'abord de rappeler le fonctionnement normal des primitives de lecture/écriture : sans indication particulière, les fonctions standards de la libC sont des appels synchrones.

Une lecture sur un socket ne possédant aucune information en attente bloquera le processus en cours jusqu'à ce qu'une quantité de données suffisante arrive sur le socket (ou jusqu'à une durée d'attente maximale suivant les fonctions utilisées). De manière similaire, une écriture sur un socket dont tous les tampons sont

déjà pleins sera bloquante jusqu'à ce qu'une place suffisante pour accueillir les nouvelles données soit disponible dans les tampons (figure 7.3). Ce comportemement peut être accepté sur un petit poste client, mais certainement pas sur un serveur, sinon un client disposant d'une connexion bas débit (par exemple un modem 56K) bloquerait de manière répétée le serveur HTTP auquel il se connecte pour télécharger un fichier de taille relativement importante.

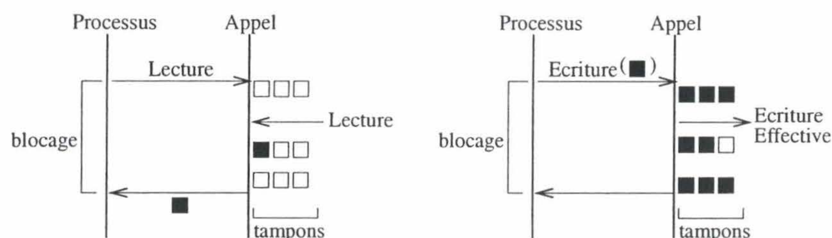

FIG. 7.3 – Entrées/Sorties synchrones.

Le processus est bloqué en attendant que l'opération d'entrée/sortie se termine.

Afin d'éviter ce comportement, les méthodes d'accès dites *asynchrones* ne bloquent jamais (figure 7.4) ; elles renvoient immédiatement le contrôle au processus appelant et lui signalent si l'opération a réussi (i.e. si la donnée à lire/écrire a effectivement été lue/écrite) ou si l'opération a échoué. Dans ce dernier cas, la valeur de retour permet au processus appelant d'identifier la cause de l'échec : un risque de blocage (EAGAIN ou EWOULDBLOCK), un socket invalide (EBADF), ... et donc le processus peut réagir en conséquence en refaisant une tentative par après ou en gérant la condition d'erreur.

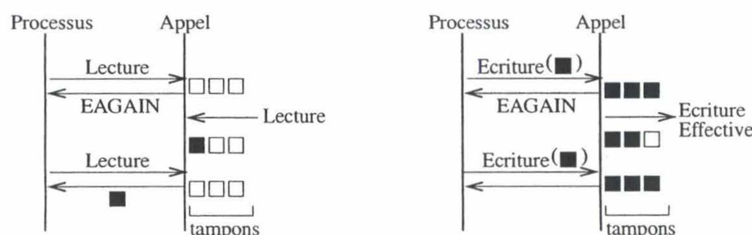

FIG. 7.4 – Entrées/Sorties asynchrones.

Le processus est immédiatement averti que la lecture/écriture n'a pas pu être effectuée ; il peut donc gérer l'erreur ou relancer l'opération par la suite.

Sous Unix, le caractère synchrone ou asynchrone des appels dépend du descripteur de fichier utilisé : par défaut, tous les appels sont synchrones (bloquants) mais la fonction *fcntl(..)* permet de leur donner un caractère asynchrone au moyen de l'option O_NONBLOCK. En outre, des fonctions de multiplexage sont disponibles afin de ne pas devoir tester de manière séquentielle tous les descripteurs de fichier utilisés : il s'agit principalement des fonctions *select(..)* et *poll(..)* qui permettent en un seul appel système de connaître

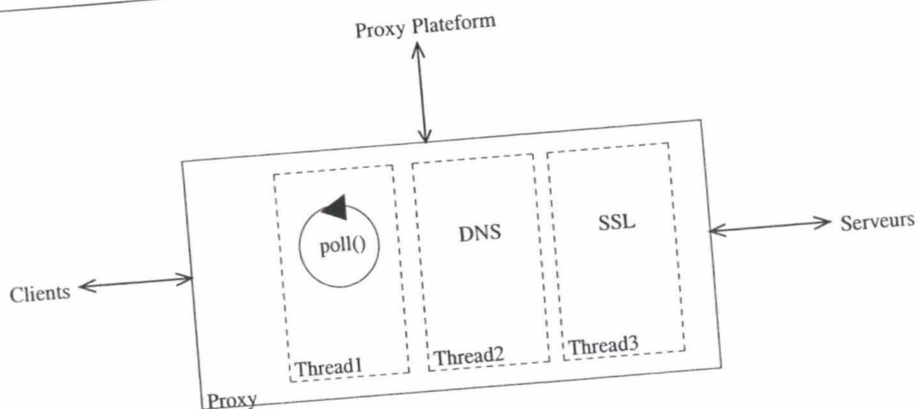

FIG. 7.6 – Proxy - vue rapprochée.

7.2 Proposition d'architecture

7.2.1 Principe

La solution au problème de montée en charge des appels *poll()* est leur démultiplication (figure 7.7) ; en effet, plusieurs appels consécutifs utilisant des paramètres plus petits sont plus rapides qu'un seul appel monolithique. Mais cette approche pose un gros problème : supposons que 5 appels consécutifs soient utilisés, un client dont la requête se trouverait prête sur un socket placé dans le cinquième appel doit attendre la complétion des 4 appels précédents avant d'être servi. Il n'est alors plus possible d'utiliser l'un des principaux avantages de *poll()* : la capacité d'endormir le processus en attendant qu'au moins un des sockets surveillés manifeste une quelconque activité. En conséquence le programme résultant sature à 100% le CPU, ce qui est difficilement acceptable sur un serveur devant par définition supporter plusieurs services en parallèle.

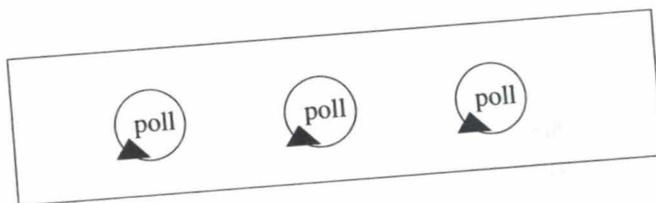

FIG. 7.7 – Solution : multiplication de l'appel *poll(..)*.

De même, la solution aux problèmes de gestionnaire de signaux unique, de mauv[illegible]sation des ressources dans les cas multi-processeurs, ... est la démultiplication des contextes [illegible]cution et donc des processus (figure 7.8). Avec deux processus distincts accédant à des espace[illegible]oire disjoints, les problèmes de mise à jour de cache ne se posent plus ; et chaque processus p[illegible] exécuter sur un processeur particulier de manière parallèle aux autres processus. Le désagrément [illegible]utilisation de processus en lieu et place de threads est que les méthodes internes de communicatior[illegible]sponibles pour les threads ne s'appliquent pas aux processus ; il convient donc d'utiliser les méthode[illegible] IPC classiques.

déjà pleins sera bloquante jusqu'à ce qu'une place suffisante pour accueillir les nouvelles données soit disponible dans les tampons (figure 7.3). Ce comportemement peut être accepté sur un petit poste client, mais certainement pas sur un serveur, sinon un client disposant d'une connexion bas débit (par exemple un modem 56K) bloquerait de manière répétée le serveur HTTP auquel il se connecte pour télécharger un fichier de taille relativement importante.

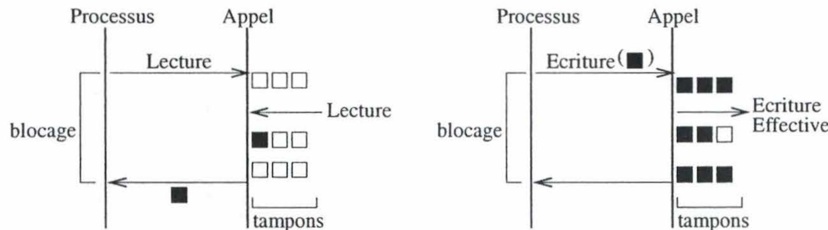

FIG. 7.3 – Entrées/Sorties synchrones.

Le processus est bloqué en attendant que l'opération d'entrée/sortie se termine.

Afin d'éviter ce comportement, les méthodes d'accès dites *asynchrones* ne bloquent jamais (figure 7.4) ; elles renvoient immédiatement le contrôle au processus appelant et lui signalent si l'opération a réussi (i.e. si la donnée à lire/écrire a effectivement été lue/écrite) ou si l'opération a échoué. Dans ce dernier cas, la valeur de retour permet au processus appelant d'identifier la cause de l'échec : un risque de blocage (EAGAIN ou EWOULDBLOCK), un socket invalide (EBADF), ... et donc le processus peut réagir en conséquence en refaisant une tentative par après ou en gérant la condition d'erreur.

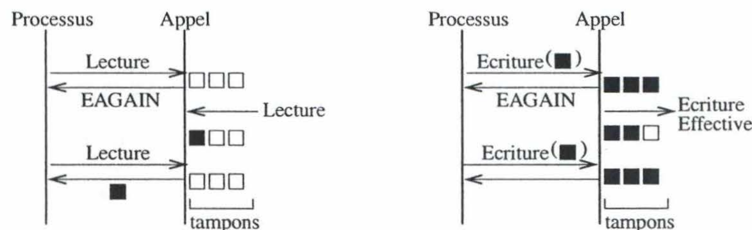

FIG. 7.4 – Entrées/Sorties asynchrones.

