POLYPUBLIE
Polytechnique Montréal

POLYTECHNIQUE MONTRÉAL
UNIVERSITÉ D'INGÉNIERIE

Titre: Synchronisation de traces dans un réseau distribué
Title:

Auteur: Éric Clément
Author:

Date: 2006

Type: Mémoire ou thèse / Dissertation or Thesis

Référence: Clément, É. (2006). Synchronisation de traces dans un réseau distribué [Master's thesis, École Polytechnique de Montréal]. PolyPublie. https://publications.polymtl.ca/7872/
Citation:

Document en libre accès dans PolyPublie
Open Access document in PolyPublie

URL de PolyPublie: https://publications.polymtl.ca/7872/
PolyPublie URL:

Directeurs de recherche: Michel Dagenais
Advisors:

Programme: Unspecified
Program:

Ce fichier a été téléchargé à partir de PolyPublie, le dépôt institutionnel de Polytechnique Montréal
This file has been downloaded from PolyPublie, the institutional repository of Polytechnique Montréal

https://publications.polymtl.ca

UNIVERSITÉ DE MONTRÉAL

SYNCHRONISATION DE TRACES DANS UN RÉSEAU DISTRIBUÉ

ERIC CLÉMENT
DÉPARTEMENT DE GÉNIE INFORMATIQUE
ÉCOLE POLYTECHNIQUE DE MONTRÉAL

MÉMOIRE PRÉSENTÉ EN VUE DE L'OBTENTION
DU DIPLÔME DE MAÎTRISE ÈS SCIENCES APPLIQUÉES
(GÉNIE INFORMATIQUE)
AOÛT 2006

© Eric Clément, 2006.

Library and Archives Canada

Published Heritage Branch

395 Wellington Street
Ottawa ON K1A 0N4
Canada

Bibliothèque et Archives Canada

Direction du Patrimoine de l'édition

395, rue Wellington
Ottawa ON K1A 0N4
Canada

Your file Votre référence
ISBN: 978-0-494-19290-0
Our file Notre référence
ISBN: 978-0-494-19290-0

NOTICE:
The author has granted a non-exclusive license allowing Library and Archives Canada to reproduce, publish, archive, preserve, conserve, communicate to the public by telecommunication or on the Internet, loan, distribute and sell theses worldwide, for commercial or non-commercial purposes, in microform, paper, electronic and/or any other formats.

The author retains copyright ownership and moral rights in this thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without the author's permission.

AVIS:
L'auteur a accordé une licence non exclusive permettant à la Bibliothèque et Archives Canada de reproduire, publier, archiver, sauvegarder, conserver, transmettre au public par télécommunication ou par l'Internet, prêter, distribuer et vendre des thèses partout dans le monde, à des fins commerciales ou autres, sur support microforme, papier, électronique et/ou autres formats.

L'auteur conserve la propriété du droit d'auteur et des droits moraux qui protège cette thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation.

In compliance with the Canadian Privacy Act some supporting forms may have been removed from this thesis.

While these forms may be included in the document page count, their removal does not represent any loss of content from the thesis.

Conformément à la loi canadienne sur la protection de la vie privée, quelques formulaires secondaires ont été enlevés de cette thèse.

Bien que ces formulaires aient inclus dans la pagination, il n'y aura aucun contenu manquant.

Canada

UNIVERSITÉ DE MONTRÉAL

ÉCOLE POLYTECHNIQUE DE MONTRÉAL

Ce mémoire intitulé:

SYNCHRONISATION DE TRACES DANS UN RÉSEAU DISTRIBUÉ

présenté par: CLÉMENT Eric

en vue de l'obtention du diplôme de: Maîtrise ès sciences appliquées

a été dûment accepté par le jury d'examen constitué de:

M. QUINTERO Alejandro, Doct., président

M. DAGENAIS Michel, Ph.D., membre et directeur de recherche

Mme. BOUCHENEB Hanifa, Doctorat, membre

à ma famille

REMERCIEMENTS

Tout d'abord, je voudrais remercier mon directeur de recherche, le professeur Michel Dagenais, présentement directeur du département de génie informatique de l'École Polytechnique de Montréal, pour sa disponibilité, son intérêt à me superviser tout au long de ma maîtrise et pour son aide financière. Je remercie également le professeur Robert Roy pour m'avoir permis d'utiliser une grappe de calcul et Louis-Alexandre Leclaire pour sa disponibilité et son aide quant à l'utilisation de cette grappe. Merci aussi à madame Hanifa Boucheneb et monsieur Alejandro Quintero pour leur participation au jury de ce mémoire.

Ensuite, j'aimerais remercier le professeur Yves Boudreault pour m'avoir choisi pour joindre son équipe de chargés de cours pour le cours ING1025, informatique, à trois reprises, ce qui fut un véritable plaisir. Cette expérience m'a donné goût à l'enseignement. Je remercie aussi le professeur Guillaume-Alexandre Bilodeau de m'avoir engagé comme chargé de cours pour le nouveau cours INF1005A, programmation procédurale. J'ai adoré notre collaboration et l'apprentissage de MATLAB, un outil dont je me sers maintenant régulièrement.

Je tiens aussi à remercier mon collègue Mathieu Desnoyers, qui m'a beaucoup aidé à démarrer mon travail, pour sa grande disponibilité et pour toutes nos discussions en lien plus ou moins direct avec mon projet. Merci aussi, pour l'avancement du projet LTTng. Le développement de cet outil a considérablement diminué ma tâche en diminuant le temps de calcul nécessaire pour la lecture des traces, mais aussi a augmenté de la précision au niveau des lectures de temps.

J'aimerais aussi remercier tous mes collègues à l'École Polytechnique avec qui j'ai passé de moments inoubliables.

Finalement, un merci tout spécial à Julie pour sa patience et sa compréhension. Merci aussi, à mes parents et mon frère pour leur support tant financier que moral.

RÉSUMÉ

L'outil LTTng, développé au laboratoire CASI, permet de générer de façon précise, flexible et robuste des traces au niveau noyau et usager (applications et librairies). Les lectures de temps sont effectuées à partir du compteur de cycle (TSC).

L'objectif de ce mémoire est de développer un outil de traçage d'événements pour les systèmes distribués en utilisant LTTng. En bref, l'idée est de générer des traces, localement, sur chaque ordinateur du réseau. Lorsque le traçage est terminé, les traces sont récupérées pour les corréler. Cette étape doit considérer le décalage de temps entre chaque noeud sujet à des déviations au cours du temps. Les mesures de décalage sont obtenues à partir des échanges de messages entre les ordinateurs lors du traçage. Cette technique est inspirée de l'algorithme de Cristian. Elle ne requière aucune structure de réseau ou matériel particulier. Par la suite, la dérive d'horloge entre deux noeuds peut être approximée à partir des mesures de décalage en fonction du temps à l'aide d'une régression linéaire des moindres carrés. Une référence est choisie afin de servir de base de temps pour la synchronisation des traces. Pour ce faire, une recherche de tous les plus petits chemins de chaque noeud vers chaque noeud est réalisée. La distance entre deux noeuds correspond à l'écart-type résiduel, E_r, obtenu lors de l'évaluation de la dérive d'horloge. La référence est donc le noeud ayant la plus grande couverture et précision avec l'ensemble des noeuds du réseau. Ensuite, les décalages de temps sont appliqués à chaque trace en utilisant le chemin le plus court reliant un noeud avec la référence.

Cette solution est contrainte à certaines limites, la précision du TSC, la variation du délai de transmission et du temps de réponse. Somme toute, nos résultats indiquent que notre outil offre une précision de quelques microsecondes, ce qui est suffisant pour les ordinateurs de nos jours.

ABSTRACT

LTTng is a tool, developed in the CASI laboratory, that generates in a precise, flexible and robust way, event traces in kernel space and user space (applications and library). The event time is recorded by reading the cycle counter (TSC).

The objective of this project is to develop an event tracing tool for distributed systems with LTTng. In short, the idea is to generate traces locally, on each computer of the network. When the tracing is finished, the traces are collected to correlate them. This stage considers the time offset between each node. The offset varies according to time due to the clock drift. Time offset estimates are obtained from messages exchanges during the tracing. This technique is inspired by the Cristian's algorithm. It does not require any specific network structure or hardware. Thereafter, the clock drift between two nodes can be approximated with offset measurements according to time using a linear least squares fit. A reference node is selected in order to be used as a time basis for the traces synchronization. With this intention, a search of shortest path between node pair is carried out. The distance between two nodes corresponds to the residual standard deviation, E_r, obtained during the evaluation of the clock drift. Thus, the node having the largest cover and precision with the remaining nodes in the network is selected as the reference. Then, the time offset is applied to each trace by using the shortest path connecting a node with the reference.

This solution is constrained by certain limits, the precision of the TSC, the variation of the transmission time and the response time. Altogether, our results indicate that our tool offers a precision of a few microseconds that is sufficient for the computers nowadays.

TABLE DES MATIÈRES

DÉDICACE iv

REMERCIEMENTS v

RÉSUMÉ vi

ABSTRACT vii

TABLE DES MATIÈRES viii

LISTE DES FIGURES xi

LISTE DES ABRÉVIATIONS xii

LISTE DES NOTATIONS ET DES SYMBOLES xv

LISTE DES TABLEAUX xvi

INTRODUCTION 1

CHAPITRE 1 REVUE DE LA LITTÉRATURE 6

1.1 Mécanismes de synchronisation 8

1.2 Algorithmes de synchronisation d'horloges 8

1.2.1 Algorithmes de synchronisation centralisé 9

1.2.1.1 Serveur passif de temps (Algorithme de Cristian) . 9

1.2.1.2 Serveur actif de temps 10

1.2.2 Algorithmes de synchronisation distribué 11

1.2.2.1 Moyenne globale 12

1.2.2.2 Moyenne locale 12

1.2.2.3 Études de cas 12

1.2.2.3.1 Distributed Time Service (DTS) 12
1.2.2.3.2 Network Time Protocol (NTP) 14
1.2.2.3.3 Algorithme de synchronisation d'horloge adaptif 14
1.2.2.4 Horloge logique de Lamport 15
1.3 Algorithmes de synchronisation de traces 17
1.4 Application parallèle avec MPI 20
1.4.1 Outils de déboguages et de profilages 22
1.4.2 Analyse de performance par traçage 25
1.4.2.1 MPE 25
1.4.2.2 Prism 26
1.4.2.3 MPICL 27
1.4.2.4 Pablo 30
1.4.2.5 Traçage d'événements pour les systèmes IBM SP . 31
1.4.2.6 Système EventSpace 32
1.4.2.7 Autres 33

CHAPITRE 2 CONCEPTS DE BASE SOUS LINUX 36
2.1 Temps sous Linux 36
2.2 Interruptions 41
2.2.1 PIC 41
2.2.2 APIC 43
2.3 Interruptions logicielles (softirqs) 46
2.4 Réception/transmission de paquets réseau 50
2.5 Appels système 60

CHAPITRE 3 ALGORITHMES ET STRUCTURES DE DONNÉES . . . 65
3.1 Concepts généraux 65
3.2 Appariement des paquets réseau 66

3.2.1 TCP vs IP 69
3.2.2 Instrumentation du noyau 72
3.2.3 Algorithme 74
3.3 Structures de données 76
3.4 Calcul de la dérive 78
3.5 Choix de la référence et des références intermédiaires 80

CHAPITRE 4 RÉSULTATS 83
4.1 Impact selon la fréquence des communications 84
4.2 Impact selon la durée du traçage 92
4.3 Impact selon la charge (temps de réponse) 96
4.4 Impact selon la distance 107

CONCLUSION 116

RÉFÉRENCES 121

LISTE DES FIGURES

Figure 1.1 Algorithme de Cristian 10
Figure 1.2 Algorithme Berkeley 11
Figure 1.3 Architecture du service DTS 13
Figure 1.4 Horloge logique de Lamport 16
Figure 1.5 Vecteur d'horloges logiques de Lamport 16
Figure 1.6 Technique de synchronisation de traces 18
Figure 1.7 Temps d'aller-retour entre deux machines i et j 19
Figure 1.8 Pipeline de quatre niveaux 22
Figure 1.9 Temps d'aller-retour entre deux machines i et j 28
Figure 2.1 Schéma du PIC 43
Figure 2.2 Schéma du APIC 45
Figure 2.3 Modèle ISO vs TCP/IP 50
Figure 2.4 Réception de paquets selon l'approche OAPI 54
Figure 2.5 Réception de paquets selon l'approche NAPI 56
Figure 2.6 Transmission de paquets 58
Figure 2.7 Espace usager et espace noyau 61
Figure 2.8 Traitement des appels système 62
Figure 3.1 Architecture du traçage d'un réseau distribué 67
Figure 3.2 Entêtes de protocoles réseau 70
Figure 3.3 Fenêtre de recherche pour trouver l'événement X_2 correspondant à l'événement X_1 75
Figure 4.1 Architecture du scénario de test 85
Figure 4.2 Marge d'erreur pour chaque aller-retour selon un intervalle de confiance de 90 et 100 % 86
Figure 4.3 Comparaison de l'approximation de la dérive selon un intervalle de confiance de 90 et 100 % 87

Figure 4.4 Mise en évidence du rôle de l'intervalle de confiance 88
Figure 4.5 Comparaison de l'écart-type résiduel pour différentes durées de traçage 92
Figure 4.6 Temps de réponse au repos par rapport à celui sous une charge au processeur (test 4 : *while(1)*) 103
Figure 4.7 Temps de réponse au repos par rapport à celui sous une charge d'appels système (test 5 : *syscall*) et à une charge de lectures et écritures sur disque (test 6 : *r/w*) 104
Figure 4.8 Temps de réponse pour les tests 1 (*bzip2*), 2 (*bzip2 + make*) et 3 (*bzip2 + make + ram*) 105
Figure 4.9 Architecture réseau de la grappe de calcul 108
Figure 4.10 Chemins directs et indirects entre le noeud$_2$ et les autres noeuds 108
Figure 4.11 Progression de la différence de l'approximation de la dérive entre un lien direct et un lien indirect en fonction du nombre de sauts 113
Figure 4.12 Comparaison des écarts-types résiduels des catégories de tests 115

LISTE DES ABRÉVIATIONS

ACPI	Advanced Configuration and Power Interface
ACPI PMT	ACPI Power Management Timer
APIC	Advanced Programmable Interrupt Controller
ATM	Asynchronous Transfer Mode
DCE	Distributed Computing Environment
DTS	Distributed Time Service
GPS	Global Positioning System
HPET	High Precision Event Timer
IP	Internet Protocol
IRQ	Interrupt ReQuest
ISO	International Standards Organization
ISR	Interrupt Service Routines
LTTng	Linux Trace Toolkit Next Generation
MPE	Multi-Processing Environment
MPI	Message Passing Interface
NAPI	New API pour la réception de paquets réseau sous Linux
NFS	Network File System
NIC	Network Interface Card
NMI	Non-Maskable Interrupts
NTP	Network Time Protocol
OAPI	Old API pour la réception de paquets réseau sous Linux
OSF	Open Software Fondation
PAG	Program Activity Graph
PIC	Programmable Interrupt Controller
PIT	Programmable Interval Timer

RTC	Real Time Clock
SAN	Storage Area Network
SCI	Scalable Coherent Interface
SCSI	Small Computer System Interface
SQL	Structured Query Language
TCP	Transmission Control Protocol
TSC	Time Stamp Counter
UDP	User Datagram Protocol
UTC	Universal Time Coordinated

LISTE DES NOTATIONS ET DES SYMBOLES

p	Précision d'une horloge	7
δ	Intervalle de resynchronisation	7, 29
$C(t)$	Valeur d'une horloge au temps t	7
T_{min}	Temps minimum de transmission dans une direction	10
D_{ij}	Décalage entre les noeuds i et j	19, 20, 28, 32, 78
E_{ij}	Marge d'erreur d'un cycle aller-retour entre les noeud i et j	19, 79
T_{aller}	Temps de transmission (décalage compris)	19
T_{retour}	Temps de réception (décalage compris)	19
$c(k)$	Correction pour l'horloge k	27
X_{ij}	Dérive entre les noeuds i et j	28, 79
$E_{X_{ij}}$	Erreur de l'estimation de la dérive entre les noeuds i et j (écart-type)	29, 79
$E_{D_{ij}}$	Erreur de l'estimation du décalage entre les noeuds i et j (écart-type)	29
(G, L)	Couple contenant une heure globale (G) correspondant à une heure locale (L)	31
R	Ratio entre la fréquence d'horloge globale et locale	31
C	Estimation du temps de transmission et de réponse pour un paquet réseau	32, 78
D^0_{ij}	Estimation du décalage initial entre les noeuds i et j	78, 79
E_r	Écart-type résiduel de l'estimation de la dérive	79
$E_{D^0_{ij}}$	Erreur de l'estimation du décalage initial entre les noeuds i et j (écart-type)	79

LISTE DES TABLEAUX

Tableau 2.1 Description des *softirqs* 47

Tableau 2.2 Comparaison des méthodes pour différer une tâche 49

Tableau 2.3 Comparaison entre la réception et la transmision 59

Tableau 3.1 Informations recueillies pour chaque événement 73

Tableau 3.2 Comparaison des algorithmes de recherche de tous les plus petits chemins 80

Tableau 4.1 Détermination de l'intervalle de confiance selon un niveau de confiance 89

Tableau 4.2 Marges d'erreur pour chaque aller-retour des tests d'une durée de dix minutes avec différentes fréquences de communications 89

Tableau 4.3 Approximation de la dérive selon les tests d'une durée de dix minutes avec différentes fréquences de communications . . . 90

Tableau 4.4 Résumé des spécifications de diverses architectures réseau . 91

Tableau 4.5 Marges d'erreur pour chaque aller-retour des tests d'une durée variable avec un échange de messages toutes les secondes 93

Tableau 4.6 Approximation de la dérive selon les tests d'une durée variable avec un échange de messages toutes les secondes . . . 94

Tableau 4.7 Description des tests utilisés afin de faire varier la charge d'un système 96

Tableau 4.8 Utilisation des ressources pour les tests 1 à 6 à titre d'indicatif 96

Tableau 4.9 Marges d'erreur pour chaque aller-retour des tests de variation de charge, d'une durée de 30 minutes avec un échange de messages toutes les secondes 98

Tableau 4.10 Marges d'erreur pour chaque aller-retour des tests de variation de charge (asymétrique), d'une durée de 30 minutes avec un échange de messages toutes les secondes 99
Tableau 4.11 Approximation de la dérive selon les tests de variation de charge, d'une durée de 30 minutes avec un échange de messages toutes les secondes 100
Tableau 4.12 Approximation de la dérive selon les tests de variation de charge (asymétrique), d'une durée de 30 minutes avec un échange de messages toutes les secondes 101
Tableau 4.13 Spécifications de la grappe de calcul 107
Tableau 4.14 Approximation de la dérive selon les tests de variation de distance, d'une durée de 30 minutes avec un échange de messages toutes les secondes 110
Tableau 4.15 Approximation de la dérive pour les liens directs faisant partie du chemin entre le $noeud_2$ et le $noeud_9$, traçage d'une durée de 30 minutes avec un échange de messages toutes les secondes 111
Tableau 4.16 Différence des paramètres de l'approximation de la dérive d'horloge d'un lien direct par rapport à un lien indirect . . . 114

INTRODUCTION

La croissance fulgurante de la puissance de calcul des ordinateurs de nos jours rend possible la résolution de problèmes de plus en plus complexes. Cette complexité s'est accentuée par l'avènement des grappes de calcul centralisées et distribuées. Des outils de mesure de performance sont donc devenus indispensables afin de valider les applications mais aussi d'en optimiser les performances. Le profilage, l'échantillonnage et le traçage d'événements sont les moyens habituels d'analyse de performance. Le profilage permet d'obtenir la distribution du temps de l'exécution des routines d'une application. Cette information est extrêmement utile pour localiser les goulots d'étranglement (*bottlenecks*). Par contre, il ne reflète pas les variations de performance en fonction du temps. D'un autre côté, l'échantillonnage périodique de l'état du système fournit des détails en fonction du temps mais, les échantillons ne sont pas corrélés avec des états particuliers de l'exécution du programme. Ainsi, l'échantillonnage ne permet pas d'identifier les causes d'une exécution anormale. En revanche, le traçage d'événements est le moyen plus général et efficace pour capturer et analyser le comportement d'une application en fonction du temps.

Ainsi, la vue de l'exécution d'un programme est représentée par une séquence d'événements correspondant à une activité physique ou logique précise (par exemple, l'entrée d'une fonction). Cette séquence d'événements est sauvegardée dans un fichier nommé trace ou trace d'événements. Typiquement, l'information associée à un événement est un identificateur de l'événement, l'heure où l'événement a eu lieu et de l'information par rapport à l'événement représentant l'état du système. Le traçage d'événements permet donc d'obtenir les informations de profilage avec en plus, la description du comportement de l'application en fonction du temps.

Linux Trace Toolkit Next Generation[1] (Desnoyers, Dagenais, 2006) (LTTng) est le traceur d'événements utilisé dans le cadre de ce mémoire. À ce jour, LTTng permet de tracer des événements dans l'espace noyau et dans l'espace usager (traçage de librairies et d'applications) sous Linux. LTTng est une version améliorée et complètement remodelée (principalement par Mathieu Desnoyers) de LTT (Yaghmour, Dagenais, 2000), la version originale réalisée par Karim Yaghmour. Ces deux projets ont été réalisés dans le laboratoire CASI (Conception et Analyse des Systèmes Informatique) sous la direction du professeur Michel Dagenais à l'École Polytechnique de Montréal. Actuellement, LTTng permet de tracer des événements de façon précise, en utilisant le TSC (Time Stamp Counter) pour les mesures de temps, tout en ayant un faible impact sur l'exécution normale du système et en offrant un haut niveau de réentrance (par exemple, il supporte les NMI).

L'objectif de ce mémoire est de développer un outil de traçage d'événements pour les systèmes distribués en utilisant LTTng. Un système distribué regroupe habituellement plusieurs architectures d'ordinateurs et environnements différents et les ressources peuvent être largement dispersées géographiquement. De façon générale, tous les ordinateurs du réseau distribué génèrent des traces (localement avec l'outil LTTng) et une fois le traçage terminé, les traces sont récupérées afin de synchroniser les événements dans le temps, permettant ainsi l'affichage de l'enchaînement des événements pouvant mener au diagnostique d'un comportement anormal. Afin de bien corréler les traces des différents noeuds du réseau, il faut s'assurer de tenir compte des décalages de temps entre chaque noeud. En effet, à un instant donné, l'horloge de chaque noeud risque de maintenir une heure différente. Un mécanisme de synchronisation des horloges est donc indispensable surtout que les horloges sont sujettes à des déviations au cours du temps. La dérive d'horloge est due aux différences (imperfections) des oscillateurs contrôlant la fréquence d'horloge (sans

[1] `http://ltt.polymtl.ca`

oublier que plusieurs cadences d'horloge différentes risquent de cohabiter dans un réseau distribué). Ainsi, une synchronisation d'horloge périodique est nécessaire afin de s'assurer que chaque horloge maintienne une heure bornée avec une certaine marge d'erreur. Elle peut être réalisée à l'aide de matériel (par exemple, un module GPS) ou d'un algorithme de synchronisation. Une autre solution est d'utiliser des événements corrélés entre les traces pour pouvoir déduire les décalages de temps entre les traces. Par conséquent, les communications réseau entre les noeuds peuvent servir de points de synchronisation entre les traces. Les échanges de messages réseau étant généralement symétriques, il est possible de calculer le décalage dans le temps, à un instant donné, entre deux noeuds en effectuant la moyenne du temps que le paquet réseau a pris pour se rendre à la destination avec le temps pris pour revenir à la source.

L'utilisation des échanges de messages lors du traçage a été choisie au détriment d'une synchronisation d'horloge afin de corréler les traces. Ce choix offre l'avantage de ne pas requérir de matériel supplémentaire et d'être moins intrusif dans le comportement du système qu'un algorithme de synchronisation d'horloge peut l'être. De plus, cette technique s'applique bien au réseau distribué puisqu'elle n'exige pas une structure réseau particulière.

En résumé, la méthodologie choisie pour synchroniser les traces d'un réseau distribué est d'effectuer une étude a posteriori des traces recueillies par chaque noeud du réseau à l'aide de l'outil LTTng. Sachant que la dérive d'horloge est linéaire en fonction du temps et que les communications réseau sont symétriques, la dérive d'horloge peut être approximée par une régression linéaire des moindres carrés à partir de plusieurs mesures de décalage en fonction du temps. Les mesures de décalage sont obtenues à l'aide des échanges de messages entre les noeuds lors du traçage. Des ajouts devront être faits à LTTng pour générer les événements d'envoi et de réception de messages réseau, et pour effectuer la corrélation des traces. Afin

de valider les résultats expérimentalement, plusieurs scénarios de test sont conçus. Ces tests vont permettre de vérifier l'hypothèse de la linéarité de la dérive d'horloge et de la symétrie des communications réseau, et aussi, de définir les limites d'une telle infrastructure.

Le problème de la synchronisation d'horloge n'est pas un problème nouveau. L'état de l'art est présenté au chapitre 1. La présentation des mécanismes et algorithmes de synchronisation est importante afin de choisir judicieusement la technique à utiliser afin de corréler les traces. De plus, plusieurs algorithmes de synchronisation de traces sont exposés dont certains sont basés sur l'utilisation de la librairie MPI.

Le chapitre 2 présente quant à lui les concepts de base sous Linux puisque l'outil LTTng est basé sur ce système d'exploitation. La gestion du temps ainsi que les intervenants influençant l'imprécision des lectures de temps sont présentés. Ainsi, les composantes permettant de suivre l'écoulement du temps sont décrites. Une explication des interruptions matérielles et logicielles et des appels système est présente puisqu'ils sont des facteurs pouvant faire varier le temps de réponse du système à une demande de lecture de temps, mais aussi au traitement des paquets réseau. Enfin, les mécanismes de réception et transmission de paquets réseau sous Linux sont décrits.

Dans le chapitre 3, l'implantation du système de traçage est expliquée. Ainsi, les concepts généraux sont décrits, suivis de la présentation du mécanisme d'appariement des paquets réseau. Il s'agit du traitement nécessaire de manière à pouvoir relier un événement de transmission d'un paquet réseau avec l'événement de réception équivalent, par le noeud destinataire. Ces appariements de paquets réseau constituent les points de synchronisation menant à l'évaluation du décalage entre deux noeuds à un instant donné. Par la suite, les structures de données et la façon d'approximer la dérive d'horloge entre les noeuds sont présentées. Enfin, le choix de

la référence de temps est expliqué. Son rôle est de servir de base de temps commune afin de synchroniser les événements des traces.

Ensuite, le chapitre 4 tente de déterminer les limites d'une telle implantation. Pour ce faire, une série de scénarios de test est élaborée en vue d'évaluer l'impact de la fréquence des communications, l'impact de la durée, l'impact de la charge et l'impact de la distance sur le traçage. Ainsi, il sera possible de statuer si un fort ou faible débit de communications dégrade la précision de l'approximation de la dérive. L'étude de l'impact de la durée du traçage indiquera le degré de linéarité de la dérive d'horloge dans le temps. L'analyse de l'impact de la charge démontrera la sensibilité de l'approximation de la dérive d'horloge à la variation du temps de réponse. Enfin, l'étude de l'impact selon la distance, permettra de connaître le niveau de dégradation des mesures lorsque l'approximation de la dérive d'horloge entre deux noeuds passe par plusieurs noeuds intermédiaires (plusieurs sauts).

CHAPITRE 1

REVUE DE LA LITTÉRATURE

Ce chapitre présente l'état de l'art en rapport avec la synchronisation de traces dans un réseau distribué. Cette synchronisation peut être obtenue de diverses façons. La première solution est de synchroniser les horloges de chaque noeud du réseau. De cette façon, la synchronisation des traces devient triviale puisqu'il suffit d'assembler les traces ensemble sans ajustement de temps. La première partie traite des mécanismes de synchronisation d'horloge, suivie d'une présentation des algorithmes de synchronisation d'horloge pour des systèmes centralisés et distribués dont DTS et NTP. Par la suite, plusieurs algorithmes de synchronisation de traces sont présentés. Ces algorithmes effectuent une synchronisation a posteriori du temps grâce aux échanges de messages à travers le réseau. L'avantage de ces algorithmes est qu'ils ne sont pas intrusifs dans le système contrairement aux algorithmes de synchronisation d'horloge, mais ils demandent toutefois une charge de calcul plus importante afin de réaliser la synchronisation des traces. Pour terminer, ce chapitre couvre les outils disponibles afin d'analyser une application parallèle utilisant MPI. Parmi ces outils, certains permettent le déboguage, le profilage et le traçage d'une application MPI.

Avant de débuter, il est important de donner quelques définitions. Tout d'abord, l'horloge d'un ordinateur est constituée de trois composantes : un oscillateur fixé à une fréquence (cristal au quartz), un registre compteur et un registre constant. Le registre compteur est mis à jour (décrémenté) à chaque oscillation de l'horloge et lorsque celui-ci atteint la valeur zéro, une interruption est lancée (nommé *tick* ou coup d'horloge) et ce registre est mis à jour avec la valeur du registre

constant. La valeur du registre constant est définie pour générer un certain nombre d'interruptions par seconde (sur Linux, ce registre est fixé afin de produire mille interruptions par seconde par défaut). Afin de synchroniser plusieurs ordinateurs avec l'heure réelle, deux autres valeurs sont nécessaires : une date de début qui est le 1er janvier 1970 à minuit pour les systèmes UNIX et le temps depuis le démarrage qui est mis à jour à chaque coup d'horloge.

L'horloge de chaque ordinateur peut dériver par rapport aux autres en raison de la variabilité des oscillateurs. Pour les oscillateurs basés sur un cristal de quartz, la dérive est d'environ une seconde toutes les millions de secondes, soit environ une seconde tous les 11,6 jours. Une horloge est définie non fautive si sa dérive avec l'heure réelle est bornée pour n'importe quel intervalle de temps (Gusella, Zatti, 1989). Soit $C(t)$ le temps d'une horloge lorsque l'heure réelle est t et soit p le taux de dérive maximal permis, alors l'horloge est non fautive si elle respecte cette inégalité :

$$1 - p \leq \frac{dc}{dt} \leq 1 + p$$

Ainsi, après un temps Δt où deux horloges non fautives ont été synchronisées, la dérive maximale entre les deux horloges sera de $2p\Delta t$. Pour s'assurer que la dérive de deux horloges ne dépasse pas δ, il faut synchroniser les horloges périodiquement à des intervalles de temps $\leq \delta/2p$. Par exemple, avec un cristal au quartz, p vaut 10^{-6}. Alors, pour avoir un $\delta \leq 1\mu s$, la période de synchronisation doit être d'au plus 0.5 seconde.

Une synchronisation mutuelle des horloges est souvent nécessaire afin de conserver une vue cohérente d'une application distribuée. Il ne s'agit pas d'un problème nouveau et diverses solutions sont présentées.

1.1 Mécanismes de synchronisation

Il existe deux types de synchronisation : synchronisation interne et externe. La première consiste à utiliser une horloge du système afin de synchroniser toutes les autres disponibles. Tandis que la deuxième utilise une source externe d'heure, nommée heure réelle. Le standard international UTC (Universal Time Coordinated) est le standard le plus utilisé. Cette heure est disponible de plusieurs sources via l'Internet, la radio, le téléphone et le GPS (Global Positioning System). Ces sources conservent l'heure grâce à des horloges atomiques au césium ayant une dérive d'une seconde tous les 1,4 millions d'années (Breakiron, 2006). La précision est aussi influencée par le médium de communication utilisé (Webb, 2006). Une source sur Internet sera grandement influencée par la bande passante du réseau. Les autres sources sont généralement accessibles via le port série d'un ordinateur et donc sujettes à la dérive de ce médium. Le dispositif GPS est le plus précis permettant d'obtenir une précision de l'ordre de 50ns (Ubik, Smotlacha, 2003). Le problème est qu'il peut être trop coûteux d'installer un tel module pour synchroniser l'ensemble d'un réseau (un module par ordinateur). Pour y remédier, une panoplie d'algorithmes de synchronisation existe afin de synchroniser tous les noeuds d'un réseau, (Anceaume, Puaut, 1997) présentent une taxonomie.

1.2 Algorithmes de synchronisation d'horloges

La complexité d'un système centralisé étant généralement moindre que celle d'un système distribué, il est plus facile de synchroniser un tel système. Un système distribué contient habituellement plusieurs architectures d'ordinateur et environnements différents, et les ressources peuvent être largement dispersées géographiquement, contrairement au système centralisé. De plus, les systèmes peuvent être

synchrones ou asynchrones. Le premier étant plus faiblement utilisé, il possède toutefois l'avantage d'être caractérisé par un temps de transmission minimal et maximal connus, le temps de communication maximal étant inconnu pour les systèmes asynchrones.

1.2.1 Algorithmes de synchronisation centralisé

Pour les systèmes centralisés, un serveur est généralement dédié pour la synchronisation d'horloge. Ce serveur peut être en mode passif ou actif, c'est-à-dire en mode écoute ou en mode interrogateur. Ainsi, dans le mode passif, ce sont les noeuds qui ont la responsabilité de se synchroniser avec le serveur en lui envoyant périodiquement des requêtes d'heure, tandis que dans le mode actif, la responsabilité est au serveur.

1.2.1.1 Serveur passif de temps (Algorithme de Cristian)

Il s'agit d'un algorithme simple et facile d'implantation. Son fonctionnement se base sur deux principes : chaque horloge est raisonnablement précise dans un court laps de temps et les communications sont symétriques. Ainsi, chaque noeud du réseau (client) se synchronise avec le serveur de temps à l'aide de requêtes périodiques. Le serveur de temps doit être connecté à une source externe d'heure UTC afin de fournir un temps précis, puisque tous les clients en dépendent. La figure 1.1 illustre le mécanisme de cet algorithme (Cristian, 1989).

