Observations et analyses quantitatives multi-niveaux d'applications à objets réparties

François-Gaël Ottogalli

UNIVERSITÉ JOSEPH FOURIER – GRENOBLE 1
SCIENCES & GEOGRAPHIE

THÈSE

pour obtenir le grade de

DOCTEUR DE L'UNIVERSITÉ JOSEPH FOURIER

Spécialité : «Informatique : Systèmes et Communication»
École doctorale : «Mathématiques-Informatique»

Présentée et soutenue publiquement

par

François-Gaël OTTOGALLI

Le 27 novembre 2001

Observations et analyses quantitatives multi-niveaux d'applications à objets réparties

---

Directeurs de thèse

Brigitte PLATEAU et Jean-Marc VINCENT

Composition du jury

Président :	Sacha KRAKOWIAK
Rapporteurs :	Guy BERNARD
	Jean-Marc GEIB
	Jean-François ROOS
Examinateurs :	Brigitte PLATEAU
	Vincent OLIVE
	Jean-Marc VINCENT

Thèse préparée au sein du département DTL/ASR - France Télécom R&D
et du laboratoire Informatique et Distribution - IMAG

à Cécile,

à ma famille,

au hasard des rencontres.

Remerciements

Je tiens tout d'abord à remercier le département DTL/ASR de France Télécom R&D pour m'avoir fourni des conditions de recherche plus que propices à mon travail, Tant au niveau humain que matériel, ainsi que le laboratoire Informatique et Distribution de l'IMAG pour le soutient scientifique qu'il m'a apporté.

Je voudrais de même remercier Sacha KRAKOWIAK de m'avoir fait l'honneur de présider mon jury de thèse ; Guy BERNARD, Jean-Marc GEIB et Jean-François ROOS pour avoir eu la patience et le courage de rapporter sur mon manuscrit ; et enfin Brigitte PLATEAU, Vincent OLIVE et Jean-Marc VINCENT pour m'avoir accompagné et dirigé tout au long de ces trois ans de thèse. A tous un grand merci ; vos avis, remarques et commentaires m'ont été d'une grande aide et font partie à part entière de mon travail final.

Prendre le chemin d'une thèse a été pour moi l'expression d'un défi, d'un formidable challenge à relever. Cette démarche est le fruit de rencontres, de contacts, de personnes pour qui la recherche est un acte naturel. Je ne saurai aller plus loin dans mes remerciement sans rentre hommage au courage et à la persévérance de Jean-Marc VINCENT. Il est vain de dire que sans lui rien de tout cela n'aurait été réalisé. Cette thèse est l'aboutissement de plus de six années de travail en commun. J'ai eu l'extraordinaire chance de rencontrer Jean-Marc lors de ma formation en Mathématiques Appliquées, lors de laquelle il m'a fait découvrir la monde de l'algorithmique parallèle. Nous avons voyagé des modèles de Géographie Quantitative jusqu'à l'analyse multi-niveaux d'applications à objets réparties. De m'avoir permis de passer des Mathématiques Appliquées aux problématiques des systèmes répartis est la réification de l'immense spectre de compétences qui est le sien. De plus, et ce n'est pas le moindre des choses, il a eu le courage et la patience de m'initier à la recherche et il a su croire en moi, sûrement plus que ce que j'ai pu le faire/croire en moi même. Ajoutez à tout cela un immense pédagogue, qui a très rapidement compris que j'avais besoin de me frotter aux problèmes et que la liberté d'action et de penser me sont nécessaires pour les appréhender, et vous aurez une image assez précise de ce que représente Jean-Marc pour moi. Pour tout cela et pour tout le reste, je te suis à jamais reconnaissant. Merci Jean-Marc.

Il serait ingrat de ma part de ne faire les louanges que d'une seule personne. Sans en amoindrir les mérites, je tiens à citer d'autres «personnages» qui ont participer à me construire scientifiquement et humainement. Comment ne pas .dire un mot de Jacques BRIAT antidote assuré de vos soucis scientifiques et techniques ; de Jean-Louis ROCH pour qui la recherche est un jeu duquel il ne se lasse pas ; de Jean-Bernard STEFANI dont l'étendu des connaissances donne le vertige ; de Jacques CHASSIN DE KERGOMMEAUX à l'humour toujours prompt et aiguisé ; ainsi que de tous ces brillants thésards qui m'ont précédé et qui me succèdent : Benhur DE OLIVEIRA STEIN (encyclopédie uni-

verselle de l'informatique au sens large du terme), Jean-Philippe FASSINO (pour qui tout n'est que librairie, assembleur et simplicité. Il est présent dans ma thèse dans les idées et les réalisations qui vous sont présentées), Alexendre CARISSIMI (Forza, Ferrari !), Eric MAILLET (l'heure c'est l'heure), Jean-Guillaume DUMAS (à qui je dois toujours un squash), Cyril LABBÉ (qui ne m'a toujours pas initié au ski de randonné et à la planche à voile), Nicolas PHILIPPE (mon premier pingouin c'est lui), Reymond VALLIER (par qui tout a commencé) et bien d'autres encore que je n'ai pas cité, je m'en excuse. J'ai eu la chance de rencontrer des personnes bienveillantes, riches en idées et en confiance. Je leurs doit beaucoup.

Comment ne pas remercier ma famille, et en premier lieu ma femme. Elle a su m'accompagner, m'encourager et me supporter durant tout mon parcours. Je ne lui est pas toujours facilité la tâche, elle n'en ai que plus valeureuse. Elle mérite à cet égard toute ma considération et ma gratitude. J'aurai aussi une pensée particulière pour mon chien, Wooffy, avec qui j'ai sillonné de nombreux chemins, autant d'occasions de faire le point et de prendre un peu de recul sur une situation pas toujours très claire.

Je vous remercie tous, ceux cités ainsi que ceux que j'ai oublié. Cette thèse est la preuve, une fois de plus, que la recherche n'est pas une activité individuelle mais tout au contraire qu'elle s'inscrit dans une dynamique de groupe. Sans vous je n'y serais jamais arrivé, sans vous je ne serais pas là à vous remercier. Je vous dédie donc tout naturellement ma thèse : au hasard des rencontres, à toi Cécile.

Table des matières

Table des figures

Chapitre 1

Introduction

> « *Le télécran recevait et transmettait simultanément. Il captait tous les sons émis par Winston au-dessus du chuchotement très bas. De plus, tant que Winston demeurait dans le champ de vision de la plaque de métal, il pouvait être vu aussi bien qu'entendu. [...] On devait vivre, on vivait, car l'habitude devient instinct, en admettant que tout son émis était entendu et que, sauf dans l'obscurité, tout mouvement était perçu.* » [**Orwell, 1950**]

Le vingt et unième siècle est annoncé comme celui de l'information et des communications. Leur quantité et leur débit croissent de jour en jour, à tel point que la question du stockage et des traitements finit par supplanter celle de la qualité, de la fraîcheur. Mais que faire de ces informations :

> « *La valeur de ce qui est transmis dépend entièrement, pour chaque individu, des modalités de sa réception, avec la présence ou non des éléments - terminologie commune, attention, intérêt, langue, paradigme - indispensables pour que ce qui est reçu acquière un sens.* » [**Blondeau et Latrive, 2000**]

On voit ici apparaître le corollaire direct de l'acquisition d'informations : son traitement. En effet, une information brute, sortie de son contexte, non traitée reste difficilement interprétable, voir vide de sens ou encore emplie de contre sens.

Plus le débit d'informations augmente, plus les moyens de traitement qui lui sont attachés deviennent critiques. On a besoin de plus de puissance de calcul, plus de vitesse, plus de complexité dans les traitements. Les systèmes de traitement de l'information deviennent rapidement d'énormes «usines à gaz» dans lesquelles il n'est pas bon d'avoir à repérer la moindre fuite. Les systèmes deviennent opaques, impénétrables, et finissent par se transformer en boîte noire dont les mécanismes et les actions nous échappent.

Afin de remédier à ce problème, les modèles à objets ou la multi-programmation légère nous

aident à mieux structurer les systèmes, à mieux utiliser les ressources d'exécution auxquelles ils accèdent. Mais qui dit mieux sous entend que l'on puisse quantifier les apports et les performances de ces modèles.

L'évaluation des performances nous aide dans cette démarche. Selon [**Lange et al., 1992**], elle peut être vue comme un processus itératif se décomposant en six phases : l'identification des problèmes, la formulation des objectifs, la définition d'un plan d'expériences,la mise en place de l'instrumentation, réalisation des mesures et l'interprétation des données collectées.

Les trois premiers points correspondent à une phase de réflexion et de modélisation permettant d'identifier les sources du problème et de définir, d'une façon théorique, comment les résoudre. Les trois points suivants représentent la concrétisation de l'étude théorique pour la confronter à la réalité expérimentale.

Cette confrontation est assujettie à un certain nombre de problèmes dont le premier est : «l'observabilité» du système. Il s'agit de définir à quel point le système est observable : *où*, *quand* et *comment*. L'objet des observations est la création d'une image de l'exécution afin que :

> « *A partir de son cadre conceptuel et de ses connaissances, le chercheur formule des hypothèses, qu'il met ensuite à l'essai.* » [**Perdijon, 1998**]

Le point de départ de ce travail est l'observation du méta ORB Jonathan [**Dumant et al., 1998**], et plus particulièrement des liaisons qu'il procure. Pour cela, l'idée initiale était de modifier le comportement des usines à liaisons (voir section 2.2) de façon à créer des liaisons instrumentées. L'instrumentation aurait dû nous permettre d'étudier les interactions entre objets, et plus particulièrement les interactions distantes.

Ce projet a évolué pour s'orienter vers un environnement d'observation plus généraliste. Il ne s'agit plus seulement d'observer des liaisons Jonathan, mais l'exécution d'objets qui interagissent (voir section 2.2). Or, comme les liaisons créées par Jonathan sont elles-mêmes instanciées par des objets, notre environnement est donc capable de répondre au projet initial.

Les observations sont réalisées à un niveau applicatif (voir section 3.4) et au niveau système (voir section 3.3). Or, la granularité et la sémantique des données obtenues au niveau applicatif sont différentes de celles des données du niveau système. Pour pouvoir interpréter les unes par rapport aux autres, il est nécessaire de les projeter dans un référentiel de sens commun, c'est-à-dire un référentiel dans lequel il est possible de les interpréter les unes par rapport aux autres (voir section 3.5).

A la suite de quoi, une analyse statique (voir section 4.3) et dynamique (voir section 4.4) de l'exécution observée est réalisée. Elles répondent à la problématique énoncée ci-dessous.