Le processus est immédiatement averti que la lecture/écriture n'a pas pu être effectuée ; il peut donc gérer l'erreur ou relancer l'opération par la suite.

Sous Unix, le caractère synchrone ou asynchrone des appels dépend du descripteur de fichier utilisé : par défaut, tous les appels sont synchrones (bloquants) mais la fonction *fcntl(..)* permet de leur donner un caractère asynchrone au moyen de l'option O_NONBLOCK. En outre, des fonctions de multiplexage sont disponibles afin de ne pas devoir tester de manière séquentielle tous les descripteurs de fichier utilisés : il s'agit principalement des fonctions *select(..)* et *poll(..)* qui permettent en un seul appel système de connaître

l'état d'un grand nombre de descripteurs de fichier (prêt à l'écriture, prêt à la lecture, erreur, ...)

7.1.2 Appel système *poll(..)*

L'appel système *poll()* (figure 7.5) va jouer un rôle central dans la suite du développement. La fonction *poll(..)* est une fonction de la libC (libraire de fonctions disponibles sur la quasi totalité des systèmes Unix et sur certains systèmes Windows) servant de relais avec les routines du noyau en charge de la gestion des descripteurs de fichier. Elle est utilisée afin de pouvoir surveiller un grand nombre de sockets et, par là, de pouvoir extraire les quelques sockets présentant une activité intéressante sans devoir les parcourir tous un par un. Outre son rôle de surveillance, la fonction *poll(..)* permet aussi de bloquer le processus pendant un certain laps de temps ou jusqu'à l'arrivée d'un évènement sur l'un des descripteurs de fichier surveillés ; le premier de ces évènements débloquant le processus. Il s'agit d'une alternative avantageuse à l'utilisation d'une boucle d'attente active qui consisterait à boucler sur la vérification des descripteurs de fichier ou à l'utilisation d'un appel de blocage temporel comme *sleep(..)* qui bloque le processus pendant la durée spécifiée, mais sans le réveiller en cas d'activité.

```
#include <sys/poll.h>

struct pollfd {
       int fd;            /* file descriptor */
       short events;      /* requested events */
       short revents;     /* returned events */
};

int poll(struct pollfd *ufds,
         unsigned int nfds,
         int timeout);
```

FIG. 7.5 – Fonction poll(..).

Le choix de la fonction *poll()* vis-à-vis de l'autre principale fonction de multiplexage (*select()*) est principalement dû à trois éléments. D'abord, l'incapacité de *select()* de gèrer de grands ensembles de descripteurs de fichiers : la fonction *select()* pose une limite sur le nombre de descripteurs utilisables lors d'un appel (en général 1024). De surcroît, les opérations de manipulation des structures de données associées à *select()* (comme FD_ZERO, FD_ISSET, ...) sont implémentées sous forme de macros peu efficaces lorsqu'elles sont employées sur données de grande taille. Un autre élément de choix concerne la qualité des conditions rapportées sur les descripteurs de fichier : *select()* n'utilise qu'un sous-ensemble des évènements rapportés par *poll()* (*select()* permet uniquement de savoir si un descripteur de fichier est prêt pour la lecture/écriture alors que *poll()* permet en plus de connaître le détail des éventuelles conditions d'erreur). Enfin, sur les systèmes suppportant *poll()* de manière native (soit la quasi totalité des Unix actuels), *select()* est émulée

par *poll()*.

L'utilisation de la fonction *poll()* est assez simple : il suffit de lui fournir un tableau de structures *pollfd* contenant les descripteurs de fichier à surveiller (ici des sockets réseau) et les évènements que l'on souhaite surveiller sur chaque descripteur de fichier (par exemple *POLLIN* pour des données disponibles en lecture, *POLLERR* pour une condition d'erreur, ...). L'appel système renvoie immédiatment le nombre de descripteurs de fichiers actifs (le détail se trouve dans le champs *revents* de la structure *pollfd*) si celui-ci est supérieur à 0. Dans le cas contraire, l'appel bloque pendant *timeout* millisecondes.

Lors des montées en charge (par exemple lors de la gestion simultanée de 1000 connexions clients-serveurs), les performances subissent une baisse importante non pas à cause du volume de données à transmettre mais bien à cause d'une limitation des fonctions de gestion d'entrées-sorties asynchrones (*poll()* en particulier). Des mesures ont permis de noter que le processus proxy passait près de 30% de sa tranche de temps à copier des paramètres vers le noyau, et ce quelle que soit l'activité réelle de ces sockets [Kegel, 2003]. Ce délai est dû à la taille de la structure *pollfd* ; en effet un élément de cette stucture représente 64 bits en mémoire (32 bits pour le *int* et deux fois 16 bits pour les *short*), le tout multiplié par un millier de sockets. Cela représente 64kbits (8 KB) soit deux pages de mémoire. Des mesures de performance permettent de voir que la copie de données de l'espace utilisateur vers le noyau Linux est très pénalisante pour les performances dès que ces données représentent plus d'une page mémoire (4096 octets par défaut sur une architecture Intel x86).

7.1.3 Mono-processus

L'architecture de base du proxy était constituée d'un seul et unique processus utilisant un seul appel *poll()*. Il arrivait donc très vite à des tailles de paramètres engendrant de grosses pénalités de performances. Sa structure interne comportait plusieurs *thread* afin de permettre une gestion plus simple des requêtes DNS et des interactions avec les libraires de cryptages OpenSSL. Cet ensemble était donc en permanence sous la menace d'une faute de l'un des threads (exception/signal non géré, faute de mémoire, ...) ou de la librairie de gestion des threads (appel système bloquant mal intercepté,...), faute qui provoque l'arrêt de l'ensemble du proxy (plus précisement, l'OS arrête le processus responsable de la faute, sans tenir compte des threads qui le composent).

Un autre élément de performance à prendre en compte est la difficulté de cette architecture à supporter l'exécution parallèle sur une machine multi-processeur : si l'OS permet à la libraire de gestion des threads de lancer plusieurs exécutions parallèles sur plusieurs processeurs différents (ce qui n'est pas toujours le cas), le fait que l'ensemble des threads partagent le même espace d'adressage provoque un grand nombre de problèmes de gestion mémoire. L'OS doit en permanence resynchroniser les différentes mémoires caches afin que les modifications faites par un thread soient immédiatement visibles par les autres. Ce travail provoque une importante baisse des performances.

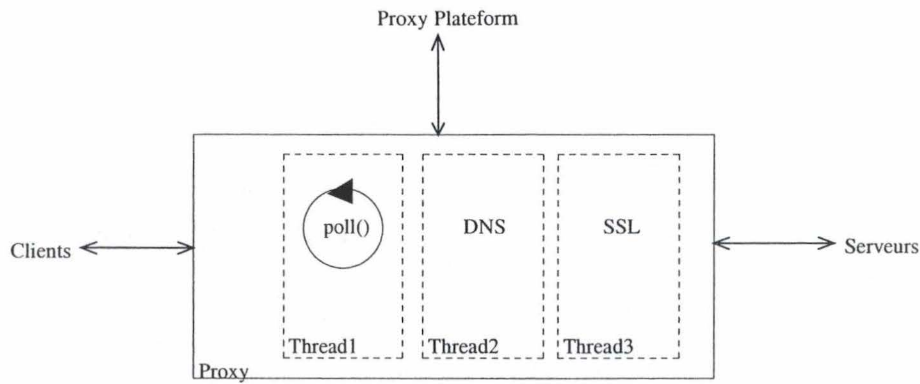

FIG. 7.6 – Proxy - vue rapprochée.

7.2 Proposition d'architecture

7.2.1 Principe

La solution au problème de montée en charge des appels *poll()* est leur démultiplication (figure 7.7) ; en effet, plusieurs appels consécutifs utilisant des paramètres plus petits sont plus rapides qu'un seul appel monolithique. Mais cette approche pose un gros problème : supposons que 5 appels consécutifs soient utilisés, un client dont la requête se trouverait prête sur un socket placé dans le cinquième appel doit attendre la complétion des 4 appels précédents avant d'être servi. Il n'est alors plus possible d'utiliser l'un des principaux avantages de *poll()* : la capacité d'endormir le processus en attendant qu'au moins un des sockets surveillés manifeste une quelconque activité. En conséquence le programme résultant sature à 100% le CPU, ce qui est difficilement acceptable sur un serveur devant par définition supporter plusieurs services en parallèle.

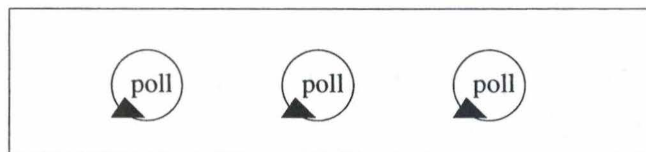

FIG. 7.7 – Solution : multiplication de l'appel *poll(..)*.