Pour commencer, le client transmet une requête au serveur en mémorisant son heure actuelle (T_0). Lorsque le serveur reçoit la requête, il y répond en transmettant l'heure UTC actuelle (T_{UTC}). Enfin, le client note l'heure de la réception de cette réponse (T_1) et ajuste son heure en additionnant le temps de transmission avec

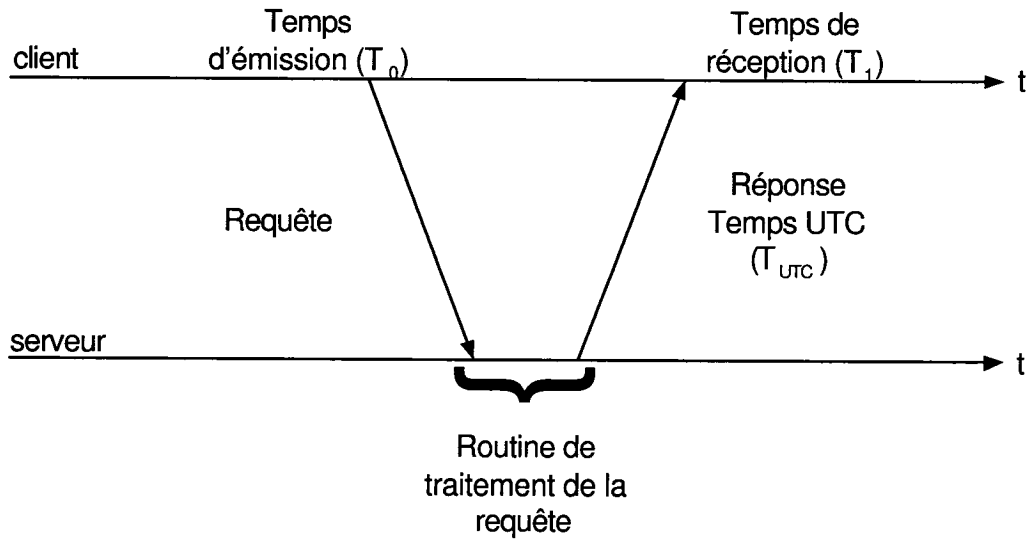


Figure 1.1 Algorithme de Cristian

l'heure UTC selon cette formule : $T = T_{UTC} + \frac{T_{aller-retour}}{2}$ où $T_{aller-retour} = (T_1 - T_0)$. Soit T_{min} , le temps minimum de transmission dans une direction, alors le client recevra la réponse du serveur dans cet intervalle de temps $[t + T_{min}, t + T_{aller-retour} - T_{min}]$. Donc, la précision est de $\pm(\frac{T_{aller-retour}}{2} - T_{min})$.

1.2.1.2 Serveur actif de temps

Approche générale Dans cette approche le serveur diffuse périodiquement son heure à tous les noeuds. Ainsi, chaque noeud ajuste son heure à chaque réception considérant que le temps de propagation du message est connu.

Algorithme Berkeley Cet algorithme fut proposé par (Gusella, Zatti, 1989) pour un réseau d'ordinateurs utilisant le système d'exploitation Berkeley UNIX. Cet algorithme ne nécessite pas une source externe d'heure. Le serveur diffuse

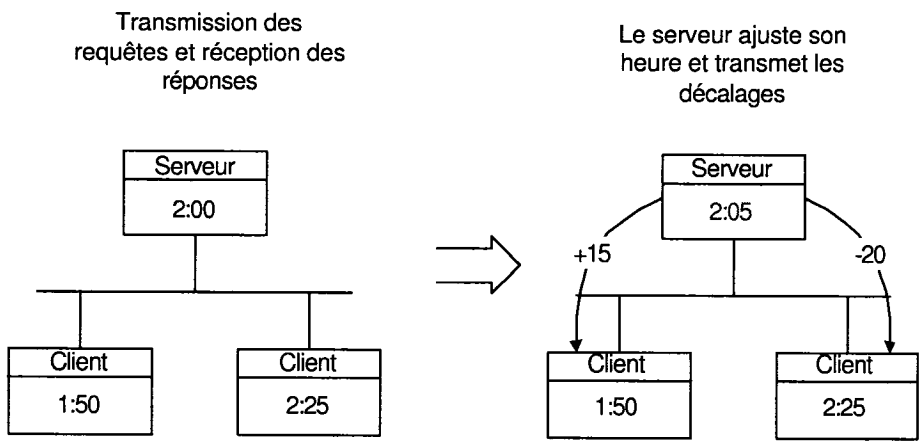


Figure 1.2 Algorithme Berkeley

périodiquement une requête d'heure aux noeuds. Les noeuds répondent à cette requête en envoyant leur heure courante. Le serveur calcule la moyenne des temps reçus en éliminant les temps trop éloignés selon une limite prédéfinie. Ensuite, le serveur ajuste son heure avec la moyenne calculée et il transmet à chaque noeud leur décalage respectif par rapport à la moyenne, avec lequel ils ajusteront leur heure. La figure 1.2 présente cette description.

1.2.2 Algorithmes de synchronisation distribué

La complexité d'un réseau distribué rend la synchronisation d'horloge plus laborieuse. Plusieurs techniques sont utilisées. Outre la technique de moyenne globale et local, trois études de cas sont présentées dont DTS, NTP et un algorithme de synchronisation d'horloge adaptif. Enfin, l'horloge logique de Lamport est décrite. Il s'agit d'une technique particulière qui permet d'ordonner des événements sans nécessité de synchronisation d'horloge.

1.2.2.1 Moyenne globale

Dans cette approche, tous les noeuds diffusent périodiquement leur heure courante. Après avoir transmis son heure, le noeud attend un certain laps de temps où il collectera les heures transmises par les autres noeuds. Une fois le temps d'attente terminé, chaque noeud ajuste son heure en fonction de la moyenne des temps reçus. Deux façons différentes peuvent être utilisées pour éliminer les heures fautives. La moyenne peut être prise avec seulement les heures ne dépassant pas une certaine limite prédéfinie ou en éliminant les n valeurs les plus éloignées. Le grand désavantage de cette méthode réside dans la génération importante de communication.

1.2.2.2 Moyenne locale

Cette approche peut être utilisée lorsque l'architecture du système distribué est en anneau, en maille ou autre. Chaque noeud échange son heure avec ses voisins et ajuste son heure en faisant la moyenne de son heure avec ses voisins. Cette approche diminue énormément le nombre requis de communications mais, le temps de convergence est plus grand. (Olson, Shin, 1994) propose une approche de ce genre en utilisant un anneau virtuel.

1.2.2.3 Études de cas

Dans cette partie, deux protocoles couramment utilisés seront étudiés, suivi d'un algorithme adaptif de synchronisation.

1.2.2.3.1 Distributed Time Service (DTS) DTS (IBM, 2001) est un service faisant partie du DCE (Distributed Computing Environment). DCE (The

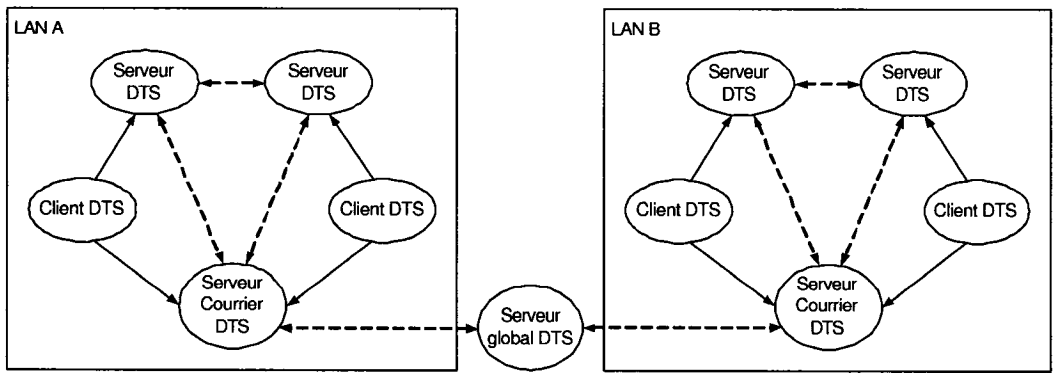


Figure 1.3 Architecture du service DTS

Open Group, 2005) est un produit de l'OSF (Open Software Fondation) et il est un standard de l'industrie pour les technologies distribuées. Il s'agit d'une plate-forme conçue pour supporter les applications distribuées indépendamment des équipements et logiciels. Il joue un rôle majeur au niveau de la sécurité, de l'Internet et des objets distribués.

Chaque noeud du DCE est configuré comme client DTS ou serveur DTS (voir la figure 1.3). Le client ajuste son heure locale en lançant des requêtes aux serveurs. Chaque client connaît ses limites de précision en fonction du matériel dont il dispose. Ainsi, le client connaît l'intervalle de temps maximal qu'il peut attendre sans dépasser le seuil d'imprécision. Une fois cet intervalle écoulé, le client doit exécuter une routine de synchronisation. Elle consiste en une série de requêtes aux serveurs pour obtenir leur heure. Elle ajuste l'heure locale du client en calculant l'heure probable courante et son imprécision en fonction des réponses reçues. L'heure courante est obtenue en faisant la moyenne des temps reçus, tout en ignorant les temps trop éloignés.

Les serveurs se synchronisent entre eux après chaque demande de synchronisation des clients. Il peut y avoir un nombre variable de serveurs par réseau, trois étant

la valeur typique. Les serveurs peuvent être synchronisés par un serveur de temps global permettant ainsi de synchroniser plusieurs réseaux entre eux. Pour ce faire, chaque réseau doit posséder un serveur courrier. Ce serveur ajuste son heure locale avec le temps global, ce qui aura pour effet de propager ce temps aux autres serveurs puisqu'ils se synchronisent entre eux. Le serveur global est généralement connecté via une source UTC.

1.2.2.3.2 Network Time Protocol (NTP) NTP (Mills, 1991) et (Mills, 1995) est un standard de l'IETF (RFC 1305) largement utilisé sous Internet, où se retrouvent plusieurs serveurs disponibles gratuitement. Les serveurs secondaires se synchronisent avec les serveurs primaires qui sont directement connectés à une source UTC. NTP peut être utilisé pour synchroniser des ordinateurs sous trois modes : le mode multidiffusion (*multicast*), le mode client-serveur, et le mode symétrique. Le mode multidiffusion est conçu pour les réseaux locaux rapides. Le serveur diffuse périodiquement son temps courant à chaque client du réseau. Les clients ajustent leur heure en fonction du temps reçu et du délai de communication.

Dans le mode client-serveur, le client envoie des requêtes au démarrage, et périodiquement par la suite, au serveur pour pouvoir ajuster son temps. Ce mode s'inspire de l'algorithme de Cristian.

Le mode symétrique est utilisé lorsqu'un haut niveau de précision est demandé. Dans ce mode, les serveurs se synchronisent entre eux en duo. Ce mode est surtout utilisé par les serveurs primaires.

1.2.2.3.3 Algorithme de synchronisation d'horloge adaptif Cette solution est présentée dans (Liao, Martonosi, Clark, 1999) basée sur l'algorithme de Cristian (section 1.2.1.1) avec quelques modifications. En effet, il n'est pas néces-

saire d'utiliser une source externe d'heure et un serveur est dédié pour synchroniser les noeuds (serveur actif). La particularité est qu'il utilise le processeur et l'horloge des cartes réseau pour estampiller le temps lors des échanges de messages. Ceci permet de libérer l'utilisation du processeur tout en obtenant un temps très précis. Ainsi, à tour de rôle, le serveur synchronise les autres noeuds en leur transmettant le décalage évalué. Cet algorithme offre deux modes : adaptif et non adaptif. La différence entre les deux réside dans le fait que le mode adaptif ajuste la durée de la resynchronisation selon l'état du système. Ainsi, avec un système chargé (par exemple, avec beaucoup d'échanges de messages), la période de resynchronisation diminue. Cette technique leur a permis d'obtenir une précision de l'ordre de la microseconde pour un système distribué basé sur Myrinet[1].

1.2.2.4 Horloge logique de Lamport

Il s'agit d'un moyen d'ordonner des événements basé sur la causalité (Lamport, 1978), (Raynal, Singhal, 1996). La causalité est déterminée par l'ordre des événements d'un processus et par les communications entre les processus. Puisque cette horloge est virtuelle, elle ne permet pas de capter la vue globale d'une application distribuée mais seulement d'ordonner les événements. Son fonctionnement est relativement simple. À chaque processus, un temps monotone unique est assigné. À chaque réception de message, un ajustement du temps est effectué si nécessaire. Le schéma à gauche de la figure 1.4 présente le temps de chaque événement pour trois processus. Le schéma à droite présente la même vue d'exécution des trois processus mais avec l'utilisation de l'horloge logique de Lamport.

Cette technique permet d'ordonner les événements seulement de façon partielle. Un

[1]Myrinet est une technologie d'interconnection de réseau local haute-vitesse développée par Myricom (Myricom, 2006).

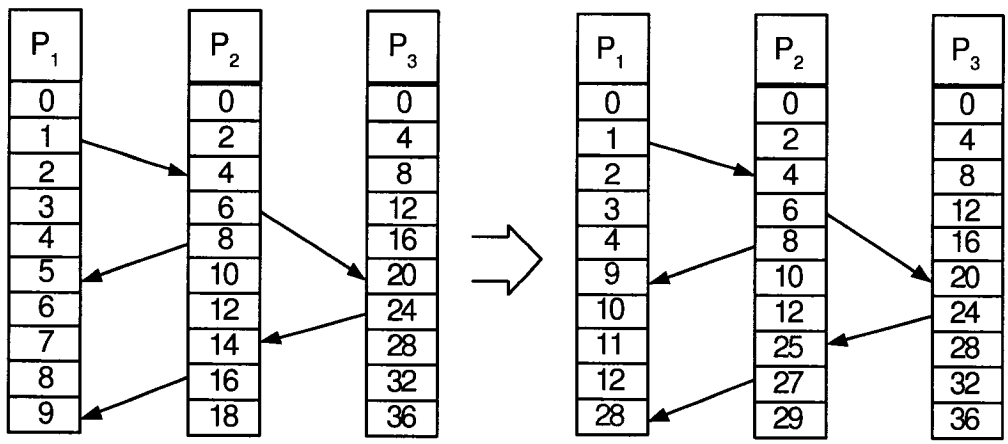


Figure 1.4 Horloge logique de Lamport

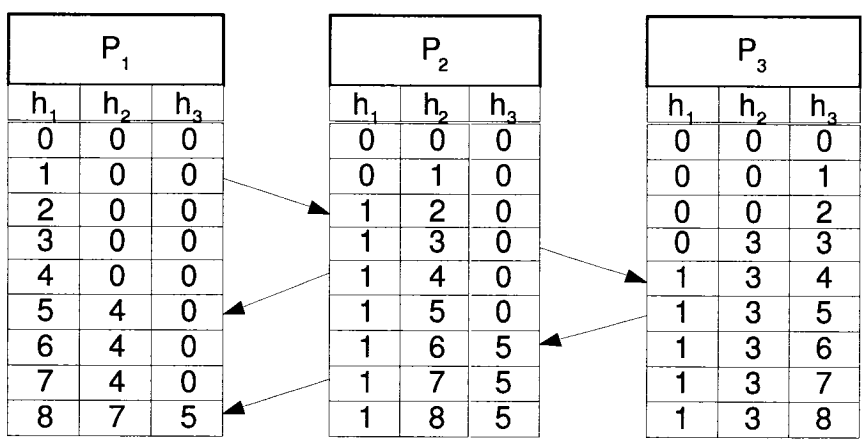


Figure 1.5 Vecteur d'horloges logiques de Lamport

vecteur d'horloges logiques[2], utilisant la même technique présentée ci-haut, permet d'avoir un ordonnancement total. Le principe est d'envoyer à chaque transmission le vecteur de temps courant du processus en question. À la réception, le noeud met à jour son vecteur, ce qui aura pour effet de propager à l'ensemble du réseau les échanges de messages (voir la figure 1.5). Le désavantage de ces techniques est qu'elles demandent une importante bande passante augmentant avec le nombre de noeuds en jeu.

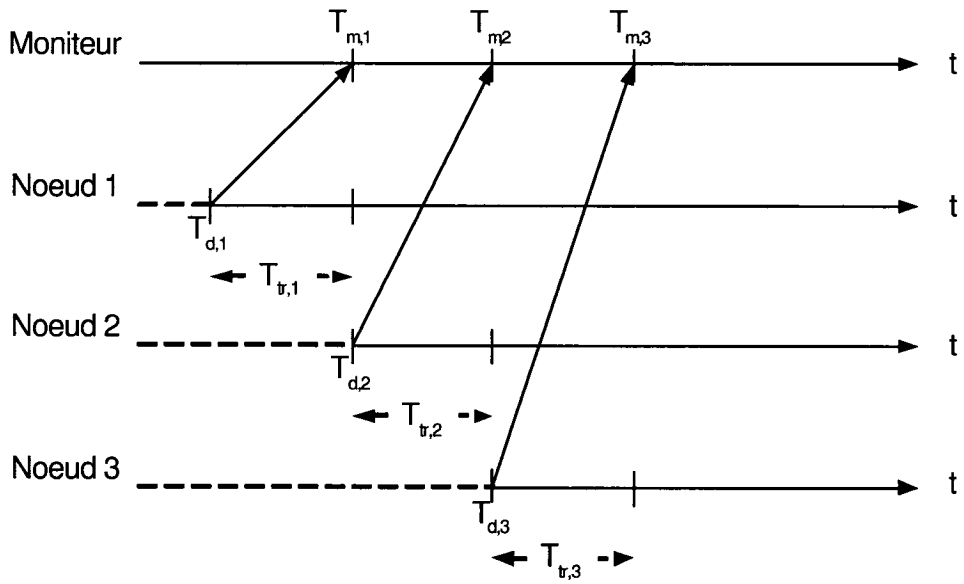
Jusqu'à présent, il a été question seulement de synchronisation d'horloge. La synchronisation d'horloge rend la synchronisation de traces triviale, puisque chaque trace se retrouve déjà synchronisée dans le temps. La section suivante introduit une autre façon de synchroniser des événements sous différents noeuds, sans nécessiter une synchronisation d'horloge, simplement en utilisant les données déjà présentes dans les traces.

1.3 Algorithmes de synchronisation de traces

Les algorithmes de synchronisation sont intrusifs dans le système, causant ainsi un impact dans les performances pouvant mener à une dégradation indésirable. Le traçage d'événements permet de diminuer cet impact en instrumentant seulement les régions pertinentes sans modifier l'heure locale des ordinateurs. Une étude a posteriori est toutefois requise afin de synchroniser les événements de chacune des traces. Ce sont les communications à travers le réseau qui servent de points de synchronisation. Plusieurs architectures de traçage peuvent être envisagées.

Normalement, une application parallèle est lancée à partir d'une machine nommé serveur. Cette machine distribue les tâches aux différents noeuds qu'elle contrôle.

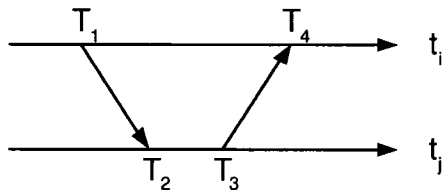
[2]F. Mattern en 1989



$$D_{m,i} = T_{m,i} - T_{d,i} - T_{tr,i}$$
$$D_{i,j} \approx D_{m,i} - D_{m,j} \; puisque \; T_{tr,i} \approx T_{tr,j}$$

Figure 1.6 Technique de synchronisation de traces

Dans ce type d'application, le serveur peut acquérir une responsabilité supplémentaire, celle de moniteur, sans demander trop d'effort. La tâche du moniteur est de synchroniser les événements tracés sur les différents noeuds. Une façon d'y parvenir, est de configurer les noeuds afin qu'ils envoient un message au moniteur avec leur heure courante, au lancement de l'application ou du traçage. Ainsi, le moniteur recueille tous ces temps en notant l'heure à laquelle il reçoit le message. De cette façon, il peut utiliser tous ces points de départ pour connaître les décalages de chaque noeud. Cette technique (Steigner, Wilke, 2001) considère que le temps de communication de chaque noeud vers le moniteur est semblable, mais aussi que les décalages restent constants tout au long du traçage. La figure 1.6 illustre cette méthode où $T_{d,i}$ est le temps du début du traçage pour le noeud i, $T_{m,i}$ est le temps de réception du message du noeud i par le moniteur, $T_{tr,i}$ est le temps de



$T_{aller} = T_1 - T_2$ et $T_{retour} = T_4 - T_3$

Figure 1.7 Temps d'aller-retour entre deux machines i et j

transmission du message pour le noeud i, $D_{i,j}$ est le décalage entre le noeud i et j et $D_{m,i}$ est le décalage entre le moniteur et le noeud i.

Une autre technique (Hofmann, Hilgers, 1998) consiste à utiliser un plus grand nombre de points sur toute la durée du traçage. Par exemple, la durée du traçage peut être divisée en un certain nombre de segments. Chaque segment contient un certain nombre de communications. L'idée est de choisir le plus petit temps d'aller et le plus petit temps de retour (en valeur absolue) entre chaque noeuds afin de respecter la causalité pour l'ensemble du traçage. Le décalage est calculé en faisant la moyenne du temps d'aller et du temps de retour (voir la section 1.2.1.1). Soit D_{ij}, le décalage entre les machines i et j, alors $D_{ij} = \frac{T_{aller}+T_{retour}}{2} = \frac{(T_1-T_2)+(T_4-T_3)}{2}$ avec comme marge d'erreur $E_{ij} = |\frac{T_{aller}-T_{retour}}{2}|$ (voir la figure 1.7). Une fois que tous les décalages et les marges d'erreur sont connus, il est nécessaire de trouver une référence pour servir de base de temps. Le choix de la référence repose sur deux facteurs : la précision et la couverture. Ainsi, la machine qui sera élue comme référence doit être celle qui communique avec le plus de noeuds du réseau, et ce avec la plus grande précision. Ainsi, un graphe de tous les chemins possibles est créé et il s'ensuit d'une recherche de tous les plus petits chemins reliant un noeud avec les autres. Les poids utilisés comme distance du graphe correspondent aux marges d'erreur calculées. La référence sera la machine ayant la sommation de tous ces chemins la plus petite. Cette méthode a l'avantage de bien fonctionner peut

importe l'architecture du réseau.

(Kortenkamp, Simmons, Milam, Fernández, 2002) propose une architecture permettant de déboguer des systèmes distribués. Les événements sont estampillés par l'horloge locale puis envoyés à un serveur qui les emmagasine grâce à une base de données SQL (MySQL[3]). C'est le serveur qui possède la responsabilité de bien assembler les événements des différents noeuds avec une base de temps commune. Pour ce faire, lorsque le serveur reçoit le premier événement d'un noeud, il démarre un processus qui envoie une requête à ce noeud afin d'évaluer le décalage du noeud par rapport à lui-même. Ce décalage sera donc ensuite appliqué à cet événement et à tous ceux qui suivront. Afin de considérer la dérive, le serveur redémarre ce processus périodiquement toutes les deux minutes. Le décalage D_{ij} est calculé selon la formule publiée dans le RFC 2030 (Mills, 1996) (voir la figure 1.7) :

$$D_{ij} = \frac{(T_2 - T_1) + (T_3 - T_4)}{2}. \tag{1.1}$$

La section 1.4.2 présente d'autres approches de traçage en relation avec la librairie MPI qui est présentée dans la section subséquente.

1.4 Application parallèle avec MPI

MPI[4] est un standard pour une librairie d'échange de messages. Il a été conçu par le forum MPI (MPI Forum[5]) qui est composé de plusieurs intervenants dans le monde du calcul parallèle dont des vendeurs d'ordinateurs, des concepteurs de librairies ainsi que des spécialistes d'applications parallèles. Plusieurs plates-formes existent

[3] `http://www.mysql.com`
[4] `http://www-unix.mcs.anl.gov/mpi/`
[5] `http://www.mpi-forum.org/`

aujourd'hui dont LAM/MPI[6] et MPICH[7] provenant du domaine public. Quand vient le temps de concevoir une application parallèle, le choix des algorithmes doit être basé sur les fonctionnalités fournies par l'architecture utilisée. Ces algorithmes sont soit optimisés pour les architectures à mémoire partagée ou pour les architectures d'échange de messages. Alors, pour un système distribué, l'approche par échange de messages est généralement privilégiée.

Le but d'une application parallèle est de diminuer le temps de calcul en effectuant plusieurs traitements simultanément sur différents processeurs. Pour y arriver, il est nécessaire de décomposer le problème en plusieurs morceaux pour profiter du parallélisme. Deux stratégies de décompositions existent : décomposition de données et décomposition de tâches. La décomposition de données est efficace pour les algorithmes récursifs. Il s'agit de la stratégie, diviser pour régner, employée entre autre dans les algorithmes de tri. Pour ce qui est de la décomposition de tâche, l'approche la plus commune est le pipeline (voir la figure 1.8). Ainsi, chaque tâche consomme et produit un travail. Pour les deux approches, la décomposition doit fournir une quantité de travail identique à chaque processeur pour obtenir des performances optimales. Par contre, il peut être très difficile de choisir la granularité du problème considérant le caractère dynamique de la majorité des applications (les données en entrée ne sont pas statiques) et le fait qu'un réseau distribué peut être hétérogène (plusieurs processeurs différents). Une solution est de diviser le problème en un nombre plus élevé que le nombre de processeurs disponibles. Il s'agit d'un bon compromis entre le coût des communications (étant le temps d'attente d'un processeur pour recevoir un message) et le temps d'inactivité d'un processeur. Cette solution permet ainsi de maximiser l'utilisation des processeurs rapides tout en diminuant le temps d'attente d'un processeur plus lent. Par exemple, avec un système de dix

[6] `http://www.lam-mpi.org/`
[7] `http://www-unix.mcs.anl.gov/mpi/mpich/index.htm`

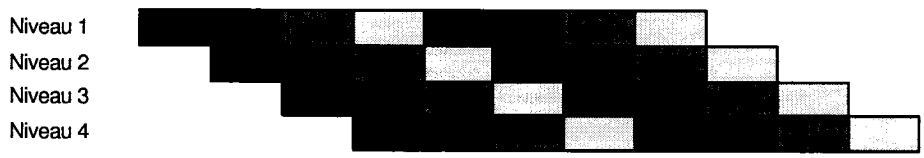


Figure 1.8 Pipeline de quatre niveaux

processeurs, le problème d'une multiplication de deux matrices 10x10 peut être divisé en dix tâches de multiplication, chaque tâche effectuant dix multiplications. Le temps de calcul est donc environ la durée de calcul de la tâche la plus lente. Par contre, si ce problème est divisé en 100 tâches, chaque processeur recevant une tâche lorsqu'il en finit une, risque de fournir de bien meilleures performances, maximisant ainsi l'utilisation des processeurs plus rapides.

1.4.1 Outils de déboguages et de profilages

Plus les problèmes sont complexes et plus il est difficile de trouver les sources d'erreur de programmation. Cette complexité est encore plus élevée pour les applications parallèles puisqu'elles peuvent engendrer des problèmes de synchronisations, dont la concurrence critique (*race condition*) et l'interblocage (*deadlock*). Une panoplie d'outils de déboguage et de profilage est disponible pour les applications traditionnelles et ceux-ci peuvent être utilisées pour les applications parallèles. Outres les méthodes d'affichage classique, en ajoutant aux endroits stratégiques une fonction telle *printf* pour le langage C, ou en générant plusieurs avertissements avec un compilateur, un débogueur peut être utilisé par exemple, *gdb*[8] et *ddd*[9]. Le problème est qu'un débogueur a besoin d'une application pour débuter le déverminage. Les programmes écrits avec MPI utilisent soit *mpirun* ou *mpiexec*

[8] `http://www.gnu.org/software/gdb/`
[9] `http://www.gnu.org/software/ddd/`

pour lancer une application parallèle. Heureusement, il existe une seconde façon d'utiliser ces débogueurs. Il est possible d'attacher *gdb* ou *ddd* à une application déjà en cours d'exécution. Ainsi, il est possible d'attacher un débogueur à chaque processus s'exécutant sur le réseau distribué. Ceci peut sembler compliqué, mais cette méthode s'avère relativement efficace avec un script et l'astuce de mettre en attente d'une entrée au clavier les processus suspects.

Plusieurs outils sont aussi à la disposition du programmeur pour profiler ces applications. Cette phase est très importante pour identifier le chemin critique d'une application afin de diriger les efforts d'optimisation sur les segments de codes ayant le plus d'impact sur les performances. La règle du 10-90 expose bien ce fait en stipulant qu'environ 10% du code d'un programme utilise 90% des ressources disponibles.

Il existe la commande UNIX *time* qui permet de déterminer la durée d'une application (de son lancement jusqu'à son arrêt) tout en spécifiant le temps passé dans l'espace usager et l'espace noyau. Cette commande exécutée dans une console C (*C shell*) permet d'obtenir encore plus d'information : le taux d'utilisation du processeur, l'usage de la mémoire partagée et non partagée, le nombre d'opérations d'entrée et de sortie, le nombre de défauts de page (*page faults*) et enfin, le nombre de permutations en mémoire (*swaps*).

Pour déterminer la durée d'un segment de code, les appels système suivants peuvent être utilisées : *time*, *gettimeofday*, etc. Le problème est qu'il faut identifier manuellement les segments critiques. Pour cette raison, il est souvent plus efficace d'utiliser un profileur. Il existe deux catégories de profileurs, actifs et passifs. Les passifs recueillent les informations sans modifier le code. Ainsi, ce genre de profileur donne par exemple une estimation du temps passé dans chaque fonction. Le problème est que ces profileurs n'affichent pas une vue d'ensemble de l'exécution d'un pro-

gramme. Ils ne permettent pas de savoir combien de fois une fonction en appelle une autre. Les profileurs actifs rajoutent automatiquement des lignes de code dans le programme pour recueillir des informations. Ces profileurs permettent d'avoir une meilleure vue de l'exécution d'un programme mais ils ont un effet plus important sur les mesures de temps prises. Deux profileurs bien connus sont *gprof*[10] et *gcov*[11]. *gprof* est un profileur à la fois actif et passif. Il permet d'obtenir à la fois des statistiques d'utilisation des fonctions mais aussi d'analyser ces statistiques avec un arbre d'appels. Il est donc possible de retracer le nombre d'appels d'une fonction vers une autre. Il est possible d'exécuter *gprof* pour une application parallèle. *gprof* génère une fichier de résultat. L'astuce pour une application parallèle est d'exécuter le programme dans le répertoire local */tmp* pour un système UNIX. Ainsi, chaque noeud créera un fichier de sortie unique évitant ainsi que chaque fichier soit écrasé en un seul dans le cas où il serait écrit dans un répertoire non local, partagé sur le serveur de fichier NFS.

L'outil *gcov* est une autre application très utilisée. Elle permet de tester tous les chemins d'une application. Il enregistre le nombre de fois qu'une ligne de code a été exécutée. Il est beaucoup utilisé pour vérifier si une série de tests permet d'exécuter au moins une fois chaque ligne de code (tests de couverture). Tout comme *gprof*, pour utiliser *gcov* il faut ajouter des options au compilateur pour rajouter du code automatiquement au programme afin de générer des statistiques. La compilation génère deux fichiers ayant les extensions .bb et .bbg contenant de l'information sur la structure du programme. Après l'exécution du programme, un troisième fichier est créé avec l'extension .da. Alors, pour utiliser *gcov*, il faut utiliser une astuce semblable à celle utilisée pour *gprof*. Il suffit de copier tous les fichiers mentionnés ci-haut avec le fichier exécutable dans le répertoire */tmp* de chaque noeud. Ensuite,

[10] `http://www.gnu.org/software/binutils/manual/gprof-2.9.1/gprof.html`
[11] `http://gcc.gnu.org/onlinedocs/gcc-3.0/gcc_8.html`

il est possible d'exécuter le programme pour observer les résultats. Il est important de noter que les résultats contenus dans les fichiers sont cumulatifs alors, selon l'utilisation faite, il peut être nécessaire d'effacer ou de recopier tous ces fichiers à chaque exécution.

1.4.2 Analyse de performance par traçage

Dans cette section, plusieurs outils de traçage d'applications MPI sont énumérés dans le but de décrire les mécanismes de synchronisation de traces utilisés dans chaque cas.

1.4.2.1 MPE

MPE[12] (Multi-Processing Environment) est un autre moyen pour profiler une application MPI. MPE fournit plusieurs outils pour analyser les performances grâce à une librairie de profilage, plusieurs utilitaires et plusieurs outils de visualisation graphique. MPE offre présentement trois librairies, une librairie de traçage (*tracing library*), une librairie de visualisation (*animation library*) et une librairie de journalisation (*loggin library*). La dernière, la plus pratique, permet de générer une collection d'événements avec un temps associé. Le principe de MPE est qu'il s'interpose entre l'appel de fonction du programme et la librairie MPI. Ainsi, à chaque appel de fonction de la librairie MPI, un traitement supplémentaire sera réalisé pour permettre de recueillir les informations. Il est aussi possible de personnaliser ses propres événements. Chaque événement est caractérisé par un identificateur unique et il est généré grâce à un appel de fonction. Avec cet appel, il est possible d'associer des informations à l'événement (un entier et une chaîne de caractères). Le

[12] `http://www-unix.mcs.anl.gov/mpi/www/www4/MPE.html`

temps associé à chaque événement a une résolution de l'ordre de la microseconde. Les événements sont recueillis par la machine virtuelle qui lance le programme MPI. Ainsi, les événements sont triés en ordre d'arrivée au serveur. Cet ordre est donc influencé par les délais de communication et peut donc être trompeur dans le cas d'événements très proches dans le temps.