1.1 Problématique

Le contexte de ce travail est le domaine des applications à objets réparties. **La problématique abordée porte sur la reconstruction post-mortem, à partir de mesures, de la dynamique d'une exécution afin de réaliser une analyse qualitative et quantitative des ressources d'exécution consommées.**

Pour cela, nous posons comme **hypothèse que l'analyse des interactions entre objets, par le prisme de l'observation multi-niveaux, procure les informations suffisantes aux études que nous menons.**

La méthodologie mise en œuvre consiste en la réalisation d'observations au niveau système (voir section 3.3) **ainsi qu'au niveau applicatif** (voir section 3.4). Les observations du niveau système nous informent sur les communications ainsi que les consommations des ressources d'exécutions (par exemple processeur, mémoire ou lien de communication). Les observations du niveau applicatif nous permettent de reconstruire la dynamique d'exécution en terme d'interaction entre objets.

Pour être significative, la mise en œuvre doit être la moins intrusive possible ou alors être quantifiée et corrigée (voir section 4.2). De plus, elle doit être aussi peu contraignante que possible quant à son utilisation. **Dans notre cas, les observations réalisées ne demandent ni modification du code original de l'application, ni phase de recompilation.**

Pour valider notre approche, nous avons développé une infrastructure d'observation multi-niveaux pour applications Java réparties (voir section 5). Le langage à objet Java étant largement répandu pour le développement d'applications, nous posons l'hypothèse que l'étude de ce type d'application recouvre une classe suffisamment large de situations.

Les traces événementielles collectées lors des exécutions sont corrigées et transformées pour être analysées par un environnement de visualisation : "Pajé" en l'occurrence (voir section 4.5).

Cette infrastructure a été testée sur des jeux d'exemples types, puis pour l'observation et l'analyse d'un serveur multimédia développé au sein du DTL/ASR de France Télécom R&D (voir section 6.5). Les principaux résultats de cette collaboration sont présentés dans la section suivante.

1.2 Principaux résultats

La validation de notre approche est réalisée par la construction d'une **infrastructure d'observation multi-niveaux** pour application Java s'exécutant sur un système d'exploitation Linux. Elle permet d'obtenir des **observations provenant du noyau du système** (informations de consomma-

tion de ressources d'exécution et utilisation des liens de communication : voir section 5.1.2) et **des JVM au travers de la JVMPI** (début et fin de méthode, utilisation des objets de synchronisation, etc. : voir section 5.2.2.1). De cette façon, **le code de l'application n'est ni altéré ni recompilé**.

Les **observations** collectées sont des **événements datés et nommés** de façon univoque. Grâce à leur datation et à leur nommage, ils sont projetés dans un référentiel de sens commun. Cette trace est convertie, lors d'un traitement post-mortem, en une trace interprétable par l'environnement de visualisation "Pajé" (voir section 5.2). Cette infrastructure a été testée pour l'observation d'un serveur multimédia.

1.3 Un exemple d'utilisation : l'observation d'un serveur multimédia

L'application étudiée est un serveur multimédia de livres et de musique au format MP3 auquel accèdent des clients. Cette application est intéressante dans le sens où elle met en œuvre des interactions de types asynchrone (accès à un livre) et synchrone (accès à un morceau de musique). Elle couvre un large spectre de situations aussi bien en termes de diversité et quantité de données à traiter (texte, image, musique) que de complexité des traitements associés à ces dernières (simple accès, décompression, interprétation et conversion de format). Les expériences observées sont composées d'un courtier, de serveurs et de clients.

Un exemple de référence : l'accès à un livre

Soit un courtierauprès duquel les serveurs viennent enregistrer les propriétés des ressources qu'ils exportent. Lorsqu'un client souhaite accéder à une ressource, il demande au courtierla référence d'un serveur susceptible de satisfaire sa demande. Le courtierlui fournit une telle référence et le client établit une liaison avec le serveur. Cette liaison sert de support aux requêtes formulées par le client.

L'exemple considéré représente la requête d'un client qui souhaite obtenir le résumé d'un livre. Le client demande donc au courtierla référence d'un serveur susceptible de lui fournir le service désiré. Une fois cette référence acquise, le client établit une liaison avec le serveur par lequel ils interagissent (invocation de méthodes permettant d'obtenir le résumé du livre et le transfert des données).

Les observations présentées correspondent à une implantation du serveur multimédia et du client en Java. Ils utilisent la personnalité David de Jonathan comme infrastructure d'interactions. Elle s'exécute sur un réseau de stations sous Linux reliées par un réseau Ethernet 100 Mbit/s. L'observation des communications et des consommations de ressources système est réalisée au niveau du noyau Linux (voir section 5.1.2) et via la JVMPI pour ce qui est du niveau applicatif (voir section 5.1.1).

Analyse de l'exécution

Les données collectées sont transformées en une trace interprétable par l'environnement de visualisation "Pajé" (voir section 5.2). L'exécution est représentée par une hiérarchie d'objets visuels présentée par la figure 5.8 représentant l'exécution de "*threads*" dans des JVM. Les "*thread*" Java utilisés sont de type «natif», c'est-à-dire qu'à chaque "*thread*" Java correspond un "*thread*" système. Ils servent de ressource de calcul à l'exécution des méthodes de l'application. L'observation des objets de synchronisation, autre ressource de calcul (voir section 2.3.1), est réalisée au travers de la JVMPI et est représentée sur les mêmes diagrammes espace-temps que les "*threads*" (voir figure 6.5.2.3).

Les interactions distantes prennent la forme de liens entre les JVM représentant les communications observées en tant qu'écritures et lectures sur des "*sockets*" système (voir figure 1.1).

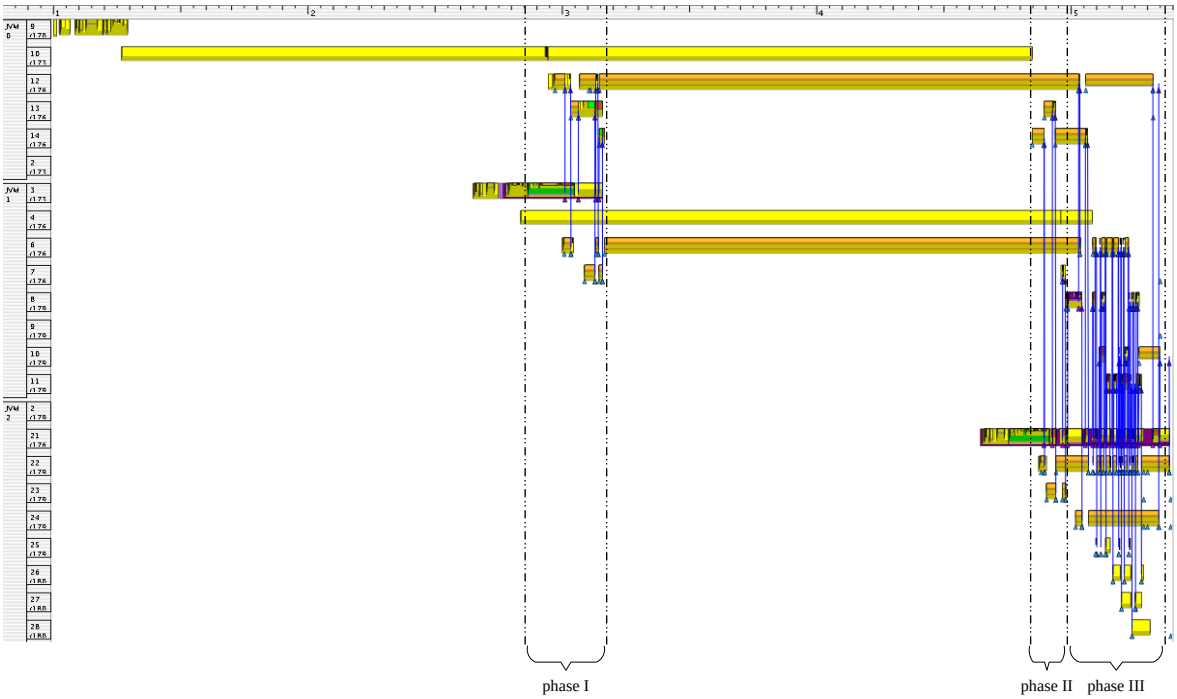


Figure 1.1 – Dynamique de l'exécution représentant l'accès à une ressource «Livre».
Trois phases sont identifiées : I) l'enregistrement des propriétés des ressources exportées par le serveur auprès du courtier(voir figure 6.13) ; II) l'acquisition, par le client, d'une référence sur un serveur pouvant répondre à sa requête (voir figure 6.14) ; III) l'exécution de la requête (voir figure 6.15).

Les observations du niveau système donnent accès à la structure de donnée attachée à chaque processus présent dans un système Linux : la `task_struct` (voir section 5.1.2). Elle procure, entre autres informations, des indices de consommations de ressources système telles que l'utilisation du processeur ou la quantité de mémoire attribuée. Ces informations sont mises en cohérence avec l'exécution de l'application afin d'identifier d'éventuels problèmes d'accès à ces ressources (voir figure 1.2).