De même, la solution aux problèmes de gestionnaire de signaux unique, de mauvaise utilisation des ressources dans les cas multi-processeurs, ... est la démultiplication des contextes d'exécution et donc des processus (figure 7.8). Avec deux processus distincts accédant à des espaces mémoire disjoints, les problèmes de mise à jour de cache ne se posent plus ; et chaque processus peut s'exécuter sur un processeur particulier de manière parallèle aux autres processus. Le désagrément de l'utilisation de processus en lieu et place de threads est que les méthodes internes de communication disponibles pour les threads ne s'appliquent pas aux processus ; il convient donc d'utiliser les méthodes IPC classiques.

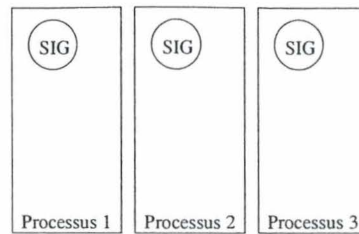

FIG. 7.8 – Solution : multiplication des processus.

Notre proposition d'architecture est la combinaison des deux solutions partielles : un ensemble de processus coordonnés, chacun gérant un certain nombre de clients et donc exécutant un appel *poll(..)* de taille restreinte (figure 7.9). Les appels *poll(..)* n'étant pas sérialisés, il est de nouveau possible de mettre les processus en attente tout en garantissant un service rapide aux différents clients.

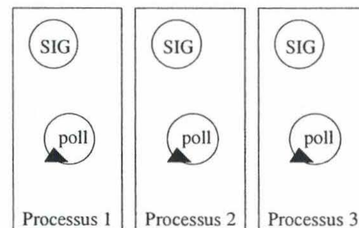

FIG. 7.9 – Solution : combinaison.

Il faut néanmoins garder à l'esprit que toutes les données reçues des clients et/ou des serveurs doivent transiter par la *Proxy Platform*. Celle-ci possède un seul point de communication avec le proxy : il s'agit d'une connexion TCP multiplexée. Ce point de passage unique représente clairement un lieu où la synchronisation des écritures des différents processus devra être gérée afin d'éviter une corruption des données et/ou des interblocages. Un processus supplémentaire a donc été ajouté afin de gérer les échanges de données entre les processus effectuant les appels *poll(..)* et entre ces processus et la *Proxy Platform*.

7.2.2 Choix de la méthode de communication

Cette section illustre comment les informations sur les diverses IPC présentées au chapitre 2 ainsi que les résultats de la classification du chapitre 6 peuvent être utilisés afin de concevoir une méthode de communication efficace.

Contraintes

Afin de clarifier les contraintes pesant sur la communication, nous les structurerons autours des critères définis au chapitre 6 (page 57) : portée, protection, parallélisme, propreté, préemption, restrictions et débit.

Pour la communication entre les différents processus du prototype, une portée locale est suffisante ; en effet, tous les processus seront créés par un seul et même processus.

La protection contre les accès non-autorisés est un pré-requis majeur : des données potentiellement sensibles peuvent transiter au travers du proxy. Il ne faut en aucun cas qu'une personne mal intentionnée puisse intercepter le contenu de la communication.

Le critère de parallélisme est ici très important : il faut que chaque processus puisse communiquer avec tous les autres (*full mesh* ou système de transit de messages).

La propreté de l'IPC sélectionnée n'est pas un réel critère de choix, dans la mesure où, si le noyau ne prend pas en charge la destruction des objets de comunications inutilisés, une procédure spécifique peut être prévue au sein du proxy.

La préemption des communications est souhaitée pour une raison simple : lors des pics de charge du proxy, le canal de comunication peut être encombré de données en cours de traitement, ce qui peut empêcher un processus de recevoir un ordre urgent. Par exemple, il est souhaitable qu'un processus réagisse immédiatement à un ordre d'arrêt.

Le critère portant sur les restrictions architecturales est ici relativement faible étant donné qu'il s'agit d'une conception sans réutilisation d'une base existante. La seule véritable contrainte est que l'IPC sélectionnée doit permettre de réveiller de manière propre un processus en attente dans un appel *poll(..)*.

Le débit de l'IPC sélectionnée doit être très important ; en effet, le proxy doit être capable de soutenir des transferts de gros volumes de données entre les clients et les serveurs. Ce qui implique que, pour ne pas ralentir l'application globale, le proxy doit au minimum supporter des débits internes égaux au débit de la connexion le reliant à la *ProxyPlatform*.

En résumé, on peut dire que les principaux critères à prendre en compte sont : la protection, le parallélisme, la préemption (pour certains messages uniquement) et le débit. Les trois autres critères (portée, propreté et restrictions) représentent plus des souhaits que de véritables contraintes (figure 7.1).

Portée	Protection	Parallélisme	Propreté	Préemption	Restrictions	Débit
⋆	⋆⋆⋆⋆	⋆⋆⋆⋆	⋆	⋆⋆⋆⋆	⋆	⋆⋆⋆⋆

TAB. 7.1 – Synthèse des éléments de choix.

IPC sélectionnées

Aucune des IPC présentées au chapitre 2 ne permet de satisfaire l'ensemble des contraintres énoncées. En conséquence, nous nous sommes tournés vers une combinaison d'IPC ; les trois IPC utilisées dans la communication interne sont les pipes, les signaux et les zones de mémoire partagée (Sys V). Chacune des IPC sélectionnées permet de résoudre de manière efficace une partie des contraintes.

L'utilisation de pipes unidirectionnels permet à chaque processus de disposer d'un canal de communication dédié avec un processus en charge du transit des messages internes. En outre, les descripteurs de fichiers identifiant les pipes peuvent être inclus dans les appels *poll(..)* permettant par là de réveiller le processus dès la réception d'un message.

L'utilisation de signaux pour les messages prioritaires (comme l'arrêt) permet d'éviter les délais parfois important pouvant survenir si le processus cible devait traiter l'ensemble des messages présents dans le pipe avant de recevoir le message d'arrêt.

Enfin, la mémoire partagée, utilisée en mode *zero-copy* (i.e. sans copie des données dans l'espace d'adressage local) permet à l'ensemble de soutenir des débits très importants. Un ensemble de sémaphores sont associés à cette zone de mémoire partagée ; leur rôle est uniquement d'empêcher les accès concurrents de se corrompre l'un l'autre. Ils ne possèdent aucune sémantique dans l'optique de la communication.

La combinaison de ces trois IPC permet bien de satisfaire les critères mis en évidence : la protection des données en transit est assurée par les permissions d'accès placées sur la zone de mémoire partagée ainsi que par le fait que les données transitant via un pipe sont invisibles à un processus extérieur. Afin d'obtenir une communication facile entre tous les processus sans multiplier le nombre de pipes, un système de transit de messages a été développé : un processus , capable de communiquer avec tous les autres processus, joue le rôle de "facteur" ; il règle le transit des messages internes. La préemption en cas de messages urgents est assurée au moyen de signaux. Enfin, l'utilisation de parties de la zone de mémoire partagée pour le transfert des données permet d'éviter que la communication interne ne se comporte comme un goulot d'étranglement dans le flux de communication global.

7.2.3 Description

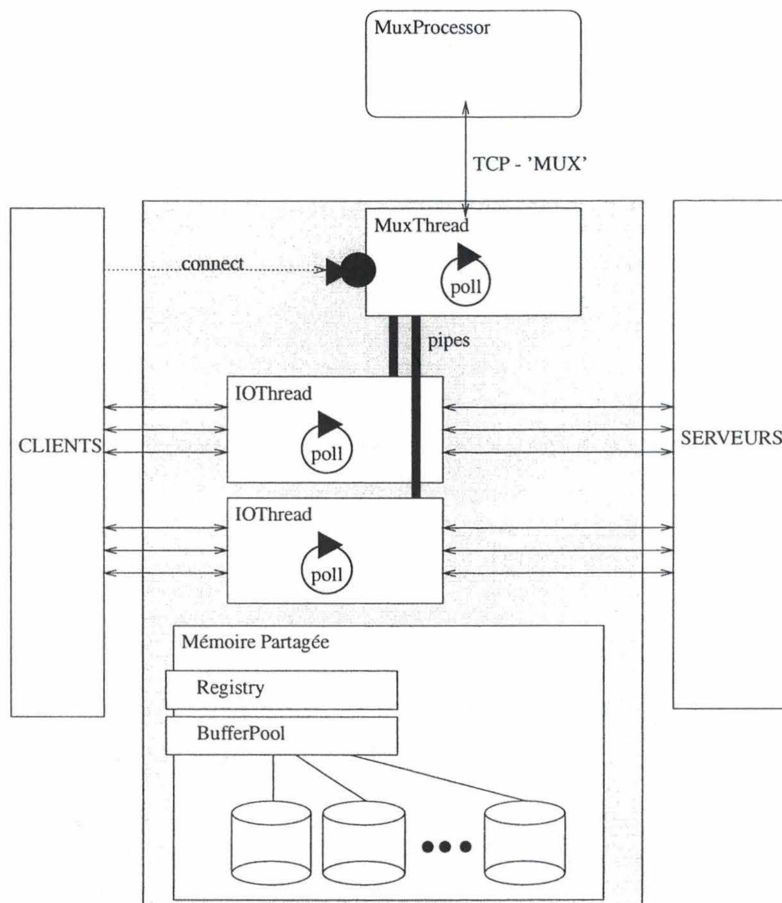

FIG. 7.10 – Architecture du prototype

La figure 7.10 présente l'architecture utilisée pour la réalisation du prototype de communication :

- Le **MuxProcessor** est un processus simulant l'action de la *Proxy Platform*. Son rôle est de filtrer les messages reçus et de les renvoyer sur le canal TCP multiplexé.
- Les **IOThread** sont les processus gérant les communications avec un ensemble de clients et/ou de serveurs. Ils contiennent les appels *poll()* utilisés pour détecter l'activité des sockets surveillés.
- Le **MuxThread** joue le rôle de coordinateur : il distribue les nouvelles connexions aux IOThread, envoie les signaux d'arrêt, gère le traffic de données entre les différents IOThread et gère la communication avec la *Proxy Platform* (ici le MuxProcessor).
- La zone de mémoire partagée entre tous les processus contient deux objets principaux :
  1. le *Registry* qui reprend l'ensemble de la configuration des différents processus.
  2. le *BufferPool* qui est une zone de mémoire partagée dans laquelle des équivalents des fonctions `malloc` et `free` vont allouer des buffers utilisables par tous les processus pour le transit des données volumineuses.
- La communication entre les différents processus sera développée plus en détail au point 7.2.4 page 77.