1.4.2.2 Prism

Prism (Sistare, Dorenkamp, Nevin, Loh, 1999) est un débogueur de programmes multi-processus MPI incluant des outils d'analyse de performance et de visualisation. L'analyse de performance se base sur le traçage d'événements. Chaque processus MPI génère un fichier de trace. Ce volumineux fichier est présent dans l'espace mémoire du processus et il est utilisé comme une liste circulaire. Ce fichier est manipulé dès la phase d'initialisation du processus afin de le mettre en cache, s'assurant ainsi de diminuer l'impact de la première écriture. L'avantage de l'utilisation du fichier comme liste circulaire est de borner la taille des traces et de diminuer les perturbations causées par les accès disques dans le cas où tout le fichier est présent en mémoire. La synchronisation des événements des traces s'effectue a posteriori. Pour ce faire, un programme de mesure de dérive roule en parallèle avec celui qui est tracé. Il s'agit d'un programme MPI ayant un processus s'exécutant sur chaque noeud. Périodiquement, chaque processus calcule le décalage avec chaque noeud selon la méthode de Cristian (section 1.2.1.1). Ainsi, chaque processus recueille l'ensemble des décalages avec les noeuds à un instant donné. Ces informations sont sauvegardées dans un fichier par processus. De cette façon, ces fichiers permettent d'appliquer les corrections de temps nécessaires lors de l'assemblage des traces dans un seul fichier.

Plus précisément, la période choisie pour calculer les décalages est de trois mi-

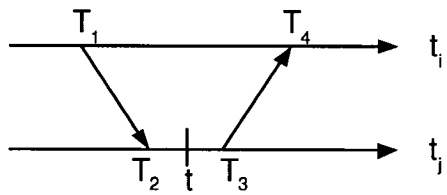
nutes (déterminée expérimentalement). Chaque calcul de décalage engendre plusieurs échanges de messages et donc divers temps de cycle aller-retour. Le plus petit temps trouvé de cycle aller-retour est considéré pour le calcul du décalage présent. Ainsi, on retrouve alors N^2 calculs de décalage, D_{ij}, périodiquement. Puisque les valeurs D_{ij} peuvent différer par rapport aux valeurs D_{ji} en raison des imprécisions des mesures, une correction par horloge, $c(k)$, est utilisée afin de minimiser l'erreur moyenne quadratique des mesures.

$$c(k) = \frac{1}{N} \sum_{i=1}^{N} \frac{(D_{ik} + D_{ki})}{2} \tag{1.2}$$

Une interpolation linéaire est utilisée pour déterminer les décalages entre deux mesures prises.

1.4.2.3 MPICL

MPICL (Worley, 2000) est une librairie portable d'instrumentation permettant de recueillir des informations sur les communications et des événements personnalisés. Elle utilise l'interface de profilage de MPI tout comme MPE. Elle fournit deux modes d'utilisation : profileur et traceur. Avec le mode profileur, il est possible de connaître le nombre d'occurrences, les statistiques et le temps écoulé lors des échanges de messages et des événements personnalisés par processeur. Le mode traceur permet de générer une trace d'événements par processeur. L'avantage de cette librairie par rapport à MPE est la diminution de l'impact du traçage puisque l'information des événements n'est pas propagée sur le réseau. Ces traces peuvent être visualisées à l'aide de l'outil *ParaGraph* (Heath, Finger, 2003). Cet outil n'a pas à se soucier de la synchronisation des événements puisque MPICL s'assure de synchroniser les horloges des processeurs et d'ajuster la dérive de celles-ci. La



$T_{aller-retour} = T_4 - T_1$

Figure 1.9 Temps d'aller-retour entre deux machines i et j

méthodologie de synchronisation (Dunigan, 1992) a été développée pour les architectures *hypercubes* mais, elle peut être généralisée pour d'autres types de systèmes. La synchronisation des horloges se base sur la méthode de Cristian (section 1.2.1.1) sans nécessiter l'usage d'une source d'heure externe (UTC). Le décalage entre deux noeuds, i et j, est calculé comme suit :

$$D_{ij} = t + \frac{T_{aller-retour}}{2} - T_4 \tag{1.3}$$

$$D_{ij} = t + \frac{T_4 - T_1}{2} - T_4 \tag{1.4}$$

où t est le temps enregistré entre T_2 et T_3 (voir la figure 1.9). La dérive correspond à la différence de fréquence et elle est définie par :

$$X_{ij} \equiv \frac{T_j(t+S) - T_j(t)}{T_i(t+S) - T_i(t)} \tag{1.5}$$

pour un intervalle de temps S. L'estimation de la dérive du noeud i par rapport à j peut se faire en prenant le ratio de deux temps sur une période S, où :

$$X_{ij} = \frac{T_j(t+S) - T_j(t)}{S} \tag{1.6}$$

L'erreur de l'estimation de la dérive $E_{X_{ij}}$ est :

$$E_{X_{ij}} < \frac{2E_{D_{ij}}}{S} \tag{1.7}$$

où $E_{D_{ij}}$ est l'erreur de l'estimation du décalage et elle est définie comme suit :

$$E_{D_{ij}} < \frac{T_{aller-retour}}{2} - T_{min} + X_{ij} \cdot T_{aller-retour} + p \tag{1.8}$$

où p est la précision de l'horloge (p vaut 1 μs pour une horloge au quartz). Connaissant le décalage et la dérive, il est possible d'ajuster l'heure du noeud i par rapport à j selon cette formule :

$$T_{i \to j}(t) = T_i(t) + D_{ij} + X_{ij} \cdot (t - T_4) \tag{1.9}$$

avec comme intervalle de resynchronisation δ :

$$\delta = \frac{T_{min} - E_{D_{ij}}}{E_{X_{ij}}}. \tag{1.10}$$

où T_{min} est le temps minimum de transmission dans une direction. De façon expérimentale, ils ont remarqué que les mesures de temps aller-retour sont constantes avec la présence de pics. Ces pointes sont causées par la perturbation de la routine

d'interruption de l'horloge système. Afin de diminuer l'impact de cette routine, ils prennent quatre mesures de temps aller-retour et ils conservent seulement le plus petit temps trouvé. Ceci est possible étant donné que, le temps aller-retour est plus petit que l'intervalle de temps entre deux occurrences d'une interruption causée par l'horloge système. L'implantation de cet algorithme dans MPICL consiste en deux fonctions. La première est appelée par chaque noeud pour synchroniser leur heure et estimer leur dérive par rapport au processeur 0 (ou noeud 0) au démarrage de l'application. La seconde est utilisée lorsque l'application veut lire l'heure d'un noeud. Les mesures sont prises en supposant que le système est en attente (aucune charge et aucun échange de message) et il n'y a pas de resynchronisation lors du déroulement de l'application.

1.4.2.4 Pablo

Pablo (Reed, Aydf, Madhyastha, Noe, Shields, Schwartz, 1992), (Noe, 1994) est un ensemble d'outils d'analyse de performance. Toutes les informations de performance sont recueillies par la librairie de traçage qui est basée, tout comme MPE, sur la librairie de profilage de MPI. Il y a une trace par processus. L'analyse de performance est surtout basée par processus mais les traces peuvent aussi être regroupées ensemble. Par défaut, les événements sont écrits dans un tampon temporaire qui est vidé périodiquement dans le fichier de trace. La librairie de traçage synchronise tous les noeuds participant au traitement de l'application parallèle de telle façon que chaque tampon temporaire sera vidé simultanément, tout en rendant disponible une heure globale aux noeuds.

1.4.2.5 Traçage d'événements pour les systèmes IBM SP

Une autre technique (Wu, Bolmarcich, Snir, Wootton, Parpia, Chan, Lusk, Gropp, 2000) permet de régénérer des traces d'événements MPI mais aussi des événements systèmes. L'astuce utilisée pour synchroniser les événements des différents noeuds est d'utiliser l'horloge interne du commutateur d'IBM SP (MHPCC, 2003). Ainsi, chaque événement est estampillé à partir de l'heure du commutateur (horloge globale) au détriment de l'heure locale du noeud en question. Par contre, l'accès à l'horloge globale est beaucoup plus coûteux que celle locale. La solution est d'accéder périodiquement à l'horloge globale. Chaque accès à cette horloge permet de sauvegarder l'heure globale correspondant à l'heure locale (G, L). Le principe est donc de générer une trace par noeud. Au début du traçage, un couple (G,L) est sauvegardé pour chaque noeud. Chaque événement est estampillé à l'aide de l'horloge locale et des couples (G,L) sont générés périodiquement. Ces couples contenus dans les traces permettent l'assemblage des celles-ci. Pour ce faire, (Wu, Bolmarcich, Snir, Wootton, Parpia, Chan, Lusk, Gropp, 2000) proposent d'évaluer le ratio R entre la fréquence d'horloge du commutateur et l'horloge locale à partir de cette formule :

$$R = \sqrt{\frac{\sum_{i=1}^{N} (\frac{G_i - G_{i-1}}{L_i - L_{i-j}})^2}{N}}. \tag{1.11}$$

Ainsi, un temps local t est ajusté par $t * R$ (temps global) et une durée d par $d * R$. Ils proposent aussi deux alternatives. La première est d'utiliser seulement le dernier couple (G, L) pour calculer le ratio R si la durée du traçage est relativement longue. La seconde est de calculer le ratio pour chaque couple (G,L), segmentant ainsi l'ajustement de l'horloge locale avec une valeur de ratio par segment.

1.4.2.6 Système EventSpace

Le système EventSpace (Bongo, Anshus, Bjørndalen, 2003) permet d'effectuer une étude de traces a posteriori d'applications parallèles en configurant les chemins de communication de la plateforme LAM/MPI. L'application parallèle est instrumentée par l'insertion de points de traçage dans les chemins de communication. Il existe trois types de chemin : les chemins de l'application, les chemins de la grappe et les chemins de l'application vers la grappe. Plus précisément, les chemins de l'application décrivent les éléments qui sont accédés par les processus. Les chemins de la grappe caractérisent la topologie et les noeuds du réseau. Enfin, les chemins de l'application vers la grappe représentent les processus et les éléments que gère un noeud. Tous ces chemins sont instrumentés permettant ainsi d'analyser toutes les communications réseau (messages réseau) et systèmes (accès à une variable partagée par exemple). Afin de synchroniser les événements des divers noeuds, ce système utilise les échanges de messages réseau afin de calculer les décalages. Un événement est collecté au début de la transmission et de la réception d'un message réseau. Le calcul du décalage est déduit à partir d'un cycle aller-retour d'échanges de messages, en effectuant la moyenne de plusieurs valeurs calculées de décalage où C correspond au temps de transmission et de réponse (voir la figure 1.7).

$$D_{ij} = T_2 - (T_1 + C) \tag{1.12}$$

avec,

$$C = \frac{(T_4 - T_1) - (T_3 - T_2)}{2} \tag{1.13}$$

Pour synchroniser plusieurs noeuds, un graphe est créé à partir de tous les chemins

(à la fois réseau et système) permettant d'identifier les chemins de communication réseau entre les noeuds. À partir de ce graphe, une recherche de l'arbre sous-tendu de poids minimal (*minimum spanning tree*) à partir du noeud de référence est effectuée. Par la suite, tous les décalages entre tous les noeuds sont calculés pour ensuite être appliqués afin d'ajuster leur heure par rapport à la référence.

La section suivante présente d'autres outils d'analyse de performance utilisant le traçage d'événements avec plus ou moins un lien direct avec la librairie MPI.

1.4.2.7 Autres

VGV (Vampir/GuideView) (Hoeflinger, Kuhn, Nagel, Petersen, Rajic, Shah, Vetter, Voss, Woo, 2001), (VGV, 2006) est un ensemble d'outils permettant l'analyse de programme parallèle écrit avec MPI et OpenMP[13] (tout comme Paraver (Caubet, Gimenez, Labarta, DeRose, Vetter, 2001)). Il s'agit d'une intégration des outils Vampir dédié pour MPI et GuideView, pour OpenMP. Vampir utilise la librairie de profilage de MPI pour générer une trace d'événements. Dans VGV, cette librairie a été modifiée afin de générer aussi les événements de OpenMP. Le traçage est donc similaire à MPE, générant un seul fichier de trace. Aujourd'hui, l'Intel Trace Collector (ITC) et l'Intel Trace Analyzer (ITA) (Intel, 2006) ont remplacé VGV puisqu'Intel a acheté les compagnies le développant.

NetLogger (Tierney, Johnston, Cowley, Hoo, Brooks, Gunter, 1998) est aussi un outil d'analyse de performance, mais pour les systèmes distribués de haute performance. L'analyse est basée sur le traçage d'événements avec une trace par noeud. Une synchronisation des horloges doit être réalisée étant donnée que NetLogger n'utilise pas une méthodologie de synchronisation des événements des différentes

[13]http ://www.openmp.org/

traces. (Tierney, Johnston, Cowley, Hoo, Brooks, Gunter, 1998) suggèrent d'utiliser NTP avec une serveur d'heure alimenté par un GPS pour chaque sous-réseau du système distribué.

TAU (Shende, Malony, Cuny, Lindlan, Beckman, Karmesin, 1998) permet le profilage et le traçage d'application utilisant l'approche orienté objet avec le langage C++. Il n'apporte toutefois rien de particulier dans sa méthodologie de traçage.

Le site Web (SciApps, 2006) présente d'autres outils de déboguage et de profilage, et l'article (Moore, Cronk, London, Dongarra, 2001) compare les performances de quelques outils de ce genre.

En résumé, ce chapitre a présenté plusieurs techniques de synchronisation de traces d'un réseau distribué. Cette synchronisation peut être assurée en synchronisant les horloges de chaque noeud du réseau à l'aide de matériel (e.g. module GPS) ou d'algorithmes ou la combinaison des deux. La synchronisation des traces peut aussi être réalisée par un traitement a posteriori en utilisant les communications réseau comme points de synchronisation. Cette façon de faire a l'avantage d'être moins intrusive dans le comportement du système en comparaison aux algorithmes de synchronisation d'horloge générant des échanges de messages sur le réseau. Enfin, il est aussi possible d'effectuer la synchronisation des traces en profitant des spécificités de l'environnement utilisé. Par exemple, la librairie de profilage de MPI permet de réaliser cette tâche et elle est utilisée par plusieurs outils directement ou avec quelques modifications. Une autre technique utilise l'horloge interne d'un commutateur reliant les noeuds du réseau pour obtenir une heure globale.

Le chapitre suivant décrit les concepts de base importants sous Linux afin d'effectuer les bons choix quant à la lecture du temps et le traçage d'événements réseau. Ces deux points sont primordiaux afin de synchroniser les traces d'un réseau dis-

tribué. Ainsi, la gestion du temps sous Linux et les facteurs pouvant influencer la précision des mesures de temps sont abordés, ainsi qu'une description de la réception et transmission de paquets.

CHAPITRE 2

CONCEPTS DE BASE SOUS LINUX

Ce chapitre a pour objectif d'expliquer la gestion du temps sous Linux et tous les intervenants influençant l'imprécision des lecture du temps en faisant varier le temps de réponse du système. Ces concepts sont importants puisqu'ils sont les éléments centraux de la synchronisation des traces d'un réseau distribué. Les temps associés aux événements réseau servent de point de synchronisation entre les traces. Ainsi, ce chaptitre présente aussi une description des interruptions matérielles et logicielles, des mécanismes de traitement de réception et de transmission des paquets réseau, et des appels système.

2.1 Temps sous Linux

Avant de décrire la gestion du temps sous Linux, une description du matériel permettant de suivre l'écoulement du temps et d'échantillonner des périodes de temps prime.

Real Time Clock (RTC) Tous les ordinateurs sont équipés de cette horloge. Elle est toujours active et elle permet de conserver l'heure même si l'ordinateur est éteint puisqu'elle est alimentée par une pile. Elle peut générer périodiquement une interruption (IRQ8) à des fréquences pouvant aller de 2 Hz à 8,192 kHz. Linux utilise cette horloge pour déterminer l'heure et la date. Les applications peuvent la programmer via le fichier /proc/rtc et le noyau via les ports 0x70 et 0x71. Les lectures et écritures sur cette horloge peuvent être réalisées avec la commande

UNIX *clock*.

Time Stamp Counter (TSC) Tous les systèmes Intel, depuis le Pentium, possèdent des microprocesseurs équipés d'un compteur de cycles d'horloge. Ce compteur est accessible via un registre de 64 bits grâce à l'instruction *rdtsc*. Ainsi, avec un microprocesseur de 1 GHz, ce compteur est incrémenté toutes les nanosecondes. Évidemment, Linux tire avantage de ce compteur lorsque vient le temps d'obtenir des mesures précises. Puisque la fréquence d'horloge ne peut être obtenue directement lors de la compilation du noyau, Linux doit approximer sa fréquence. Il le fait lors du démarrage (*boot time*) à l'aide de la fonction *calibrate_tsc()* en comptant le nombre de coups d'horloge dans un intervalle d'environ 5 millisecondes obtenu en programmant le PIT.

Programmable Interval Timer (PIT) Le PIT est une horloge que toutes les architectures compatibles avec l'ordinateur personnel IBM supportent. Ce compteur lance une interruption perpétuellement à une fréquence fixe. Cette fréquence est fixée à 1000 Hz pour le noyau Linux depuis sa version 2.5 pour les architectures i386. Ainsi, à toutes les millisecondes une interruption est lancée sur l'IRQ0 et la routine d'interruption de l'horloge système est donc exécutée.

CPU Local Timer Le composant Local APIC supporté récemment dans les processeurs Intel possède une horloge. Cette horloge est très similaire au PIT puisqu'elle a la capacité de générer périodiquement une interruption. Par contre, plusieurs différences sont notables. Le compteur du Local APIC est de 32 bits comparativement à 16 bits pour le PIT. Aussi, le compteur du Local APIC génère des interruptions seulement sur le processeur auquel il est relié (interruption locale) contrairement au PIT où n'importe quel processeur peut le traiter (interruption

globale). Le compteur du Local APIC est basé sur le bus du signal d'horloge (ou le bus de l'APIC pour les plus vieilles machines). Il peut être programmé seulement à une fréquence multiple du signal d'horloge du bus (1, 2, 4, 8, 16, 32, 64 ou 128). Le PIT quant à lui peut être programmé plus simplement puisqu'il possède son propre signal d'horloge.

High Precision Event Timer (HPET) Il s'agit d'un nouveau compteur développé conjointement par Intel et Microsoft. Il est encore peu utilisé mais les noyaux 2.6 le supporte déjà. Le HPET inclut huit compteurs 32 bits ou 64 bits indépendants. Chaque compteur possède son propre signal d'horloge dont la fréquence est minimalement 10 MHz. À chaque compteur est associé 32 horloges qui sont composées d'un comparateur et d'un registre. Le comparateur génère une interruption lorsque la valeur du compteur est identique à celle du registre. Quelques compteurs peuvent aussi être configurés pour générer des interruptions périodiquement. Les prochaines générations de cartes mères vont supporter le HPET et le PIT mais le HPET est destiné à remplacer le PIT.

ACPI Power Management Timer (ACPI PMT) Cette horloge est incluse dans la majorité des cartes mères basées sur l'ACPI. Sa fréquence est de 3.58 MHz et, elle est préférée au TSC si le système d'opération a la capacité de dynamiquement diminuer la fréquence d'horloge ou le voltage des processeurs dans le but de diminuer la consommation et ainsi préserver les batteries. Dans ce cas la fréquence du TSC va changer, risquant de causer des comportements indésirables tandis que le ACPI PMT a une fréquence constante. Par contre, le TSC permet d'obtenir de loin une meilleure précision pour de petits intervalles de temps.

Comme mentionné ci haut, le PIT génère des interruptions à une fréquence fixe. Cette fréquence est définie par la directive suivante pour les architectures 80x86 et

à partir des noyaux 2.5 :

```
#define Hz 1000
```

La variable globale *jiffies* conserve le nombre d'interruptions générées par le PIT depuis le démarrage de la machine et elle est définie ainsi :

```
extern unsigned long volatile jiffies;
```

Il est donc possible de savoir depuis combien de temps une machine a démarré simplement en effectuant cette division *jiffies/Hz* pour obtenir le nombre de secondes. Pour des fins de compatibilité avec les versions antérieures de Linux, la variable globale *jiffies* est définie sous 32 bits. Le problème est que la fréquence *Hz* a augmenté de dix faisant ainsi passer de 497 jours à 49,7 jours le temps nécessaire pour que le contenu de cette variable repasse à zéro. Pour pallier à ce problème, une autre variable globale est définie (*jiffies_64*) de 64 bits rendant ainsi le temps à écouler pour obtenir un débordement suffisamment grand pour que personne ne s'en rende compte. Le contenu de la variable *jiffies* est donc tiré des 32 bits les moins significatifs de la variable *jiffies_64*. La variable *xtime* permet de conserver la date et l'heure courante. Cette variable est une structure de type *timespec* et ainsi le champ *tv_sec* vaut le nombre de secondes écoulées depuis le 1er janvier 1970. Cette variable est mise à jour à chaque interruption du PIT de même que la variable *jiffies_64*. Cette variable peut être modifiée par un usager à sa convenance. Ceci peut parfois causer un problème car certaines applications ont besoin de manipuler un temps monotone. Pour le bien de la cause, la variable *wall_to_monotonic*, de même type que *xtime*, conserve le temps à ajouter à *xtime* pour obtenir un temps monotone.

Enfin, plusieurs horloges plus précises que le PIT ont été déjà présentées. Linux tire avantage de ces horloges pour augmenter la précision du temps. La variable

cur_timer contient l'horloge la plus précise disponible selon cet ordre de préférence :

1. HPET
2. ACPI PMT
3. TSC
4. PIT
5. dummy (aucune horloge disponible).

Grâce à une horloge plus précise, Linux peut par interpolation obtenir un gain considérable en précision.

Pour avoir une meilleure idée générale de la gestion du temps sous Linux, un résumé de la phase d'initialisation et de la routine d'interruption suit. La phase d'initialisation a lieu lors du démarrage de la machine. Voici dans l'ordre des étapes effectuées pour un système monoprocesseur :

1. Initialiser *xtime* à partir du RTC.
2. Initialiser *wall_to_monotonic.*
3. Si le HPET est supporté, initialiser celui-ci pour être l'horloge système sinon utiliser le PIT.
4. Choisir l'horloge la plus précise pour *cur_timer.*
5. Affecter au IRQ0 le HPET ou le PIT.

Voici maintenant les étapes effectuées dans la routine d'interruption de l'horloge système :

1. Acquérir le verrou de protection des variables de temps.
2. Selon l'horloge la plus précise disponible (*cur_timer*)
   (a) vérifier si des coups d'horloge ont été perdus en mettant la variable *jiffies_64* à jour si nécessaire

   (b) sauvegarder la valeur courante du compteur (*cur_timer*)
   (c) note : dans le cas du ACPI PMT et du TSC, ces horloges sont utilisées pour obtenir une meilleure résolution.
3. Mettre à jour la date, l'heure et les statistiques du système.
4. Mettre à jour les statistiques du processeur.
5. Si l'horloge est synchronisée avec une source extérieure (appel système *adjtimex()* exécuté), telle NTP, mettre la valeur de RTC à jour (seulement toutes les onze minutes).

La section suivante traite des interruptions. Les interruptions influencent la gestion du temps puisque leur utilisation fait varier le temps de réponse du système, ce qui induit une incertitude lors des lectures du temps.

2.2 Interruptions

Les interruptions sont divisées en deux familles : synchrone et asynchrone ou exception et interruption dans le langage des processeurs Intel. Les interruptions synchrones sont générées par le processeur et causées par des erreurs de programmation ou des conditions d'exécution anormales. Les interruptions asynchrones sont produites par des périphériques. Les interruptions asynchrones peuvent aussi être catégorisées en interruptions masquables et non masquables (NMI).

2.2.1 PIC

Chaque périphérique peut générer des interruptions à une (ou plusieurs) ligne(s) IRQ (Interrupt ReQuest) assignées. Chaque ligne IRQ est connectée en entrée du PIC (Programmable Interrupt Controler) et sont numérotées de 0 à 15. Le PIC

est le gestionnaire des interruptions. Il permet d'informer le CPU d'exécuter la routine d'interruption associée au périphérique qui a lancé la requête. Si plusieurs interruptions sont lancées simultanément, la ligne IRQ avec la plus petite valeur a la priorité.

Le PIC permet d'activer/désactiver une ligne IRQ à la demande. La désactivation d'une ligne IRQ n'implique pas la perte d'interruption, le PIC informera le CPU de l'interruption seulement lorsque la ligne IRQ sera réactivée. Cette fonctionnalité est fréquemment utilisée par les routines d'interruption permettant d'exécuter chaque interruption une à la suite de l'autre. Il est important de noter qu'une interruption peut interrompre une routine d'interruption, alors désactiver une ligne IRQ évite pour cette ligne que la même routine d'interruption s'empile.

Le PIC est composé d'un contrôleur 8259 dans les systèmes PC/XT (1983) et de deux contrôleurs 8259 en cascade dans les systèmes PC/AT (1984) et plus récents (voir la figure 2.1). Voici un exemple de configuration :

Master 8259

IRQ0	PIT
IRQ1	clavier
IRQ2	ligne d'interruption du 8259 esclave
IRQ3	port série COM2 et COM4
IRQ4	port série COM1 et COM3
IRQ5	port parallèle LPT2
IRQ6	contrôleur de disquette
IRQ7	port parallèle LPT1

Slave 8259

IRQ8	RTC

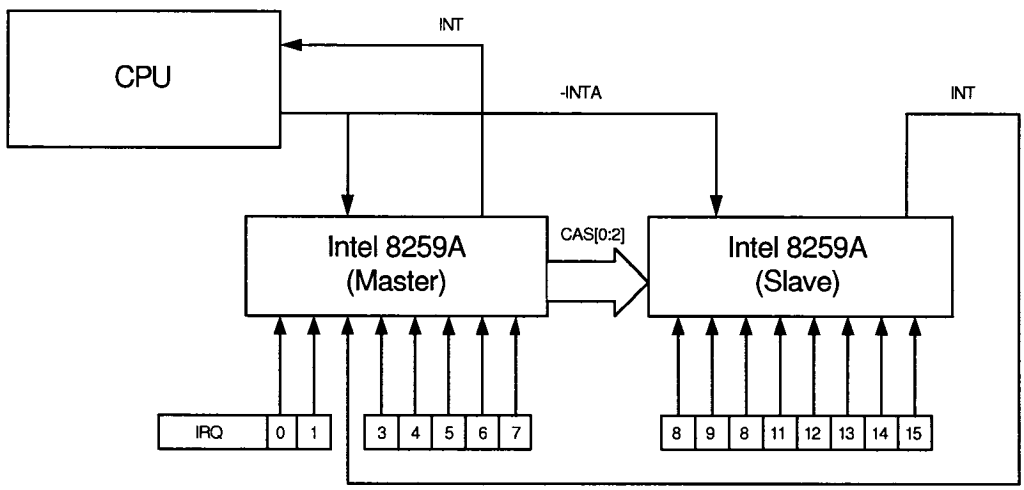


Figure 2.1 Schéma du PIC

IRQ9	ACPI
IRQ10	carte réseau (périphérique PCI)
IRQ11	non assigné
IRQ12	souris PS/2
IRQ13	coprocesseur mathématique
IRQ14	contrôleur primaire IDE
IRQ15	contrôleur secondaire IDE

2.2.2 APIC

De nos jours le PIC est remplacé par le APIC depuis l'arrivée des Pentium III d'Intel. L'APIC est développé pour une meilleure gestion des interruptions dans les systèmes multiprocesseurs (SMP, Symmetric Multi-Processor).

L'APIC est composé du Local APIC (LAPIC) et du IO-APIC (voir la figure 2.2).

Chaque processeur possède un Local APIC qui est connecté au IO-APIC via un bus. L'IO-APIC est le successeur du PIC. Même les systèmes récents uniprocesseurs sont munis d'un IO-APIC pouvant être configuré en deux modes : comme le PIC standard en désactivant le Local APIC ou comme le IO-APIC standard.

Le IO-APIC est composé de 24 lignes IRQs, de 24 registres programmables (*Interruption Redirection Table*) et d'un contrôleur de bus permettant de transmettre et recevoir des messages du bus APIC. La gestion des priorités est différente de celle du PIC. Chaque entrée de la table de redirection peut être configurée en indiquant le vecteur d'interruption, la priorité, le processeur destinataire et comment le processeur est sélectionné.

Il y a 256 vecteurs disponibles, les 32 premiers (0x00 à 0x1F) étant réservés pour les exceptions et les 16 suivants (0x20 à 0x2F) sont généralement utilisés pour les interruptions logicielles. La priorité d'une interruption est déterminée par son vecteur d'interruption divisé par 16. Ainsi, le vecteur 0x74 est moins prioritaire que le vecteur 0x84 puisque la priorité appartient au vecteur ayant le plus grand premier chiffre hexadécimal, ce qui implique aussi que les vecteurs 0x74 et 0x7C ont la même priorité. Contrairement au PIC, si plusieurs interruptions sont lancées simultanément, la priorité du IRQ n'est pas en jeu pour désigner le choix du IRQ à traiter en premier. Le IO-APIC vérifie à tour de rôle chaque IRQ et distribue le premier IRQ actif qu'il trouve. Par contre, un Local APIC n'interrompra pas son exécution en cours, si la priorité de l'interruption arrivée est inférieure au processus en cours.

Les requêtes d'interruptions provenant de périphériques peuvent être distribuées de façon statique ou dynamique sur tous les processeurs disponibles. En mode statique, la requête est transmise au Local APIC selon l'entrée de table de redirection correspondante. En mode dynamique, la requête est au Local APIC exécutant le

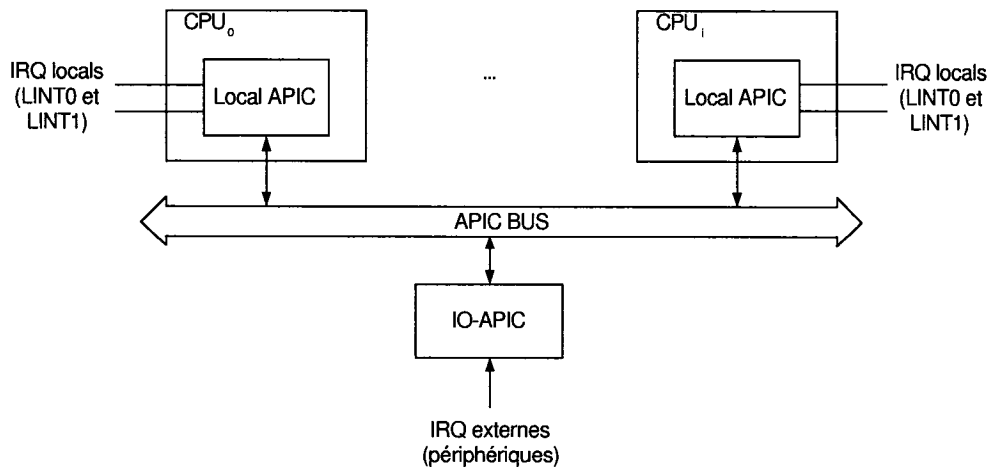


Figure 2.2 Schéma du APIC

processus avec la plus faible priorité. Un mécanisme d'arbitrage est lancé s'il y a égalité entre plusieurs processeurs. Chaque Local APIC possède un registre contenant une priorité d'arbitrage assignée allant de 0 à 15 où 15 est la priorité la plus forte. La valeur 0 est écrite dans ce registre chaque fois qu'une interruption est délivrée à un processeur tandis que les autres voient ce registre incrémenté de un ou assigné de la valeur du processeur gérant l'interruption si la valeur dépasse 15. La distribution des interruptions se voit donc être uniforme pour chaque processeur.

Un autre avantage de ce système se situe au niveau de l'empilement des interruptions. Avec le PIC, c'est au programmeur de s'assurer de bien gérer l'empilement possible des interruptions. Le Local APIC va interrompre l'exécution du processeur seulement si la priorité de l'interruption à gérer est plus forte que la priorité du processus courant, évitant ainsi l'empilement d'interruptions de même priorité.

Une autre caractéristique très intéressante de l'APIC est qu'il permet de générer des interruptions directement entre processeurs. Cette caractéristique est fréquemment utilisée pour des fins de synchronisation.

En résumé, avec le PIC, toute routine d'interruption peut se faire interrompre par une autre interruption, peu importe la priorité. La priorité est en jeu seulement lorsqu'il y a plusieurs interruptions lancées simultanément. Avec le APIC, le choix de la priorité des interruptions est plus important puisqu'une interruption de faible priorité ne peut en interrompre une de plus forte priorité. Dans ce cas, il faudrait prioriser les interruptions de l'horloge système (PIT) et celles associées au traitement réseau afin de minimiser le temps de réponse dans ces cas.

2.3 Interruptions logicielles (softirqs)

Puisque les interruptions ne sont pas prédictibles en terme de temps et de fréquence d'arrivée, il faut absolument minimiser le plus possible le temps de traitement d'une interruption pour maintenir un temps de réponse court aux interruptions. Le problème est que certaines interruptions nécessitent beaucoup de calcul à réaliser, par exemple le traitement d'émission et de réception de paquets par une carte réseau, d'où l'utilité des interruptions logicielles. Elles permettent de diminuer la taille des routines d'interruption en remettant à plus tard certains traitements.

La gestion des interruptions sous Linux est divisée en deux parties, la première partie est la routine d'interruption (*top half*) et la seconde partie est utilisée pour différer du traitement (*bottom half*). Avec un noyau 2.6 de Linux, il existe trois mécanismes pour différer du traitement :

- Softirq
- Tasklet
- Work queues (introduit depuis les noyaux 2.5).

Les *softirqs* et les *tasklets* sont similaires puisque ces derniers sont bâtis sur les *softirqs*. Par contre, les *works queues* font monde à part en étant bâtis sur les processus

noyaux (*kernel threads*). Les *softirqs* sont alloués statiquement lors de la compilation du noyau. Il est donc impossible de créer et détruire dynamiquement des *softirqs* contrairement aux *tasklets*. Seuls six *softirqs* sont définis sur une possibilité de 32 :

Tableau 2.1 Description des *softirqs*

Softirq	Priorité	Description
HI_SOFTIRQ	0	Tasklet de forte priorité
TIMER_SOFTIRQ	1	Routine de l'horloge système
NET_TX_SOFTIRQ	2	Pour l'envoie de paquets réseau
NET_RX_SOFTIRQ	3	Pour la réception de paquets réseau
SCSI_SOFTIRQ	4	Routine *bottom haft* pour le SCSI
TASKLET_SOFTIRQ	5	Tasklet de faible priorité

Actuellement, seuls les traitements réseau et SCSI utilisent directement les *softirqs* ainsi que les horloges système (*kernel timers*) et les *tasklets* bâtis sur les *softirqs*. Les *softirqs* sont requis par les systèmes ayant une fréquence très élevée d'utilisation et utilisés par un nombre important de processus.