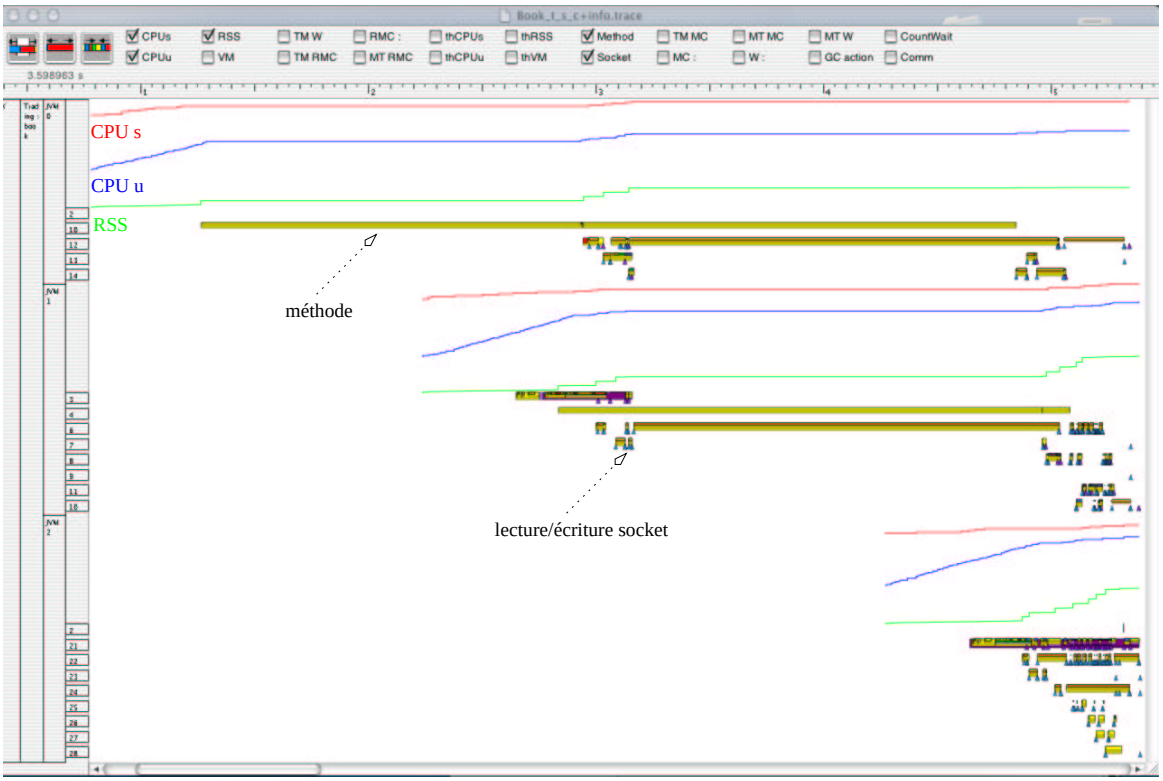


FIG. 1.2 – Exemple du «Livre» : consommation des ressources d'exécution.
Cette visualisation met en corrélation les consommations de la ressource processeur en mode utilisateur (CPU u), en mode système (CPU s) et la taille en mémoire occupée par le JVM (RSS) avec l'exécution de l'application. Les mesures réalisées sont en centième de seconde (la granularité de l'ordonnancement des processus). Elles sont ici exprimées en milliseconde.

Enfin, nous avons identifié des séquences d'appels de méthodes qui se reproduisent tout au long de l'exécution. L'attente d'une requête de type IIOP, ainsi que le retour de cette dernière, sont étudiés en détail dans la section 6.5.3. Cette étude possède un double intérêt : pédagogique puisqu'il retrace l'enchaînement exacte des méthodes et fait apparaître les mécanismes de gestion des requêtes par un "*pool*" de "*threads*" dans Jonathan ; et simplificateur car l'identification de telles séquences peut permettre de simplifier la trace originale en remplaçant les séquences par une abstraction de plus haut niveau.

Après avoir présenté la problématique traitée et dressé les grandes lignes de la solution que nous lui avons apportée, nous allons positionner notre travail par rapport aux projets de recherche déjà existants.

1.4 Environnements d'observation d'applications réparties

Un environnement d'observation est mis en œuvre afin d'acquérir des informations sur un système. Celui-ci est instrumenté de façon à collecter des données et générer des informations sur son comportement. Cette instrumentation peut être matérielle, logicielle ou hybride (voir Chapitre 3). Dans tous les cas, l'objectif est de capturer les changements d'état et de créer de l'information sur le contexte qui s'y rattache. Les observations pourront être réalisées à différents niveaux d'abstraction tels que applicatif, système ou matériel (voir section 3.1).

Pour ce qui est de l'instrumentation logicielle, l'ajout du code pour l'observation peut être automatique (comme dans Paradyn (voir section 1.4.5)), dirigée (comme dans TAU (voir section 1.4.6)) ou manuelle (comme dans DSKI (voir section 1.4.4) ou Pablo (voir section 1.4.3)).

Les données sont collectées dans le cadre de deux types d'analyse. La première correspond à la construction d'une vue statique de l'exécution, sous la forme de résumé statique sur une période de temps fixée (il s'agit ici de "*profiling*"). Dans ce cas, une fonction de traitement est appliquée pour construire l'indice résumé voulu (DSKI, TAU). La seconde cherche à reconstruire la dynamique de l'exécution. Pour cela, chaque observation est identifiée et ordonnée par le système d'observation de façon à représenter l'enchaînement des événements (Paradyn, TAU).

Une fois les données acquises, elles sont exportées à l'extérieur de l'instrument d'observation. Cela peut représenter la sauvegarde des informations dans une base de données (MODINOS) ou un périphérique virtuel (DSKI), ou encore la mise en place d'un modèle d'infrastructure de type client/serveur comme c'est le cas par exemple dans TAU (voir section 1.4.6).

A la suite de quoi, les données sont traitées et analysées. Il peut s'agir de traitements statistiques, de conversion de format ou de mise en cohérence pour la construction de traces événementielles. Des outils d'analyse automatique ont été développés tel que celui de Paradyn ("w^3 *search*") qui recherche automatiquement certaines classes de problèmes de performance. Finalement, le résultat des analyses, ou une modélisation des données observées, est présenté à l'utilisateur.

Cette section a pour objet de présenter un panorama de différentes architectures et implantation d'environnement d'observation et de nous positionner par rapport aux solutions proposées.

1.4.1 OMIS : définition d'un standard pour les interfaces de "*monitoring*"

Le projet OMIS ("*On-line Monitoring Interface Specification*") [**Ludwig et al., 1996**, **Ludwig et al., 1997**, **Ludwig et Wismüeller, 1997**] propose la définition d'une interface standard pour le "*monitoring*" [1]. L'environnement d'observation définit deux interfaces : une avec le système observé et une

[1] Le "*monitoring*" recouvre les actions d'observation, d'enregistrement et de transport

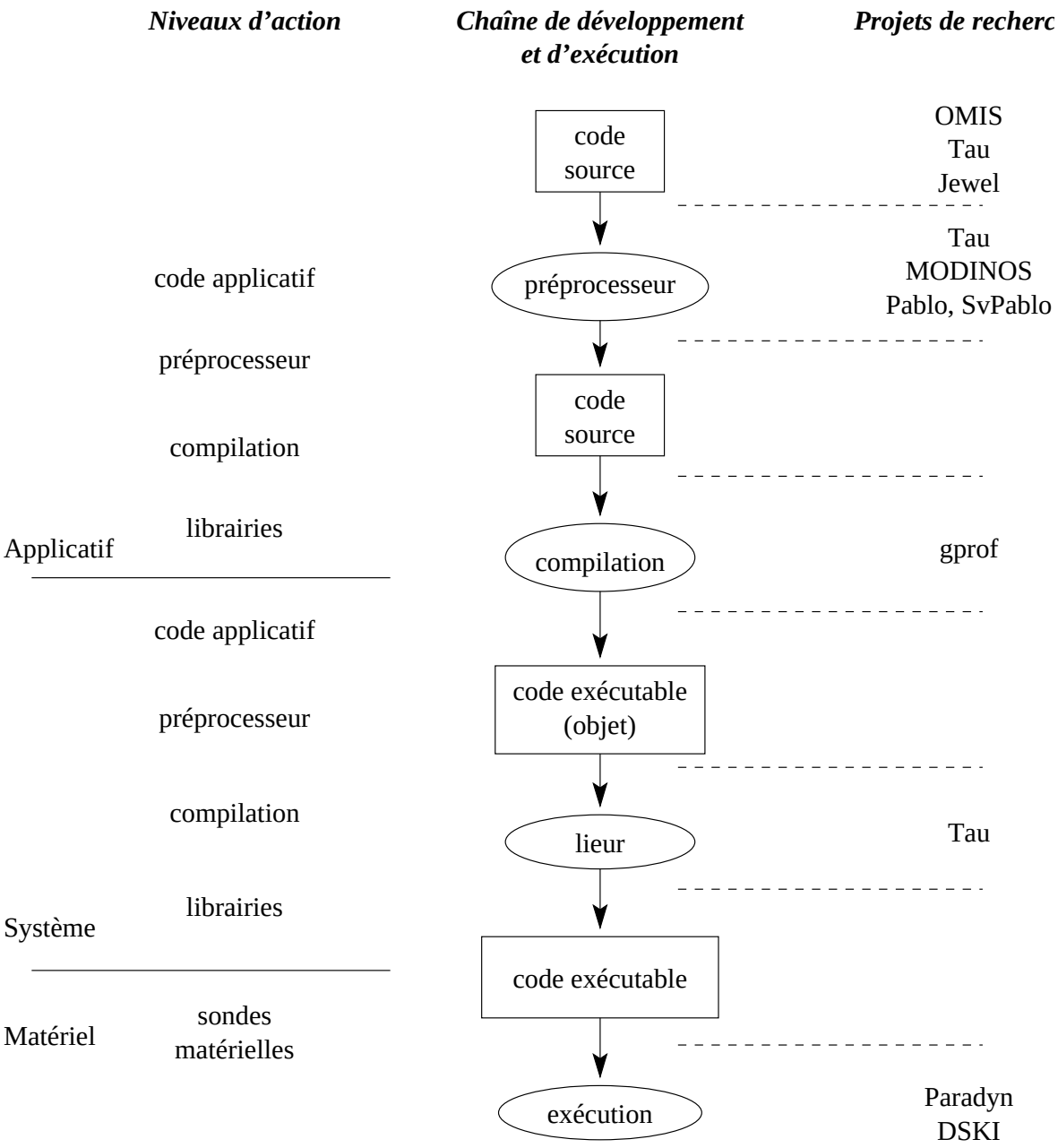


Figure 1.3: Synthèse des points d'accroche des différents environnement d'observation.

avec les outils à valeur ajoutée[2] qui traitent les observations. OMIS propose une standardisation de l'interface exportée vers ces outils. Elle a pour fonction l'observation et le contrôle de l'application durant son exécution. Cette normalisation offre la possibilité de brancher/utiliser plusieurs outils sur le même environnement de façon à fédérer leurs actions et leurs observations.

Le système d'observation repose sur un modèle d'exécution de type événement/action (voir figure 1.4). Ainsi, l'occurrence d'un événement déclenche l'exécution de l'action qui lui est associée. De

[2]Il s'agit par exemple d'outil de débogage, de contrôle ou d'évaluation de performances [**Boutrous Saab, 2000**]

ce fait, le système d'observation est programmable dans le sens où l'on définit les couples événement/action. Dans ce modèle, l'environnement d'observation a en charge la détection des événements et la mise en œuvre de la ou des actions associées.