Le transit d'une requête se fait de la manière suivante (figure 7.10) :

1. Le MuxThread détecte une demande de nouvelle connexion, il sélectionne un IOThread en fonction de sa charge (un nouvel IOThread peut être créé si aucun n'est disponible) et l'avertit de la disponibilité d'un nouveau client.
2. L'IOThread (client) sélectionné accepte le client, lit sa requête et la transmet vers le MuxThread.
3. Le MuxThread reçoit les données de la requête ; il les envoie alors vers la *Proxy Platform*.
4. Le MuxThread reçoit les données (éventuellement modifiées) de la *Proxy Plateform*, il sélectionne alors un IOThread pour prendre en charge la communication avec le serveur correspondant et l'avertit de la disponibilité d'une requête.
5. L'IOThread (serveur) reçoit la requête et la transmet au serveur.
6. l'IOThread (serveur) reçoit la réponse du serveur.
7. Le trajet de retour de la réponse vers le client original est ensuite parfaitement symétrique avec l'envoi (passage par le MuxThread, puis envoi vers la *Proxy Platform*, transit par le MuxThread, envoi vers l'IOThread (client) et transmission au client).

7.2.4 Communication Interne

Mémoire partagée

La mémoire partagée contient deux objets principaux : le *Registry* et le *BufferPool*. Le premier est simplement un ensemble de couples (données, mécanisme de verrouillage) contenant les informations de configuration des différents processus. Le mécanisme de verrouillage est utilisé afin de garantir que les informations de cette "mémoire du système" ne puissent pas être altérées lors d'accès concurrents.

L'autre composant est le *BufferPool* (figure 7.11) : il s'agit d'une zone de mémoire partagée de taille relativement importante (pouvant atteindre la limite Sys V de 32 Mo) formatée de manière à pouvoir permettre une implémentation simple des fonctions *bf_malloc(..)* et *bf_free(..)* (figure 7.11).

```
#include ''bufferPoll.h''

extern void* bf_malloc(int size);
extern void bf_free(void* ptr);
```

FIG. 7.11 – Emulation de *malloc(..)/free(..)* en mémoire partagée.

Ces deux fonctions possèdent des prototypes équivalents aux fonctions standards `malloc` et `free`. Leur rôle est de gérer l'allocation et la désallocation de parties de la zone de mémoire partagée. Les tampons ainsi alloués sont alors utilisés pour le transfert des données entre les différents processus. Ici aussi les accès à cette partie de la mémoire sont protégés des accès concurrents par un sémaphore.

cet évènement et tenter de remettre le proxy dans un état stable. Si le processus détruit était un *IOThread*, les autres processus vont terminer les connexions interrompues et reprendre le fonctionnement normal. Dans le cas où c'est le processus de contrôle lui-même (*MuxThread*) qui est victime de la faute, les autres processus n'ont pas d'autres choix que de terminer leur exécution, le composant en charge du routage interne des messages ayant été détruit. Dans ce dernier scénario, les ressources globales allouées par le système pour les IPC (en particulier pour les sémaphores et les mémoires partagées) ne sont pas détruites, le processus en charge de leur élimination (*MuxThread*) n'étant plus en mesure d'effectuer l'opération. En conséquence, ces ressources restent présentes sur le système jusqu'à une intervention manuelle de l'administrateur ou jusqu'au redémarrage de la machine.

Un autre problème survenant en cas de crash d'un des processus est lié à la création du fichier *core*. Ce fichier reprend une image du contenu de la zone mémoire du processus ayant provoqué la faute et est utilisé pour permettre d'identifier celle-ci. Mais la zone de mémoire partagée n'est pas reprise dans ce fichier *core* (figure 7.14).

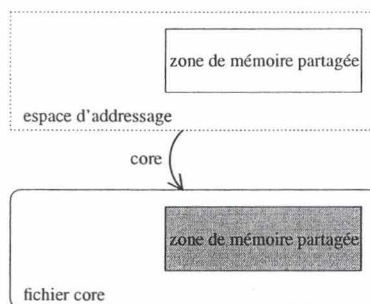

FIG. 7.14 – Création de fichier *core* incomplet.

La zone de mémoire partagée, bien que faisant partie de l'espace d'adressage du processus, n'est pas reprise dans le fichier core.

Opportunités

Avec l'arrivée des noyaux Linux 2.6, sont apparus des mécanismes jouant le même rôle que *poll(..)* mais souffrant beaucoup moins des effets liés à la taille de leurs paramètres ; ces mécanismes (*epoll(..)* et /dev/poll en particulier) peuvent représenter une alternative à la multiplication des processus. Mais les premiers noyaux 2.6 sont apparus trop tardivement (fin décembre 2003) pour pouvoir entrer en ligne de compte dans le choix de l'architecture. Il s'agit néanmoins d'une opportunité pour le futur : ces évolutions de *poll(..)* permettent de gérer plus de connexions pour un même nombre de processus ou, de manière symétrique, de diminuer le nombre de processus tout en conservant le même temps de réponse.

Une autre aspect lié aux noyaux 2.6 est l'utilisation possible des *futex* ; ces dispositifs sont des verrous situés dans l'espace utilisateur qui ne font appel aux mécanismes de protection du noyau que dans les cas où un

Le transit d'une requête se fait de la manière suivante (figure 7.10) :

1. Le MuxThread détecte une demande de nouvelle connexion, il sélectionne un IOThread en fonction de sa charge (un nouvel IOThread peut être créé si aucun n'est disponible) et l'avertit de la disponibilité d'un nouveau client.
2. L'IOThread (client) sélectionné accepte le client, lit sa requête et la transmet vers le MuxThread.
3. Le MuxThread reçoit les données de la requête ; il les envoie alors vers la *Proxy Platform*.
4. Le MuxThread reçoit les données (éventuellement modifiées) de la *Proxy Plateform*, il sélectionne alors un IOThread pour prendre en charge la communication avec le serveur correspondant et l'avertit de la disponibilité d'une requête.
5. L'IOThread (serveur) reçoit la requête et la transmet au serveur.
6. l'IOThread (serveur) reçoit la réponse du serveur.
7. Le trajet de retour de la réponse vers le client original est ensuite parfaitement symétrique avec l'envoi (passage par le MuxThread, puis envoi vers la *Proxy Platform*, transit par le MuxThread, envoi vers l'IOThread (client) et transmission au client).

7.2.4 Communication Interne

Mémoire partagée

La mémoire partagée contient deux objets principaux : le *Registry* et le *BufferPool*. Le premier est simplement un ensemble de couples (données, mécanisme de verrouillage) contenant les informations de configuration des différents processus. Le mécanisme de verrouillage est utilisé afin de garantir que les informations de cette "mémoire du système" ne puissent pas être altérées lors d'accès concurrents.

L'autre composant est le *BufferPool* (figure 7.11) : il s'agit d'une zone de mémoire partagée de taille relativement importante (pouvant atteindre la limite Sys V de 32 Mo) formatée de manière à pouvoir permettre une implémentation simple des fonctions *bf_malloc(..)* et *bf_free(..)* (figure 7.11).

```
#include ''bufferPoll.h''

extern void* bf_malloc(int size);
extern void bf_free(void* ptr);
```

FIG. 7.11 – Emulation de *malloc(..)/free(..)* en mémoire partagée.

Ces deux fonctions possèdent des prototypes équivalents aux fonctions standards `malloc` et `free`. Leur rôle est de gérer l'allocation et la désallocation de parties de la zone de mémoire partagée. Les tampons ainsi alloués sont alors utilisés pour le transfert des données entre les différents processus. Ici aussi les accès à cette partie de la mémoire sont protégés des accès concurrents par un sémaphore.

Entre IOThread et MuxThread

La communication entre chaque couple de processus (IOThread, MuxThread) s'effectue au moyen d'une combinaison de pipes, de mémoire partagée et de signaux.

Deux pipes sont utilisés : un pour chaque direction de communication, afin de ne pas devoir tenter de déterminer à quel processus les messages sont destinés. Ce principe de fonctionnement permet d'utiliser le pipe de manière plus efficace : il n'est pas nécessaire d'attendre que le premier message soit lu avant d'écrire le suivant (dans la limite de la taille du pipe) car tous les messages du pipe sont destinés au même processus. Dans le cas de messages de taille relativement faible (de l'ordre de quelques octets à quelques dizaines d'octets), les messages sont transmis tels quels dans le pipe (figure 7.12), tandis que les messages de taille plus importante sont placés dans un tampon alloué en mémoire partagée et seul un pointeur vers cette zone transite via le pipe (figure 7.13). Ce procédé permet de transmettre de grandes quantités de données en un temps très court (la quantité de données transmise par cycle dépend uniquement de la taille du plus grand tampon de mémoire partagée disponible).