Dans le cas général, les *tasklets* sont préférés aux *softirqs* puisqu'ils sont beaucoup plus simples d'utilisation en permettant la création et la destruction dynamiquement et avec une interface et des règles de verrouillage (*locking*) plus légères. Les *tasklets* sont, à toutes fins pratiques, des *softirqs* pouvant avoir deux priorités différentes (HI_SOFTIRQ et TASKLET_SOFTIRQ).

Les développeurs du noyau de Linux ont dû faire des compromis quant à leur choix d'implantation pour l'exécution des *softirqs* et donc aussi des *tasklets*. Les *softirqs* sont dans la plupart des cas générés à la suite d'une interruption mais un *softirq* peut lui-même s'activer. Alors, considérant la possibilité d'une forte génération de *softirqs* et la caractéristique d'autoactivation d'un *softirq*, le processeur risque d'occuper tout son temps à traiter les interruptions et les *softirqs* laissant les ap-

plications à leur faim. Le paradoxe est le suivant : faut-il simplement exécuter les *softirqs* lorsqu'ils se présentent ou ne pas exécuter directement les *softirqs* lorsqu'ils sont activés par eux-mêmes ? La réponse est ni l'un ni l'autre. Le problème est que si les *softirqs* sont traités dès leur arrivée, le noyau ne fera qu'exécuter les *softirqs* dans un système très chargé monopolisant ainsi le processeur. De l'autre côté, si tous les *softirqs* (avec ceux en attentes) sont exécutés seulement lors de l'arrivée d'une interruption, le temps d'attente peut être très long (au pire au prochain coup d'horloge) avant que les *softirqs* activés par eux-mêmes soient traités. Les développeurs ont choisi de ne pas traiter immédiatement les *softirqs* lorsqu'ils sont activés par eux-mêmes. Par contre, si le nombre de *softirqs* devient important, le noyau réveille une famille de processus noyau (*ksoftirqd*) pour traiter la demande. Ces processus noyau ont la plus faible priorité d'exécution (*nice 19*) pour s'assurer que leur exécution n'interrompra pas des processus importants, tout en assurant à ce fort flot de *softirqs* d'être traité éventuellement. Cette solution permet d'utiliser efficacement un système dans l'état d'attente (*idle*) ainsi qu'un système ayant plusieurs processeurs à sa main puisqu'il y a un processus noyau associé à chaque processeur (*ksoftirqd/0* et *ksoftirqd/1* pour un système avec deux processeurs).

Pour leur part, les *works queues* permettent de différer un traitement à l'aide d'un processus noyau. Les *works queues* bénéficient donc de tous les avantages des processus, dont la capacité d'être suspendus. Les *works queues* ne sont qu'une interface permettant la création de processus noyau réalisant des tâches mises en queue nommés *worker threads*. Cette interface prévoit même des *worker threads* par défaut nommés *events/0, events/1*, ... selon le nombre de processeurs disponibles (un *worker thread* par processeur). Règle générale, ces *worker threads* par défaut sont utilisés à moins que le traitement demandé soit très lourd et très critique, dans ce cas mieux voudra définir une version personnalisée.

Le tableau 2.2 résume les différences entre les trois moyens permettant de différer un

Tableau 2.2 Comparaison des méthodes pour différer une tâche

Méthode	**Contexte d'exécution**	**Exécution**
Softirq	Interruption	Concurrente entre processeur
Tasklet	Interruption	En série pour le même Tasklet
Work queues	Processus	Ordonnancée tel un processus

traitement. Les *softirqs* maximisent la puissance de calcul d'un système possédant plusieurs processeurs mais ils nécessitent de prévoir leur propre mesure de protection des variables globales partagées en les dupliquant par processeur pour être efficace. Les *tasklets* sont beaucoup plus simples d'utilisation que les *softirqs* en n'ayant pas de concurrence entre les mêmes *tasklets* tout en offrant une excellente performance dans le cas général. Enfin, les *works queues* présentent l'avantage de pouvoir être ordonnancés et ainsi d'être mis en attente.

En résumé, les interruptions logicielles permettent d'alléger la taille des routines d'interruption, ce qui a pour effet de diminuer le temps de réponse pour les interruptions. Par contre, les *softirqs*, les *tasklets* et les *works queues* partagent l'usage du processeur avec tous les processus en cours. Ainsi, lorsque le système est très chargé, le temps de réponse des interruptions logicielles se retrouve augmenté car l'ordonnanceur essaie de donner la chance à chaque processus d'utiliser le processeur. Ce point est important puisque le traitement des paquets réseau (présenté à la section suivante) utilise les *softirqs*. Ainsi, une lecture de temps à l'intérieur d'un *softirq* présente une incertitude influencée par l'arrivée d'interruptions (et d'exceptions) mais aussi par la charge sur le processeur.

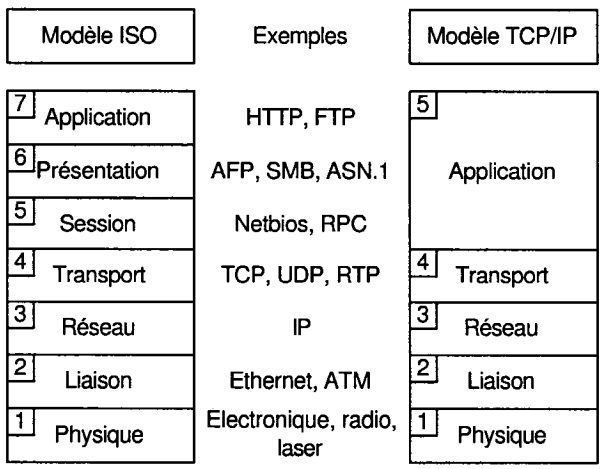


Figure 2.3 Modèle ISO vs TCP/IP

2.4 Réception/transmission de paquets réseau

Le traitement des paquets réseau est essentiel pour la synchronisation de traces dans un réseau distribué puisque les échanges de message sont utilisés comme point de synchronisation. Voici les deux modèles principaux de protocoles réseau : le modèle ISO et le modèle TCP/IP. Le modèle ISO est souvent utilisé comme référence lors de discussion mais c'est le modèle TCP/IP qui est le plus souvent implanté de nos jours dans les systèmes d'exploitation comme Linux.

Plusieurs protocoles sont associés à chaque couche de ces modèles, l'image 2.3 en énumère quelques-uns. Comme il a été mentionné ci-haut, Linux se base sur le modèle TCP/IP pour gérer les communications réseau. Un résumé de son implantation suit en débutant par la réception d'un paquet réseau.

Lorsqu'un paquet est reçu par la carte réseau (NIC, Network Interface Card), il faut trouver un moyen d'en informer le noyau. Il existe principalement trois techniques : faire une scrutation périodique, générer une interruption ou combiner ces 2

techniques. En effectuant une scrutation, le temps de réponse de la réception d'un paquet est optimal, mais cette technique va gaspiller inutilement des ressources. Générer une interruption à chaque arrivée de paquet réseau représente la meilleure technique lorsque l'activité réseau est faible. Par contre, lorsque l'activité est élevée, la performance en souffre. En gros, le traitement de l'arrivée d'un paquet réseau est divisé en deux étapes. La première étape (routine d'interruption) ne fait que copier ce paquet dans une queue et la seconde étape (*softirq*) décode chaque paquet dans la queue. Ainsi, avec une fréquence élevée de communication, le processeur n'aura pas le temps de traiter les paquets réseau puisqu'il passera tout son temps à exécuter la routine d'interruption. Une variante de cette méthode est de mettre en queue plusieurs paquets lors d'une interruption jusqu'à un nombre maximal fixé. Cette méthode permet ainsi de diminuer le nombre d'interruptions mais elle aura un impact sur les autres périphériques car cette routine désactive les interruptions. Une autre technique est de vérifier périodiquement la réception de paquets réseau à l'aide d'une horloge. Ainsi, le périphérique génère périodiquement une interruption et vérifie s'il y a eu réception, ou encore mieux génère une interruption seulement si une réception a eu lieu. Dans ce cas, tous les paquets arrivés sont mis en queue. La performance dépend alors de la granularité de l'horloge. Par contre, peu de périphériques possèdent une horloge interne. Ce manque pourrait être remplacé par une horloge du système mais cela risque de gaspiller inutilement des ressources, ce qui revient carrément à scruter périodiquement le périphérique.

Linux offre deux modèles d'interaction entre la carte réseau et le noyau, dont le NAPI (New API) implanté depuis les versions 2.5 du noyau. Il a aussi été introduit dans les récents noyaux 2.4 (à partir du 2.4.19). L'autre modèle, étant encore très utilisé, est aussi décrit et est identifié par Old API (OAPI). L'approche du OAPI est de mettre en queue plusieurs paquets, limités à une borne supérieure, lors de la routine d'interruption. Cette borne permet de diminuer le nombre d'interruptions

mais aussi un partage équitable des ressources du système. L'approche du NAPI est quelque peu différente. Au lieu de désactiver toutes les interruptions et laisser la routine d'interruption placer les paquets en queue, cette méthode désactive les interruptions seulement pour le périphérique réseau en question et délègue le noyau pour remplir la queue en scrutant le périphérique. Ainsi, l'approche du NAPI est un mélange entre le modèle par interruption et par scrutation. Elle présente plusieurs avantages au détriment du OAPI. Premièrement, la charge sur le processeur est considérablement réduite pour un état de communication élevé en raison de la diminution du nombre d'interruptions, sachant que pour le modèle OAPI une interruption se produit à chaque arrivée de paquet réseau. Par contre, selon des tests, la performance est légèrement diminuée lorsqu'il y a peu de communication en augmentant quelque peu l'usage du processeur. Le NAPI permet aussi un partage plus équitable entre les périphériques réseau assurant le même temps d'attente pour un périphérique communiquant peu que pour un autre communiquant beaucoup.

Avant de décrire ces deux approches, il est important de décrire les types de données en jeu, principalement ces trois enregistrements : *struct sk_buff*, *struct net_device* et *struct softnet_data*. L'enregistrement *sk_buff* (souvent déclaré avec l'identificateur *skb*) est l'endroit où est sauvegardé un paquet. Il est structuré de telle sorte que toutes les couches du modèle TCP/IP y sont représentées. L'enregistrement *net_device* (souvent déclaré avec l'identificateur *dev*) est utilisé pour caractériser chaque périphérique réseau présent dans le système. Il contient tant les informations sur la configuration matérielle que logicielle du périphérique réseau. Enfin, chaque processeur possède une déclaration de l'enregistrement *softnet_data*. Ceci permet de diminuer les verrous en maximisant la puissance d'un système multiprocesseur. Cet enregistrement est utilisé par les deux approches mais différemment. Il contient à la fois les informations pour la réception et la transmission de paquets. Voici sa définition :

```
struct softnet_data
{
    int             throttle;
    int             cng_level;
    int             avg_blog;
    struct sk_buff_head input_pkt_queue;
    struct list_head            poll_list;
    struct net_device           *output_queue;
    struct sk_buff              *completion_queue;
    struct net_device           backlog_dev;
};
```

Les champs *throttle*, *avg_blog* et *cng_level* sont utilisés pour la gestion des congestions. Le champ *input_pkt_queue* est une queue utilisée pour emmagasiner les paquets reçus. Elle est partagée par chaque périphérique réseau et elle est seulement utilisée dans l'approche du OAPI. Le champ *backlog_dev* représente en fait un périphérique bidon qui est seulement utilisé par l'approche OAPI. Le champ *poll_list* est une liste de périphériques réseau ayant des paquets reçus prêts à être traités. Le champ *output_queue* est une liste de périphériques ayant des paquets à transmettre et le champ *completion_queue* est une liste de paquets ayant été transmis.

L'image 2.4 présente la vue générale du modèle OAPI. Chaque périphérique réseau possède soit une mémoire interne ou une mémoire réservée dans le noyau (*rx_ring*). Une fois qu'un paquet est reçu par la carte réseau, une interruption est générée. Dans le cas d'un système multiprocesseur, cette interruption est captée par le IO-APIC qui la distribue à un processeur. Le processeur lance ensuite la routine d'interruption associée (ISR). L'ISR débute en initialisant un enregistrement *sk_buff*

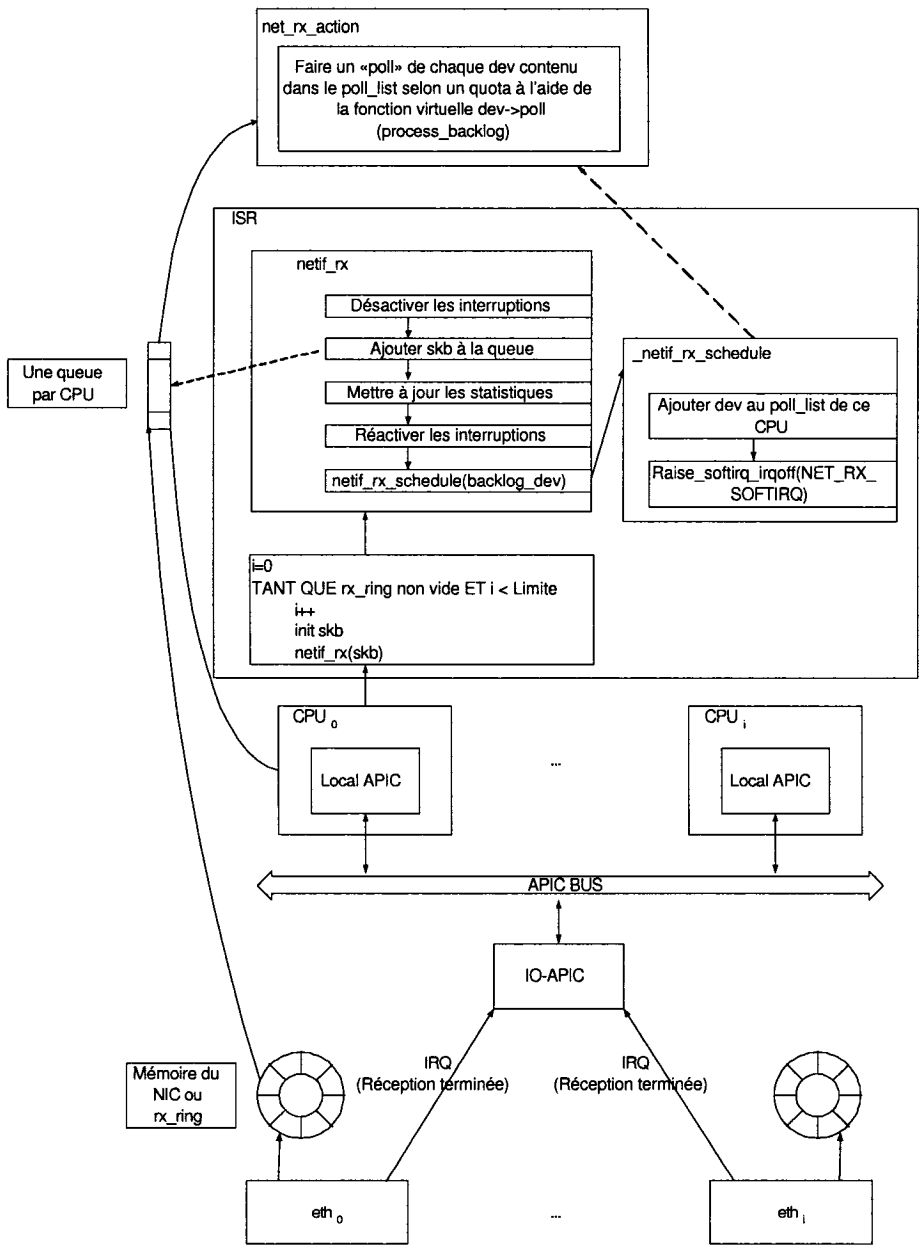


Figure 2.4 Réception de paquets selon l'approche OAPI

(*skb*), par exemple initialiser le protocole utilisé, puis elle appelle la fonction *netif_rx()*. Cette fonction désactive les interruptions pour ce processeur pour pouvoir accéder à l'enregistrement *softnet_data* dans le but d'ajouter le pointeur du paquet reçu (et non le contenu du paquet) à la queue de réception (*input_pkt_queue*). Les interruptions sont réactivées après avoir mis à jour les statistiques d'utilisation. Par la suite, la fonction *netif_rx_schedule()* est appelée pour placer dans la liste, *poll_list*, le périphérique réseau *backlog_dev* et enfin de lancer le *softirq* NET_RX_SOFTIRQ. Ces instructions sont répétées tant qu'il reste des paquets reçus, sans dépasser une limite pour assurer un partage équitable des ressources du système. À noter que la fonction *netif_rx_schedule()* ne fait rien si le périphérique a déjà fait une demande d'ordonnancement sans avoir été traitée.

L'image 2.5 présente la vue générale du modèle NAPI. La différence avec l'approche OAPI débute par l'ISR. Tout ce qu'elle fait est d'appeler la fonction *netif_rx_schedule()* pour placer le périphérique réseau en jeu (*dev*) dans la liste *poll_list* et lancer le *softirq* NET_RX_SOFTIRQ.

Les deux approches utilisent la même routine de *softirq* (*net_rx_action*) qui est présentée dans la figure suivante. Cette routine débute par la désactivation des interruptions du processeur pour accéder à l'enregistrement *softnet_data* afin de vérifier si la *poll_list* est vide. En supposant qu'elle n'est pas vide, que la limite totale de traitement de paquet n'est pas dépassée et qu'il reste du temps disponible sur le processeur, les interruptions sont activées et le premier périphérique (*dev*) de la liste, *poll_list* est choisi. Si son quota n'est pas dépassé, la fonction virtuelle *dev->poll* est appelée pour traiter les paquets reçus de ce périphérique. La différence entre les deux approches se situe à ce niveau. Pour l'OAPI, la fonction virtuelle est la même pour chaque périphérique et est nommée *process_backlog*. Pour le NAPI, chaque périphérique doit fournir sa propre fonction virtuelle. Comme les images 2.4 et 2.5 le montrent, les principales différences entre ces deux fonctions virtuelles sont

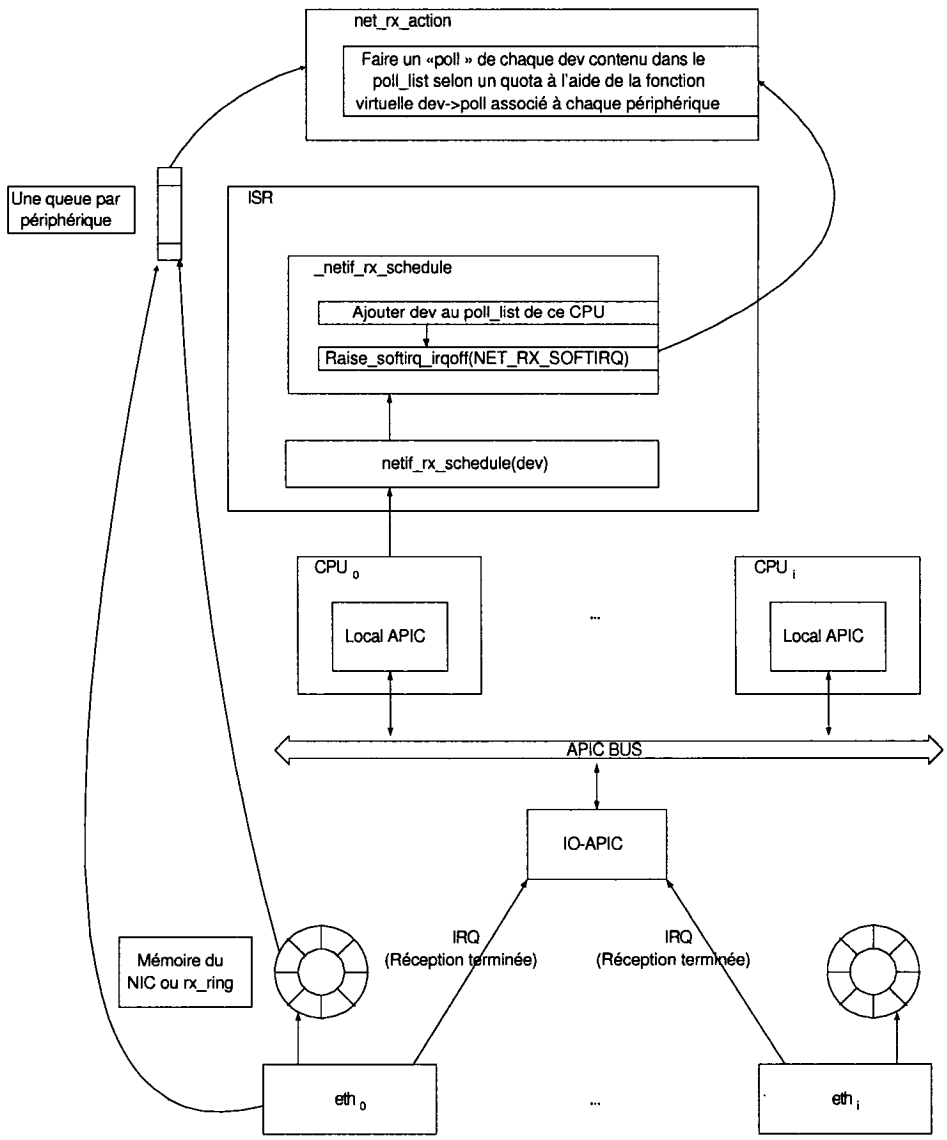


Figure 2.5 Réception de paquets selon l'approche NAPI

que la fonction virtuelle *process_backlog* doit accéder l'enregistrement *softnet_data* en désactivant les interruptions tandis que pour l'approche NAPI, la fonction virtuelle accède directement à la queue du périphérique puisque les interruptions de celui-ci sont déjà désactivées. La conclusion à retenir est que l'approche NAPI nécessite moins l'usage de verrous pour accéder aux queues de réception de paquets. Elle permet aussi un meilleur partage des ressources entre les périphériques réseau. Contrairement au OAPI, chaque périphérique possède sa propre queue et beaucoup moins d'interruptions sont générées. Pour l'OAPI, chaque périphérique partage la même queue (une par processeur), et une interruption est lancée à chaque réception de paquets.

Le principe pour la transmission est relativement semblable à la réception (voir le tableau 2.3) en ayant par contre une seule approche. La figure 2.6 illustre l'idée générale. La queue *completion_queue* est semblable à la queue *input_pkt_queue* qui possèdent seulement des pointeurs vers des paquets (*skb*). La *completion_queue* contient les pointeurs des paquets prêts à être libérés. Aussi, la queue *output_queue* est semblable à la queue *poll_list* contenant toutes les deux des périphériques (*dev*). Les périphériques placés dans la *output_queue* possèdent des queues de paquets n'ayant pu être transmis pour diverses raisons (périphérique inactif, congestion, ...). De plus, la transmission de paquets utilise un *softirq* qui peut être ordonnancé par trois fonctions : *netif_schedule*, *netif_wake_queue* et *dev_kfree_skb_irq*. Les deux dernières fonctions sont seulement utilisées par le pilote du périphérique réseau. La fonction *netif_wake_queue* indique que le périphérique est maintenant actif. Dans ce cas, il est nécessaire d'ordonnancer le *softirq net_tx_action* pour s'assurer qu'il n'y a pas de paquets de ce périphérique en attente d'être transmis. La fonction *dev_kfree_skb_irq*, tant qu'à elle, place le pointeur du paquet transmis dans la queue *completion_queue* et ordonnance ce *softirq* pour libérer son espace mémoire. Enfin, la fonction *netif_schedule* est appelée par les autres couches réseau lorsque la trans-

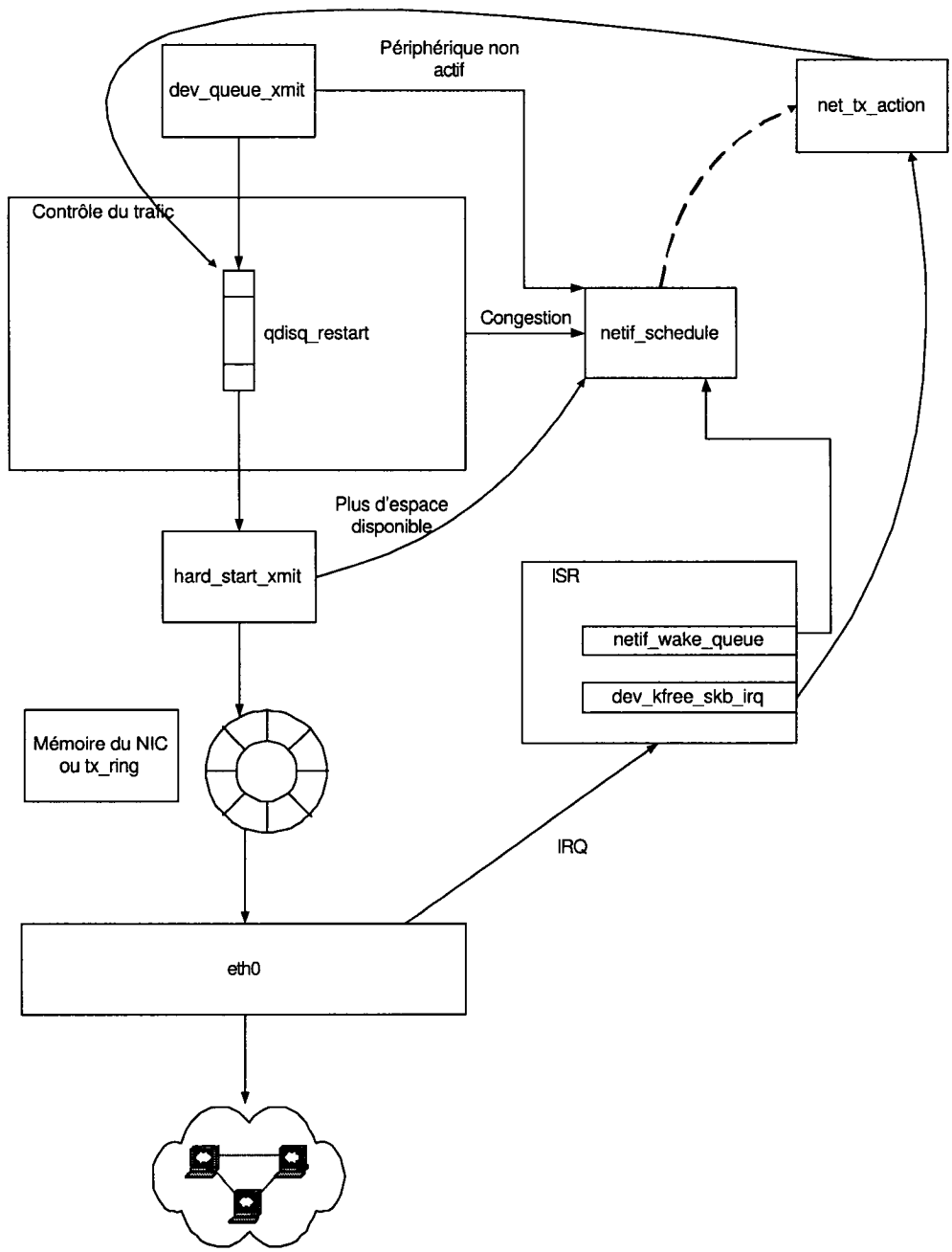


Figure 2.6 Transmission de paquets

mission de paquets n'est pas possible afin de tenter, plus tard, de transmettre ces paquets. Cette fonction place le périphérique dans la *output_queue* et ordonnance aussi ce *softirq*.

Tableau 2.3 Comparaison entre la réception et la transmision

Réception	**Transmission**
poll_list	output_queue
input_pkt_queue	completion_queue
netif_rx_schedule	netif_schedule netif_wake_queue dev_kfree_skb_irq

La fonction *dev_queue_xmit* permet de placer un paquet (*skb*) à la queue de transmission d'un périphérique (*dev*). Si le périphérique est actif, ce paquet sera mis en queue de transmission seulement si le contrôleur de trafic (*qdisq_restart*) ne signale pas de congestion. Par la suite, les paquets sont envoyés dans la mémoire du périphérique réseau ou *tx_ring* s'il reste de l'espace disponible, et enfin les paquets sont transmis au réseau. Une fois la transmission complétée, une interruption est générée pour l'indiquer au noyau. La routine d'interruption place le pointeur du paquet transmis dans une queue (*completion_queue*) pour différer le travail de libérer la mémoire au *softirq*, *net_tx_softirq*. Ce *softirq* exécute deux tâches bien précises. Premièrement, il libère l'espace mémoire de tous les pointeurs présents dans la queue *completion_queue*. Ensuite, il essaie de transmettre tous les paquets en queue non transmis de chaque périphérique en parcourant cette queue *output_queue* pour déplacer les paquets non transmis au dispositif de contrôle de trafic, donnant ainsi une seconde chance à ces paquets d'être transmis.

Le traitement des paquets réseau est une tâche complexe pouvant générer un fort débit d'interruptions. Pour cette raison, les développeurs de Linux ont mis beaucoup d'efforts afin d'optimiser le plus possible ce traitement, en réduisant le temps

de réponse d'un événement réseau, tout en minimisant l'impact du temps de réponse pour les autres tâches du système. Deux modèles permettent de traiter la réception de paquets. Le NAPI offre une meilleure performance que le OAPI en diminuant l'usage de verrous et la génération d'interruptions. Le défi est de trouver le meilleur endroit dans le traitement des paquets réseau pour recueillir les événements réseau nécessaires dans l'intention de synchroniser les traces d'un réseau distribué. Cette partie est traitée dans la section 3.2. Pour terminer le chapitre, les appels système sont décrits. Ils sont fortement utilisés dans les applications usager et leur utilisation peut influencer les lectures de temps.

2.5 Appels système

Généralement, une application se doit d'interagir avec le noyau, que ce soit pour communiquer avec celui-ci ou avec des périphériques. Pour des questions de sécurité et de stabilité, aucune application au niveau espace usager ne peut directement exécuter un segment de code du noyau (espace noyau). La raison est que l'espace mémoire utilisé au niveau noyau est protégé en lecture et écriture. Alors, pour répondre à ce besoin, le noyau offre une panoplie d'interfaces permettant l'interaction entre l'espace usager et l'espace noyau dont les appels système sont le seul véritable moyen, en ne considérant pas les exceptions et les déroutements (*traps*). D'autres interfaces sont disponibles comme les fichiers contenus dans le répertoire /proc, mais elles reviennent toutes à être manipulées par des appels système.

Pour des fins de portabilité, les appels système ne sont pas invoqués directement mais plutôt par l'entremise d'un API basé sur le standard POSIX (de l'IEEE). Cet API est défini en grande partie dans la librairie standard du langage C. Cette librairie comprend donc le standard du langage C mais aussi l'interface aux appels système sous Linux.

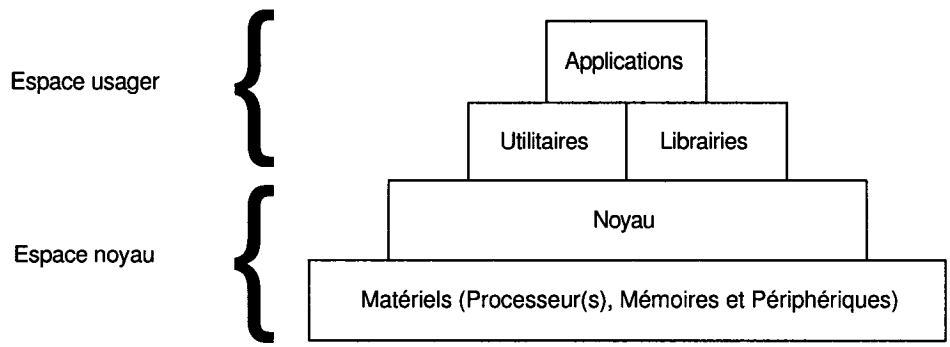


Figure 2.7 Espace usager et espace noyau

Chaque appel système a un numéro unique assigné qui ne peut jamais être modifié afin de permettre le bon fonctionnement des applications déjà existantes. Ainsi, lorsqu'un numéro est désigné pour un appel système, il ne peut pas être supprimé ou recyclé pour un autre usage. Pour exécuter un appel système, il faut trouver un mécanisme qui peut informer le noyau de l'arrivée de cet événement sans perdre de vue qu'une application ne peut directement accéder à l'espace noyau. La solution est de lancer une interruption (synchrone) par l'entremise d'une exception (voir la figure 2.8). De cette façon, lorsque le processeur traite l'exception, le système passe de l'espace usager à l'espace noyau, et la routine d'interruption de l'exception est lancée correspondant à la routine de traitement des appels système. Pour différencier chaque appel système, une information supplémentaire est requise. Dans les systèmes Intel, cette routine vérifie le contenu du registre *eax* contenant le numéro de l'appel système. Les registres *ecx*, *edx*, *esi*, et *edi* contiennent respectivement la valeur des cinq paramètres de l'appel système et le registre *eax* contient la valeur de retour correspondant au code de l'erreur s'il y a lieu.