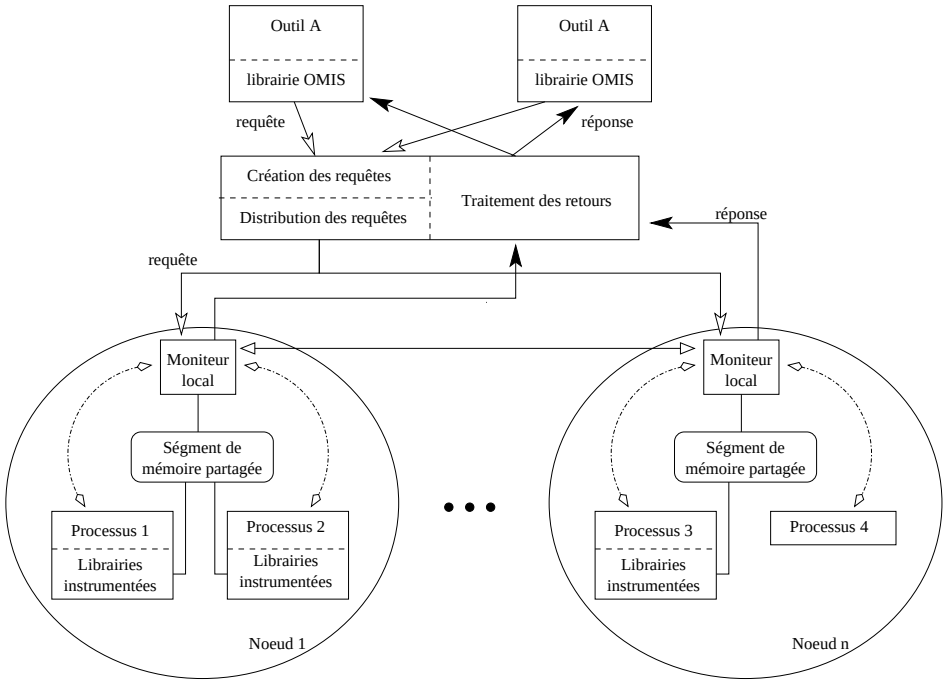


Figure 1.4: Architecture générale proposée dans OMIS.
Les outils de traitement formulent des requêtes qui sont interprétées et réparties par une unité de distribution. Ils réalisent le lien entre les outils et les applications observés. Chaque nœud observé possède un moniteur local, connecté à l'unité de distribution, qui a en charge la gestion des requêtes et le retour de leurs résultats. Des événements spéciaux peuvent être envoyés entre moniteurs locaux afin de réaliser, par exemple, des opérations de synchronisation.

L'application observée est modélisée par une hiérarchie de classes d'objets. Par exemple, une application basée sur un modèle d'exécution à base de "*threads*" qui communiquent par passage de messages est modélisée sous la forme :

de nœuds qui servent d'unité de distribution de l'exécution parallèle.

de processus qui représente les unités d'exécution. Chaque processus est associé à un seul nœud et ne peut appartenir à plusieurs.

de "threads" qui appartiennent à un seul processus et ne peuvent migrer vers un autre.

de file de messages qui représentent les communications entre processus.

Selon les besoins de l'observation, des objets représentant des requêtes conditionnelles ou des événements définis par l'utilisateur peuvent être rajoutés.

Pour chaque classe d'objets, une liste de service est spécifiée. Les services sont de trois catégories : information, manipulation ou services d'événement. La dernière catégorie permet de notifier un changement d'état de l'objet. Ces services permettent de construire des requêtes d'observation depuis les outils de traitement. Une requête est constituée d'un événement de service suivi d'une liste d'actions. Chacune des actions est une action d'information ou de manipulation. Si le «service d'événement» est indéfini, la requête est dite *inconditionnelle* et est exécutée immédiatement. Sinon, la requête est dite *conditionnelle* et la liste d'action est exécutée lorsque l'événement a lieu. Une fonction de retour est associée aux requêtes conditionnelles de façon à permettre la communication des résultats des actions de façon asynchrone.

Le modèle défini dans OMIS a été mis en œuvre dans le système d'observation OCM [**Wismüller et al., 1998**]. Il s'agit d'un environnement d'observation pour applications réparties utilisant PVM pour leurs communications et s'exécutant dans un réseau de station sous UNIX.

Positionnement de notre travail :
OMIS se focalise sur la partie intermédiaire d'un environnement d'évaluation de performances. C'est-à-dire qu'il ne traite pas des problèmes liés à l'acquisition des observations, ni à leur traitement, mais à la connexion entre les outils de traitement et les sources d'observation. A ce titre, il ne traite pas du problème de la cohérence et de l'ordonnancement des observations dans un système distribué ou parallèle. Toutefois, la capacité à fédérer l'action de plusieurs sources d'information rend l'approche intéressante dans le cadre de l'observation multi-niveaux.

1.4.2 MODINOS : architecture pour le "*monitoring*"

MODINOS ("*Managed Object-base Ditributed Monitoring System*") [**Zieliński et al., 1995**] est une architecture et un modèle d'activités pour le "*monitoring*" (voir figure 1.5). MODINOS se structure en couches. Des interfaces et des protocoles sont définis pour les communications entre couches. De cette façon, il offre un contrôle fin du comportement de l'environnement d'observation.

Un environnement de "*monitoring*" est décomposé en quatre couches :

L'environnement
Il s'agit de la couche dans laquelle les objets sont observés via les moniteurs locaux. Les observations collectées dans l'application sont converties selon un modèle uniforme ("*Uniform Model*"). Par conséquence, au dessus de la première couche, seule la sémantique abstraite du modèle uniforme est accessible.

L'interopérabilité
Elle représente la couche de communication entre les moniteurs locaux et le moniteur global. Différentes stratégies de transport des observations sont possibles : socket, proxy, gateway, etc.

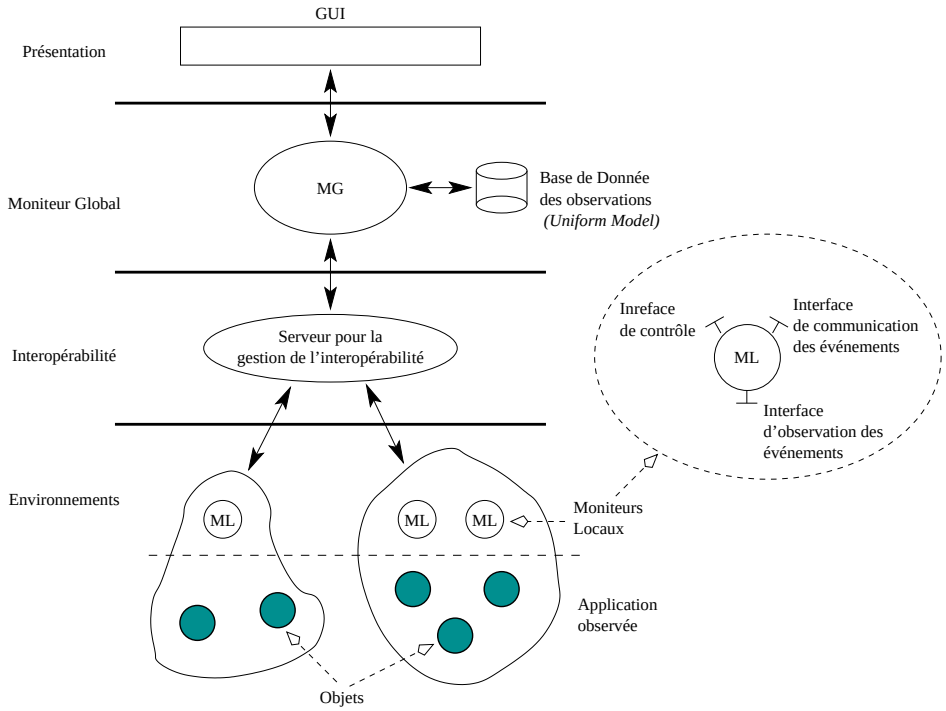


Figure 1.5: Architecture générale proposée dans MODINOS.
Les applications sont observées par au travers des moniteurs locaux par lesquels l'environnement peut exercer des opérations de contrôle. La couche d'interopérabilité joue le rôle de glu entre les applications observées et le moniteur global. Ce dernier enregistre les événements dans une base de données qui sert aux outils d'analyse et de traitement.

Cette couche garantit l'interopérabilité et l'insertion dans des environnements hétérogènes. De ce fait, l'hétérogénéité est supportée au niveau logiciel et infrastructure de communication.

Le moniteur global

Il correspond à la centralisation des informations recueillies au niveau local. Les observations sont agrégées et enregistrées dans une base de donnée nommée "*Internal Model*". Cette couche réalise l'interface avec celle de présentation.

La présentation

C'est la couche d'interaction avec l'utilisateur. Les outils de traitement et de visualisation viennent se connecter à l'environnement par la couche de présentation qui offre une vue uniforme des informations recueillies.

Positionnement de notre travail :

MODINOS, tout comme OMIS, propose une architecture de la partie intermédiaire d'un environnement d'évaluation de performances. Son point fort réside en sa capacité à s'intégrer dans des environnements hétérogènes (du point de vue logiciel et des communications) et répartis. Toutefois, la partie liée à l'instrumentation et à la cohérence et l'ordonnancement des observations n'est pas

intégrée dans l'environnement.

1.4.3 SvPablo : un environnement de mesure de performance

SvPablo [**Rose et al., 1998**] est un environnement de mesure de performances multi-langages (C, Fortran 77 et 90, HPF) et multi-systèmes (séquentiels et parallèles). Il a pour vocation d'observer les exécutions sur de grandes périodes de temps (i.e. plusieurs heures ou plusieurs jours). L'information construite est de nature statistique et non pas dynamique.

Le code de l'application est instrumenté de façon automatique ou par un processus interactif. Par exemple, dans le cas du langage HPF, l'instrumentation peut être automatiquement insérée grâce à l'emploi d'un compilateur spécialisé. Pour ce qui est de l'instrumentation interactive, elle est réalisée à l'ide d'une analyse séquentielle du code. L'environnement propose des points d'instrumentation au niveau des appels de méthodes situées à l'extérieur des boucles.

Le programme instrumenté est lié à la bibliothèque de SvPablo qui réalise la gestion et le traitement des données collectées. Elle permet d'intégrer des données venant d'autres sources que le programme par exemple de points d'observation du niveau matériel. Une interface a été développée pour prendre en compte les données provenant des capteurs matériels présents sur SGI MIPS R10000. Elle fournit aussi quelques fonctions de traitement des données brutes pour construire les indices statistiques.