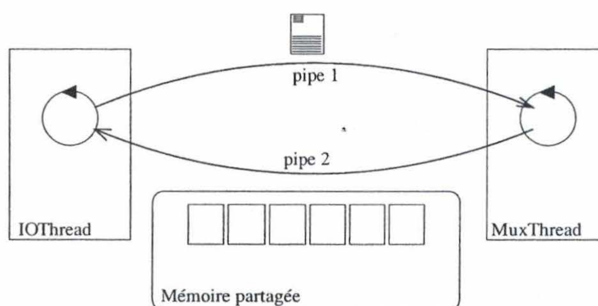

FIG. 7.12 – Communication composite : Message court.

Dans ce cas, le message est transmis tel quel sur le pipe de communication.

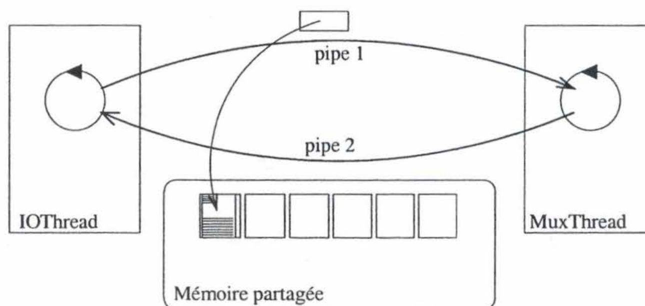

FIG. 7.13 – Communication composite : Message long.

Dans ce cas, seul un pointeur vers un tampon de mémoire partagée est transmis via le pipe, les données sont placées en mémoire partagée.

Un avantage de ce choix de design de communication interne est que les données dont la taille dépasse quelques dizaines de bytes circulent dans le proxy en mode *zero-copy* : elles ne sont jamais recopiées. Les données lues sur le réseau sont directement placées dans des buffers de mémoire partagée et circuleront entre les différents processus à l'intérieur de ce buffer jusqu'à leur sortie du proxy. Les temps nécessaires à la copie des données d'un buffer à un autre à chaque passage de processus sont ainsi évités.

L'utilisation de pipes pour la communication entre les processus répond aussi à un autre impératif : afin de permettre à un processus de réagir le plus vite possible à un message, il faut s'assurer que ce dernier ne puisse pas rester en attente dans un appel *poll()* alors que des données à traiter sont disponibles. Le pipe étant géré au moyen de descripteurs de fichiers, ce problème peut être facilement résolu en ajoutant le descripteur de fichier correspondant au pipe dans la liste des sockets surveillés par *poll()*. Ce fonctionnement n'aurait pas été possible avec d'autres moyens de communication comme les files de messages ou les signaux[17].

Entre MuxThread et MuxProcessor

La communication entre le MuxThread et le MuxProcessor est effectuée au moyen d'une connexion TCP traditionnelle en suivant un protocole de multiplexage simple : les parties de messages lues sur le réseau sont encapsulées dans une structure reprennant les informations sur le client (ou le serveur) source et sont placées dans la file des messages en attente de transfert. Ce transfert est optimisé afin d'éviter au maximum de transmettre des trames TCP incomplètes (qui engendrent à chaque fois le coût d'une écriture vers le réseau alors que la quantité de données à écrire est non optimale).

Conclusion

Ce prototype simple regroupe déjà un nombre conséquent d'IPC : les pipes, les mémoires partagées, les sockets réseau (tcp), les sockets réseaux avec protocoles (mux), les sémaphores, les signaux et la communication par fichier placé sur le disque (pour le système de logging).

Prises individuellement, ces IPC ne supportent chacunes qu'une petite partie de la communication, mais de l'ensemble nait une communication fiable, structurée et capable de résister à des montées en charge.

7.2.5 Critique de l'architecture

Problèmes connus

Le principal problème non-résolu par cette architecture est celui de l'arrêt en cas de crash ; si un processus effectue une opération invalide (faute mémoire, ...) et se termine, les autres processus vont eux-aussi réagir à

[17]Le cas des signaux pose encore plus de problèmes : il faut être certain que le processus cible est dans son appel *poll()* avant de lancer le signal, sous peine de voir le processus interrompu en plein milieu d'une opération de lecture/écriture

cet évènement et tenter de remettre le proxy dans un état stable. Si le processus détruit était un *IOThread*, les autres processus vont terminer les connexions interrompues et reprendre le fonctionnement normal. Dans le cas où c'est le processus de contrôle lui-même (*MuxThread*) qui est victime de la faute, les autres processus n'ont pas d'autres choix que de terminer leur exécution, le composant en charge du routage interne des messages ayant été détruit. Dans ce dernier scénario, les ressources globales allouées par le système pour les IPC (en particulier pour les sémaphores et les mémoires partagées) ne sont pas détruites, le processus en charge de leur élimination (*MuxThread*) n'étant plus en mesure d'effectuer l'opération. En conséquence, ces ressources restent présentes sur le système jusqu'à une intervention manuelle de l'administrateur ou jusqu'au redémarrage de la machine.

Un autre problème survenant en cas de crash d'un des processus est lié à la création du fichier *core*. Ce fichier reprend une image du contenu de la zone mémoire du processus ayant provoqué la faute et est utilisé pour permettre d'identifier celle-ci. Mais la zone de mémoire partagée n'est pas reprise dans ce fichier *core* (figure 7.14).

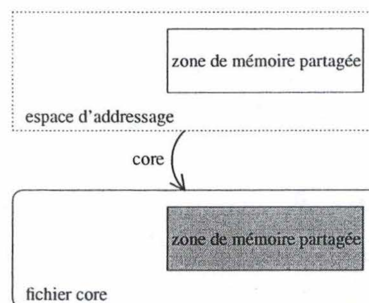

FIG. 7.14 – Création de fichier *core* incomplet.

La zone de mémoire partagée, bien que faisant partie de l'espace d'adressage du processus, n'est pas reprise dans le fichier core.

Opportunités

Avec l'arrivée des noyaux Linux 2.6, sont apparus des mécanismes jouant le même rôle que *poll(..)* mais souffrant beaucoup moins des effets liés à la taille de leurs paramètres ; ces mécanismes (*epoll(..)* et /dev/poll en particulier) peuvent représenter une alternative à la multiplication des processus. Mais les premiers noyaux 2.6 sont apparus trop tardivement (fin décembre 2003) pour pouvoir entrer en ligne de compte dans le choix de l'architecture. Il s'agit néanmoins d'une opportunité pour le futur : ces évolutions de *poll(..)* permettent de gérer plus de connexions pour un même nombre de processus ou, de manière symétrique, de diminuer le nombre de processus tout en conservant le même temps de réponse.

Une autre aspect lié aux noyaux 2.6 est l'utilisation possible des *futex* ; ces dispositifs sont des verrous situés dans l'espace utilisateur qui ne font appel aux mécanismes de protection du noyau que dans les cas où un

objet utilisateur ne peut pas résoudre le conflit. Leur utilisation permet de diminuer le nombre d'appels de synchronisation envoyés au noyau et donc d'augmenter les performances.

7.3 Mesures de performances

7.3.1 Procédure de test

Les mesures de performances effectuées visent à mesurer le temps nécessaire à un client HTTP pour recevoir la réponse à une requête émise à travers le proxy. Afin de simuler au mieux ce comportement, le prototype fut placé entre un ensemble de programmes écrits en C simulant les clients et un serveur *boa* (*Boa* est un serveur HTTP simple et économe en ressources ; la version utilisée est la version 0.94.13) (figure 7.15)

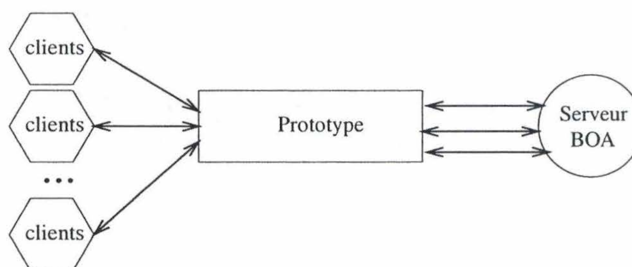

FIG. 7.15 – Configuration de test.

Le délai mesuré est celui entre le moment où le client émet sa requête et le moment où il reçoit sa réponse.

Lors des tests, les clients, le prototype et le serveur *boa* étaient tous implantés sur la même machine, et ce afin d'éviter autant que possible les latences dues au réseau de communication. Avant chaque test, tous les éléments étaient redémarrés afin de les remettre dans leur configuration par défaut. En outre, un délai minimum de 2 minutes (correspondant au *time-out TCP*) était observé entre chaque phase de test afin que la charge introduite sur le noyau par la gestion de milliers de connexions en phase de TIME_WAIT[18] ne perturbe pas les mesures.