Linux définit environ 250 appels système pour les systèmes Intel, ce qui est très

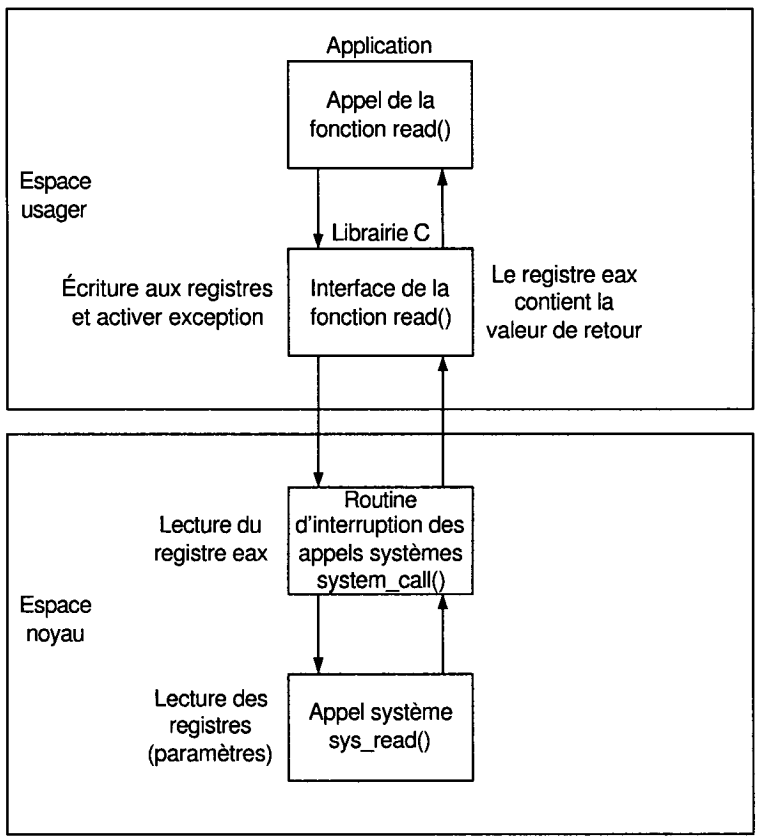


Figure 2.8 Traitement des appels système

peu considérant que d'autres systèmes en utilisent plus d'un millier. De plus, le traitement des appels système sous Linux est extrêmement rapide, ce qui s'explique en grande partie par l'efficacité de ses changements de contexte et aussi par leur simplicité de conception. Puisque les appels système génèrent des exceptions, ils influencent les lectures de temps, mais aussi le traitement des paquets réseau puisqu'il utilise les *softirqs* qui peuvent être interrompus par ces exceptions.

En résumé, ce chapitre a décrit la gestion du temps sous Linux. Parmi les horloges disponibles présentement dans les ordinateurs (système Intel) de nos jours, le TSC est le compteur le plus précis. C'est pour cette raison qu'il s'agit du compteur utilisé pour les lectures de temps des événements lors du traçage. Par contre, sa fréquence étant inconnue, elle doit être approximée au démarrage de Linux. Ceci rajoute une imprécision dans la lecture du temps. D'autres facteurs influencent cette imprécision dont le temps de réponse. Dépendamment de l'endroit où sont effectuées ces lectures, le temps de réponse varie. Ainsi, on peut espérer avoir le temps de réponse le plus court à l'intérieur d'une routine d'interruption. Toutefois, il est possible qu'une telle routine soit elle-même interrompue par une interruption ou en attente d'être exécutée dans le cas où les interruptions sont désactivées. Afin, de diminuer le plus possible la latence des interruptions, les interruptions logicielles sont nées. Elles permettent de différer un traitement à plus tard, allégeant ainsi les routines d'interruption. Toutefois, une lecture de temps à l'intérieur d'une routine d'interruption logicielle (par exemple, un *softirq*) risque d'avoir un temps de réponse plus lent puisqu'ils partagent l'usage du processeur avec les autres processus du système. Ensuite, le traitement des paquets réseau a été exposé dans le but de comprendre leur fonctionnement et de faire un choix éclairé quant à l'endroit choisi pour recueillir les événements réseau dans les traces. Pour terminer, les appels système ont été présentés. Il s'agit d'un autre facteur pouvant influencer les lectures de temps puisqu'ils génèrent des exceptions et ils sont fortement uti-

lisés par les applications usager. Le chapitre suivant présente les choix utilisés dans l'implantation du système de traçage afin de rendre possible la synchronisation des traces tout en obtenant des mesures de temps précises.

CHAPITRE 3

ALGORITHMES ET STRUCTURES DE DONNÉES

Ce chapitre décrit l'implantation utilisée pour tracer les événements d'un réseau distribué. La description débute par la présentation des concepts généraux du système de traçage afin de permettre la synchronisation de traces d'un réseau distribué. Par la suite, la façon d'apparier les paquets réseau est détaillée. Il s'agit du traitement nécessaire de manière à pouvoir relier un événement de transmission d'un paquet réseau d'un noeud source avec l'événement de réception du paquet réseau en question, par le noeud destinataire. Ces appariements de paquet réseau constituent les points de synchronisation menant à l'évaluation du décalage entre deux noeuds à un instant donné. Ensuite, les structures de données utilisées sont décrites, suivies de l'explication du calcul de l'approximation de la dérive d'horloge à partir des informations sur le décalage en fonction du temps obtenues lors de l'appariement des paquets réseau. Pour terminer, le choix de la référence et des références intermédiaires est exposé. Il s'agit d'une étape nécessaire dans l'intention de synchroniser les traces sur une base de temps commune. Les références intermédiaires sont nécessaires dans le cas où des noeuds se retrouvent isolés par rapport à d'autres noeuds du réseau.

3.1 Concepts généraux

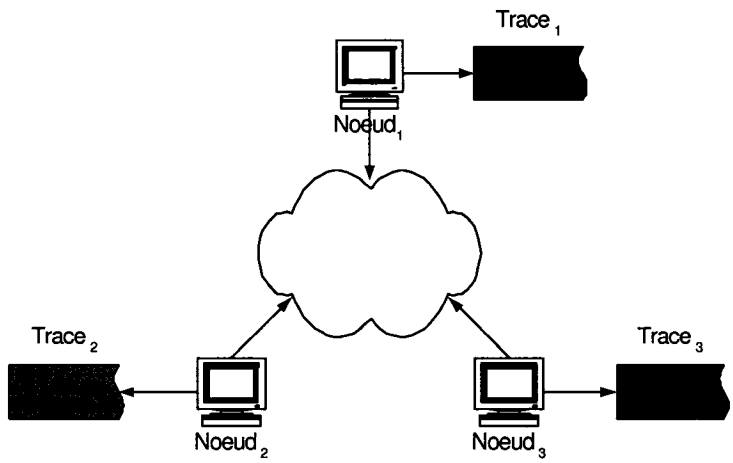
Linux Trace Toolkit Next Generation[1] (Desnoyers, Dagenais, 2006) (LTTng) est le traceur d'événements utilisé. Le principe est d'installer ce traceur dans le noyau

[1] `http://ltt.polymtl.ca`

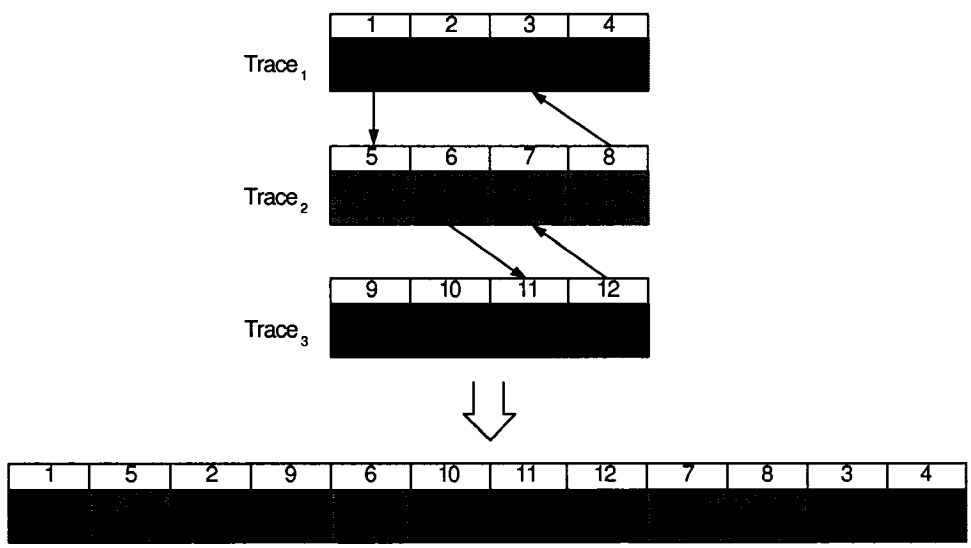
de chaque noeud du réseau distribué. Pour analyser le réseau distribué, il suffit de démarrer le traçage sur chaque noeud afin de recueillir, à la fin du traçage, toutes les traces générées (une par noeud). Ayant récupéré toutes les traces, il faut synchroniser les événements de chacune des traces dans le temps. Pour ce faire, il est nécessaire d'avoir des points de synchronisation, utilisés comme repères, dans les traces. Ces repères sont les échanges de messages durant la période de traçage. Sans communication, la synchronisation d'événements est impossible. Le calcul des points de synchronisation est inspiré de la méthode de Cristian (section 1.2.1.1). Ainsi, un cycle de communication aller-retour permet de déduire un décalage dans le temps entre deux noeuds. Une étude expérimentale a permis de constater que le décalage entre deux noeuds varie de façon linéaire dans le temps. Ainsi, avec un ensemble de points de décalage, une régression linéaire permet d'approximer cette dérive. Connaissant ainsi tous les décalages entre chaque noeud, il faut trouver une base de temps commune afin de synchroniser les événements contenus dans chaque trace. Cette référence correspond au noeud communiquant avec le plus grand nombre possible de noeuds et ce de la façon la plus précise. Des références intermédiaires sont choisies selon le même principe pour les noeuds isolés de la référence choisie, c'est-à-dire qui ne présentent aucun échange de messages (direct ou indirect) avec la référence. Les sections suivantes décrivent en détails cette implantation (voir la figure 3.1) en débutant par le mécanisme d'appariement des paquets réseau.

3.2 Appariement des paquets réseau

Cette étape est nécessaire afin d'identifier les points de synchronisation. Pour obtenir la meilleure précision possible, ces événements doivent être placés au plus bas niveau i.e. dans la routine d'interruption du périphérique réseau. Les routines



(a) Traçage de tous les noeuds du réseau distribué



(b) Synchronisation des événements de chaque trace

Figure 3.1 Architecture du traçage d'un réseau distribué

d'interruption ont le temps de réponse le plus court. Ainsi, à ce niveau, les lectures de temps sont moins influencées par l'état du système car, seule une interruption peut interrompre son exécution. Toutefois, ce choix présente plusieurs désavantages. Premièrement, chaque périphérique possède sa propre routine d'interruption empêchant ainsi d'avoir une version générique du traçage des événements réseau. Deuxièmement, il est possible que le pilote du périphérique réseau utilise l'approche NAPI (voir la section 2.4) où il n'y a pas nécessairement une interruption générée à chaque réception de paquet. Troisièmement, à ce niveau, les paquets réseau reçus ne sont pas décodés, ce traitement étant différé. Le décodage des paquets réseau est indispensable pour pouvoir associer le paquet émis d'un noeud avec le paquet reçu équivalent sur un autre noeud. Le décodage permet de recueillir les informations pertinentes permettant l'appariement des paquets réseau. Dans ce cas, il faudrait donc copier le contenu des paquets réseau au complet dans la trace et les décoder au moment de la lecture de celle-ci. L'impact serait important au niveau de la lecture des traces mais aussi au niveau du traçage puisque beaucoup de données devraient alors être écrites dans les traces.

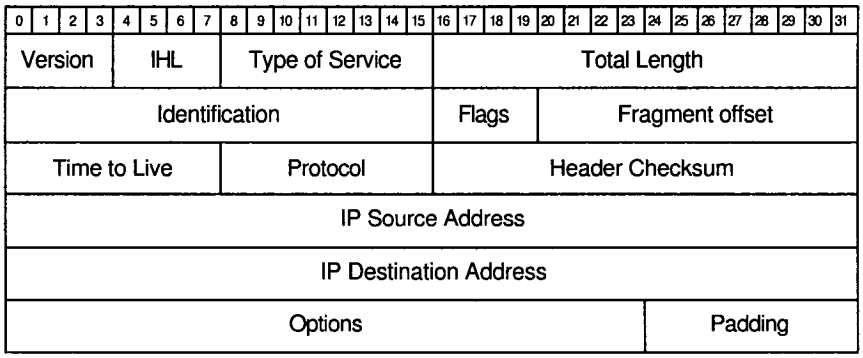
Une autre possibilité est de placer le traçage d'événements réseau au niveau des *softirqs*. Encore une fois, il est important de choisir l'endroit le plus près du début du traitement des paquets réseau dans les *softirqs* afin d'associer le temps le plus juste à l'événement réseau tout en diminuant le temps de réponse (diminuer le risque d'être interrompu). Le plus bas niveau pour la réception de paquets réseau est la fonction virtuelle de traitement *dev->poll*. Toutefois, il s'agit d'une autre solution non générique (pour l'approche NAPI) et les paquets réseau reçus ne sont toujours pas décodés.

La solution est de placer les points de traçage au plus bas niveau du noyau. En d'autres mots, à l'endroit où le noyau lance la transmission d'un paquet réseau et où, il reçoit un paquet. Au moment de la transmission, le noyau connaît les

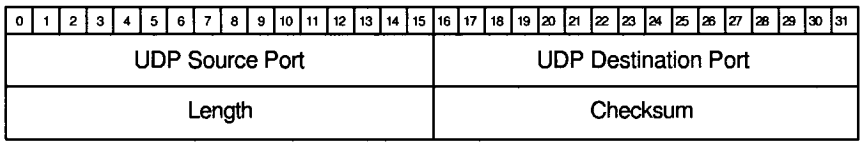
informations de chaque couche réseau d'un paquet grâce à l'enregistrement *sk_buff*. Par contre, pour la réception, les couches supérieures à la couche physique ne sont pas encore décodées. Pour pallier à ce problème, lors de la réception de paquets réseau, deux événements peuvent être générés. Le premier correspond à l'entrée du paquet dans le noyau et le deuxième est placé à la couche réseau désirée. Il est possible de lier ces deux événements à l'aide du pointeur vers un enregistrement de type *sk_buff*, puisqu'il est impossible que le noyau utilise le même pointeur pour traiter deux paquets réseau différents simultanément. Ainsi, la contrainte principale est d'avoir le temps le plus précis en réception comme en transmission. De plus, il est nécessaire d'avoir accès aux contenus de l'entête TCP des paquets (voir la section suivante) pour identifier les paires d'événements qui correspondent à l'envoi et à la réception d'un même paquet. En réception, les paquets doivent être décodés et ce traitement est relativement long. Pour ces raisons, deux événements sont utilisés pour la réception et un seul pour la transmission. Maintenant, il faut déterminer la couche réseau la plus appropriée pour tracer les événements réseau.

3.2.1 TCP vs IP

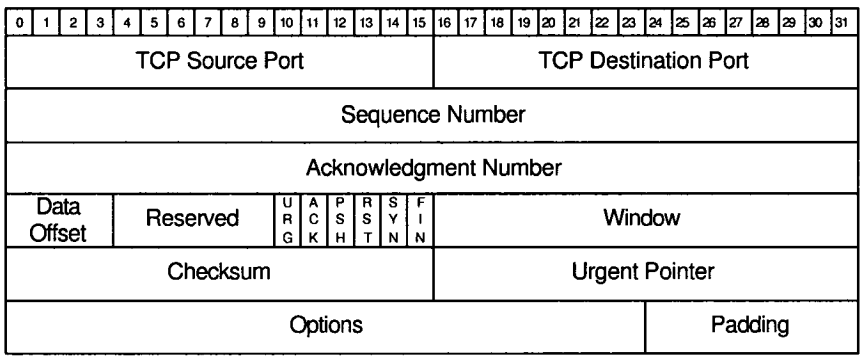
Le choix de la couche réseau utilisée pour l'appariement de paquets réseau doit permettre l'identification unique de chaque message pour identifier les paires d'événements qui correspondent à l'émission et à la réception d'un même paquet, mais aussi être un protocole transparent et largement utilisé. Le protocole IP serait donc le choix idéal. Le schéma 3.2(a) présente l'entête du protocole IP. Ce protocole permet la fragmentation de datagramme afin de transmettre des messages dépassant la taille maximale des trames utilisée par la couche liaison. Puisque plusieurs fragments de datagrammes différents peuvent être transmis simultanément, un mécanisme d'identification est présent, assurant un réassemblage adéquat. Ainsi, la valeur du champ identification est choisie afin d'assurer l'unicité, tout au long de



(a) Entête du protocole IP



(b) Entête du protocole UDP



(c) Entête du protocole TCP

Figure 3.2 Entêtes de protocoles réseau

la transmission, du datagramme pour un couple source-destination selon un protocole (ces quatre champs sont donc utilisés : source address, destination address, identification et protocol).

Malheureusement, ce protocole ne fournit pas assez d'information pour assurer un appariement de paquets réseau sans faille. Tout d'abord, la valeur du champ iden-

tification est souvent réutilisée pour un protocole et une paire source-destination. Ceci n'a pas d'incidence pour le réassemblage des fragments mais risque de créer de mauvais appariements lors de la lecture des traces, puisque le traçage ne débute pas nécessairement en même temps pour chaque noeud du réseau distribué. Ainsi, deux datagrammes différents peuvent être confondus. De plus, ce mécanisme ne permet pas d'identifier les retransmissions de datagrammes. Les retransmissions de fragments peuvent être détectées en suivant l'assemblage du datagramme lors de la lecture des traces sans toutefois pouvoir détecter les retransmissions de datagrammes (générées par une couche supérieure, TCP par exemple). Les retransmissions sont généralement causées lorsque le système est dans un état d'erreur et elles peuvent donc induire des erreurs lors de la phase d'appariement. Il est donc important de les détecter pour les ignorer comme points de synchronisation.

La couche transport fournit plus d'information sur les paquets réseau. UDP et TCP sont les deux protocoles les plus utilisés à ce niveau. L'entête du protocole UDP (figure 3.2(b)) ne rajoute pas assez d'information pour pallier aux problèmes présentés avec le protocole IP. Par contre, le protocole TCP (figure 3.2(c)) permet d'apparier, de façon adéquate, les paquets réseau et de détecter les retransmissions grâce à son mécanisme d'identification et d'accusé réception des datagrammes. Ainsi, l'unicité est assurée grâce aux adresses IP d'origine et de destination, aux ports TCP d'origine et de destination, au numéro de séquence et au numéro d'accusé réception. La détection d'une retransmission est possible en recherchant un duplicat du datagramme entre le premier paquet réseau (envoyé ou reçu) et l'accusé réception. De plus, il faut vérifier la présence de duplicats d'accusé réception dans la fenêtre du délai d'attente maximal que le protocole TCP spécifie pour générer une retransmission. Il s'agit donc du protocole choisi pour l'appariement des paquets réseau. Le désavantage de ce choix est que plusieurs échanges de messages réseau seront ignorés lors du calcul de la dérive d'horloge dont les messages UDP et

ICMP. Toutefois, le protocole TCP est largement utilisé entre autre par la librairie MPI pour les échanges de messages.

Afin de valider les mesures prises expérimentalement, un serveur UDP est utilisé avec lequel tous les noeuds communiqueront périodiquement avec des paquets facilement identifiables. Ce protocole étant plus léger que le protocole TCP, l'augmentation du trafic aura un faible impact. Chaque datagramme UDP transmis contient un entier dont la valeur est incrémentée à chaque envoi. Ainsi, avec le couple d'adresses IP source-destination, le couple de ports UDP source-destination et cet entier, l'unicité est assurée. La section suivante présente l'implantation du traçage des communications réseau dans le noyau de Linux.

3.2.2 Instrumentation du noyau

Trois événements sont utilisés dans le but d'identifier les points de synchronisations dans les traces. Pour le traçage de messages transmis, l'endroit choisi dans le noyau est la fonction *dev_queue_xmit()* présente dans le fichier *net/core/dev.c*. Pour la réception de paquets réseau, deux événements sont utilisés : le premier, au début de la réception au noyau dans la fonction *netif_receive_skb()* du fichier *net/core/dev.c* et le deuxième, à l'endroit où les paquets réseau sont décodés au niveau TCP dans la fonction *tcp_v4_rcv()* du fichier *net/ipv4/tcp_ipv4.c*. Le tableau 3.1 présente les informations recueillies pour chaque événement. Il est important de noter qu'un temps est associé à chaque événement, pris à partir du TSC, et que, pour la réception d'un paquet réseau, c'est le temps du premier événement qui est utilisé lors du calcul des points de décalage.

Pour les communications UDP générées afin de valider les résultats obtenus, les informations recueillies sont : les adresses IP d'origine et de destination, les ports

Tableau 3.1 Informations recueillies pour chaque événement

Protocoles	**Événement de transmission**	**Événement de réception 1 (paquet reçu)**	**Événement de réception 2 (paquet décodé)**
-	-	Adresse du pointeur *sk_buff*	Adresse du pointeur *sk_buff*
IP	Adresse d'origine Adresse de destination Total length IHL	- - - -	Adresse d'origine Adresse de destination Total length IHL
TCP	Port d'origine Port de destination Sequence number Acknowledgement number Data offset ACK SYN FIN	- - - - - - - -	Port d'origine Port de destination Sequence number Acknowledgement number Data offset ACK SYN FIN

UDP d'origine et de destination et un entier qui est incrémenté à chaque transmission. Les points de traçage sont identiques sauf pour le deuxième événement de réception. Il est placé dans la fonction *udp_rcv()* du fichier *net/ipv4/udp.c*.

De plus, il est nécessaire de recueillir les adresses IP des noeuds afin d'associer celles-ci avec une trace. Pour ce faire, au début du traçage, l'état des interfaces réseau IP est sauvegardé à l'aide du module *statedump*. Il s'agit d'un module présent dans LTTng permettant la sauvegarde de l'état initial du noyau au début du traçage. Ce module parcourt la variable globale *dev_base*, pointeur de type *struct net_device*, où tous les périphériques réseau sont listés. Ensuite, il s'agit de sauvegarder toutes les adresses IP associées à chaque périphérique. Enfin, il est possible que des adresses IP changent au cours du traçage. Pour considérer ces changements, des événements

sont tracés lors de l'activation et la désactivation d'un périphérique en sauvegardant les adresses IP courantes. Ces événements sont placés dans la fonction *inet-dev_event()* du fichier *net/ipv4/devinet.c*.

La section suivante décrit le traitement de lecture des traces afin de trouver tous les points de synchronisation (appariement des paquets réseau).

3.2.3 Algorithme

Une fois le traçage complété, l'étape suivante est de lire toutes les traces pour identifier tous les échanges de messages. Puisqu'à cette étape les décalages ne sont pas connus entre les noeuds, la recherche des événements d'émission et de réception correspondant risque d'être laborieuse car, dans le pire cas pour un événement donné, elle peut nécessiter un parcours complet d'une trace. Afin de borner la fenêtre de recherche, il est suggéré de synchroniser l'heure des noeuds avant le traçage. Par exemple, NTP garantit, dans les meilleurs cas, un écart moyen de 1 à 5 millisecondes (Ubik, Smotlacha, 2003) mais, pouvant dépasser les 50 millisecondes (Mills, 2004) lorsque la source est un serveur secondaire disponible sur l'Internet.

Avant de débuter la lecture, il faut associer, pour chaque trace, la (ou les) adresse(s) IP utilisée(s) par le noeud. Il suffit de lire les informations sur les interfaces réseau générées par le module *statedump*. Les traces contiennent aussi les événements de changement d'état de ces interfaces permettant l'actualisation des adresses IP. Ensuite, la correspondance entre l'heure et le temps monotone fourni par le TSC doit être réaliséee. Au début du traçage, un événement spécial est généré en indiquant l'heure réelle (*do_gettimeofday()*) pour une valeur du TSC, relativement proche, avec la fréquence approximée de cette horloge par le noyau au démarrage (*cpu_khz*). L'heure réelle est nécessaire comme base absolue entre les traces et la

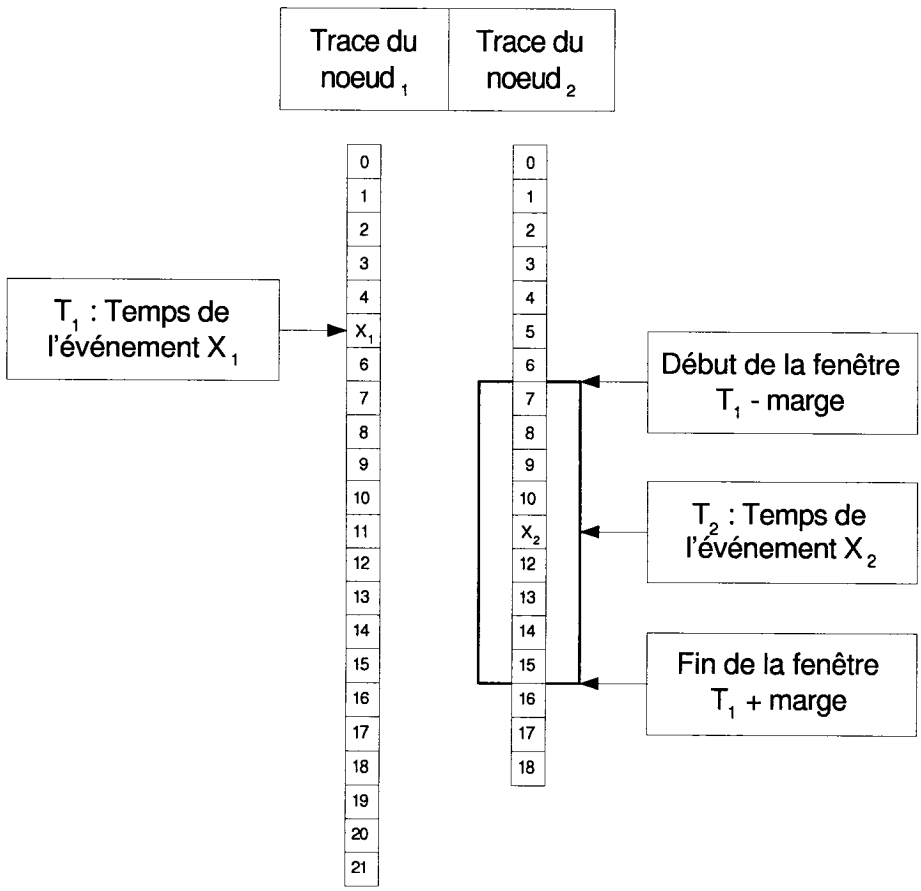


Figure 3.3 Fenêtre de recherche pour trouver l'événement X_2 correspondant à l'événement X_1

valeur du TSC avec sa fréquence indiquent le temps écoulé entre deux événements de la même trace.

L'idée est de parcourir les traces progressivement, à tour de rôle et sans retour arrière. Le traitement débute par la lecture de la trace d'un noeud jusqu'à la découverte d'un événement de communication dont les intervenants font partie du réseau distribué. Selon la valeur du TSC lors de cet événement, il est possible de connaître le temps écoulé depuis le début du traçage et donc de connaître son heure

réelle d'arrivée. Ainsi, les bornes de la fenêtre de recherche peuvent être déduites en appliquant une marge d'erreur (voir la figure 3.3). Une lecture de la seconde trace en jeu s'ensuit jusqu'à la détection du paquet recherché ou la limite de la fenêtre. Ce demi cycle aller-retour est sauvegardé. Tous les paquets rencontrés ne correspondant pas à la requête sont placés dans une queue. Ensuite, le traitement de la seconde trace débute en vérifiant s'il n'y a pas de paquets en queue, sinon une lecture de la trace s'ensuit. Ayant un paquet en main, il faut lancer une recherche dans la trace correspondante. Selon la fenêtre de recherche, il faut peut-être lire la queue et/ou la trace. Si la recherche est fructueuse, il faut vérifier s'il est possible de compléter un cycle aller-retour. Si tel est le cas alors, un point de synchronisation (valeur de décalage) est calculé et sauvegardé, sinon ce demi cycle aller-retour est sauvegardé. Ce procédé se poursuit jusqu'à la fin du parcours des traces et il est présenté par l'algorithme 1.

La section suivante développe les principales structures de données utilisées.

3.3 Structures de données

Le traitement de synchronisation des traces prises d'un réseau distribué utilise essentiellement trois structures de données. La première sert à mémoriser tous les points de décalage. Elle est conçue de n vecteurs d'arbres binaires balancés où n est le nombre de noeuds. Ainsi, chaque noeud possède un vecteur conservant les informations de décalage avec les autres noeuds, triées en fonction du temps. Ces informations sont indispensables pour le calcul de la dérive (voir la section 3.4) qui est évalué par segment. En effet, les traces sont segmentées en plusieurs morceaux puisque la dérive est linéaire pour une courte durée. Ainsi, la seconde structure sauvegarde les informations du calcul de la dérive. Elle est composée d'un tableau de n vecteurs de mesures de dérive où n est le nombre de noeuds. Chaque

Algorithme 1 Appariement des paquets réseau (version simplifiée, épurée des problèmes de retransmission et de l'actualisation de l'état des interfaces réseaux)

```
Initialiser les informations sur le temps
Initialiser les informations sur les interfaces réseau
TANT QUE le parcours des traces n'est pas terminé FAIRE
  POUR i allant de 1 au nombre de traces FAIRE
    SI la queue de paquets déjà lus n'est pas vide ALORS
      Prendre un paquet de la queue
    SINON SI la lecture de la trace i n'est pas terminée ALORS
      Parcourir la trace i jusqu'à l'obtention d'un paquet réseau
    SINON
      Indiquer que le parcours de la trace i est terminé
    FIN
    Assigner j au numéro de la trace communiquant avec i
    Calculer la fenêtre de recherche de la trace j
    SI un parcours de la queue de paquets en attente est nécessaire ALORS
      Rechercher le paquet
    FIN
    SI le paquet n'est pas trouvé ALORS
      Lire la trace j jusqu'à découverte du paquet ou la fin de la fenêtre
      Mettre en queue les paquets visités ne correspondant pas à celui recherché
    FIN
    SI le paquet est trouvé ALORS
      SI il est possible de compléter un cycle aller-retour ALORS
        Calculer un point de décalage {l'autre demi cycle est déjà calculé}
        Sauvegarder ce point
      SINON
        Sauvegarder ce demi cycle aller-retour
      FIN
    FIN
  FIN
FIN
```

élément du tableau caractérise un segment et chaque élément permet de conserver les informations sur la dérive entre chaque noeud. Enfin, la dernière structure sert à mémoriser la structure du réseau pour chaque segment. Il s'agit d'un tableau dont chaque élément qualifie un segment. Chaque élément contient une matrice de distances, une matrice de voisins adjacents et un vecteur de références (voir la section 3.5). Ces deux matrices permettent de conserver tous les chemins les plus courts entre chaque noeud. Le vecteur de références contient la référence trouvée pour chaque noeud. La section suivante présente le calcul de la dérive d'horloge.

3.4 Calcul de la dérive

La dérive est évaluée selon la méthode de régression linéaire des moindres carrés à partir de cette formule :

$$D_{ij} = X_{ij} * T + D_{ij}^0 \tag{3.1}$$

sachant que (voir la figure 1.7)

$$\begin{aligned} D_{ij} &= T_2 - (T_1 + C) \text{ avec,} \\ C &= \frac{(T_4 - T_1) - (T_3 - T_2)}{2} \end{aligned} \tag{3.2}$$

où, D_{ij} est le décalage entre les noeuds i et j pour un temps, T, donné avec X_{ij} comme dérive et D_{ij}^0 le décalage initial. Autrement dit, D_{ij}^0 correspond au décalage

initial entre les noeuds i et j s'ils avaient débuté leur traçage exactement en même temps. La recherche des cycles aller-retour de paquets réseau fournit des valeurs de décalages (D_{ij}) en fonction du temps (T). La régression linéaire permet ainsi d'évaluer les inconnues X_{ij} et D^0_{ij} en procurant aussi la précision des paramètres approximés (écart-type) grâce aux formules suivantes :

$$X_{ij} = \frac{n\sum_{k=1}^{n} T_k D_k - \sum_{k=1}^{n} T_k \sum_{k=1}^{n} D_k}{n\sum_{k=1}^{n} T_k^2 - (\sum_{k=1}^{n} T_k)^2} \tag{3.3}$$

$$D^0_{ij} = \frac{\sum_{k=1}^{n} T_k^2 \sum_{k=1}^{n} D_k - \sum_{k=1}^{n} T_k \sum_{k=1}^{n} T_k D_k}{n\sum_{k=1}^{n} T_k^2 - (\sum_{k=1}^{n} T_k)^2} \tag{3.4}$$

$$E_r = \sqrt{\frac{1}{n-2}\sum_{k=1}^{n}(D_k - X_{ij}T_k - D^0_k)^2} \tag{3.5}$$

$$E_{X_{ij}} = E_r \sqrt{\frac{\sum_{k=1}^{n} T_k^2}{n\sum_{k=1}^{n} T_k^2 - (\sum_{k=1}^{n} T_k)^2}} \tag{3.6}$$

$$E_{D^0_{ij}} = E_r \sqrt{\frac{n}{n\sum_{k=1}^{n} T_k^2 - (\sum_{k=1}^{n} T_k)^2}} \tag{3.7}$$

où E_r correspond à l'écart-type résiduel. Il s'agit d'une estimation de l'erreur commise sur la mesure du décalage D_{ij}. La section suivante expose le choix d'une base de temps commune afin de synchroniser les traces d'un réseau distribué.

3.5 Choix de la référence et des références intermédiaires

Dans l'intention d'appliquer les décalages de temps entre les noeuds, il faut trouver une base de temps commune. Mais avant, il faut estimer les meilleurs chemins à prendre i.e. les chemins les plus précis. Pour ce faire, un graphe de tous les chemins possibles est construit avec l'imprécision (E_r) comme poids des arêtes. Par la suite, il faut trouver tous les plus petits chemins reliant ensemble les noeuds. Plusieurs algorithmes existent pour réaliser ce traitement et ils sont présentés dans le tableau 3.2 (Sedgewick, 2002). L'avantage de l'algorithme de Floyd est sa simplicité d'implantation mais son coût est trop élevé. L'algorithme de Dijkstra et de Johnson (Johnson, 1977) ont la même complexité. L'algorithme de Dijkstra base son traitement sur l'utilisation d'une queue de priorité tandis que Johnson, sur les propriétés des matrices creuses (*sparses matrix*). L'algorithme choisi est celui de Dijkstra pour sa simplicité d'implantation (moins de calcul) et le gain de l'algorithme de Johnson sera faible parce que le graphe risque d'être complet dans la plupart des cas.