La trace générée est au format "*Self-Defining Data Format*" (SDDF) définit dans Pablo [**Aydt, 1994**]. Ce format permet de complètement dissocier le format d'enregistrement des données de l'application observée. L'interopérabilité des différents composants de Pablo est basée sur ce format pivot. Il existe des traducteurs de différents formats de trace tels que PICL, AIMS ou IBM VT vers le format SDDF. L'environnement d'analyse de SvPablo est ainsi utilisable à partir de traces produites par d'autres environnements d'observation.

L'outil de visualisation fournit de multiples représentations des indices statistiques construits. De plus, il réalise la correspondance entre le code source de l'application et les indices de performance observés. Les analyses peuvent être menées sur plusieurs traces ouvertes simultanément pour être comparées.

Positionnement de notre travail :
SvPablo traite des problèmes de l'instrumentation pour l'obtention des observations. Un des points forts de ce projet repose sur la définition du format de trace autodéfinie SDDF. Toutefois, il n'est pas conçu pour représenter la dynamique d'exécution des applications, et ne traite pas des problèmes de cohérence et d'ordonnancement des observations dans un système distribué ou parallèle.

1.4.4 DSKI : collecte de données au niveau noyau pour l'évaluation de performance

DSKI ("*Data Stream Kernel Interface*") [**Buchanan et al., 1998**] propose une interface standard de collecte de données provenant du noyau du système hôte. Les implantations actuelles de ce projet sont sous Linux sur PC et Digital Unix sur DEC Alpha. Elles prennent la forme d'un pilote de pseudo-terminal depuis lequel on accède aux données collectées.

Les données sont définies par des "objets" représentant des collections de données. L'ensemble des données collectées est nommé "flux de données" ("*Data Stream*"). Trois type de flux de données sont définis :

Un flux de données actif ("*Active Data Stream*")
Il enregistre des séquences de données appelées événements (changement d'état dans le noyau). Ce type de flux est basé sur la définition d'une fonction d'enregistrement qui place dans une file, appartenant à l'espace d'adressage noyau (donc sans changement de contexte), les événements observés. L'utilisateur accède aux enregistrements via un appel à la fonction `read()` sur le descripteur de fichier associé au flux. Il s'agit d'un mécanisme coûteux puisqu'il demande la mise à jour et le maintien en mémoire de la totalité de la file d'observation durant l'exécution.

Un flux de données passif ("*Passive Data Stream*")
Il tient à jour une structure de données représentant les états des objets observés. Le coût est minimum puisque les états ne sont lus que lorsqu'une lecture est réalisée sur le descripteur de fichier associé au flux.

Un flux de données cumulatif ("*Cumulative Data Stream*")
Son rôle est de procurer des informations statistiques sur les événements ayant lieu dans le noyau. Cette fonctionnalité permet d'observer le comportement global sur du long terme lorsque l'on ne s'intéresse pas aux détails de la situation. Elle requiert la définition des fonctions statistiques qui traiteront les événements observés.

Ainsi, pour obtenir de nouveaux indices de performance il suffit de définir un nouvel objet englobant les données recherchées. Par exemple, si l'on souhaite observer le nombre de paquets TCP/IP qui transitent par un port donné, il suffit de définir un objet dont le rôle est d'observer les paquets et d'y accéder via un flux de données cumulatif.

DSKI est implanté sous la forme d'un service ("*DSKI driver*") donnant accès à un pseudo terminal qui réalise la passerelle entre les observations de bas niveau et l'utilisateur. Le noyau tient en permanence à jour un ensemble d'objets d'observations définis à l'initialisation du noyau. La façon dont les objets sont utilisés est définie lors de l'ouverture du pseudo terminal. C'est à cet instant que le mode d'accès des objets fixé, par l'utilisation d'un type de flux de données, et que les mécanismes d'accès

ou de traitement sont mis en œuvre. DSKI offre des fonctions d'enregistrement et de traitement afin de créer des instances de flux de données.

A titre d'exemple, nous pouvons citer les travaux de [**Niehaus et al., 1999**] dans lesquels DSKI est utilisé pour mesurer les performances de KURT : un noyau Linux temps réel, ainsi que les travaux de [**Goipnath et al., 1998**, **Nimmagadda et al., 1999**] où DSKI est utilisé pour l'évaluation de performance d'ORB CORBA.

Positionnement de notre travail :
L'intérêt de DSKI consiste en la construction d'un environnement d'observation du niveau noyau, accessible depuis l'espace utilisateur via l'utilisation du pseudo système de fichier `/proc`, auquel on associe des fonctions de traitement : les flux de données. Toutefois, ce projet n'a pas été conçu pour traiter des problèmes liés à la distribution, ni pour intégrer, par exemple, des informations du niveau applicatif dans la création des flux de données. De plus, le noyau doit être modifié et recompilé pour supporter les fonctionnalités de DSKI.

1.4.5 Paradyn : outil de mesure de performances pour applications parallèles et réparties

Paradyn [**Miller et al., 1994a**] est un outil de mesure de performances pour applications parallèles et réparties. Il repose sur les notions d'instrumentation et de mesures de performances dynamiques. L'instrumentation est contrôlée par le module "*Performance Consultant*" qui insère automatiquement les points d'instrumentation durant l'exécution en fonction du problème de performance à traiter.

Paradyn se base sur des heuristiques de reconnaissance de goulots de performances, ainsi que sur la structure du programme, pour choisir les points d'instrumentation. De plus, il évalue les perturbations liées à la prise de trace afin qu'elles ne dépassent pas une limite définie par l'utilisateur et que l'observation ne modifie "pas trop" l'exécution initiale.

L'environnement Paradyn consiste en une entité centrale de contrôle, des démons d'observation et des processus externes de visualisation. L'unité centrale contient le "*Performance Consultant*", "*Visualization Manager*", "*Data Manager*", et le "*User Interface Manager*" (voir figure 1.6).

Le consultant de Performances
Il contrôle la recherche automatique de problèmes en performances en fonction des données obtenues par le gestionnaire de données.

Le gestionnaire de données
Il fait le lien entre les requêtes et les démons qui réalisent les prises de mesures.

Le gestionnaire de l'interface utilisateur
Il procure une interface en TCL/TK à l'utilisateur lui donnant accès aux informations et fonc-

tionnalités de Paradyn.

Le gestionnaire de visualisation

Il réalise le lien avec des outils de visualisation externes à Paradyn. Il a donc en charge la gestion des requêtes réalisées par ces outils. Plusieurs outils de visualisation peuvent être employés simultanément.

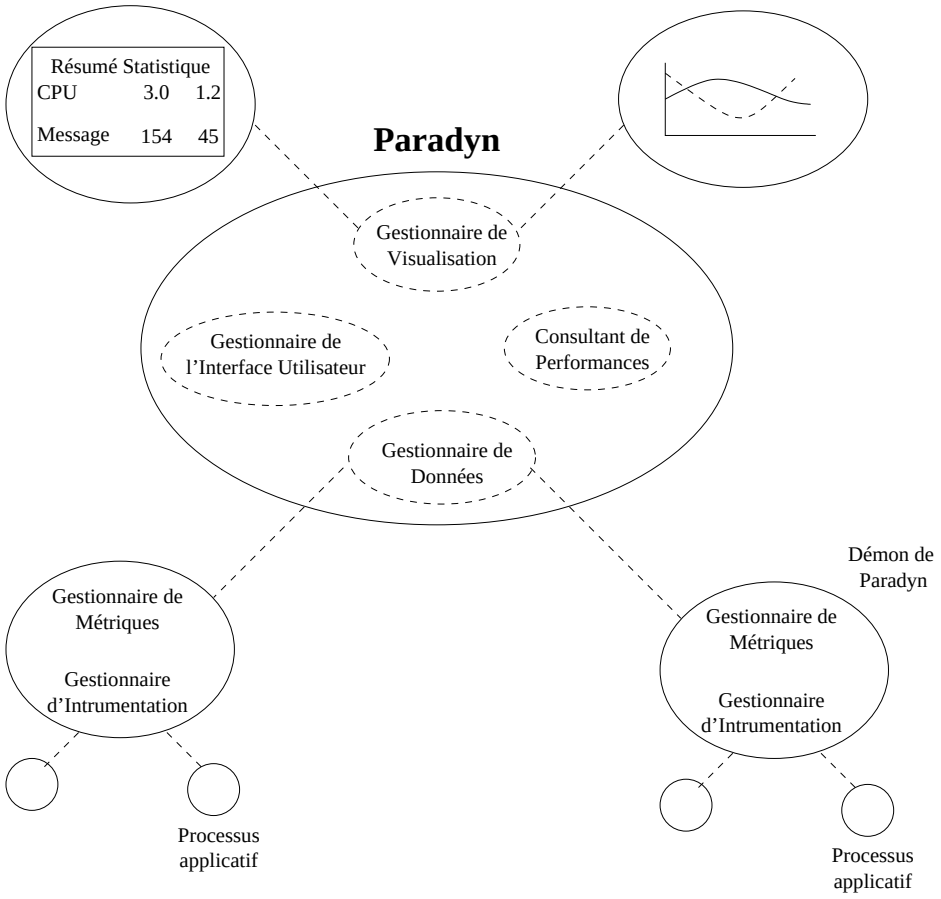


FIG. 1.6 – Architecture générale de Paradyn.

Les démons d'observation possèdent un gestionnaire d'instrumentation ayant la charge de l'insertion dynamique du code d'instrumentation (voir figure 1.7).

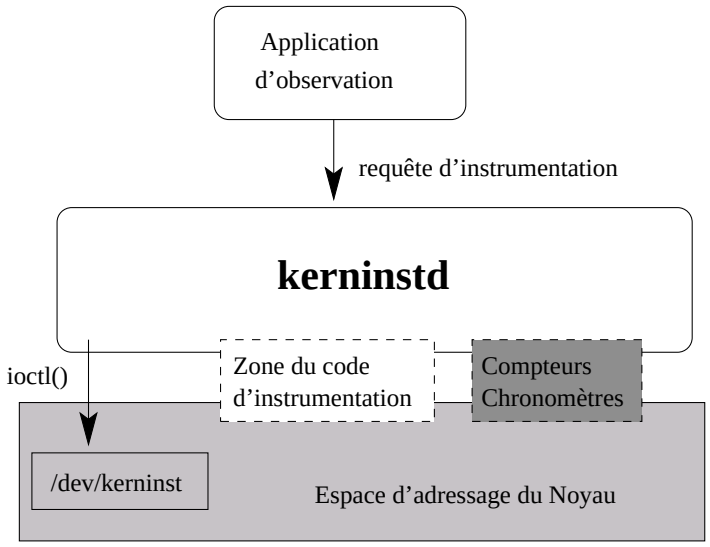


FIG. 1.7 – Kerninstd permet l'insertion dynamique de code.
Kerninstd permet d'insérer dynamiquement du code dans l'application ou le noyau du système. Cette instrumentation permet d'insérer des compteurs et des chronomètres. Elle repose sur les notions de *points d'instrumentation*, de *primitives* (qui permettent de changer la valeur d'un compteur ou d'un "*timer*") et de *prédicats* (qui permettent de calculer une condition avant d'appliquer une primitive). Les points d'instrumentation accessibles sont les débuts et fins de méthode et les appels de méthode.