Pour les besoins du test, le nombre maximum de descripteurs de périphérique autorisés pour l'utilisateur courant a été augmenté de 1024 à 4096. Les options suivantes ont été modifiées dans la configuration par défaut du serveur *boa* afin de permettre un plus grand nombre de connexions persistantes HTTP 1.1 simultanées :

[18] [Kegel, 2003] contient une estimation du coût en matière de performances causé par la gestion des ces connexions en attente de terminaison.

```
# KeepAliveMax: Number of KeepAlive requests to allow
#               per connection
# Comment out,or set to 0 to disable keepalive processing
KeepAliveMax 10000

# KeepAliveTimeout: seconds to wait before keepalive
#                   connection times out
KeepAliveTimeout 120
```

FIG. 7.16 – Options du serveur *boa*.

Remarque : Les mesures effectuées dans la suite visent uniquement à permettre d'évaluer les performances du prototype seul ; elles ne permettent en rien de comparer le prototype au proxy existant. En effet, le prototype ne supporte qu'un sous-ensemble des fonctions du proxy complet et donc la comparaison des performances des deux architectures n'aurait que peu de sens.

7.3.2 Résultats

La forme concise de notation des principales informations de configuration est utilisée dans la suite :

A - B - C : D - E

où[19] :

A représente le nombre minimum de processus en charge de la gestion des clients/serveurs (*IOThread*).

B représente le nombre maximum de processus en charge de la gestion des clients/serveurs (*IO-Thread*).

C représente le nombre maximum de connexions simultanées autorisées pour un *IOthread*.

D représente le nombre de connexions persistantes HTTP 1.1 traversant le prototype avant le début des mesures. Ce paramètre est utilisé afin de simuler le comportement hors charge (aucune connexion avec le début des mesures) et le comportement en charge (de nombreuses connexions traversent déjà le proxy avant le début des mesures, mais ces connexions sont inactives pendant les mesures).

E représente le nombre de connexions clients simultanées utilisées afin de mesurer le temps moyen de réponse envers le client.

Par exemple, 2-4-500 :300-500 signifie que les mesures ont été effectuées sur un prototype dont le nombre de *IOThread* variait dynamiquement entre 2 et 4, chacun d'entre eux acceptant un maximum de 500 connexions simultanées. Au début de ce test, le prototype était traversé par 300 connexions persistantes HTTP 1.1 et le test a porté sur la mesure du temps nécessaire à 500 clients pour être servis.

Afin de pouvoir comparer ces résultats aux temps de traitement des requêtes en l'absence du prototype, le temps de réponse du serveur *boa* a chaque fois été indiqué sur les graphiques sous la forme *Boa : D - E*.

Chaque test a été répété 10 fois afin de fournir un temps moyen relativement indépendant des éventuelles perturbations extérieures.

Remarques :

1. Lors des tests, le nombre maximum de processus en charge des I/O clientes a été plafonné à 8 ; en effet, au delà de cette limite, les performances commencent à chuter de manière importante en raison de la charge représentée par le scheduling et la protection de multiples processus concurrents. En outre les charges simulées lors des tests n'ont jamais permis de saturer 8 processus.
2. L'adaptation dynamique du nombre de processus à la charge n'a pas été employée dans les tests.
3. Les nombre maximum de connexions simultanées sur le serveur *boa* fut lui aussi plafonné à un maximum de 1000, valeur au delà de laquelle le temps de réponse du serveur *boa* devient incertain. L'utilisation d'un serveur HTTP plus puissant (comme *apache*) pourrait permettre de dépasser cette

[19] Pour être complet, il faut signaler que les valeurs de ces cinq paramètres sont liées : le test ne peut être effectué dans de bonnes conditions que si $B * C >= 2 * (D + E)$, ce qui revient à dire que le prototype doit pouvoir au minimum ouvrir une connexion entrante et une connexion sortante pour chaque client

limite, mais la charge représentée par un serveur *apache* local aurait faussé les résultats des mesures sur le prototype.

Résultats hors-charge

Dans ce premier cas de figure, le prototype était exempt de toute charge avant les tests.

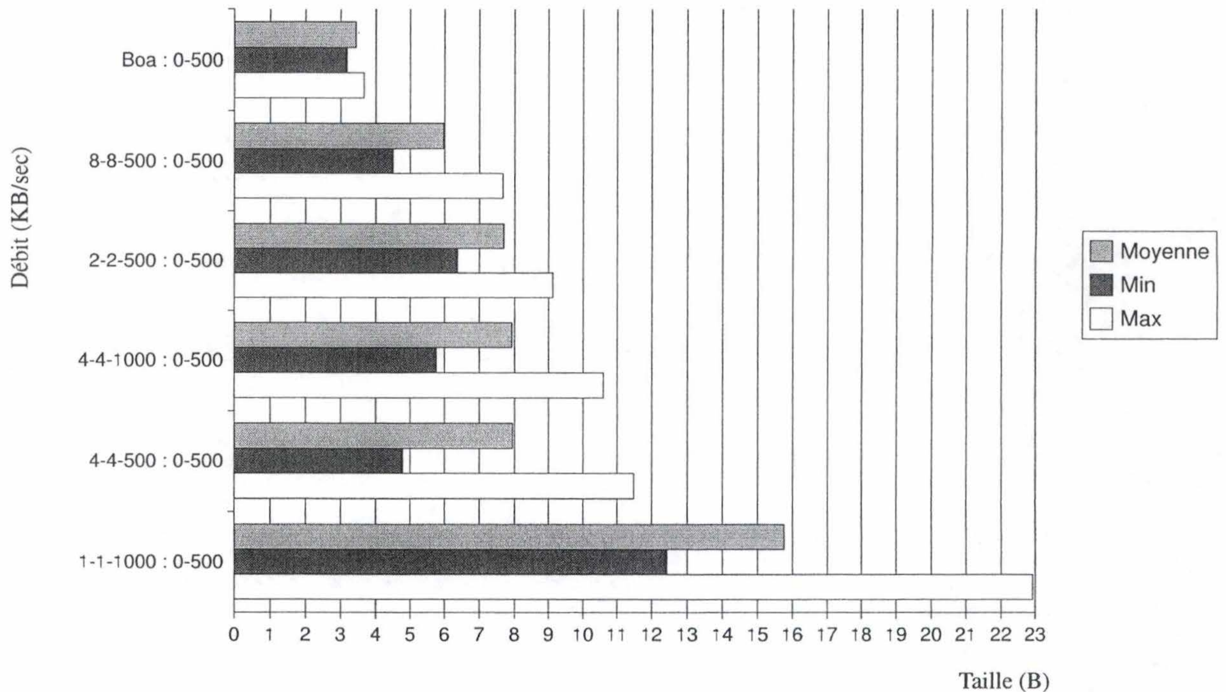

FIG. 7.17 – Résultats hors charge.

Tri sur le temps moyen. Tous les temps sont exprimés en secondes.

Configuration	Moyenne	VAR	STDEV	Ecart max	Pop	Min	Max
Boa : 0-500	3,435	0,55%	25,81%	7,49%	10	3,178	3,666
8-8-500 : 0-500	5,975	16,90%	38,23%	28,32%	10	4,496	7,667
2-2-500 : 0-500	7,686	8,07%	16,32%	18,57%	10	6,384	9,113
4-4-1000 : 0-500	7,930	47,19%	24,40%	33,55%	10	5,776	10,590
4-4-500 : 0-500	7,949	78,87%	31,50%	44,50%	10	4,781	11,486
1-1-1000 : 0-500	15,752	65,48%	20,39%	45,55%	10	12,442	22,927

TAB. 7.2 – Résultats hors charge.

Résultats des différentes configurations hors charge ; tri sur le temps moyen.

Tous les temps sont exprimés en secondes.

Les résultats de ce test (figure 7.17 et tableau 7.2) sont éloquents : les performances moyennes de la solu-

tion mono-processus sont inférieures d'un facteur trois aux performances moyennes de la meilleure solution multi-processus (8-8-500). En outre, dans cette configuration, le prototype réagit dans les même délais que le serveur *boa* : le temps de complétion des requêtes clients au travers du proxy est le double du temps nécessaire à la complétion de ces mêmes requêtes lorsqu'elles sont envoyées directement sur le serveur *boa* (cas *Boa :0-500* sur la figure 7.17).

Ce test permet aussi de constater que plus le nombre de processus est grand[20], plus les performances augmentent (le cas 2-2-500 ne dépasse les cas à 4 processus que d'une fraction de seconde ; ce délai provient probablement de la plus grande charge de scheduling engendrée par un doublement du nombre de processus).

[20]Tout en restant sous la limite de 8 processus d'entrée/sortie.

Résultats en charge

Lors de ce test, le prototype était chargé de 500 connexions HTTP 1.1 persistantes avant le début du test.