Tableau 3.2 Comparaison des algorithmes de recherche de tous les plus petits chemins

Contraintes sur les poids	**Algorithmes**	**Coût**
Valeurs non négatives	Floyd	V^3
Valeurs non négatives	Dijkstra	VElogV
Aucun cycle négatif	Floyd	V^3
Aucun cycle négatif	Johnson	VElogV

Maintenant, il faut trouver une base de temps commune de manière à trier les événements de l'ensemble des traces dans le temps. Tous les plus courts chemins étant estimés, il s'agit d'identifier le noeud qui est le plus proche de l'ensemble des noeuds du réseau. Une sommation de la distance de chaque noeud avec toutes les autres est calculée à l'aide des poids calculés en passant par les plus courts chemins.

Le noeud ayant la plus petite somme est désigné comme la référence. La référence pourrait aussi être choisie manuellement comme étant, par exemple, le noeud où l'application parallèle démarre. La référence étant trouvée, le tri des événements des traces dans le temps est accessible. Il suffit d'appliquer le décalage dans le temps de chaque trace par rapport à la référence en passant par le chemin le plus court les reliant (chaque noeud risque d'utiliser un chemin différent).

Par contre, il est possible que certains noeuds ne communiquent pas directement ou indirectement avec la référence. Dans ces cas, il faut déterminer une ou plusieurs références intermédiaires de la même façon présentée ci haut pour les noeuds isolés. A priori, on peut penser que si deux sous-graphes ne communiquent pas, l'utilisation d'une référence de temps différente importe peu car ces traces sont complètement indépendantes. Néanmoins, il est possible que d'autres événements corrélés soient présents autres que les communications réseau. Par exemple, deux noeuds échangeant des commandes de lecture et écriture sur une même unité de disque partagée (par exemple un SAN (Tate, Kanth, Telles, 2005)). L'ordre de ces événements pourrait être important pour diagnostiquer un comportement anormal.

Pour conclure, ce chapitre a décrit l'implantation du système de traçage rendant possible la synchronisation des traces d'un réseau distribué. Les événements réseau servent de point de synchronisation des traces. Afin d'avoir un temps précis et de sauvegarder seulement les informations nécessaires pour l'appariement des paquets réseau, deux événements sont utilisés pour la réception et un seul, pour la transmission. L'information des paquets réseau est recueillie au niveau TCP car le protocole IP ne permet pas d'identifier les paires d'événements qui correspondent à l'envoi et à la réception d'un même paquet contrairement au protocole TCP. Le désavantage d'utiliser le protocole TCP est qu'un grand nombre d'échanges de messages n'est pas considéré dans l'évaluation de la dérive d'horloge (par exemple, les communications UDP, ICMP, ...). Par contre, l'usage de communication TCP reste

important et ce protocole est utilisé lors des échanges de messages utilisés par la librairie MPI. L'approximation de la dérive d'horloge entre les noeuds est réalisée par une régression linéaire des moindres carrés à partir de plusieurs points de décalage en fonction du temps. Ces points sont obtenus par la technique de Cristian. Une fois que les décalages en fonction du temps entre chaque noeud sont connus, une référence est désignée pour servir de base de temps. Le choix de la référence est basé sur la précision et la couverture sur l'ensemble du réseau. Par la suite, les décalages de temps sont appliqués sur les traces de chaque noeud par rapport à la référence, en utilisant le plus petit chemin reliant un noeud avec la référence. Le chapitre suivant présente les résultats obtenus selon différents scénarios de test permettant de cerner les limites de cette implantation.

CHAPITRE 4

RÉSULTATS

Ce chapitre présente les résultats de plusieurs scénarios de test permettant de cerner les limites de l'implantation du système de traçage distribué exposé au chapitre précédent. Tout d'abord, il est important de connaître l'impact de la fréquence des communications. Est-ce qu'un fort débit d'échange de messages dégrade la précision de l'approximation de la dérive ? Est-ce qu'un débit minimal de communications est requis ? La première section tentera de répondre à ces questions. Ensuite, l'impact de la durée du traçage est étudié. L'objectif de cette étude est de déterminer la durée idéale du traçage pour laquelle on peut considérer que la dérive d'horloge varie de façon linéaire en fonction du temps. Pour une longue période de temps, il est possible que la dérive se caractérise plutôt par une légère courbe. Connaissant le degré de linéarité de la dérive d'horloge, il sera possible de statuer sur la nécessité de fragmenter le calcul de la dérive et dans quel ordre de grandeur. Autrement dit, pour un traçage d'une longue durée, est-il nécessaire de segmenter le calcul de la dérive d'horloge sur plusieurs intervalles de temps ? Par exemple, un traçage d'une durée T secondes pourrait être segmenté en S segments. Pour chaque segment, une approximation linéaire de la dérive serait effectuée et considérée lors de la synchronisation des événements.

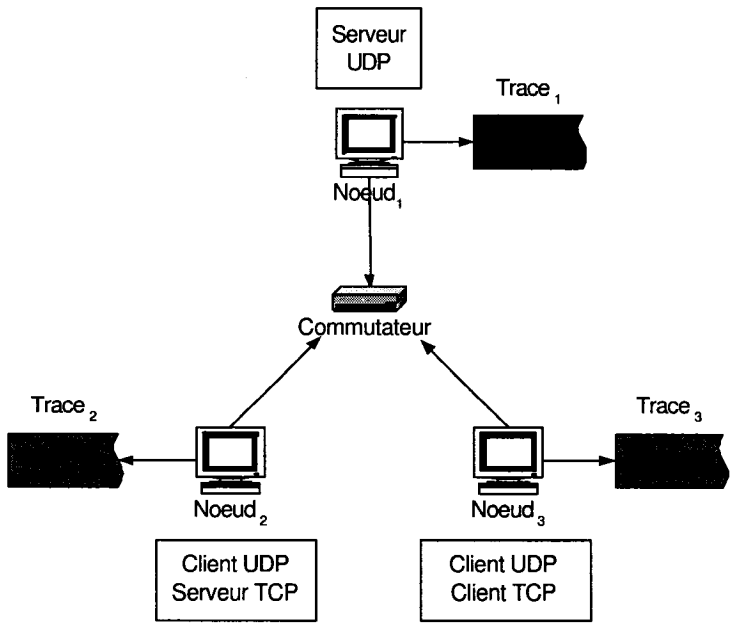
Par la suite, une étude a pour but de déterminer la sensibilité de la précision de l'approximation de la dérive pour des systèmes sous différentes charges. Ces tests vont induire une variabilité du temps de réponse des systèmes et risquent d'influencer les mesures de temps au niveau des échanges de messages. Ensuite, une analyse de la proximité des noeuds est présentée dans l'intention de déterminer l'impact

sur la précision de l'approximation de la dérive lorsque les noeuds sont éloignés. En effet, il est possible que deux noeuds ne communiquent jamais directement. Dans ce cas, le calcul de la dérive doit être réalisé en utilisant une ou des machines intermédiaires créant un chemin logique reliant ces deux noeuds. Dans cette section, l'objectif est de déterminer la perte de précision en fonction du nombre de sauts nécessaires afin de relier deux machines.

Le premier scénario de test, présenté à la section suivante, permet d'étudier l'impact de la fréquence des communications sur le traçage.

4.1 Impact selon la fréquence des communications

L'échange de messages est l'élément central pour la synchronisation du système de traçage réparti. Il est donc important de déterminer la fréquence idéale de communication pour obtenir une synchronisation d'événements précise. La figure 4.1 montre la structure du réseau utilisé. L'objectif est de mesurer la précision de la synchronisation des traces entre les noeuds 2 et 3. Le noeud_1 agit comme moniteur des deux autres machines. Chaque noeud communique avec celui-ci en envoyant un datagramme UDP toutes les secondes. Ce moniteur est actif pour chaque cas de tests pour des fins de validation des résultats. Les échanges de messages sont générés à l'aide d'un client-serveur TCP. Le client communique avec le serveur selon une fréquence fixée. Chaque cas de test est d'une durée de dix minutes et répété à dix reprises. Puisqu'un fort débit d'échange de messages engendre une charge élevée (voir la section 2.4), on pose comme hypothèse que la variabilité entre le temps d'aller et le temps de retour est plus important ayant comme conséquence de diminuer la précision de l'approximation de la dérive d'horloge (en effet, pour le calcul du décalage, on suppose que le temps d'aller est sensiblement le même que le temps de retour).



La trace du serveur UDP est utilisée seulement pour valider les résultats des traces 2 et 3.

Figure 4.1 Architecture du scénario de test

Le tableau 4.2 compare les marges d'erreur ($E_{ij} = |\frac{T_{aller}-T_{retour}}{2}|$, voir la figure 1.7) obtenus avec l'ensemble des cas de tests choisis en effectuant la moyenne des dix essais, selon un intervalle de confiance. Puisque le nombre d'échantillons des essais est grand, la distribution des E_{ij} suit approximativement une loi normale. L'intervalle de confiance est donc obtenu selon les valeurs du tableau 4.1 où $\bar{x}$ est la moyenne de l'ensemble des échantillons et $\sigma_{\bar{x}}$ est l'écart-type de ce même ensemble. L'intervalle de confiance sert en fait d'intervalle d'inclusion de données pour l'évaluation de la dérive. Il permet de retirer du calcul, les mesures éloignées de la moyenne ponctuellement pertubées pour diverses raisons (stabilité du réseau, temps de réponse, ...). Ces mesures sont donc soumises à un délai supplémentaire. Ainsi, avec un niveau de confiance de 90%, 10% des mesures sont retirées, prises à partir de celles les plus éloignées de la moyenne. La figure 4.2 montre la distribution

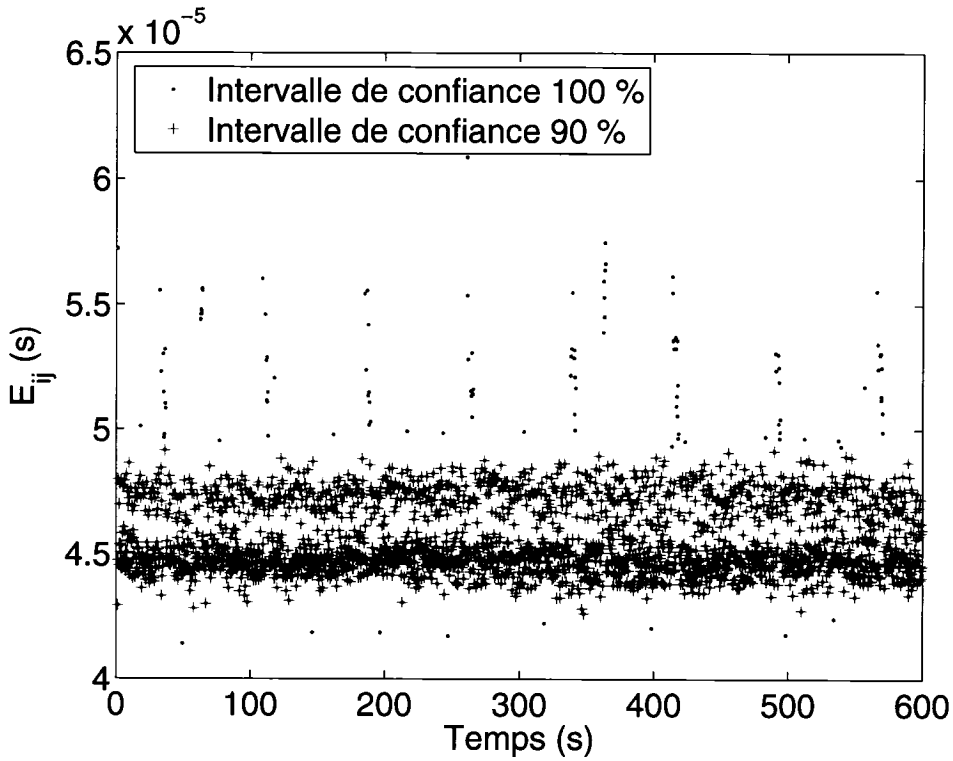
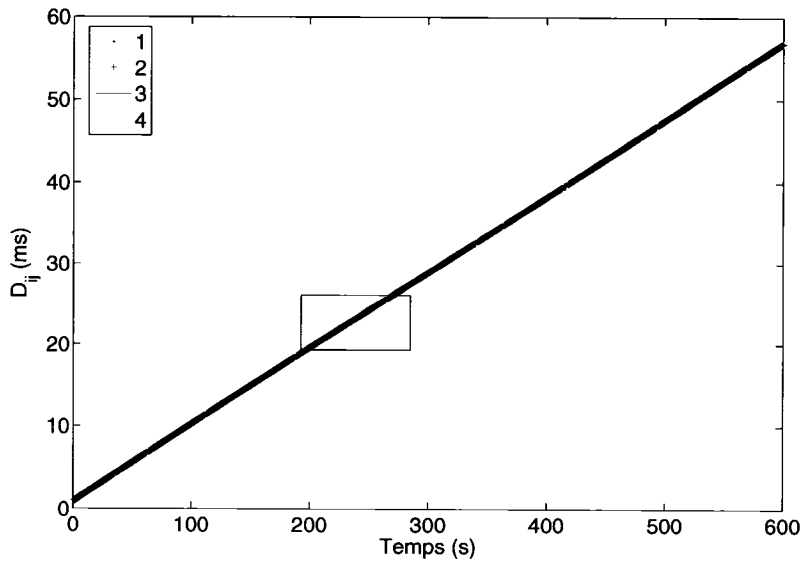


Figure 4.2 Marge d'erreur pour chaque aller-retour selon un intervalle de confiance de 90 et 100 %

de la marge d'erreur pour chaque aller-retour. On observe un nuage de points autour d'une moyenne avec quelques valeurs éloignées. Ces valeurs sont retirer du calcul en utilisant un intervalle de confiance de 90 %. Les figures 4.3 et 4.4 montrent l'effet du retrait de ces valeurs dans le calcul de l'approximation de la dérive d'horloge entre les deux ordinateurs. On remarque avec la figure 4.4(a) trois points éloignés de la droite. Ces points ont été retirés du calcul avec un intervalle de confiance de 90 %, ce qui a pour effet d'augmenter la précision de l'approximation. La figure 4.4(b) illustre une légère différence entre l'approximation de la dérive selon un intervalle de confiance 90 et 100 %.

Le premier test utilise la fréquence maximale i.e. le client TCP transmet la quantité

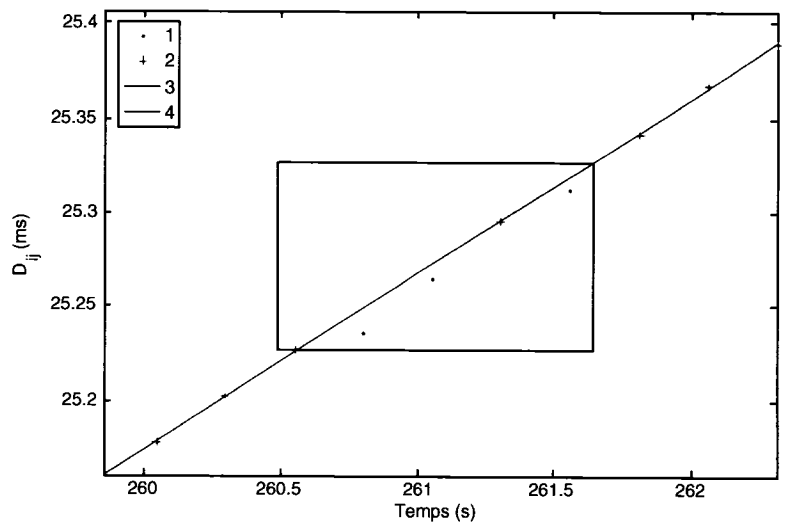


1. Décalage, D_{ij}, selon un intervalle de confiance de 100 %
2. Décalage, D_{ij}, selon un intervalle de confiance de 90 %
3. Approximation de la dérive selon un intervalle de confiance de 100 %
4. Approximation de la dérive selon un intervalle de confiance de 90 %

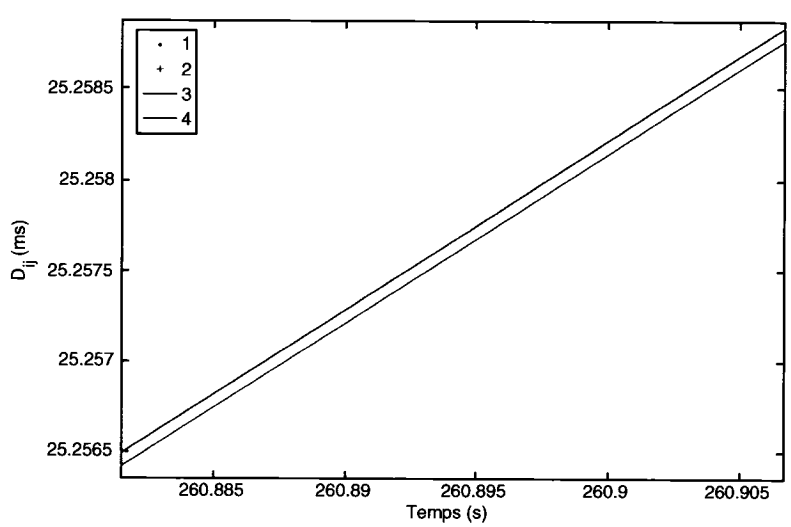
Figure 4.3 Comparaison de l'approximation de la dérive selon un intervalle de confiance de 90 et 100 %

maximale de messages qui lui est physiquement possible. Pour les autres tests, une période est désignée pour générer un cycle aller-retour d'échange de messages. Selon les résultats présentés au tableau 4.2, la marge d'erreur moyenne la plus faible est obtenue avec la plus petite période de communication. Toutefois, en moyenne 2,5 μs, environ, séparent la plus petite avec la plus grande marge d'erreur (une différence de l'ordre de 5%) qui est obtenue avec une période de 10 s et 20 s. La marge d'erreur moyenne tourne autour de 45 μs, ce qui est plausible compte tenu de la latence unidirectionnelle d'un lien *Fast Ethernet* qui est de 20 μs (voir le tableau 4.4 (Juhász, Charaf, 2004)) et du temps de réponse.

Le tableau 4.3 montre les résultats du calcul de la dérive pour chaque cas de



(a) Zoom du graphique 4.3



(b) Zoom du graphique 4.4(a)

1. Décalage, D_{ij}, selon un intervalle de confiance de 100 %
2. Décalage, D_{ij}, selon un intervalle de confiance de 90 %
3. Approximation de la dérive selon un intervalle de confiance de 100 %
4. Approximation de la dérive selon un intervalle de confiance de 90 %

Figure 4.4 Mise en évidence du rôle de l'intervalle de confiance

Tableau 4.1 Détermination de l'intervalle de confiance selon un niveau de confiance

Niveau de confiance	Intervalle de confiance
90%	$\bar{x} - 1,64\sigma_{\bar{x}} < \mu < \bar{x} + 1,64\sigma_{\bar{x}}$
95%	$\bar{x} - 1,96\sigma_{\bar{x}} < \mu < \bar{x} + 1,96\sigma_{\bar{x}}$
99%	$\bar{x} - 2,58\sigma_{\bar{x}} < \mu < \bar{x} + 2,58\sigma_{\bar{x}}$

$\bar{x}$: moyenne de l'ensemble des échantillons
$\sigma_{\bar{x}}$: écart-type de l'ensemble des échantillons

Tableau 4.2 Marges d'erreur pour chaque aller-retour des tests d'une durée de dix minutes avec différentes fréquences de communications

Période	Intervalle de confiance	Marge d'erreur E_{ij}		
		Moyenne (s)	Maximum (s)	Minimum (s)
≈0	100%	4.42E-005	1.76E-004	4.06E-005
	90%	4.39E-005	4.72E-005	4.12E-005
	95%	4.39E-005	4.78E-005	4.09E-005
	99%	4.39E-005	4.90E-005	4.06E-005
1s	100%	4.54E-005	5.70E-005	4.19E-005
	90%	4.52E-005	4.82E-005	4.25E-005
	95%	4.51E-005	4.86E-005	4.20E-005
	99%	4.52E-005	4.97E-005	4.19E-005
10s	100%	4.68E-005	6.25E-005	4.29E-005
	90%	4.65E-005	5.06E-005	4.33E-005
	95%	4.65E-005	5.10E-005	4.32E-005
	99%	4.66E-005	5.30E-005	4.29E-005
20s	100%	4.69E-005	5.59E-005	4.31E-005
	90%	4.65E-005	5.15E-005	4.32E-005
	95%	4.65E-005	5.15E-005	4.32E-005
	99%	4.66E-005	5.48E-005	4.31E-005
30s	100%	4.66E-005	6.25E-005	4.32E-005
	90%	4.61E-005	5.11E-005	4.33E-005
	95%	4.61E-005	5.11E-005	4.32E-005
	99%	4.63E-005	5.53E-005	4.32E-005
1m	100%	4.65E-005	5.77E-005	4.33E-005
	90%	4.58E-005	5.27E-005	4.33E-005
	95%	4.58E-005	5.27E-005	4.33E-005
	99%	4.60E-005	5.63E-005	4.33E-005

Tableau 4.3 Approximation de la dérive selon les tests d'une durée de dix minutes avec différentes fréquences de communications

Période	Intervalle de confiance	Régression linéaire			
		X_{ij} (s/s)	E_r (s)	$E_{X_{ij}}$ (s/s)	$E_{D^0_{ij}}$ (s)
≈0	100%	-2.07E-004	3.29E-006	2.46E-011	8.64E-009
	90%	-2.07E-004	2.82E-006	2.16E-011	7.57E-009
	95%	-2.07E-004	2.83E-006	2.16E-011	7.56E-009
	99%	-2.07E-004	2.84E-006	2.16E-011	7.58E-009
1s	100%	-2.07E-004	1.55E-006	3.65E-010	1.28E-007
	90%	-2.07E-004	1.01E-006	2.46E-010	8.63E-008
	95%	-2.07E-004	1.04E-006	2.51E-010	8.79E-008
	99%	-2.07E-004	1.08E-006	2.60E-010	9.09E-008
10s	100%	-2.07E-004	1.55E-006	1.13E-009	3.88E-007
	90%	-2.07E-004	1.03E-006	8.08E-010	2.83E-007
	95%	-2.07E-004	1.06E-006	8.11E-010	2.81E-007
	99%	-2.07E-004	1.15E-006	8.60E-010	2.99E-007
20s	100%	-2.07E-004	1.80E-006	1.82E-009	6.14E-007
	90%	-2.07E-004	1.24E-006	1.34E-009	4.66E-007
	95%	-2.07E-004	1.29E-006	1.37E-009	4.71E-007
	99%	-2.07E-004	1.45E-006	1.50E-009	5.15E-007
30s	100%	-2.07E-004	1.51E-006	1.84E-009	6.08E-007
	90%	-2.07E-004	8.46E-007	1.11E-009	3.71E-007
	95%	-2.07E-004	8.68E-007	1.12E-009	3.71E-007
	99%	-2.07E-004	1.15E-006	1.45E-009	4.87E-007
1m	100%	-2.07E-004	1.65E-006	2.74E-009	8.50E-007
	90%	-2.07E-004	1.46E-006	2.69E-009	8.74E-007
	95%	-2.07E-004	1.46E-006	2.69E-009	8.74E-007
	99%	-2.07E-004	1.49E-006	2.62E-009	8.38E-007

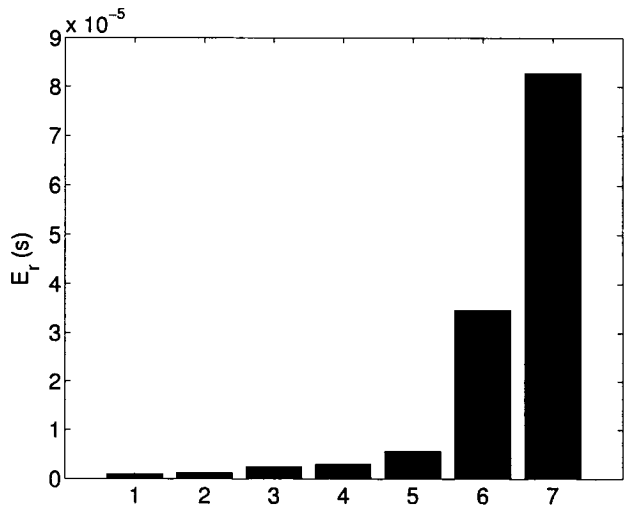
Tableau 4.4 Résumé des spécifications de diverses architectures réseau

	Fast Ethernet	**Gigabit Ethernet**	**SCI**	**ATM**	**Myrinet**
Structure réseau	bus	bus	commuté	commuté	commuté
Latence unidirectionnelle	20 μs	20 μs	5 μs	120 μs	5 μs
Bande passante	100 Mbps	1 Gbps	4 Gbps	155 Gbps	1.2 Gbps
Longueur du câble par lien	200 m	200 m	10 m	100 m	10 m

test. L'approximation de la dérive est réalisée à l'aide de la régression linéaire des moindres carrés à partir des équations 3.1 à 3.7. Ce calcul est effectué pour chaque intervalle de confiance. Ainsi, chaque valeur E_{ij} à l'extérieur de l'intervalle de confiance n'est pas considérée lors du calcul de la régression linéaire. En observant ce tableau, on s'aperçoit que l'écart-type résiduel (E_r) le plus élevé (environ 2.5 μs) se retrouve avec la période de $\approx$ 0 seconde. Dans les autres cas, l'écart-type résiduel est de l'ordre de la microseconde (une différence de l'ordre de 60%).

On peut donc conclure qu'un fort débit de communications dégrade quelque peu les résultats. Toutefois, un faible débit n'influence pas cette imprécision. En effet, on ne remarque aucune dégradation avec une période d'une minute entre deux cycles aller-retour de messages même si seulement dix mesures de décalage sont prises.

La section suivante tente d'identifier le degré de linéarité de la dérive d'horloge en étudiant son approximation selon différentes durées de traçage.



Test 1 : Test d'une durée de 10 minute
Test 2 : Test d'une durée de 30 minutes
Test 3 : Test d'une durée de 45 minutes
Test 4 : Test d'une durée d'une heure
Test 5 : Test d'une durée de 2 heures
Test 6 : Test d'une durée de 5 heures
Test 7 : Test d'une durée de 12 heures

Figure 4.5 Comparaison de l'écart-type résiduel pour différentes durées de traçage

4.2 Impact selon la durée du traçage

L'objectif de ce test est de déterminer quelle est la durée idéale d'un segment de traçage. Cette durée est un facteur très important qui permettra d'identifier le degré de fragmentation nécessaire pour la synchronisation d'événements de traces de longue durée. La même méthode que le scénario de test précédent est utilisée, et ce pour différentes durées avec un échange de messages toutes les secondes. Encore une fois, chaque test est effectué à dix reprises. On pose comme hypothèse que la dérive d'horloge est linéaire à moyen terme et une légère courbe à long terme.

Les résultats sont présentés dans les tableaux 4.5 et 4.6. On observe que les marges

Tableau 4.5 Marges d'erreur pour chaque aller-retour des tests d'une durée variable avec un échange de messages toutes les secondes

Durée	**Intervalle de confiance**	**Marge d'erreur E_{ij}**		
		Moyenne (s)	**Maximum (s)**	**Minimum (s)**
10m	100%	4.54E-005	5.77E-005	4.19E-005
	90%	4.52E-005	5.03E-005	4.25E-005
	95%	4.52E-005	5.03E-005	4.20E-005
	99%	4.52E-005	5.17E-005	4.19E-005
30m	100%	4.54E-005	5.91E-005	4.19E-005
	90%	4.53E-005	5.00E-005	4.24E-005
	95%	4.53E-005	5.07E-005	4.20E-005
	99%	4.53E-005	5.17E-005	4.19E-005
45m	100%	4.53E-005	7.53E-005	4.17E-005
	90%	4.52E-005	5.06E-005	4.26E-005
	95%	4.51E-005	5.11E-005	4.20E-005
	99%	4.52E-005	5.22E-005	4.17E-005
1h	100%	4.54E-005	1.25E-004	4.13E-005
	90%	4.53E-005	5.04E-005	4.23E-005
	95%	4.52E-005	5.11E-005	4.21E-005
	99%	4.53E-005	5.22E-005	4.13E-005
2h	100%	4.50E-005	6.17E-005	4.16E-005
	90%	4.48E-005	4.76E-005	4.25E-005
	95%	4.48E-005	4.81E-005	4.20E-005
	99%	4.48E-005	4.92E-005	4.16E-005
5h	100%	4.54E-005	9.20E-005	4.13E-005
	90%	4.53E-005	4.83E-005	4.20E-005
	95%	4.52E-005	4.88E-005	4.17E-005
	99%	4.52E-005	4.98E-005	4.16E-005
12h	100%	4.50E-005	1.46E-004	4.15E-005
	90%	4.48E-005	4.97E-005	4.20E-005
	95%	4.48E-005	5.02E-005	4.15E-005
	99%	4.48E-005	5.13E-005	4.15E-005

Tableau 4.6 Approximation de la dérive selon les tests d'une durée variable avec un échange de messages toutes les secondes

Durée	Intervalle de confiance	Régression linéaire			
		X_{ij} (s/s)	E_r (s)	$E_{X_{ij}}$ (s/s)	$E_{D^0_{ij}}$ (s)
10m	100%	1.13E-004	1.54E-006	3.64E-010	1.29E-007
	90%	1.13E-004	9.54E-007	2.31E-010	8.22E-008
	95%	1.13E-004	9.84E-007	2.37E-010	8.41E-008
	99%	1.13E-004	1.01E-006	2.43E-010	8.63E-008
30m	100%	1.13E-004	1.69E-006	7.69E-011	8.15E-008
	90%	1.13E-004	1.24E-006	5.78E-011	6.12E-008
	95%	1.13E-004	1.25E-006	5.80E-011	6.15E-008
	99%	1.13E-004	1.27E-006	5.88E-011	6.23E-008
45m	100%	1.13E-004	2.79E-006	6.90E-011	1.08E-007
	90%	1.13E-004	2.49E-006	6.32E-011	9.89E-008
	95%	1.13E-004	2.50E-006	6.30E-011	9.86E-008
	99%	1.13E-004	2.51E-006	6.30E-011	9.86E-008
1h	100%	1.13E-004	3.28E-006	5.26E-011	1.10E-007
	90%	1.13E-004	3.01E-006	4.97E-011	1.04E-007
	95%	1.13E-004	3.02E-006	4.95E-011	1.03E-007
	99%	1.13E-004	3.03E-006	4.95E-011	1.03E-007
2h	100%	1.13E-004	5.74E-006	3.26E-011	1.36E-007
	90%	1.13E-004	5.61E-006	3.28E-011	1.36E-007
	95%	1.13E-004	5.62E-006	3.25E-011	1.36E-007
	99%	1.13E-004	5.62E-006	3.24E-011	1.35E-007
5h	100%	1.13E-004	3.47E-005	4.98E-011	5.18E-007
	90%	1.13E-004	3.47E-005	5.11E-011	5.33E-007
	95%	1.13E-004	3.46E-005	5.08E-011	5.28E-007
	99%	1.13E-004	3.46E-005	5.06E-011	5.26E-007
12h*	100%	-2.07E-004	8.29E-005	3.20E-011	7.99E-007
	90%	-2.07E-004	8.28E-005	3.28E-011	8.17E-007
	95%	-2.07E-004	8.28E-005	3.26E-011	8.14E-007
	99%	-2.07E-004	8.28E-005	3.25E-011	8.10E-007

* La valeur de l'approximation de la dérive est différente pour cette durée car, il y a eu un redémarrage des ordinateurs.

d'erreur sont relativement similaires pour chaque durée. Ce qui est normal puisque la charge est identique dans chaque cas. De plus, on constate que l'approximation de la dérive (X_{ij}) est sensiblement la même dans chaque situation sauf pour la durée de 12 heures. Cette différence s'explique par le fait qu'il y a eu un redémarrage des ordinateurs pour ce test. Les mesures de temps étant prises à partir du TSC, ce comportement est tout à fait normal puisque la fréquence de cette horloge est approximée lors de la phase de démarrage de Linux. Par contre, une différence est notable par rapport à la précision de l'approximation. En effet, l'écart-type résiduel (E_r) est relativement le même pour une durée allant de 10 à 30 minutes et, elle augmente de façon progressive pour une durée du traçage de 45 minutes et plus. E_r passe d'environ 1 ou 2 μs à environ 5,6 μs pour une durée de 2 heures, et à environ 83 μs pour une durée de 12 heures. Cette augmentation croissante de l'écart-type résiduel (présentée par la figure 4.5) s'explique par la variabilité de la dérive dans le temps. Ainsi, pour une longue durée, la dérive n'est plus linéaire en fonction du temps mais, elle se définit plutôt par une légère courbe. Cette courbe étant fortement aplatie, il est possible de la représenter par plusieurs droites. Nous pouvons donc conclure en proposant de fragmenter le calcul de la dérive d'horloge en plusieurs segments de 30 minutes. Cette période de temps s'avère un bon compromis entre la précision obtenue et la charge de calcul nécessaire afin de synchroniser les événements. Il s'agit d'un choix conservateur puisqu'il est basé seulement sur l'écart-type résiduel, E_r. En effet, on remarque que pour les durées allant de 10 minutes à 5 heures, l'évaluation de la dérive d'horloge n'a pratiquement pas changée mais, seulement son incertitude.

La partie qui suit traite de la précision de l'évaluation de la dérive d'horloge d'un système soumis à des charges distinctes.