L'instrumentation est construite en fonction des métriques évaluées. Elle est dirigée par des fichiers de configuration écrit dans le "*Paradyn Configuration Laguage*". L'insertion du code d'observation correspond à un déroutement du code initial vers le code à exécuter (voir figure 1.8).

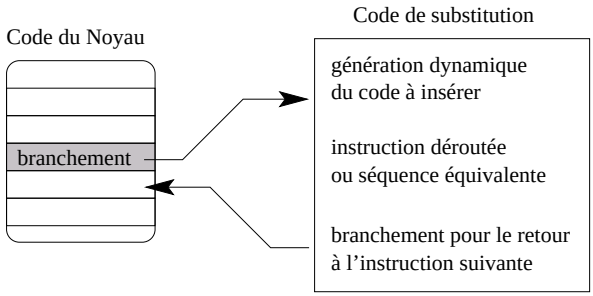


FIG. 1.8 – Insertion de code par déroutement.
Le code original est remplacé par un branchement vers le code à insérer. Ce dernier contient l'instrumentation, le code original ou un équivalent, ainsi que le retour au pas suivant de l'exécution.

Les données collectées peuvent être stockées dans un tampon ou communiquées à l'unité centrale[3]. Dans tous les cas, elles alimentent le consultant de performances qui se base sur le "*W3 search*

[3]Dans [**Waheed et al., 1996**] une étude est réalisée la consommation de ressources liée aux communications entre les démons et l'unité central de Paradyn. Il définit et justifie l'emploi d'une nouvelle politique de communications des

model" [**Miller et al., 1994a**, **Miller et al., 1994b**] pour identifier les problèmes de performances. Pour cela trois questions sont posées : *pourquoi* ("*why*") l'application est inefficace ; *où* ("*where*") se trouve les problèmes de performance ; *quand* ("*when*") est-ce que ces problèmes ont lieu.

Pourquoi ("*why*") l'application est inefficace
Le système inclut des hypothèses sur les causes potentielles de manque de performances. Ces causes de problèmes sont représentées par des hypothèses et des tests.

Où ("*where*") se trouve les problèmes de performance
L'analyse de performances est réalisée en fonction des ressources (parties du programmes ou foci) qui posent problèmes.

Quand ("*when*") est-ce que ces problèmes ont lieu
L'exécution est décomposée en phases. L'hypothèse est faite que c'est lors des changements de phase que les problèmes de performances surgissent.

Parmi les exemples d'utilisation de Paradyn, nous pouvons citer le travail de [**Miller et al., 1994b**] sur l'étude d'applications réparties utilisant PVM pour communiquer, ainsi que les travaux de [**Newhall et Miller, 1998**, **Newhall et Miller, 2000**] portant sur l'instrumentation dynamique de JVM "*byte code*".

Positionnement de notre travail :
Nous considérons Paradyn comme étant un des projets des plus intéressants, et des plus avancés, en ce qui concerne l'insertion dynamique et automatique de code d'instrumentation. De plus, le couplage réalisé avec le consultant de performances aide à l'identification automatique de problèmes de performances classiques. Toutefois, les analyses présentées ne prennent ni en compte la dynamique de l'exécution telle que nous l'avons définie dans la section 4.4, ni les aspects multi-niveaux (voir section 5.1). De plus, les problèmes de cohérence et d'ordonnancement des observations dans un système distribué ou parallèle ne sont pas abordés.

1.4.6 Tau (τ) : environnement d'observation et de contrôle pour les programmes scientifiques parallèles

"*Tuning and Analysis Utilities*" (TAU ou τ) est un environnement d'observation et de contrôle développé pour l'étude de programmes scientifiques parallèles. Il intégre des fonctionnalités de débogage, d'analyse de performances et de visualisation.

Il est conçu selon [**Mohr et al., 1996**] pour :

observations. Il démontre qu'il est préférable de stocker les observations dans un tampon et de les envoyer par paquet plutôt que une à une.

donner une vue utilisateur de l'exécution en termes de classes, méthodes, fonctions qui s'exécutent.

prendre en compte les langages de programmation parallèles d'un haut niveau d'abstraction en réalisant des liens entre les observations de bas niveau (d'abstraction) et les concepts du langage.

s'intégrer avec des environnements d'exécution et de compilation.

permettre la portabilité et l'extensibilité de façon à pouvoir facilement prendre en compte de nouveaux environnements d'exécution.

augmenter "l'utilisabilité" par un couplage avec des outils graphiques d'analyse.

L'instrumentation peut être manuelle ou automatique. L'instrumentation manuelle consiste à insérer des "macros" dans le code source de l'application [**Mohr et al., 1996**, **Shende et al., 1998**], ou les librairies utilisées [**Shende et al., 2001**, **Shende et Malony, 2001**]. Elles sont interprétées lors de la compilation. L'instrumentation automatique [**Shende et al., 2001**] est réalisée en utilisant, par exemple, DyninstAPI [**Buck et Hollingsworth, 2000**] comme c'est le cas pour Paradyn 1.4.5. Les données peuvent donc provenir du code de l'application, des bibliothèques, mais aussi des compteurs matériels présents dans certains processeurs [**Shende, 1999**] via les bibliothèques PCL [**Berrendorf et Ziegler, 1998**] ou PAPI [**Browne et al., 2000**].

Une architecture est proposée dans [**Sheehan et al., 1999**] quant à la mise en œuvre de l'observation (voir figure 1.9). Un agent d'observation est présent dans l'espace d'adressage de l'application. C'est lui qui réalise le lien avec l'extérieur. Les données observées sont placées dans une zone spécifique. Un verrou permet de garantir un accès exclusif en écriture, pour la mise à jour des données, ou en lecture, pour la communication des données à un client par l'agent d'observation. Le client réalise alors les traitements d'analyse et de mise au format pour la visualisation.

L'environnement τ possède des modules de navigation, d'analyse et de visualisation des données collectées [**Mohr et al., 1996**]. Toutefois, il offre la possibilité d'utiliser d'autres outils de visualisation [**Shende et Malony, 2001**] tel que VAMPIR [**Brunst et al., 2001**].

A titre d'exemple, nous pouvons citer les travaux de [**Mohr et al., 1996**, **Shende et al., 1998**] où τ a été mis en œuvre pour l'étude d'applications parallèles écrites en C++ [**Mohr et al., 1996**, **Shende et al., 1998**], et ceux de [**Shende et Malony, 2001**] pour des applications Java réparties.

Positionnement de notre travail :
τ est sans doute le projet le plus proche de notre problématique car :

- il aborde le domaine de l'observation d'application à objet répartie.
- il intègre des observations provenant de multiples niveaux d'abstraction (par exemple Java et MPI [**Shende et Malony, 2001**]).

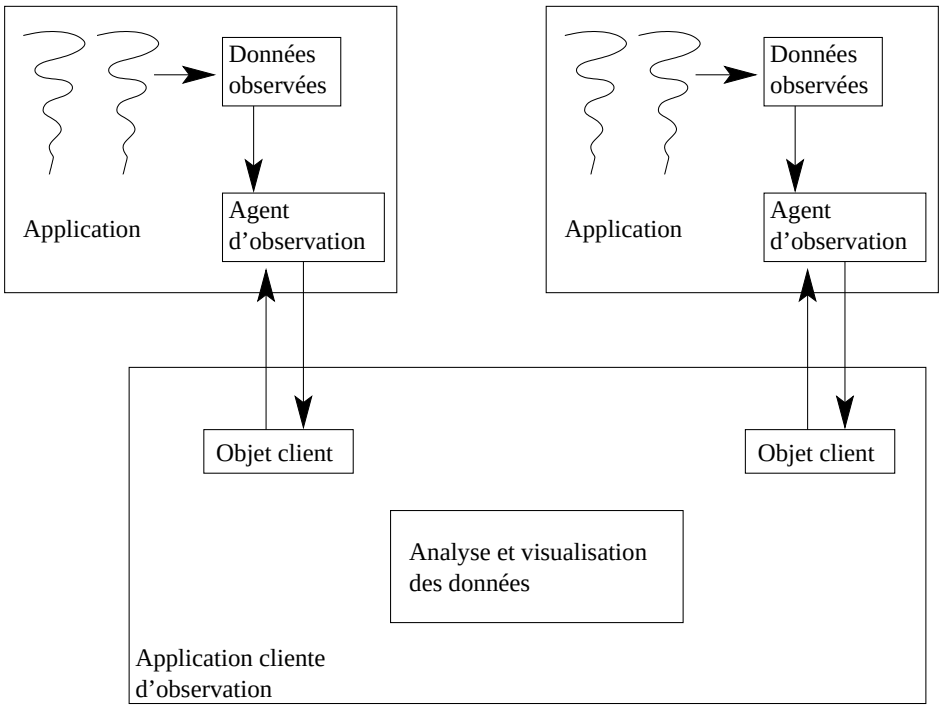


FIG. 1.9 – Architecture générale de Tau.
Les observations sont placées dans une zone de stockage. Cette zone est accédée en exclusion mutuelle pour garantir la cohérence de son contenu. L'agent d'observation, présent dans chaque application observée, fait le lien avec l'extérieur. Du côté de l'application de traitement des observations, un objet client est en relation avec chaque objet d'observation. Plusieurs objets clients peuvent être utilisés en même temps.

– il s'interface avec Vampir, un outil qui permet de représenter la dynamique de l'exécution et des indices statistiques s'y rapportant.

Néanmoins, la prise en compte des problèmes de cohérence et d'ordonnancement des observations n'est pas intégrée à l'environnement. De plus, dans le cas Java, la répartition est observée grâce à l'emploi d'une version Java des bibliothèques de communication MPI (mpiJava). C'est au niveau de cette bibliothèque qu'est réalisée la mise en cohérence des communications avec l'application. En restant à un niveau d'exécution utilisateur, il est coûteux d'obtenir des informations de consommations des ressources d'exécution. En effet, leur obtention demande d'accéder à l'espace d'adressage du noyau, ce qui génère des changements de contexte supplémentaires.