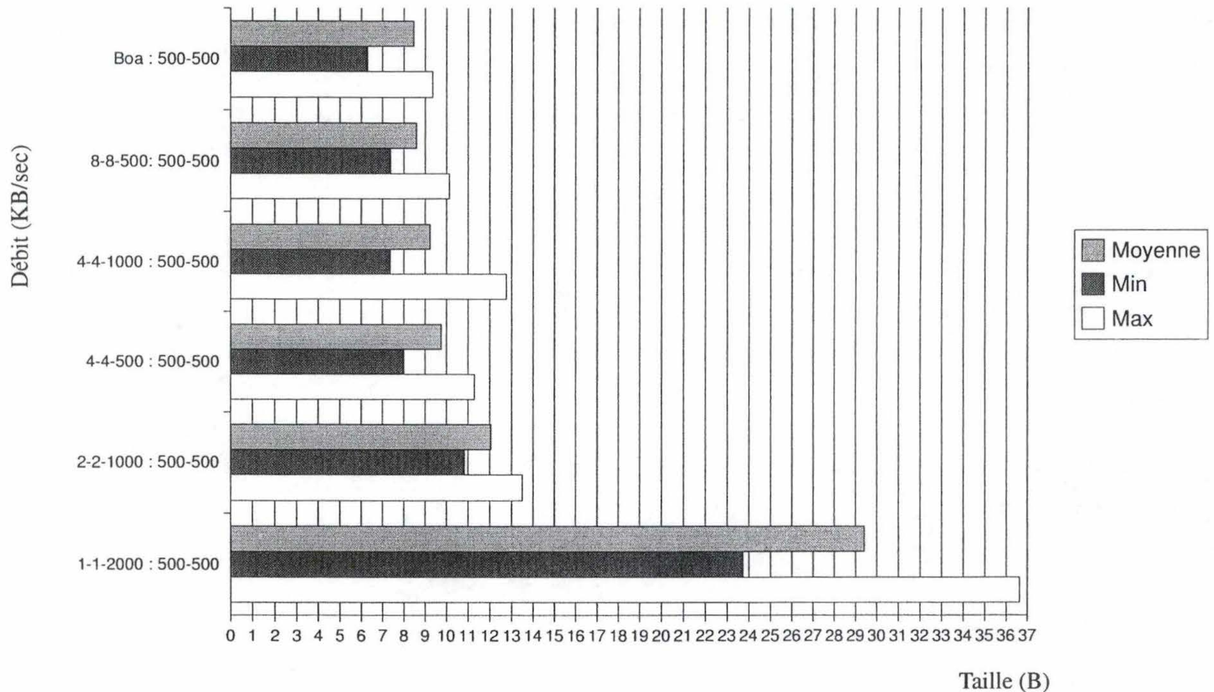

FIG. 7.18 – Résultats en charge.

Tri sur le temps moyen. Tous les temps sont exprimés en secondes.

Configuration	Moyenne	VAR	STDEV	Ecart max	Pop	Min	Max
Boa : 500-500	8,462	20,81%	18,73%	25,66%	10	6,291	9,360
8-8-500 : 500-500	8,595	15,16%	13,28%	17,93%	10	7,370	10,136
4-4-1000 : 500-500	9,245	26,63%	24,33%	38,05%	10	7,368	12,763
4-4-500 : 500-500	9,764	14,40%	12,14%	18,42%	10	7,965	11,309
2-2-1000 : 500-500	12,068	6,29%	7,22%	11,88%	10	10,838	13,502
1-1-2000 : 500-500	29,417	85,44%	21,75%	24,51%	10	23,729	36,628

TAB. 7.3 – Résultats en charge.

Résultats des différentes configurations en charge ; tri sur le temps moyen.

Tous les temps sont exprimés en secondes.

Les résultats des tests en charge (figure 7.18 et tableau 7.3) viennent confirmer les résultats hors-charge :

de nouveau les configurations présentant le plus grand nombre de processus montrent de meilleures performances que les configurations plus simples.

En outre, on peut voir que le serveur *boa* approche de son seuil limite de performances avec 1000 connexions persistantes simultanées. Il agit même comme un facteur de limitation des performances de la solution *8-8-500* (8.462 sec pour les 500 requêtes envoyées directement vers le serveur *boa* contre 8.595 sec lorsque ces requêtes sont envoyées au travers du prototype *8-8-500*).

Résultats généraux

La compilation des tests hors charge et en charge fait ressortir un élément révélateur (figure 7.19 et tableau 7.4) : les configurations mono-processus (*1-1-X*) sont largement distancées par les configurations multi-processus. Cet écart est tel que les configuration multi-processus en charge sont plus performantes que la configuration mono-processus hors charge.

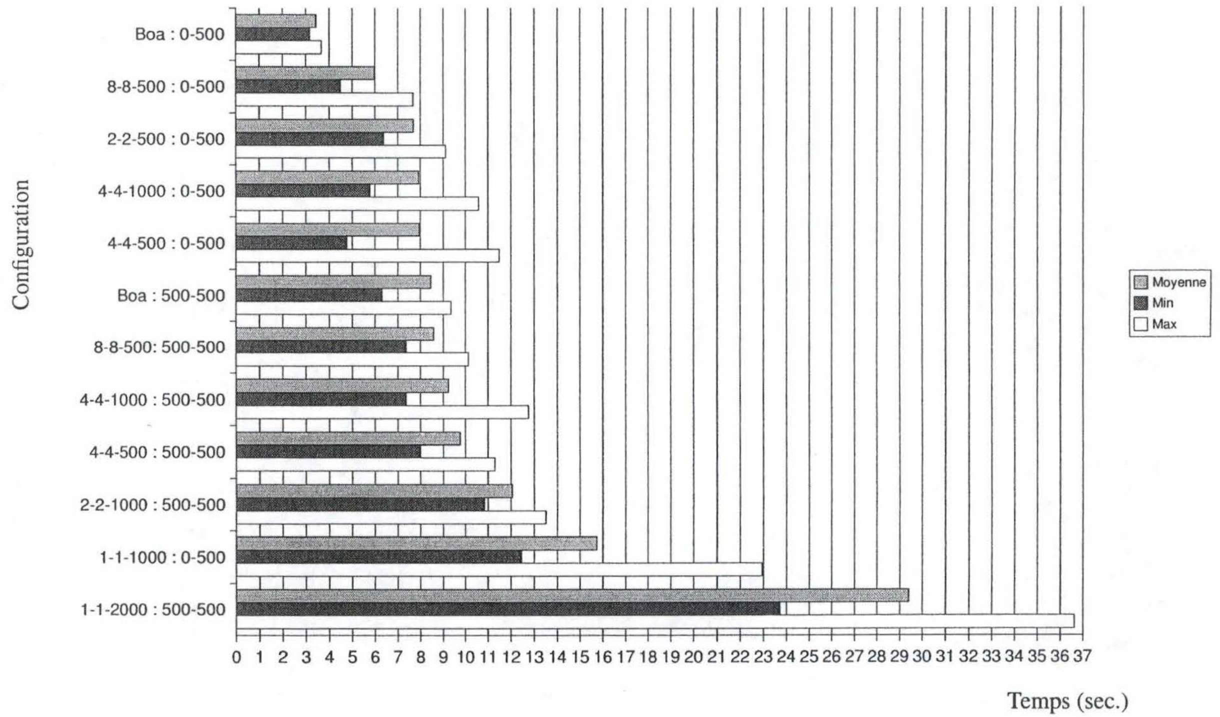

FIG. 7.19 – Résultats

Résultats des différentes configurations ; tri sur le temps moyen. Tous les temps sont exprimés en secondes.

Configuration	Moyenne	VAR	STDEV	Ecart max	Pop	Min	Max
Boa : 0-500	3,435	0,55%	25,81%	7,49%	10	3,178	3,666
8-8-500 : 0-500	5,975	16,90%	38,23%	28,32%	10	4,496	7,667
2-2-500 : 0-500	7,686	8,07%	16,32%	18,57%	10	6,384	9,113
4-4-1000 : 0-500	7,930	47,19%	24,40%	33,55%	10	5,776	10,590
4-4-500 : 0-500	7,949	78,87%	31,50%	44,50%	10	4,781	11,486
Boa : 500-500	8,462	20,81%	18,73%	25,66%	10	6,291	9,360
8-8-500 : 500-500	8,595	15,16%	13,28%	17,93%	10	7,370	10,136
4-4-1000 : 500-500	9,245	26,63%	24,33%	38,05%	10	7,368	12,763
4-4-500 : 500-500	9,764	14,40%	12,14%	18,42%	10	7,965	11,309
2-2-1000 : 500-500	12,068	6,29%	7,22%	11,88%	10	10,838	13,502
1-1-1000 : 0-500	15,752	65,48%	20,39%	45,55%	10	12,442	22,927
1-1-2000 : 500-500	29,417	85,44%	21,75%	24,51%	10	23,729	36,628

TAB. 7.4 – Résultats

Résultats des différentes configurations ; tri sur le temps moyen.Tous les temps sont exprimés en secondes.

Chapitre 8 : Conclusion

Les méthodes de communication inter-processus couvrent toute la gamme des besoins de communication des processus. Que ce soit pour une communication ponctuelle ou pour un flux continu, pour quelques octets ou pour plusieurs méga-octets, il existe toujours une IPC capable de supporter la communication.

La présentation des différentes méthodes a permis de constater la grande similitude existant entre les IPC des deux plateformes analysées : les même concepts se retrouvent dans les deux cas (par exemple : les mémoires partagées, les tubes anonymes ou nommés, les signaux, ...). Ces concepts possèdent chaque fois le même type d'avantages et de limitations.

Chaque IPC possède ses spécificités propres en termes de portée, de débit maximum, de sécurité, ... Toutes ces caractéristiques déterminent les conditions dans lesquelles les performances de l'IPC seront maximales et donc aussi les situations où l'utilisation de cet IPC sera à privilégier.