Tableau 4.7 Description des tests utilisés afin de faire varier la charge d'un système

Tests		Description
#	Nom	
1	*bzip2*	Script d'archivage des fichiers sources d'un noyau
2	*bzip2* + *make*	Script de compilation d'un noyau avec le script du test 1
3	*bzip2* + *make* + *swap*	Programme générant plusieurs permutations avec le script 1 (*bzip2*) et le script 2 (*make*)
4	*while(1)*	Programme exécutant une boucle infinie (*while(1)*)
5	*syscall*	Programme de création et destruction de processus infinies
6	*r/w*	Programme de lecture et écriture d'un gros fichier sur disque
7	-	Idem que 1 sauf qu'un seul noeud l'exécute, l'autre est au repos
8	-	Idem que 2 sauf qu'un seul noeud l'exécute, l'autre est au repos
9	-	Idem que 3 sauf qu'un seul noeud l'exécute, l'autre est au repos
10	-	Idem que 4 sauf qu'un seul noeud l'exécute, l'autre est au repos
11	-	Idem que 5 sauf qu'un seul noeud l'exécute, l'autre est au repos
12	-	Idem que 6 sauf qu'un seul noeud l'exécute, l'autre est au repos

Tableau 4.8 Utilisation des ressources pour les tests 1 à 6 à titre d'indicatif

Test	Temps CPU	Nombres d'appels système	Nombres d'accès disque
1	élevé	moyen	élevé
2	élevé	élevé	moyen
3	élevé	élevé	élevé
4	très élevé	nul	nul
5	très élevé	très élevé	nul
6	faible	très élevé	très élevé

4.3 Impact selon la charge (temps de réponse)

L'objectif de ce test est d'identifier l'impact de la synchronisation des traces selon la charge des noeuds. Pour ce faire, la même configuration est utilisée avec un échange de messages toutes les secondes pour une durée de 30 minutes. Le tableau 4.7

présente les différents tests utilisés pour faire varier la charge des noeuds. Les deux premiers tests exécutent des applications typiques sous Linux. Ainsi, le premier test génère une charge élevée au niveau du processeur et des accès disques. Ce test utilise un script qui archive, de façon continuelle, le code source d'un noyau de Linux avec l'application *bzip2*. Le second test rajoute une charge, au test précédent, avec un script compilant perpétuellement un noyau de Linux. Le test suivant ajoute au deuxième test en exécutant en plus un programme effectuant des calculs sur de grands tableaux entraînant le système à continuellement effectuer des permutations en mémoire. Dans ce test, le système sera dans un état de saturation complète. Les trois tests suivants isolent un type de charge à la fois afin d'identifier ce qui influence le plus l'imprécision de l'approximation de la dérive. Ainsi, le quatrième test génère une utilisation continue du processeur grâce à un programme exécutant une boucle infinie. Le cinquième test génère sans cesse des appels système déclenchant ainsi un nombre important de réordonnancements. Les appels système sont produits en créant (*fork*) et détruisant (*kill*) sans cesse des processus. Le sixième test engendre énormément d'interruptions et d'utilisations de ressources en réalisant des lectures et écritures sur disques. Les tests 7 à 12 correspondent respectivement aux tests 1 à 6 sauf qu'un seul noeud les exécute, l'autre noeud étudié étant sans charge.

Commençons par analyser les résultats pour les tests 1 à 6 présentés dans les tableaux 4.9 et 4.11. On remarque que la marge d'erreur E_{ij} est semblable pour un système en attente (voir tableau 4.5) qu'un système ayant une pleine charge au processeur (test 4, boucle *while(1)*). Toutefois, l'écart-type résiduel dans cette situation est supérieur passant de 1.5 μs, pour un système inactif, à 3.7 μs pour un système en pleine charge au processeur. Cette hausse peut s'expliquer par l'augmentation du temps de réponse du processeur. La figure 4.6 illustre ce fait en comparant le temps de réponse d'un système en état de repos et, en état de charge au processeur.

L'évaluation du temps de réponse d'un système est obtenue à l'aide du programme

Tableau 4.9 Marges d'erreur pour chaque aller-retour des tests de variation de charge, d'une durée de 30 minutes avec un échange de messages toutes les secondes

Test	Intervalle de confiance	Marge d'erreur E_{ij}		
		Moyenne (s)	Maximum (s)	Minimum (s)
1 *bzip2*	100%	6.70E-05	1.41E-04	4.27E-05
	90%	6.69E-05	7.67E-05	5.61E-05
	95%	6.69E-05	7.84E-05	5.43E-05
	99%	6.69E-05	8.18E-05	5.10E-05
2 *bzip2 + make*	100%	6.77E-05	1.25E-04	4.35E-05
	90%	6.65E-05	8.40E-05	4.93E-05
	95%	6.67E-05	8.78E-05	4.62E-05
	99%	6.73E-05	9.44E-05	4.35E-05
3 *bzip2 + make + swap*	100%	6.36E-05	1.64E-04	4.36E-05
	90%	6.26E-05	8.10E-05	4.36E-05
	95%	6.26E-05	8.42E-05	4.36E-05
	99%	6.28E-05	9.06E-05	4.36E-05
4 *while(1)*	100%	4.59E-05	8.38E-05	4.13E-05
	90%	4.57E-05	5.12E-05	4.26E-05
	95%	4.57E-05	5.18E-05	4.20E-05
	99%	4.56E-05	5.32E-05	4.13E-05
5 *syscall*	100%	5.47E-05	1.58E-04	4.60E-05
	90%	5.39E-05	6.62E-05	4.61E-05
	95%	5.39E-05	6.84E-05	4.61E-05
	99%	5.40E-05	7.25E-05	4.60E-05
6 *r/w*	100%	7.29E-05	1.84E-04	4.56E-05
	90%	7.17E-05	1.01E-04	4.56E-05
	95%	7.17E-05	1.05E-04	4.56E-05
	99%	7.18E-05	1.13E-04	4.56E-05

Tableau 4.10 Marges d'erreur pour chaque aller-retour des tests de variation de charge (asymétrique), d'une durée de 30 minutes avec un échange de messages toutes les secondes

Test	**Intervalle de confiance**	**Marge d'erreur E_{ij}**		
		Moyenne (s)	**Maximum (s)**	**Minimum (s)**
7 *bzip2*	100%	5.34E-05	9.27E-05	4.28E-05
	90%	5.32E-05	5.93E-05	4.28E-05
	95%	5.32E-05	6.02E-05	4.28E-05
	99%	5.32E-05	6.21E-05	4.28E-05
8 *bzip2 + make*	100%	5.16E-05	1.40E-04	4.26E-05
	90%	5.08E-05	7.19E-05	4.26E-05
	95%	5.08E-05	7.47E-05	4.26E-05
	99%	5.09E-05	8.00E-05	4.26E-05
9 *bzip2 + make + swap*	100%	5.24E-05	1.64E-04	4.24E-05
	90%	5.14E-05	6.59E-05	4.24E-05
	95%	5.14E-05	6.91E-05	4.24E-05
	99%	5.16E-05	7.45E-05	4.24E-05
10 *while(1)*	100%	4.61E-05	1.43E-04	4.16E-05
	90%	4.59E-05	5.16E-05	4.18E-05
	95%	4.59E-05	5.24E-05	4.18E-05
	99%	4.59E-05	5.36E-05	4.16E-05
11 *syscall*	100%	4.89E-05	8.10E-05	4.19E-05
	90%	4.82E-05	6.05E-05	4.19E-05
	95%	4.82E-05	6.24E-05	4.19E-05
	99%	4.82E-05	6.53E-05	4.19E-05
12 *r/w*	100%	5.36E-05	1.86E-04	4.27E-05
	90%	5.27E-05	7.00E-05	4.27E-05
	95%	5.27E-05	7.08E-05	4.27E-05
	99%	5.28E-05	7.99E-05	4.27E-05

Tableau 4.11 Approximation de la dérive selon les tests de variation de charge, d'une durée de 30 minutes avec un échange de messages toutes les secondes

Test	Intervalle de confiance	Régression linéaire			
		X_{ij} (s/s)	E_r (s)	$E_{X_{ij}}$ (s/s)	$E_{D^0_{ij}}$ (s)
1 *bzip2*	100%	9.57E-05	2.50E-05	1.13E-09	1.18E-06
	90%	9.57E-05	2.44E-05	1.16E-09	1.21E-06
	95%	9.57E-05	2.45E-05	1.14E-09	1.19E-06
	99%	9.57E-05	2.46E-05	1.13E-09	1.17E-06
2 *bzip2 + make*	100%	9.55E-05	8.70E-06	3.94E-10	4.10E-07
	90%	9.55E-05	6.99E-06	3.33E-10	3.46E-07
	95%	9.55E-05	7.23E-06	3.39E-10	3.53E-07
	99%	9.55E-05	8.05E-06	3.70E-10	3.85E-07
3 *bzip2 + make + swap*	100%	9.54E-05	2.78E-05	1.26E-09	1.31E-06
	90%	9.54E-05	2.69E-05	1.27E-09	1.31E-06
	95%	9.54E-05	2.70E-05	1.26E-09	1.30E-06
	99%	9.54E-05	2.72E-05	1.25E-09	1.30E-06
4 *while(1)*	100%	9.47E-05	4.02E-06	1.82E-10	1.89E-07
	90%	9.47E-05	3.72E-06	1.75E-10	1.82E-07
	95%	9.47E-05	3.74E-06	1.75E-10	1.81E-07
	99%	9.47E-05	3.76E-06	1.74E-10	1.81E-07
5 *syscall*	100%	9.50E-05	9.99E-06	4.53E-10	4.71E-07
	90%	9.50E-05	9.09E-06	4.25E-10	4.42E-07
	95%	9.50E-05	9.13E-06	4.25E-10	4.41E-07
	99%	9.50E-05	9.22E-06	4.26E-10	4.42E-07
6 *r/w*	100%	9.50E-05	1.21E-05	5.49E-10	5.71E-07
	90%	9.50E-05	7.62E-06	3.56E-10	3.68E-07
	95%	9.50E-05	7.88E-06	3.64E-10	3.77E-07
	99%	9.50E-05	8.24E-06	3.78E-10	3.92E-07

Tableau 4.12 Approximation de la dérive selon les tests de variation de charge (asymétrique), d'une durée de 30 minutes avec un échange de messages toutes les secondes

Test	Intervalle de confiance	Régression linéaire			
		X_{ij} (s/s)	E_r (s)	$E_{X_{ij}}$ (s/s)	$E_{D^0_{ij}}$ (s)
7 *bzip2*	100%	9.71E-05	9.39E-04	4.26E-08	4.42E-05
	90%	9.71E-05	3.37E-05	1.60E-09	1.67E-06
	95%	9.71E-05	3.37E-05	1.58E-09	1.64E-06
	99%	9.71E-05	3.39E-05	1.55E-09	1.62E-06
8 *bzip2 + make*	100%	9.70E-05	2.13E-03	9.68E-08	1.01E-04
	90%	9.68E-05	6.06E-06	2.86E-10	2.95E-07
	95%	9.68E-05	6.08E-06	2.82E-10	2.94E-07
	99%	9.69E-05	1.17E-03	5.43E-08	5.57E-05
9 *bzip2 + make + swap*	100%	9.65E-05	6.32E-03	2.86E-07	2.98E-04
	90%	9.60E-05	2.42E-05	1.13E-09	1.18E-06
	95%	9.61E-05	1.27E-03	6.00E-08	6.15E-05
	99%	9.63E-05	3.83E-03	1.76E-07	1.82E-04
10 *while(1)*	100%	9.59E-05	1.39E-03	6.29E-08	6.54E-05
	90%	9.58E-05	4.84E-06	2.27E-10	2.36E-07
	95%	9.58E-05	4.86E-06	2.26E-10	2.35E-07
	99%	9.58E-05	4.89E-06	2.26E-10	2.35E-07
11 *syscall*	100%	9.65E-05	1.63E-05	7.39E-10	7.69E-07
	90%	9.65E-05	1.56E-05	7.31E-10	7.50E-07
	95%	9.65E-05	1.57E-05	7.30E-10	7.48E-07
	99%	9.65E-05	1.57E-05	7.29E-10	7.47E-07
12 *r/w*	100%	9.54E-05	2.03E-05	9.20E-10	9.56E-07
	90%	9.54E-05	1.70E-05	7.79E-10	8.08E-07
	95%	9.54E-05	1.70E-05	7.79E-10	8.09E-07
	99%	9.54E-05	1.72E-05	7.88E-10	8.18E-07

Realfeel2 écrit par (Morton, 2002) à partir de la version originale *Realfeel* conçue par (Hahn, 2001). Ce programme s'exécute dans l'espace usager. Il programme l'horloge RTC (voir la section 2.1) afin de générer périodiquement des interruptions. Puisque la fréquence de l'horloge est connue (elle est programmable), il est possible d'évaluer le temps de réponse en effectuant la différence entre la période idéale (connue) et la période réelle (l'instant où l'application est informée de l'interruption). Les mesures de temps sont effectuées à partir du TSC (sa fréquence est interpolée par ce programme) pour un million d'échantillons.

Pour les 5 autres tests, un accroissement de la marge d'erreur est noté. Pour les tests 1, 2 et 3, on observe une augmentation d'environ 20 μs par rapport à un système inactif, et de 10 μs et 25 μs pour les tests 5 et 6 respectivement. Ainsi, lire et écrire de façon intensive sur un disque dur sont les traitements qui altèrent le plus la variabilité des communications. Ce qui est tout à fait normal puisque ces traitements produisent énormément d'interruptions interférant avec celles générées lors des communications réseau. La même explication est valable pour les tests 1, 2 et 3. L'impact est plus faible puisque le processeur partage plusieurs tâches différentes amenant un usage plus modéré du disque dur. Pour ce qui est du test 5, l'impact est plus faible puisqu'un appel système correspond en réalité à une exception. Les exceptions sont des interruptions synchrones risquant d'interrompre le traitement des paquets réseau. Toutefois, la routine d'interruption dédiée à cet appel système est tellement simple que son traitement est beaucoup plus court que celle associée à l'accès d'un disque dur (de même que celle pour le traitement réseau). L'impact du traitement des appels système se limite pratiquement seulement au temps de changement de contexte qui est constant pour les noyaux 2.6 (complexité $\mathcal{O}(1)$).

La variation de la marge d'erreur E_{ij} ne se reflète pas directement dans l'écart-type résiduel E_r. D'autres facteurs doivent être considérés telle que la variation du temps de réponse du processeur. On s'aperçoit qu'il est possible d'obtenir un écart-type

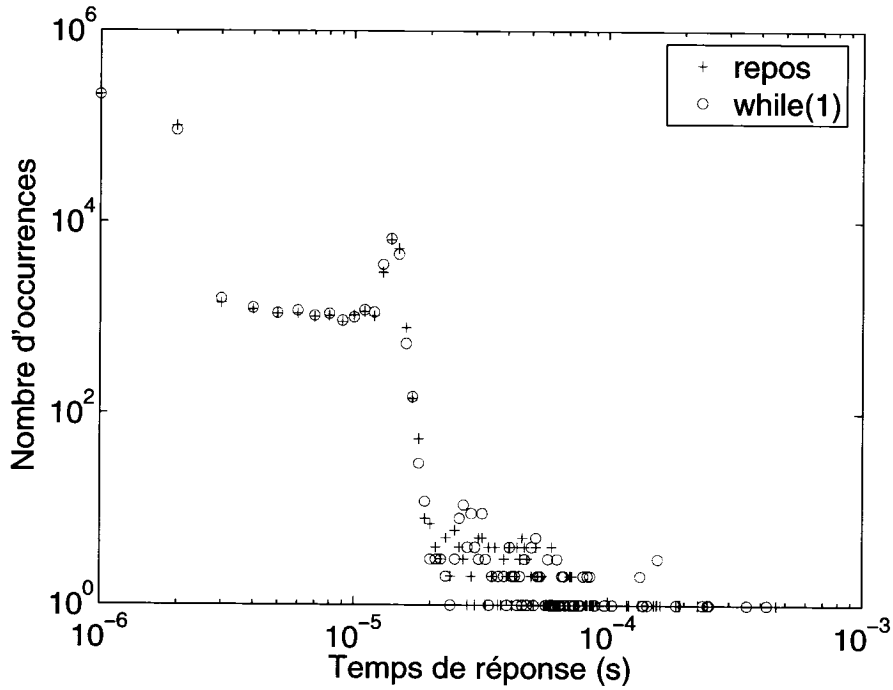


Figure 4.6 Temps de réponse au repos par rapport à celui sous une charge au processeur (test 4 : *while(1)*)

résiduel plus faible avec une marge d'erreur plus importante. Ce fait se présente dans les tests 5 et 6. Ainsi, le test générant un nombre intensif d'appels système présente une augmentation de 10 μs de la marge d'erreur, par rapport un système inactif, et avec un écart-type résiduel d'environ 9 μs. Cependant, on note, pour le test de lecture/écriture intensives, une augmentation de la marge d'erreur plus importante, environ 25 μs mais, avec un plus faible écart-type résiduel, d'environ 8 μs. Dans ces cas, le temps de réponse est élevé mais, il reste constant tout au long du traçage. C'est cette constance qui explique cet écart. Le test 5 génère énormément de courtes interruptions avec un usage intensif du processeur tandis que le test 6 produit copieusement de plus longues interruptions avec un usage faible du processeur. En effet, on observe avec la figure 4.7 que la manipulation du disque dur engendre un temps de réponse du système plus lent par rapport à la

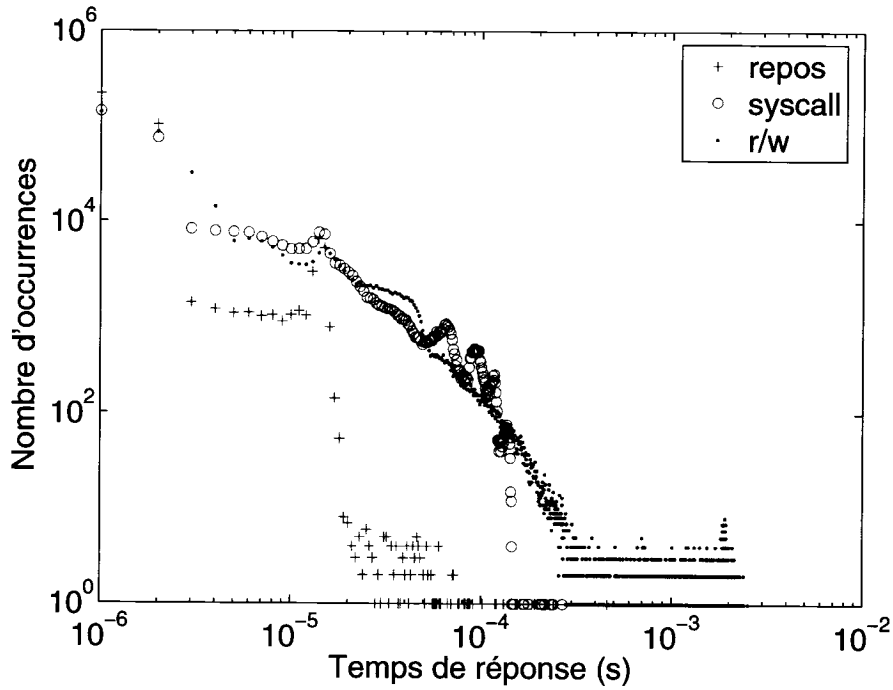


Figure 4.7 Temps de réponse au repos par rapport à celui sous une charge d'appels système (test 5 : *syscall*) et à une charge de lectures et écritures sur disque (test 6 : *r/w*)

manipulation d'appels système.

Enfin, les tests 1, 2 et 3 présentent des marges d'erreur similaires mais avec des écarts-types résiduels E_r différents. Le test 3 obtient sans surprise l'E_r le plus élevé. Dans ce test, le système est complètement saturé combinant une charge élevée au processeur et un nombre élevé de permutations en mémoire. Cette saturation du système entraîne donc une plus grande imprécision dans les mesures de temps des événements recueillis durant le traçage. Le test 1 engendre une saturation légèrement inférieure au test 3 en combinant une charge élevée au processeur avec un nombre important d'accès disque. On constate effectivement un E_r du test 1 légèrement inférieur au test 3. Pour ce qui est du test 2, on remarque que l'E_r est légèrement inférieur au test 6 et nettement inférieur aux test 1 et 3. Normalement,

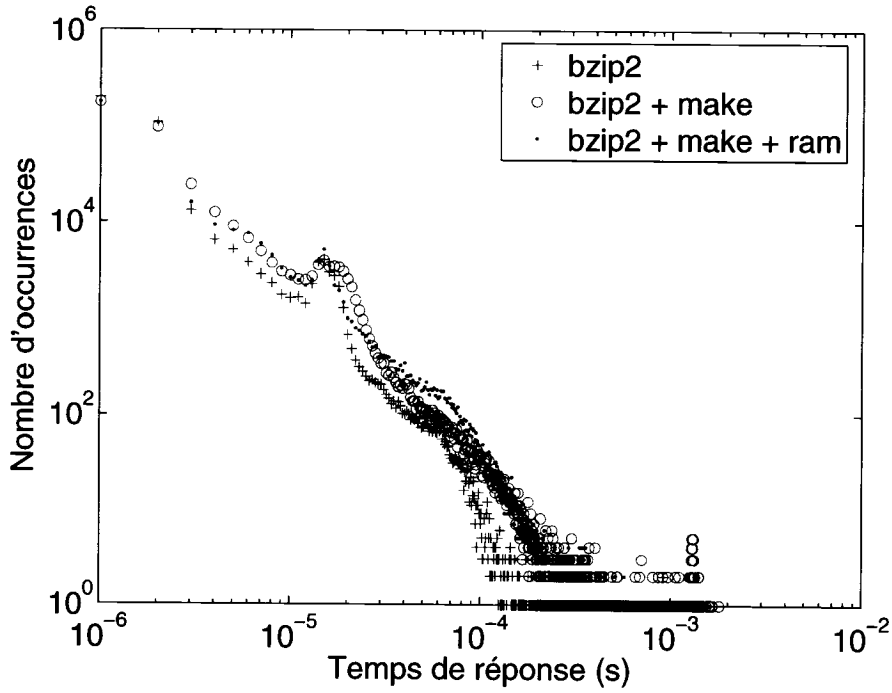


Figure 4.8 Temps de réponse pour les tests 1 (*bzip2*), 2 (*bzip2 + make*) et 3 (*bzip2 + make + ram*)

on aurait dû s'attendre à des résultats similaires entre les tests 1, 2 et 3. En effet, on remarque, avec la figure 4.8, que les temps de réponse sont relativement similaires pour les trois tests. Toutefois, le test 2 obtient un E_r plus faible puisque le temps de réponse semble être moins variable que les tests 1 et 3.

En résumé, la variabilité du temps de transmission et du temps de réponse sont les facteurs influençant la précision de l'approximation de la dérive. En effet, la charge sur le processeur entraîne une variation du temps de réponse. Évidemment, un système saturé a un temps de réponse lent vis-à-vis la précision des mesures de temps des événements, mais ce temps de réponse est énormément variable. En d'autres mots, un temps de réponse variable induit beaucoup plus d'imprécision dans l'évaluation de la dérive qu'un temps de réponse lent mais, constant.

Passons maintenant à l'analyse des tests 7 à 12 (voir les tableaux 4.10 et 4.12) où une machine est inactive et l'autre, soumise à une charge. On constate que généralement une baisse de 10 μs de la marge d'erreur E_{ij}, sauf pour le test 10 où la marge d'erreur reste inchangée, par rapport aux résultats des tests 1 à 6. Ces résultats étaient prévisibles puisqu'une machine a un temps de réponse rapide et l'autre, long. Autrement dit, l'augmentation de la marge d'erreur causée par une augmentation de la charge (temps de réponse) par rapport à une machine inactive est diminuée d'environ 50 % comparativement aux tests 1 à 6. Les résultats des marges d'erreur du test 4 et 10 sont similaires puisque la constance du temps de réponse s'apparente à celui d'un système inactif. Cependant, le test 10 présente une légère augmentation de l'écart-type résiduel. Ce fait s'explique par l'augmentation du temps de réponse seulement sur un processeur. Ainsi, selon cette inégalité des temps de réponse, les écarts-types résiduels sont plus important dans l'approximation de la dérive d'horloge pour les tests 7 à 12 sauf pour les tests 8 et 9 qui connaissent une légère baisse. Cela signifie donc, que la variabilité du temps de réponse n'a pas augmenté pour ces deux tests et que la diminution de la marge d'erreur explique cette baisse de l'écart-type résiduel. En résumé, lorsque les noeuds sont soumis à des charges différentes, l'impact sur la précision de l'évaluation de la dérive est plus grande puisque la différence entre le temps d'aller et le temps de retour n'est plus approximativement nulle parce qu'une machine est plus rapide que l'autre à traiter les événements réseau.

La prochaine section examine l'impact de la précision de l'évaluation de la dérive d'horloge en fonction du nombre de sauts présent dans le chemin reliant deux noeuds.

4.4 Impact selon la distance

Pour pouvoir étudier l'impact de la distance (nombre de sauts) sur la précision de l'approximation de la dérive, une grappe de calcul est utilisée et décrite à la figure 4.9 et au tableau 4.13. Ainsi, neuf noeuds sont à notre disposition pour ce test (le serveur étant indisponible). Chaque noeud est muni de deux processeurs et ne possède aucun disque dur, ce qui nous limite dans la taille que peuvent avoir les traces. Chaque noeud peut communiquer directement avec ses voisins grâce à deux commutateurs. Tout comme les tests précédents, un noeud (le noeud_1) est désigné comme moniteur UDP avec lequel tous les autres échangent un message toutes les secondes (afin de valider les résultats). Durant le traçage, des clients/serveurs TCP sont configurés pour générer des cycles aller-retour de messages selon une fréquence fixée entre tous les noeuds. En d'autres mots, tous les noeuds communiquent entre eux.

Tableau 4.13 Spécifications de la grappe de calcul[1]

Noeuds	Serveur x330 IBM
Processeurs	2 P3 (866 MHz)
Mémoire vive	512 Mo
Interface réseau (fichiers)	Ethernet 10/100 (Intel ePro 100)
Interface réseau (Communication)	Ethernet 10/100 (Intel ePro 100)

La figure 4.10 illustre les chemins utilisés afin d'évaluer la précision des mesures en fonction du nombre de sauts. Puisque chaque noeud communique avec l'ensemble des noeuds, il est possible de comparer les résultats obtenus à partir d'un lien direct avec un lien indirect passant par plusieurs noeuds. Le lien indirect correspond en fait

[1] `http://www.polymtl.ca/drap/infras.htm#skjellum`

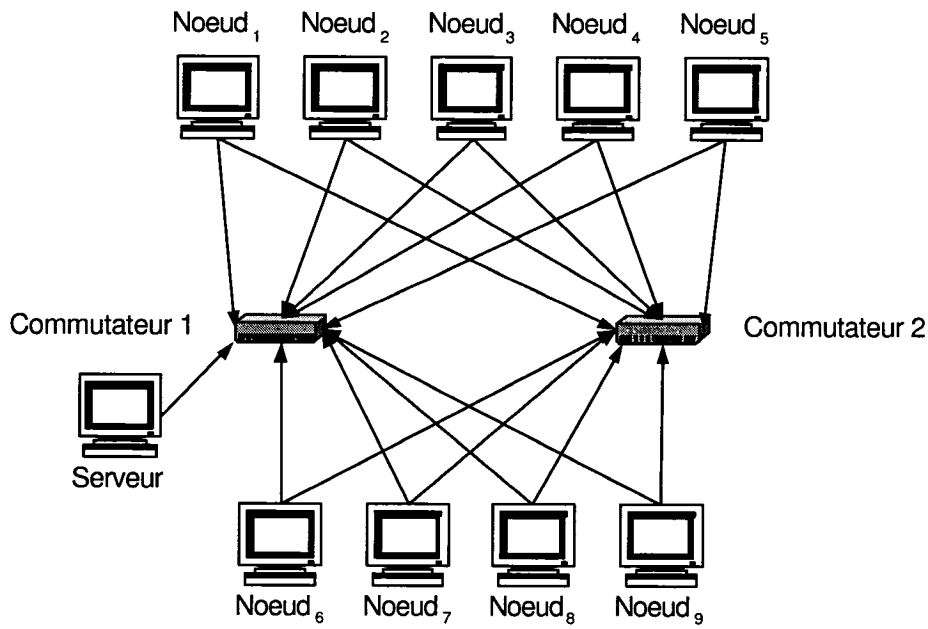


Figure 4.9 Architecture réseau de la grappe de calcul

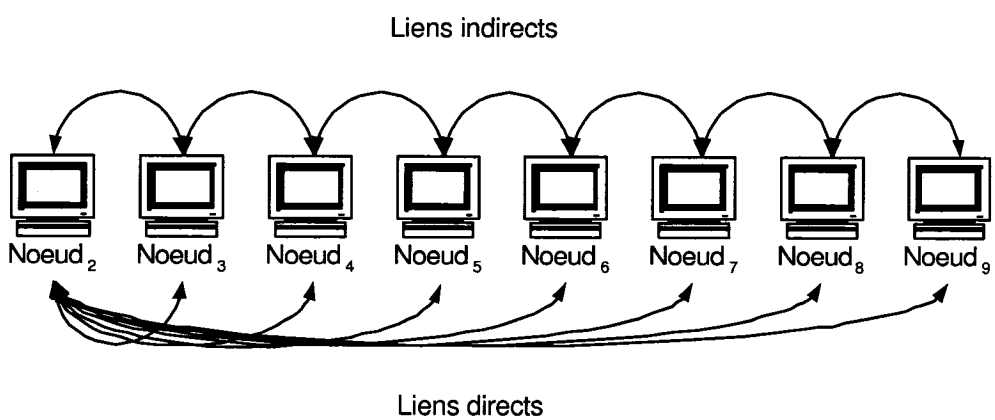


Figure 4.10 Chemins directs et indirects entre le noeud$_2$ et les autres noeuds

à un chemin virtuel entre deux noeuds dans le cas où ils ne communiqueraient pas directement. Par exemple, la dérive entre le noeud$_2$ et le noeud$_9$ est calculée à partir de deux chemins différents. Le premier correspond au lien direct, noeud$_2$-noeud$_9$, et le second, au lien indirect, noeud$_2$-noeud$_3$-noeud$_4$...-noeud$_9$, nécessitant ainsi sept sauts. Tous les chemins indirects sont conçus de la même façon en commençant par le noeud source vers le noeud de destination en parcourant de façon croissante les noeuds (selon leur numéro). L'équation suivante présente le calcul de la dérive indirect entre le noeud$_2$ et le noeud$_4$:

$$D_{2,4} = X_{2,4} * T + D^0_{2,4} \tag{4.1}$$

$$X_{2,4} * T + D^0_{2,4} = X_{2,3} * T + D^0_{2,3} + X_{3,4} * T + D^0_{3,4} \tag{4.2}$$

L'écart-type résiduel (E_r) correspond à la somme de chaque E_r obtenu pour les noeuds intermédiaires.