1.5 Positionnement de notre infrastructure par rapport aux travaux présentés

La réalisation de ce travail de thèse a été inspirée par un certain nombre des projets présentés. En ce qui concerne l'observation du niveau système (voir section 5.1.2), la solution de déroutage des appels système s'inspire directement des mécanismes mis en œuvre dans Paradyn (voir section 1.4.5).

Bien que nous n'ayons pas eu connaissance des travaux de [**Shende et Malony, 2001**] au moment de la réalisation de notre infrastructure, notre approche pour l'observation de l'exécution des JVM (voir section 5.1.1) est tout à fait similaire à celle retenue par τ (voir section 1.4.6). Pour ce qui est de la conversion des traces dans un format interprétable par "Pajé", il s'agit de la même problématique que celle traitée dans le cadre de SvPablo avec le format de trace SDDF (voir section 1.4.3) ou dans MODINOS, Paradyn et τ (voir section 1.4.2, 1.4.5 et 1.4.6) qui permettent de choisir un outil de visualisation extérieur.

1.6 Organisation du document

La suite du document est organisée en six chapitres. Le chapitre 2 présente les modèles de traitement d'informations, d'exécution et de ressources utilisés dans nos analyses. Les chapitres 3 et 4 correspondent à une analyse théorique de l'instrumentation et des traitements à appliquer aux observations pour construire une trace événementielle multi-niveaux représentant l'exécution d'une application à objets répartie. Les chapitres 5 et 6 répondent aux deux chapitres précédents en présentant l'implantation (voir chapitre 5) et la validation (voir chapitre 6) d'une infrastructure d'observation pour application Java distribuée. Enfin le chapitre 7 conclut la présentation de nos travaux et ouvre les perspectives dans lesquelles il nous paraît intéressant de s'engager.

Chapitre 2

Modèles pour l'exécution d'application répartie

> « *Cette modélisation, selon l'élégante expression du physicien français Perrin, sert à "substituer au visible compliqué de l'invisible simple.*" » [**Thom, 1980**]

Ce chapitre a pour objectif de présenter différents modèles de système représentant l'exécution d'une application répartie sur un ensemble de systèmes de traitement d'informations (voir section 2.1). Le modèle d'application considéré repose sur la notion d'*objets répartis*. L'objet forme la brique de base dans l'architecture logique de l'application. Il sert à découper et à isoler les différentes fonctionnalités et comportements de l'application.

Dans un modèle à objets répartis, une application consiste en un ensemble d'objets qui interagissent [**Zieliński et al., 1995**]. Une interaction fait référence à un changement d'état provoqué par l'extérieur de l'objet (voir section 2.2). Elle prend place sur une des interfaces exportées par l'objet (voir figure 2.1).

L'exécution d'un objet s'effectue au sein d'un «site d'exécution». Il correspond à une entité logique qui contient les ressources nécessaires à l'exécution des objets. Il s'agit, par exemple, d'un processus ou d'une JVM. Il n'existe pas obligatoirement de correspondance entre un site d'exécution et un lieu physique de l'exécution (un processus peut être ordonnancé sur l'ensemble des processeurs présents dans une machine multi-processeurs).

Les ressources utilisées par l'application sont nommées «**ressources d'exécution**». Dans notre travail, nous les divisons en deux principaux types : les ressources de calcul et les ressources d'interaction (voir les détails en fonction des niveaux dans la section 2.3). La gestion de ces ressources est réalisée par le système d'exploitation des sites hôtes et cela indépendamment du modèle d'exécution considéré.

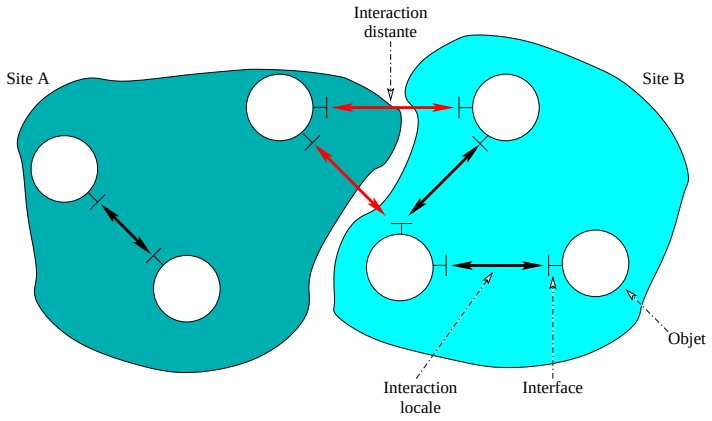


FIG. 2.1 – Modèle général d'application à objets répartis.

Le comportement d'une application se modélise alors par l'enchaînement de calculs et d'interactions (voir figure 2.2). Les interactions sont, du point de vue de l'exécution, locales ou distantes. Elles sont locales si les objets appartiennent au même site d'exécution. Elles sont distantes dans les cas contraires. Dans tous les cas, elles nécessitent la mise en œuvre d'une liaison pour relier les protagonistes.

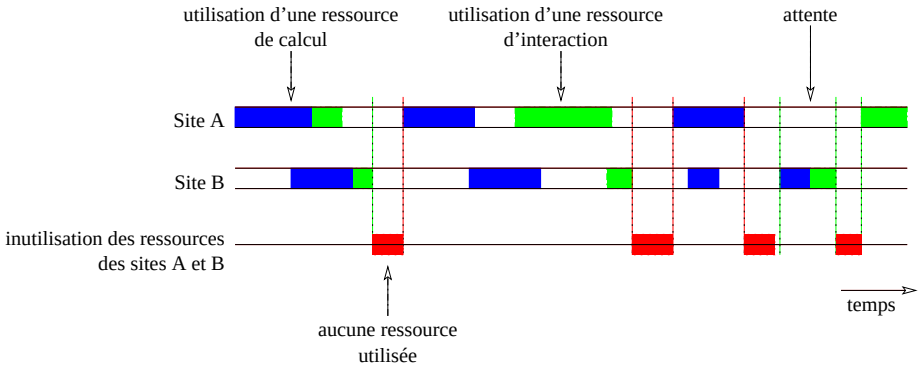


FIG. 2.2 – Entrelacement des calculs et des interactions.
L'utilisation du système est qualifiée en mesurant l'utilisation des ressources et donc par les périodes d'inactivité.

Dans le cadre de *l'analyse quantitative* d'une exécution, nous cherchons à faire apparaître les «*modes*» de consommation des ressources. L'objectif de ce type d'analyse est triple : a) mettre en évidence des situations de pénurie de ressources entraînant des problèmes de performances (des *goulots d'étranglement*), b) identifier les ressources mises en cause, c) qualifier leur utilisation[1]. L'approche que nous avons adoptée, pour mener à bien ce type d'analyse, repose sur la mise en œuvre d'une infrastructure d'observation, de collecte et d'analyse de données quantitatives multi-niveaux. De cette façon, nous pouvons identifier des schémas types d'exécution et repérer d'éventuels problèmes de performance.

[1]Il s'agit typiquement d'estimer le taux d'utilisation d'une ressource afin de juger de la façon dont le système est dimensionné.

2.1 Modèles de systèmes de traitement d'informations

D'un point de vue formel, si un système correspond à un «ensemble cohérent de notions, de principes liés logiquement et considérés dans leur enchaînement» [**Hac, 1997**] c'est-à-dire qu'il renvoie à un «ensemble d'éléments en interaction telle qu'une modification quelconque de l'un entraîne une modification de tous les autres» [**Bertalanffy, 1973**], alors il est caractérisé par les transformations qu'il opère sur les données en entrée et les sorties obtenues. Trois ensembles sont donc à définir : l'ensemble des entrées (i.e. le domaine des données du programme), l'ensemble des transformations (i.e. le programme en lui-même) et l'ensemble des sorties (i.e.le domaine des résultats attendus).

D'autre part :

> « *Un **système informatique** est un ensemble de matériels et de logiciels destinés à réaliser des tâches qui mettent en jeu le traitement automatique de l'information. Un tel système est relié au monde extérieur par des organes d'accès, qui lui permettent de communiquer avec des usagers humains ou d'interagir avec des dispositifs physiques qu'il est chargé de commander ou de surveiller.* » [**Krakowiak, 1985**]

La jonction de ces deux définitions précise les contours du domaine étudié. Il s'agit d'énoncer les problèmes, les moyens de les aborder, le domaine des résultats attendus et de réaliser les traitements à l'aide de moyens automatisés : le système informatique[2]. **L'objectif corollaire de l'évaluation de performance est de quantifier l'utilisation du système informatique afin d'en optimiser l'utilisation**. Il s'agit ici de *minimiser des fonctions de coût* de type : temps d'exécution ou utilisation des ressources d'exécution. A titre d'exemple, considérons un système réparti à l'intérieur duquel les communications reposent sur un réseau Ethernet. Si l'on cherche à minimiser la durée des communications, la fonction coût correspondante est formée à partir des composants modélisant l'accès et l'utilisation du réseau (les "*sockets*", les piles protocolaires, ...) et elle mesure le débit et la latence des communications.

2.1.1 Architecture générale

Un système est modélisé par un ensemble de composants. Ils correspondent à des abstractions des ressources d'exécution mises à disposition de l'application. Nous distinguerons deux grands types de composants : les composants de calcul et les composants d'accès aux données[3].

Les composants de calcul

Ils représentent les "moteurs de traitements". Au niveau du matériel, il s'agit par exemple des processeurs ou des contrôleurs de périphériques. Au niveau du système d'exploitation, les composants de calcul font référence aux mécanismes qui permettent de gérer l'exécution d'un

[2] Dans la suite du document l'emploi de "système" fera référence à un "système informatique".

[3] Il s'agit d'une dénomination générique qui englobe les communications.

programme : les processus, les "*threads*", l'ordonnanceur, ou les mécanismes de synchronisation (voir section 2.3.1). Au niveau applicatif, on retrouve les mêmes abstractions qu'au niveau du système d'exploitation plus les notions de services ou serveurs qui prennent à leur charge une partie du travail de façon opaque pour l'application.