Il n'existe pas (ou rarement) d'IPC parfaitement adaptée à une situation de communication, il n'existe que des combinaisons d'IPC permettant de s'approcher au plus près du résultat désiré. En utilisant plusieurs IPC comme support à la communication, le programmeur peut tirer profit des avantages de chacunes et éviter autant que possible leurs points faibles. Le choix des IPC dépend aussi de l'investissement auquel le programmeur est prêt à consentir afin de garantir la gestion de la communication. Les méthodes les plus simples à utiliser étant souvent les méthodes les moins performantes.

Les méthodes entièrement gérées par le noyau sont les méthodes les plus simples d'emploi car tous les incidents possibles lors de la communication sont gérés en interne par le noyau. Ces IPC représentent malheureusement souvent les méthodes les moins performantes, l'intervention du noyau se devant d'être générique. Cette situation s'oppose à une communication entièrement gérée par le programmeur où seules les tâches de gestion/synchronisation réellement nécessaires sont exécutées et donc où les performances sont maximales.

Le choix de l'IPC optimale est donc à faire au cas par cas, en fonction de tous les éléments disponibles (portée, protection, parallelisme, propreté, préemption, restrictions sur l'architecture, débit, ...). Il n'existe à ce jour aucune recette associant un contexte de communication et l'IPC optimale à utiliser. Pas plus qu'il n'existe de méthode complète permettant de décider si une combinaison d'IPC sera plus effiace qu'une autre sans devoir passer par l'implémentation de prototypes et la mesure des performances atteintes sur quelques communications de test.

Le choix des IPC utilisées est à faire très tôt dans le cycle de développement d'une application ; car il conditionne parfois fortement les choix architecturaux possibles. Il convient de bien spécifier les souhaits et les contraintes relatifs à la communication afin de déterminer au mieux les IPC suceptibles de convenir à la

situation.

Le présentation des IPC distribuées, nous a permis de constater quelques-uns des problèmes engendrés par la mise en relation de processus distants. Le domaine des IPC distribuées est en plein essor, suivant par là la croissance rapide des technologies liées à l'Internet. Les IPC distribuées représentent plus que probablement la prochaine évolution permettant d'augmenter la puissance disponible sans pour autant faire exploser les coûts. Les technologies permettant de combiner les puissances de calcul de différentes machines (comme les technologies de *clustering*) en sont les prémisses.

Enfin, l'approche des IPC complètement indépendantes de la plateforme nous a permis de constater que ce type de méthodes n'est utilisée que dans des situations très particulières et afin de répondre à des demandes très précises. En effet, la mise en place d'une telle IPC pose des problèmes très similaires à ceux créés par la mise en place d'une IPC distribuée sans pour autant en apporter les avantages en terme de ditribution.

Annexes

Annexe I : Protocole de mesure de performances des IPC Unix

La machine utilisée pour les mesures de performances possédait la configuration suivante :

Processeur	AMD Athlon Thunderbird C 1200Mhz
Mémoire	512 Mo SD-Ram
Noyau	Linux 2.4.25
Distribution	Debian Woody 3.0r2 (stable)

Toutes les mesures ont été effectuées au moyen de programmes ANSI-C spécifiques. Les outils suivants ont été utilisés pour la compilation :

Compilateur	Gcc 2.95.4
Linker	GNU ld version 2.12.90.0.1 20020307
LibC	libC 2.2.5-11.5

Tous les programmes utilisaient les optimisations avancées de *gcc* (option -03). Les mesures de temps ont été effectuées au moyen de la fonction *gettimeofday(..)* dont la granularité de $1\mu sec$ s'est avérée parfois insuffisante (cas des transferts par mémoire partagée sans copie dans l'espace local[21]).

```
#include <sys/time.h>

struct timeval {
   long tv_sec;   /* seconds */
   long tv_usec;  /* microseconds */
};

int gettimeofday(struct timeval *tv, struct timezone *tz);
```

FIG. 9.1 – Fonction *gettimeofday(..)*

Chaque test s'est déroulé de la manière suivante (figure 9.2) : un processus père créait un processus fils et lui transmettait au moyen de l'IPC choisie un bloc binaire dont la taille était fournie en paramètre. Le temps mesuré était le temps entre le moment ou le père émet le premier octet du bloc binaire et le moment où le fils recevait le dernier octet de celui-ci. Il est important de noter ici que les mesures ne prennent pas en compte le temps nécessaire à la mise en place et à la destruction de l'IPC.

[21]Ce cas de figure extrême engendre des délais théoriques de l'ordre de quelques cycles CPU. Cette granularité est inaccessible aux fonctions classiques de mesure du temps.

Chaque test a été conduit sur une machine inactive (*idle*) et reproduit à plusieurs reprises afin d'éviter autant que possible les interférences éventuelles avec d'autres processus du système. La valeur présentée dans les graphiques est la moyenne des quotients de la taille de données transférées et des temps mesurés.

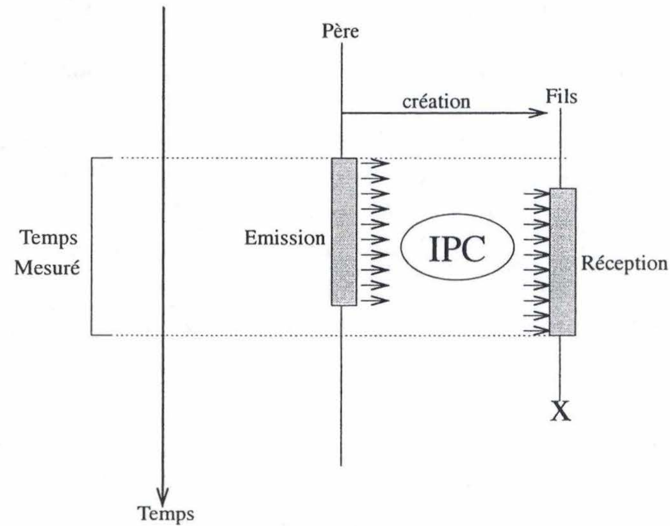

FIG. 9.2 – Mesure du temps de transfert.

Le temps mesuré est celui entre le début de l'émission par le père et la fin de la réception par le fils.

Annexe II : Bibliographie

[Besaw, 1987] Besaw, L. (1987). Berkeley unix system calls and interprocess communication. revised September 1987, January 1991 by Marvin Solomon.

[Emmerich, 2000] Emmerich, W. (2000). Engineering Distributed Objects. John Wiley & Sons, Ltd.

[Englebert, 2003] Englebert, V. (2003). Cours de systèmes coopératifs. `http://www.info.fundp.ac.be/~ven/CIS`.

[George Coulouris and Kindberg, 1994] George Coulouris, J. D. and Kindberg, T. (1994). Distributed Systems : Concept and Design - Second Edition. Addison-Wesley.

[Group, 2003] Group, T. O. (2003). The authorized guide to the single unix specification, version 3.

[Hanus, 2004] Hanus, M. (2004). Multi-paradigm declarative programming in curry. Chistian-Albrechts-Universität Kiel.

[Hemsley, 2002] Hemsley, R. (2002). Sys v ipc vs. unix pipes vs. unix sockets. `http://rikkus.info/sysv_ipc_vs_unix_sockets.html` dernière visite le 20 janvier 2004.

[Kegel, 2003] Kegel, D. (2003). The c10k problem. `http://www.kegel.com/c10k.htm` dernière visite le 16 décembre 2003, dernière mise à jour le 16 août 2003.

[Kortesmaa, 2003] Kortesmaa, T. (2003). Multi-process programming and inter-process communication (ipc). `http://users.evtek.fi/~tk/rtp/multi-process.html` dernière visite le 20 janvier 2004.

[Kurose and Ross, 2003] Kurose, J. F. and Ross, K. W. (2003). Computer Networking : A Top Down Approach Featuring The Internet - Second Edition. Addison-Wesley.

[Labo-Unix, 2002] Labo-Unix (2002). Communication inter-processus. `http://www.labo-unix.net` dernière visite le 20 janvier 2004.

[Microsoft, 2004] Microsoft (2004). Ipc in win32 environment. `http://msdn.microsoft.com/library/default.asp?url=/library/en-us/ipc/ba%se/interprocess_communications.asp` (Dernière visite : 6 mars 2004).

[Ramaekers, 1999a] Ramaekers, J. (1999a). Systèmes d'exploitation. Syllabus du cours de systèmes d'exploitation.

[Ramaekers, 1999b] Ramaekers, J. (1999b). Systèmes d'exploitation distribués. Syllabus du cours de systèmes d'exploitation - 2.

[Rusling, 1999] Rusling, D. A. (1999). The linux kernel. `http://www.tldp.org/LDP/tlk/tlk.html` Dernière visite : 28 mars 2004.

[Samuel J. Leffler, 1994] Samuel J. Leffler, R. S. F. e. a. (1994). An advanced 4.4 bsd interprocess communication tutorial.

[Stevens, 1993] Stevens, W. R. (1993). Advanced Programming In The Unix Environment. Addison-Wesley.

[Stover, 2002] Stover, G. M. (2002). Comparing some ipc methods on unix. `http://www.lisp-p.org/throughput/throughput.html` dernière viste le 20 janvier 2004.

[Willms, 1997] Willms, G. (1997). Le Langage C. Micro Application.

[Zabatta and Ying, 1998] Zabatta, F. and Ying, K. (1998). A thread performance comparison : Windows nt and solaris on a symmetric multiprocessor. In Proceedings of the 2nd Usenix Windows NT Symposium, Seattle, Washington.