Le tableau 4.14 compare les résultats obtenus par les liens directs vis-à-vis des liens indirects et le tableau 4.15 présente les résultats obtenus dans chaque saut du chemin indirect. Chaque résultat correspond à la moyenne de dix tests d'une durée de 30 minutes avec un cycle aller-retour d'échange de messages toutes les secondes. Une augmentation, non négligeable, de l'imprécision est notée quant à l'évaluation des paramètres X_{ij} et D^0_{ij} présentés au tableau 4.16. La figure 4.11 illustre la progression de l'imprécision en fonction du nombre de sauts du chemin indirect. Des pentes abruptes sont présentes pour 4 et 5 sauts dans la figure 4.11(a) et, pour 2 et 4 sauts dans la figure 4.11(b). Ces points correspondent aux liens possédant les plus grandes imprécisions (E_r). Cette observation peut être faite à partir du tableau 4.16 pour les liens noeud$_3$->noeud$_4$, noeud$_5$->noeud$_6$ et noeud$_6$->noeud$_7$

Tableau 4.14 Approximation de la dérive selon les tests de variation de distance, d'une durée de 30 minutes avec un échange de messages toutes les secondes

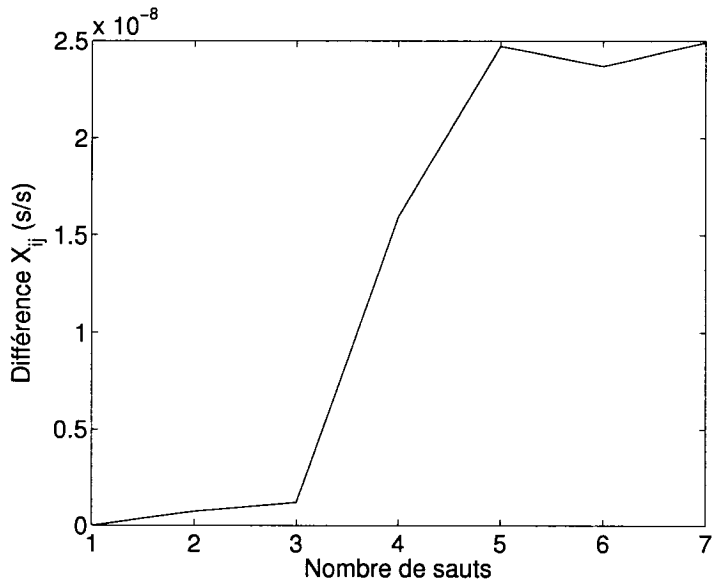
Noeud		Nbre de sauts	Intervalle de confiance	Régression linéaire			
				Lien direct		Lien indirect	
src	dest.			X_{ij} (s/s)	E_r (s)	X_{ij} (s/s)	E_r (s)
02	03	1	100%	-4.85E-06	2.74E-06	-	-
			90%	-4.85E-06	6.72E-07		
			95%	-4.85E-06	6.98E-07		
			99%	-4.85E-06	7.35E-07		
02	04	2	100%	1.44E-05	2.97E-06	1.44E-05	5.84E-06
			90%	1.44E-05	1.15E-06	1.44E-05	1.83E-06
			95%	1.44E-05	1.17E-06	1.44E-05	1.89E-06
			99%	1.44E-05	1.21E-06	1.44E-05	1.96E-06
02	05	3	100%	1.25E-06	2.86E-06	1.25E-06	7.93E-06
			90%	1.25E-06	8.35E-07	1.25E-06	2.61E-06
			95%	1.25E-06	8.51E-07	1.25E-06	2.69E-06
			99%	1.25E-06	8.65E-07	1.25E-06	2.79E-06
02	06	4	100%	1.67E-04	2.26E-06	1.67E-04	2.13E-05
			90%	1.67E-04	1.51E-06	1.67E-04	9.99E-06
			95%	1.67E-04	1.53E-06	1.67E-04	1.28E-05
			99%	1.67E-04	1.55E-06	1.67E-04	1.56E-05
02	07	5	100%	3.09E-06	2.69E-06	3.12E-06	3.57E-05
			90%	3.09E-06	7.01E-07	3.10E-06	1.18E-05
			95%	3.09E-06	7.18E-07	3.10E-06	1.46E-05
			99%	3.09E-06	7.50E-07	3.09E-06	1.73E-05
02	08	6	100%	6.10E-06	1.22E-06	6.13E-06	3.76E-05
			90%	6.10E-06	6.64E-07	6.11E-06	1.26E-05
			95%	6.10E-06	7.04E-07	6.11E-06	1.54E-05
			99%	6.10E-06	7.61E-07	6.10E-06	1.83E-05
02	09	7	100%	-4.155E-05	1.26E-06	-4.151E-05	4.27E-05
			90%	-4.155E-05	7.81E-07	-4.154E-05	1.34E-05
			95%	-4.155E-05	7.98E-07	-4.155E-05	1.63E-05
			99%	-4.155E-05	8.35E-07	-4.156E-05	1.91E-05

Tableau 4.15 Approximation de la dérive pour les liens directs faisant partie du chemin entre le noeud$_2$ et le noeud$_9$, traçage d'une durée de 30 minutes avec un échange de messages toutes les secondes

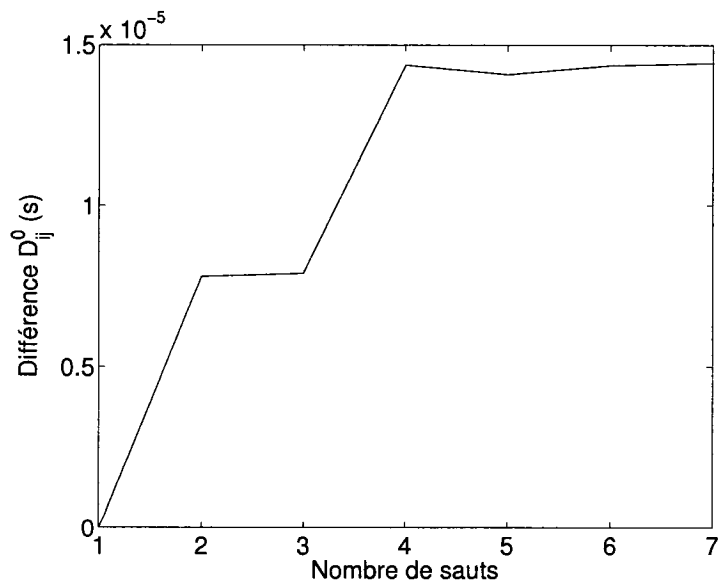
Noeud		**Intervalle de confiance**	**Régression linéaire**	
			Lien direct	
src	**dest.**		X_{ij} (s/s)	E_r (s)
02	03	100%	-4.85E-06	2.74E-06
		90%	-4.85E-06	6.72E-07
		95%	-4.85E-06	6.98E-07
		99%	-4.85E-06	7.35E-07
03	04	100%	1.93E-05	3.09E-06
		90%	1.93E-05	1.16E-06
		95%	1.93E-05	1.19E-06
		99%	1.93E-05	1.22E-06
04	05	100%	-1.32E-05	2.10E-06
		90%	-1.32E-05	7.75E-07
		95%	-1.32E-05	8.02E-07
		99%	-1.32E-05	8.30E-07
05	06	100%	1.66E-04	1.33E-05
		90%	1.66E-04	7.38E-06
		95%	1.66E-04	1.01E-05
		99%	1.66E-04	1.28E-05
06	07	100%	-1.64E-04	1.44E-05
		90%	-1.64E-04	1.79E-06
		95%	-1.64E-04	1.79E-06
		99%	-1.64E-04	1.79E-06
07	08	100%	3.01E-06	1.94E-06
		90%	3.01E-06	8.12E-07
		95%	3.01E-06	8.64E-07
		99%	3.01E-06	9.04E-07
08	09	100%	-4.76E-05	5.09E-06
		90%	-4.77E-05	8.07E-07
		95%	-4.77E-05	8.19E-07
		99%	-4.77E-05	8.41E-07

correspondant respectivement au nombre de sauts 2, 4 et 5. Puisque les erreurs sont cumulatives, les liens moins fiables (moins précis) influencent énormément la précision obtenue. Il est donc important de choisir le meilleur chemin entre deux noeuds afin d'évaluer la dérive d'horloge. Somme toute, la précision obtenue reste excellente. Avec sept sauts, l'écart-type résiduel E_r reste inférieur à 20 μs pour un système inactif. Il s'agit d'une dégradation similaire à celle d'un lien direct dont les noeuds sont soumis à une forte charge.

En conclusion, la figure 4.12 compare les écarts-types résiduels des quatre premières sections du chapitre. On remarque que le débit des communications a un faible d'impact quant à la précision de l'évaluation de la dérive d'horloge. Les mesures de temps sont beaucoup plus sensibles à la charge d'un système et donc à la variabilité du temps de réponse. En effet, les mesures de temps prises lors des événements de réception et de transmission de paquets sont effectuées au niveau des *softirqs*. Ainsi, ces mesures peuvent être interrompues en tout temps par une interruption matérielle ou une exception (e.g. un appel système). En plus, les *softirqs* doivent partager l'usage du processeur avec les autres processus s'exécutant sous le système. Aussi, lorsque les noeuds sont soumis à des charges différentes, l'évaluation de la dérive peut en souffrir puisqu'une inégalité entre le temps d'aller et temps de retour est induite. De plus, on constate qu'il est nécessaire de segmenter le calcul de la dérive lorsque le traçage s'effectue sur une longue période de temps, puisqu'elle se caractérise par une légère courbe dans ce cas. Il suffit de fragmenter ce calcul en segments de 30 minutes. Enfin, le choix du chemin utilisé pour évaluer la dérive d'horloge est aussi très important car les imprécisions de chaque lien s'accumulent dans le cas où le chemin présente plusieurs sauts. Dans le cas idéal, il faudrait que lors du traçage les noeuds soient soumis à une charge relativement constante et similaire, et que les chemins reliant chaque noeud utilisent le moins de sauts possibles.



(a) Progression de la différence de X_{ij} en fonction du nombre de sauts (intervalle de confiance de 90%)

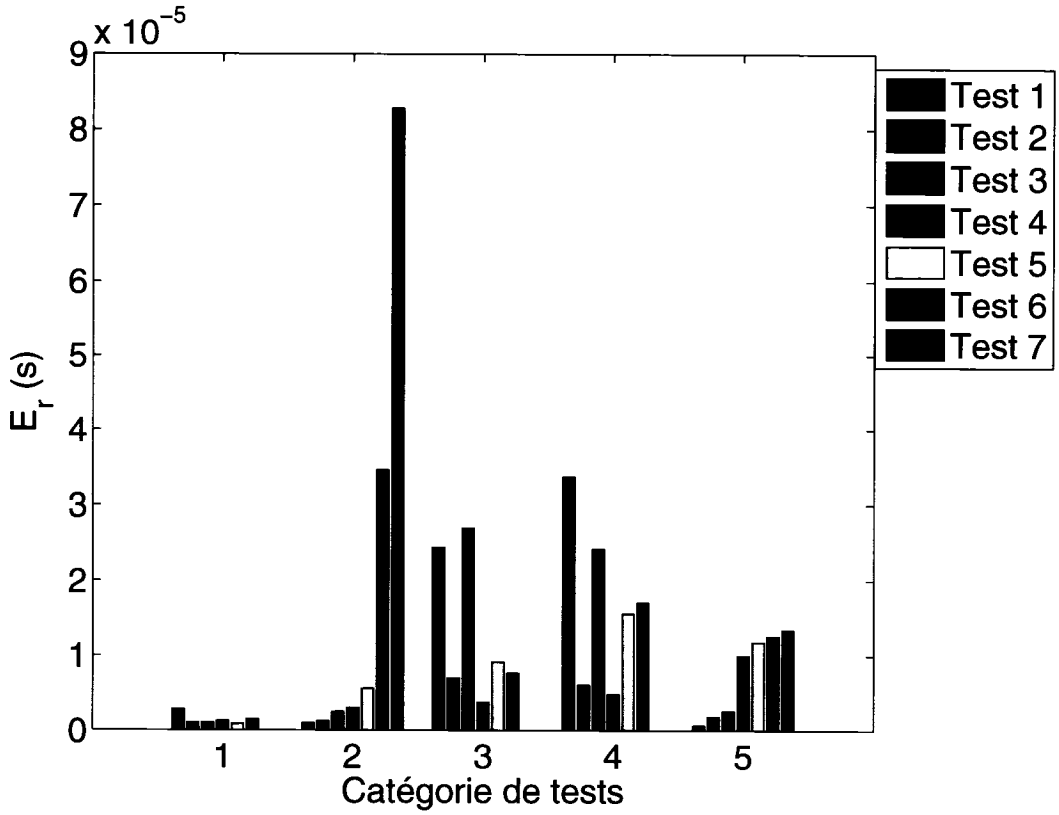


(b) Progression de la différence de D^0_{ij} en fonction du nombre de sauts (intervalle de confiance de 90%)

Figure 4.11 Progression de la différence de l'approximation de la dérive entre un lien direct et un lien indirect en fonction du nombre de sauts

Tableau 4.16 Différence des paramètres de l'approximation de la dérive d'horloge d'un lien direct par rapport à un lien indirect

Lien	Nombre de sauts	Intervalle de confiance	Différence entre lien direct - indirect	
			X_{ij} (s/s)	D^0_{ij} (s)
$N_2 - > N_3$	1	100%	0.00E+000	0.00E+000
		90%	0.00E+000	0.00E+000
		95%	0.00E+000	0.00E+000
		99%	0.00E+000	0.00E+000
$N_2 - > N_4$	2	100%	1.27E-009	7.55E-006
		90%	7.22E-010	7.80E-006
		95%	8.27E-010	7.76E-006
		99%	8.87E-010	7.73E-006
$N_2 - > N_5$	3	100%	2.40E-009	8.02E-006
		90%	1.19E-009	7.90E-006
		95%	1.30E-009	7.97E-006
		99%	1.44E-009	8.00E-006
$N_2 - > N_6$	4	100%	3.36E-008	2.14E-005
		90%	1.59E-008	1.44E-005
		95%	2.20E-008	1.69E-005
		99%	3.14E-008	2.06E-005
$N_2 - > N_7$	5	100%	3.67E-008	1.22E-005
		90%	2.47E-008	1.41E-005
		95%	1.92E-008	1.65E-005
		99%	1.51E-008	2.03E-005
$N_2 - > N_8$	6	100%	3.54E-008	1.26E-005
		90%	2.37E-008	1.44E-005
		95%	1.82E-008	1.69E-005
		99%	1.52E-008	2.07E-005
$N_2 - > N_9$	7	100%	4.53E-008	1.01E-005
		90%	2.49E-008	1.44E-005
		95%	1.94E-008	1.69E-005
		99%	1.56E-008	2.08E-005



Catégorie 1 Variation de la période de communication

- **Test 1** ≈0
- **Test 2** 1s
- **Test 3** 10s
- **Test 4** 20s
- **Test 5** 30s
- **Test 6** 1m

Catégorie 2 Variation de la durée du traçage

- **Test 1** 10m
- **Test 2** 30m
- **Test 3** 45m
- **Test 4** 1h
- **Test 5** 2h
- **Test 6** 5h
- **Test 7** 12h

Catégorie 3 Variation de la charge

- **Test 1** *bzip2*
- **Test 2** *bzip2 + make*
- **Test 3** *bzip2 + make + swap*
- **Test 4** *while(1)*
- **Test 5** *syscall*
- **Test 6** *r/w*

Catégorie 4 Variation de la charge (asymétrique)

- **Test 1** Test 7
- **Test 2** Test 8
- **Test 3** Test 9
- **Test 4** Test 10
- **Test 5** Test 11
- **Test 6** Test 12

Catégorie 5 Variation du nombre de sauts

- **Test 1** 1 saut
- **Test 2** 2 sauts
- **Test 3** 3 sauts
- **Test 4** 4 sauts
- **Test 5** 5 sauts
- **Test 6** 6 sauts
- **Test 7** 7 sauts

Figure 4.12 Comparaison des écarts-types résiduels des catégories de tests (intervalle de confiance de 90%)

CONCLUSION

L'utilisation d'outils d'analyse de performance est rendue inévitable dans le contexte de développement d'un logiciel, compte tenu du niveau de complexité élevé des problèmes informatique de nos jours. Ce degré de complexité est encore plus élevé pour les applications parallèles. D'ailleurs, ces outils permettent de valider le fonctionnement d'une application et, d'autre part, ils s'avèrent indispensables pour optimiser les performances. Le type d'outil le plus général et efficace est le traçage d'événements. Il permet de capturer et d'analyser le comportement d'une application en fonction du temps. L'objectif de ce mémoire est de développer un outil de traçage d'événements pour les systèmes distribués, basé sur LTTng. LTTng permet le traçage d'événements dans l'espace noyau et usager d'un ordinateur ayant Linux comme système d'exploitation. Les lectures de temps sont effectuées à partir du TSC. Le traçage d'événements sur un réseau distribué est plus compliqué que sur un seul système, car une synchronisation des traces est nécessaire afin de s'assurer de respecter l'enchaînement réel des événements, tout en étant fidèle à l'écoulement du temps entre les événements.

Pour cette raison, plusieurs mécanismes et algorithmes de synchronisation d'horloge ont été analysés. Ils nécessitent souvent une source de temps externe, servant de base pour synchroniser l'ensemble des noeuds du réseau. Ces techniques rendent la tâche de synchronisation des traces triviale puisqu'une heure globale est assurée avec une certaine marge d'erreur. Malheureusement, cette synchronisation a un coût en demandant parfois l'usage de matériel (par exemple, un module GPS), tout en étant intrusive dans le comportement de l'application (génération d'échanges de messages). En effet, puisque les horloges des systèmes sont sujettes à des déviations au cours du temps, les noeuds du réseau doivent être synchronisés périodiquement provoquant une génération de trafic qui peut devenir importante en fonction de la

taille du réseau.

La technique choisie pour synchroniser les traces, est d'utiliser des événements corrélés afin d'évaluer le décalage de temps entre deux noeuds. De cette façon, aucun matériel supplémentaire et aucun traitement de synchronisation n'est requis (ce traitement étant fait a posteriori). Ces événements correspondent aux échanges de messages réseau. Ainsi, avec un cycle aller-retour de communications réseau, il est possible d'obtenir une indication du décalage de temps entre deux noeuds à un instant donné. Un autre avantage de cette technique est le fait qu'elle peut être utilisée peu importe la structure du réseau. Il s'agit d'un critère important pour tracer un système distribué. Le seul critère requis est la symétrie des communications réseau. Une approximation de la dérive d'horloge doit être faite afin de suivre l'évolution du décalage en fonction du temps. Dans l'optique d'éliminer les mesures erronées, nous utilisons un intervalle de confiance de 90 %, ce qui permet d'exclure les mesures les plus éloignées, perturbées ponctuellement par diverses raisons, lors du calcul de la dérive. Il est réalisé à l'aide d'une régression linéaire des moindres carrés étant donné que la dérive est linéaire. Après l'évaluation de la dérive d'horloge entre chaque noeud, une référence doit être choisie pour servir de base de temps. Pour ce faire, une recherche de tous les plus petits chemins reliant chaque noeud est effectuée. L'algorithme de Dijkstra est utilisé, basé sur la théorie des graphes. Ainsi, l'imprécision de l'approximation de la dérive sert de poids pour les arcs. Il s'agit de l'écart-type résiduel, E_r, calculé lors de la régression linéaire. La référence élue est donc le noeud ayant la plus petite somme de tous les plus petits chemins la reliant avec l'ensemble des noeuds du réseau. Ainsi, la référence est le noeud dont l'évaluation de la dérive est la plus précise avec l'ensemble du réseau, tout en ayant la plus grande couverture. En effet, il est possible que deux sous-graphes ne communiquent jamais lors du traçage. Dans ce cas, une référence intermédiaire est choisie selon la même technique. Selon le cas, la référence de temps

pourrait être choisie manuellement comme étant l'ordinateur central du réseau ou le serveur d'une application parallèle.

Au chapitre 4, plusieurs expériences ont été réalisées dans le but de cerner les limites de notre système de synchronisation de traces d'un réseau distribué. Nous avons constaté que le débit des échanges de messages n'influence presque pas la précision de l'approximation de la dérive d'horloge. L'écart-type résiduel, E_r, est de l'ordre de la microseconde lorsque le système n'est soumis qu'à une faible charge, comprenant le démon récoltant les données du traçage avec l'application générant du trafic réseau. L'E_r passe environ de 1 μs à 2.5 μs, lorsque les noeuds sont soumis à un fort débit de communications.

De plus, nous avons remarqué que la dérive d'horloge n'est pas tout à fait constante en fonction du temps à long terme. En effet, elle se caractérise plutôt par une légère courbe fortement aplatie. Elle peut donc être représentée par plusieurs segments de droite. De façon expérimentale, nous avons établi que le traçage sur une longue période doit fragmenter l'évaluation de la dérive d'horloge en plusieurs segments de 30 minutes au plus. La raison est que l'E_r est relativement le même pour une durée allant de 1 à 30 minutes, pour ensuite augmenter de façon significative lorsque la durée dépasse une heure. L'E_r passe d'environ 1 μs ou 2 μs pour une durée inférieure à 30 minutes, à environ 35 et 80 μs pour une durée de 5 et 12 heures respectivement.

D'autre part, nous avons observé que l'imprécision de l'approximation de la dérive est grandement influençable par le temps de réponse du système. La variabilité du temps de transmission et du temps de réponse sont les facteurs pouvant dégrader la précision des lectures de temps. Autrement dit, un système ayant un temps de réponse variable induit beaucoup plus d'imprécision dans l'évaluation de la dérive qu'un système avec un temps de réponse lent mais, constant. Pour cette raison, un

système complètement saturé sera caractérisé par un E_r élevé pouvant atteindre 30 μs. De plus, l'asymétrie du temps de réponse rajoute une imprécision aux mesures. Cette asymétrie peut être obtenue par une inégalité de la charge soumise aux noeuds. Ainsi, un noeud sera en mesure de répondre plus rapidement à une lecture de temps qu'un autre. Ceci a pour effet de fausser l'hypothèse que le temps d'aller et le temps de retour d'un message réseau sont sensiblement identiques. De cette façon, une imprécision s'ajoute une fois de plus dans l'évaluation de la dérive.

Nous avons aussi étudié l'incertitude qu'apporte l'utilisation de plusieurs sauts afin d'évaluer la dérive d'horloge entre deux noeuds. Puisque les erreurs sont cumulatives, les liens moins précis vont grandement influencer la précision obtenue. Il est donc primordial de choisir le meilleur chemin reliant deux noeuds afin d'estimer la dérive. Somme toute, la précision obtenue est excellente. Avec sept sauts, nous avons obtenu un E_r inférieur à 20 μs pour un système inactif.

Notre mécanisme de synchronisation de traces d'un réseau distribué est très prometteur. Certes, nous n'obtenons pas une précision aussi élevée que des systèmes de synchronisation d'horloge à l'aide de matériel. Avec un dispositif GPS, il est possible d'atteindre une précision de l'ordre des 50 ns (Ubik, Smotlacha, 2003). Une autre technique (Liao, Martonosi, Clark, 1999) leur a permis d'obtenir une précision de l'ordre de la microseconde pour un système distribué basé sur Myrinet. De notre côté, nous pouvons obtenir, dans les meilleurs cas, une précision de l'ordre de la microseconde et d'une quarantaine de microsecondes lorsque les noeuds sont soumis à une forte charge. Notre système est moins précis que les deux techniques présentées, mais nous avons l'avantage d'être une solution plus générale, car elle ne requiert pas l'utilisation de matériel spécifique. Par contre, si une précision accrue est nécessaire, rien n'empêche l'utilisateur de rajouter un mécanisme de synchronisation d'horloge selon ses besoins. Toutefois, notre technique s'avère un bien meilleur choix qu'une simple utilisation de NTP car, il garantit un écart moyen de

1 à 5 ms (Ubik, Smotlacha, 2003) mais, pouvant dépasser les 50 ms (Mills, 2004) lorsque la source est un serveur secondaire disponible sur l'Internet.

Travaux futurs

Plusieurs ajouts seraient souhaitables à l'implantation du système de traçage décrit dans ce mémoire, afin de guider le programmeur dans l'analyse de performance de son application parallèle. Ainsi, un module pourrait être ajouté à LTTng afin de construire un graphe d'activités du programme (PAG). Ce graphe donne une vue de l'exécution du programme. À partir de ce graphe, plusieurs métriques d'analyse de performance d'applications parallèles ((Hollingsworth, Miller, 1992) comparent plusieurs métriques) peuvent être appliquées permettant d'identifier les faiblesses de l'application sous divers angles. Par exemple, le chemin critique de profilage donne généralement de bons résultats.

De plus, il serait intéressant d'ajouter un mécanisme actif de filtrage d'événements lors du traçage. De cette façon, il serait possible de diminuer la taille des traces qui peuvent rapidement devenir importante, surtout en considérant que parfois les noeuds d'une grappe de calcul ne possèdent pas de disque dur. LTTng possède déjà un mode, nommée *flight recorder*, qui permet d'utiliser une trace comme une liste circulaire. Ainsi, la taille de la trace se retrouve donc bornée. Le problème est qu'il n'y a pas de discrimination entre les événements. Ainsi, un événement avec un fort débit risque de monopoliser l'espace disponible dans la trace, et à l'extrême, écraser tous les événements de communications réseau nécessaires pour synchroniser les traces du réseau distribué. Il serait donc fort utile de développer un mécanisme de filtrage assurant un équilibre de l'usage de la trace lorsque le système est limité en espace mémoire.

RÉFÉRENCES

ANCEAUME, E., PUAUT, I. 1997. « A Taxonomy of Clock Synchronization Algorithms », *IRISA resarch report.* Publication interne numéro 1103 (1 mai 1997). 26 pages.

BENVENUTI, C. 2005. *Understanding Linux Network Internals.* États-Unis : O'Reilly. 1062p.

BONGO, L. A., ANSHUS, O. J. et BJØRNDALEN, J. M. 2003. « Using a Virtual Event Space to Understand Parallel Application Communication Behavior », *Department of Computer Science, University of Tromsø*

BOVET, D. P., CESATI, M. 2005. *Understanding the Linux Kernel.* 3rd Edition. États-Unis : O'Reilly. 942p.

BREAKIRON, L. A. 2006. Cesium Atomic Clock In *Site du Time Service Department, United States Naval Observatory.* [En ligne]. `http://tycho.usno.navy.mil/cesium.html` (Page consultée le 25 avril 2006)

CAUBET, J., GIMENEZ, J., LABARTA, J., DEROSE, L. et VETTER, J. 2001. « An Integrated Performance Visualizer for OpenMP/MPI Programs », *Proc. Workshop on OpenMP Applications and Tools (WOMPAT).*

COULOURIS, G., DOLLIMORE, J. et KINDBERG, T. 2001. *Distributed Systems, Concepts and Design.* 3rd Edition. Toronto : Addison-Wesley. 772p.

COMER, D. E. 1991. *Internetworking with TCP/IP Volume I Principles, Protocols, and Architecture.* 2ed Edition. New Jersey : Prentice hall. 547p.

COMER, D. E., STEVENS, D. L. 1991. *Internetworking with TCP/IP Volume II Design, Implementation, and Internals.* New Jersey : Prentice hall. 524p.

CRISTIAN, F. 1989. « Probabilistic clock synchronization », *Distributed Computing*. Volume 3 : Number 3 (September 1989). 146-158.

DESNOYERS, M., DAGENAIS, M. 2006. « Low Disturbance Embedded System Tracing with Linux Trace Toolkit Next Generation », *CE Linux Technical Conference*

DUNIGAN, T. H. 1992. « Hypercube clock synchronization », *Concurrency : Practice and Experience*. Volume 4 : Issue 3 (May 1992). pages 257-268.

GUSELLE, R., ZATTI, S. 1989. « The Accuracy of the Clock Synchronization Achieved by TEMPO in Berkeley UNIX 4.3BSD », *IEEE Transactions on Software Engineering*. Volume 15 : Issue 7 (July 1989). 847-853.

HAHN, M. 2001. Test de performance Realfeel. [En ligne]. `http://brain.mcmaster.ca/~hahn/realfeel.c` (Page consultée le 25 avril 2006)

HEATH, M. T., FINGER, J. E. 2003. « ParaGraph : A Performance Visualization Tool for MPI (User Guide) », *Center for Simulation of Advanced Rockets, University of Illinois at Urbana-Champaign*. [En ligne]. `http://www.csar.uiuc.edu/software/paragraph/userguide.pdf` (Page consultée le 25 avril 2006)

HOEFLINGER, J., KUHN, B., NAGEL, W., PETERSEN, P., RAJIC, H., SHAH, S., VETTER, J., VOSS, M. et WOO, R. 2001. « An Integrated Performance Visualizer for OpenMP/MPI Programs », *Proc. Workshop on OpenMP Applications and Tools (WOMPAT)*.

HOFMANN, R., HILGERS, U. 1998. « Theory and Tool for Estimating Global Time in Parallel and Distributed Systems », *University of Erlangen*

HOLLINGSWORTH, J. K., MILLER, B. P. 1992. « Parallel Program Performance Metrics : A Comparison and Validation », *Proceedings of the 1992 ACM/IEEE*

conference on Supercomputing. 4-13.

IBM. 2001. IBM$^{(R)}$ Distributed Computing Environment Version 3.2 for AIX$^{(R)}$ and Solaris : Introduction to DCE In *Site d'IBM Software.* [En ligne]. `http://www-306.ibm.com/software/network/dce/library/publications/dceintro/html/DCEINT02.HTM` (Page consultée le 25 avril 2006)

Intel. 2006. Intel Cluster Tools. In *Site de Intel Software Products.* [En ligne]. `http://www.intel.com/cd/software/products/asmo-na/eng/cluster/index.htm` (Page consultée le 25 avril 2006)

JOHNSON, D. B. 1977. « Efficient Algorithms for Shortest Paths in Sparse Networks », *Journal of the ACM (JACM).* Volume 24 : Issue 1 (January 1977). 1-13.

JUHÁSZ, S., CHARAF, H. 2004. « Exploiting fast ethernet performance in multiplatform cluster environment », *Proceedings of the 2004 ACM symposium on Applied computing.* pages 1407-1411.

KORTENKAMP, D., SIMMONS, R., MILAM, T. et FERNÁNDEZ, J. L. 2002. « A Suite of Tools for Debugging Distributed Autonomous Systems », *Proceedings of the IEEE International Conference on Robotics and Automation.* May 2002.

LAMPORT, L. 1978. « Time, Clocks, and the Ordering of Events in a Distributed System », *Communication of the ACM.* Volume 21 : Number 7 (July 1978). 558-565.

LIAO, C., MARTONOSI, M. et CLACK, D. W. 1999. « Experience with an adaptive globally-synchronizing clock algorithm », *Proceedings of the eleventh annual ACM symposium on Parallel algorithms and architectures.* June 27-30, 1999. Saint Malo, France. pages 106-114

LOVE, R. 2005. *Linux Kernel Development.* 2ed Edition. États-Unis : Sam Publishing. 432p.

MHPCC (Maui High Performance Computing Center). 2003. IBM SP Hardware/Software Overview. [En ligne]. `http://www.mhpcc.edu/training/workshop/ibmhwsw/MAIN.html` (Page consultée le 25 avril 2006)

MILLS, D. L. 1991. « Internet Time Synchronization : The Network Time Protocol », *IEEE Transactions on Communications.* Volume 39 : Number 10 (October 1991). pages 1482-1493.

MILLS, D. L. 1995. « Improved Algorithms for Synchronizing Computer Network Clocks », *ACM Transaction on Networks.* Volume 3 : Number 3 (June 1995). pages 317-327.

MILLS, D. L. 1996. Simple Network Time Protocol (SNTP) Version 4 for IPv4, IPv6 and OIS. [En ligne]. `http://www.faqs.org/rfcs/rfc2030.html` (Page consultée le 25 avril 2006)

MILLS, D. L. 2004. « NTP Performance Analysis », *University of Delaware.* [En ligne]. `http://www.eecis.udel.edu/~mills` (Page consultée le 25 avril 2006)

MOORE, S., CRONK, D., LONDON, K. et DONGARRA, J. 2001. « Review of Performance Analysis Tools for MPI Parallel Programs », *Proceedings of the 8th European PVM/MPI Users' Group Meeting on Recent Advances in Parallel Virtual Machine and Message Passing Interface.* pages 241-248

MORTON, A. 2002. Test de performance Realfeel2. *Compris dans le paquetage amlat.* [En ligne]. `http://www.zip.com.au/~akpm/linux/schedlat.html#amlat` (Page consultée le 25 avril 2006)

MYRICOM. 2006. Myrinet Documentation. [En ligne]. http://www.myri.com/scs/documentation (Page consultée le 25 avril 2006)

NAGEL, W. E., ARNOLD, A., WEBER, M., HOPPE, H.-Ch. et SOLCHENBACK, K. 1996. VAMPIR : Visualization and Analysis of MPI Resources. [En ligne]. http://www.netlib.org/benchmark/top500/reports/report95/vampir/vampir.html. (Page consultée le 25 avril 2006)

NOE, R. J. 1994. « Pablo Instrumentation Environment User's Guide », *Department of Computer Science, University of Illinois.*

OLSON, A., SHIN, K. G. 1994. « Fault-Tolerant Clock Synchronization in Large Multicomputer Systems », *IEEE Transactions on Parallel and Distributed Systems.* Volume 5 : Number 9 (September 1994). 912-923.

THE OPEN GROUP. 2005. The Open Group Debuts Open Source Licensing of DCE Source Code In *Site du The Open Group.* [En ligne]. http://www.opengroup.org/comm/press/05-01-12.htm (Page consultée le 25 avril 2006)

RAYNAL, M., SINGHAL, M. 1996. « Logical Time : Capturing Causality in Distributed Systems », *IEEE Computer.* Février 1996. pages 49-56.

REED, D. A., AYDT, R. A., MADHYASTHA, T. M., NOE, R. J., SHEILDS, K. A. et SCHWARTZ, B. W. 1992. « An Overview of the Pablo Performance Analysis Environment », *Department of Computer Science, University of Illinois.* November 1992.

RIEKER, M. 2002. Advanced Programmable Interrupt Controller. [En ligne]. http://osdev.berlios.de/pic.html (Page consultée le 25 avril 2006)

SCIENTIFIC APPLICATIONS GROUP. 2006. Parallel Tools Links. In *Site du Scientific Applications Group.* [En ligne]. `http://www.sdsc.edu/SciApps/links.html` (Page consultée le 25 avril 2006)

SEDGEWICK, R. 2002. *Algorithms in C : Part 5 : Graph Algorithms.* 3th Edition. États-Unis : Addison-Wesley. 482p.

SHENDE, S., MALONY, A. D., CUNY, J., LINDLAN, K., BECKMAN, P. et KARMSIN, S. 1998. « Portable Profiling and Tracing for Parallel, Scientific Applications using C++ », *Proceedings of SPDT'98 : ACM SIGMETRICS Symposium on Parallel and Distributed Tools* (August 1998) pages 134-145.

SISTARE, S., DORENKAMP, E., NEVIN, N. et LOH, E. 1999. « MPI support in the Prism^{TM} programming environment », *Proceedings of the 1999 ACM/IEEE conference on Supercomputing.* Article Number 22.

SLOAN, J. D. 2004. *High Performance Linux Clusters with OSCAR, Rocks, OpenMosix, and MPI.* États-Unis : O'Reilly. 360p.

STEIGNER, C., WILKE, J. 2001. « Multi-Source Performance Analysis of Distributed Software », *University of Koblenz-Landau, Germany*

TATE, J., KANTH, R., TELLES, A. 2005. *Introduction to Storage Area Networks.* Third Edition. IBM Redbooks International technical support organization. 256p.

TIERNEY, B., WILLIAM, J., COWLEY, B., HOO, G., BROOKS, C. et GUNTER, D. 1998. « The NetLogger Methodology for High Performance Distributed Systems Performance Analysis », *Proceedings of the The Seventh IEEE International Symposium on High Performance Distributed Computing.* pages 260-267.

UBIK, S., SMOTLACHA, V. « Precise Measurement of One-Way Delay and Analysis of Synchronization Issues », *CESNET, Prague*

Vampir/GuideView (VGV). [En ligne]. `http://www.llnl.gov/icc/lc/DEG/vgv.html` (Page consultée le 25 avril 2006)

WEBB, D. 2006. Coordinated Universal Time (UTC) In *Site du The Southwest's Source for Regional Space Information.* [En ligne]. `http://www.spacearchive.info/utc.htm` (Page consultée le 25 avril 2006)

WINDL, U., DALTON, D. et MARTINEC, M. 2003. The NTP FAQ and HOWTO In *Site du The University of Delaware.* [En ligne]. `http://www.eecis.udel.edu/~ntp/ntpfaq/NTP-a-faq.htm` (Page consultée le 25 avril 2006)

WORLEY, P. H. MPICL (Portable Instrumentation Library). In *Site du Computer Science and Mathematics, Oak Ridge National Laboratory* [En ligne]. `http://www.csm.ornl.gov/picl/mpicl.html` (Page consultée le 25 avril 2006)

WU, C. E., BOLMARCICH, A., SNIR, M., WOOTON, D., PARPIA, F., CHAN, A., LUSK, E. et GROPP, W. 2000. « From Trace Generation to Visualization : A Performance Framework for Distributed Parallel Systems », *Proceedings of the 2000 ACM/IEEE conference on Supercomputing.* Article number 50. (November 2000).

YAGHMOUR, K., DAGENAIS, M. 2000. « The Linux Trace Toolkit », *Linux Journal.* May 2000.