⇒ *Les indices de performance qui leur sont associés représentent, par exemple, la durée d'utilisation du processeur, le nombre de changement de contexte des processus dans les différents modes ou le nombre d'objets de synchronisation en attente, etc.*

Les composants d'accès aux données

Ils représentent les mécanismes de stockage et de communication des données (voir section 2.3.2). Au niveau matériel, il est ici fait référence aux différents types de mémoire (registre, cache, mémoire physique (RAM, ROM, flash...), unité de stockage (disque dur, CD-ROM, DVD...)) ainsi qu'à l'infrastructure de communication (réseau filaire, radio). Au niveau du système d'exploitation, il s'agit par exemple des mécanismes d'adressage, de gestion de niveaux de cache, ou des protocoles de communication. Au niveau applicatif, ces composants représentent les moyens d'écritures et de lecture associés aux données.

⇒ *Les indices de performance communément présentés correspondent à la quantité de mémoire consommée, au temps de lecture et écriture ou encore à la bande passante du média de communication utilisée ou à sa latence.*

2.1.2 Architecture physique et fonctionnelle

Les systèmes peuvent être classés selon des critères physiques ou fonctionnels.

L'architecture physique d'un système [**Silberschatz et Galvin, 1994**]

Trois catégories de système sont identifiées :

Centralisée

Un système centralisé est constitué de composants fortement connectés, c'est-à-dire reliés par un bus de communication de faible envergure tel que l'on peut le trouver dans une station de travail.

Parallèle

Un système parallèle est composé de plusieurs systèmes centralisés, même minimaux (processeur et mémoire), partageant un bus de communication, parfois une horloge et quelquefois de la mémoire et des périphériques. Un tel système est appelé système fortement couplé.

Réparti

Un système réparti est composé de plusieurs systèmes centralisés ou parallèles reliés par des connexions potentiellement de grande longueur (LAN ou WAN). Il n'est fait aucune hypo-

thèse d'homogénéité des composants du système, ni de la présence de mémoire ou d'horloge partagée. Par opposition aux systèmes parallèles, ces systèmes sont dits faiblement couplés.

En ce qui concerne l'évaluation de performances, la nature physique du système joue sur les mécanismes à mettre en œuvre pour l'observation (voir chapitre 3). Par exemple, si l'on se place dans le cadre d'un système réparti, et que l'on cherche à mesurer les durées de communication, il est nécessaire de construire un système de datation globale afin de comparer les dates d'envoi et de réception des messages.

L'architecture opérationnelle d'un système

Elle représente l'organisation des abstractions logiques qui permettent d'accéder aux composants physiques. C'est le rôle qui est dévolu au *système d'exploitation*. La définition d'une abstraction permet de se focaliser sur un sous ensemble des caractéristiques ou des fonctionnalités du composant. Il s'agit d'extraire ce qui est significatif dans un contexte donné. Par exemple, si l'on considère les opérations de lecture ou d'écriture sur un disque dur, le système d'exploitation va fournir les abstractions de lecture et d'écriture quels que soit le disque considéré et donc les opérations de bas niveaux mises en oeuvre pour ces opérations. Dans ce cas, les opérations de bas niveaux sont définies dans le pilote du périphérique ("*driver*")[4] du disque.

> « *Un système d'exploitation offre une machine virtuelle à l'utilisateur, et aux programmes qu'il exécute. Il s'exécute sur une machine physique qui possède une interface de programmation de bas niveau, et il fournit des abstractions de haut niveau et une interface de programmation et d'utilisation plus évoluée.* » [**Card et al., 1998**]

Par symétrie aux architectures physiques, il est possible d'identifier deux catégories de système d'exploitation :

Les systèmes d'exploitation centralisés

Ils correspondent à une situation où les abstractions sont définies et accédées dans un même système physique centralisé.

Les systèmes d'exploitation répartis

Ils correspondent à une situation où les abstractions sont définies de façon locale sur chacun des sites, et accessibles par tous.

> « *Un système réparti offre à l'utilisateur la possibilité d'accéder aux différentes ressources situées dans les sites éloignés* » [**Silberschatz et Galvin, 1994**]

Ces différentes définitions posent le cadre de l'exécution des applications. Ainsi, dans une architecture centralisée, l'exécution a lieu dans un seul et même site physique, et cela grâce aux seules

[4] Le pilote d'un périphérique définit les sous-routines qui lui sont spécifiques. Un pilote sait comment les tampons, les drapeaux, les bits de contrôle et les bits d'état d'un périphérique particulier doivent être utilisés [**Silberschatz et Galvin, 1994**].

ressources présentes localement. Par contre, une exécution ayant lieu dans un système parallèle ou réparti peut prétendre à plus de ressources que celles présentes sur son site d'exécution d'origine.

La principale différence, de nature opérationnelle, qu'il est possible de réaliser entre un système parallèle et réparti, repose sur le mode d'utilisation des ressources. Dans un système parallèle, l'ensemble des ressources est mis à la disposition d'un seul utilisateur. A contrario, lors d'une exécution répartie, les ressources du système sont dynamiquement partagées de façon "équitable" entre les utilisateurs présents ou à venir.

La problématique du partage des ressources se retrouve dans les systèmes partagés [**Silberschatz et Galvin, 1994**, **Krakowiak, 1985**, **Card et al., 1998**], c'est-à-dire à l'intérieur desquels plusieurs exécutions ont lieu en même temps, qu'ils soient centralisés, parallèles ou distribués. Ce partage est la cause de nombreux problèmes de performances car les différentes exécutions sont en compétition pour les mêmes ressources. Dans ce cas, les fonctions de coût construites doivent tenir compte des interdépendances. Pour analyser la consommation de ressources d'une application, il devient non seulement nécessaire de mesurer ses propres consommations, mais aussi de tenir compte du contexte dans lequel l'exécution prend place. En effet, dans le cas d'un système partagé, un blocage sur une entrée/sortie peut, par exemple, être dû à un verrou posé par une autre exécution qui accède à la même ressource ((voir figure 2.3)).

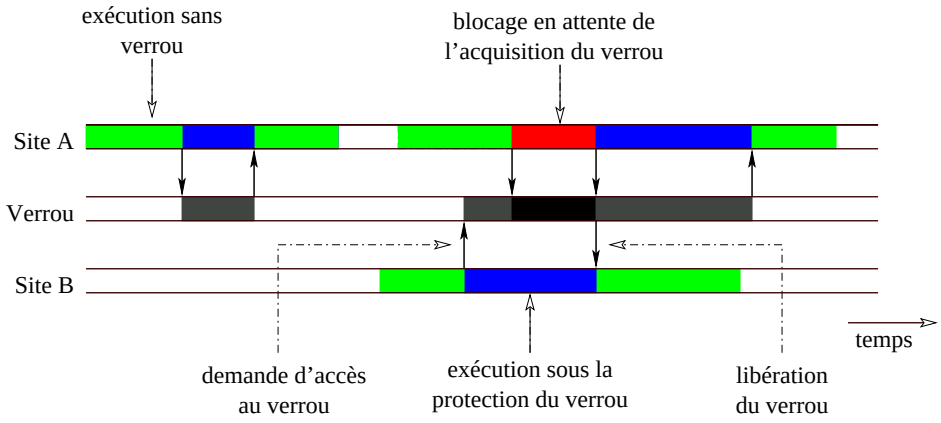


FIG. 2.3 – Blocage en attente de la libération d'un verrou.
Un mécanisme de verrou est utilisé pour garantir l'unicité d'accès à une ressource, ou l'exécution séquentielle d'une partie de l'application. Lorsque deux exécutions tentent d'acquérir le même verrou, l'ordre d'accès est sérialisé. Lorsque le site B est en possession du verrou, le site A doit attendre la libération de ce dernier pour pouvoir l'acquérir.

Les problèmes liés au partage de ressources sont projetés au niveau du modèle d'exécution qui nous aide à gérer la concurrence et les exécutions simultanées.

2.2 Le modèle d'exécution de RM-ODP

L'étude présentée porte sur un modèle d'exécution à base d'objets communiquants. Le modèle RM-ODP ("*Reference Model-Open Distributed Processing*") offre un cadre conceptuel générique pour la définition et l'implantation de systèmes répartis [**Iec, 1995a**]. Ce modèle a été notamment mis en œuvre dans le projet Jonathan développé à France Télécom R&D [**Dumant et al., 1998**].

Tout système conforme au modèle RM-ODP est construit comme une configuration ou une composition d'objets qui interagissent. L'environnement d'exécution et de communication dans lequel se trouvent les objets est a priori hétérogène.

Dans ce modèle, un système réparti est représenté comme un ensemble de composants qui interagissent. Les composants peuvent être localisés sur des sites physiques distincts. Dans ce cas, les interactions entre composants sont supportées par des communications qui peuvent être longues ou même défaillantes [**Iec, 1995b**].

Un composant correspond à un objet[5] ou une agrégation d'objets au sens logiciel du terme. L'objet est caractérisé par un identifiant unique, une capsule et un comportement. La capsule de l'objet délimite sa portée et son comportement définit les états auxquels il peut accéder. La granularité d'un objet est fixée par la notion d'*atomicité*. Un objet est considéré comme atomique, à un niveau d'abstraction fixé, lorsque ce dernier ne peut plus être subdivisé. **La granularité des objets fixe celle des observations et donc la précision des mesures réalisées.**

Le changement d'état d'un objet représente une action. Les actions associées à un objet sont décomposées en actions internes et en **interactions**. Une action interne représente un changement d'état spontané de l'objet, sans intervention de l'extérieur. **Une interaction fait référence à un changement d'état provoqué par l'extérieur de l'objet via une de ses interfaces.**

L'état d'un objet, à une date t, est défini par les conditions et contraintes qui lui sont associées[6]*. Ainsi l'état d'un objet est construit à partir de l'observation de son contexte plutôt que par l'observation de l'objet lui-même. Il s'agit d'une approche contextuelle où l'on se concentre sur les relations plutôt que sur l'objet.*

L'objet est encapsulé, ce qui signifie que son état n'est observable de l'extérieur que s'il possède une interface dédiée à cet usage (voir figure 2.4) ou si le support d'exécution est réflexif[7]. Dans le cas contraire, la seule façon de modéliser son comportement est d'observer ses interactions avec l'extérieur. En effet, chaque interaction est rattachée à une interface unique. Il est donc possible, au

[5] Un objet représente tout ce qui peut être pris en considération lors d'une exécution, que ce soit concret ou abstrait.

[6] Cette définition diffère de celle communément employée dans laquelle l'état est représenté par un vecteur de valeurs représentatives.

[7] Les mécanismes de réflexion ou d'encapsulation que nous avons testés semblent actuellement coûteux à mettre en œuvre et à utiliser, ce serait toutefois une piste intéressante à explorer.

travers de l'observation des interactions, de savoir quelle interface est utilisée et donc quelle est la sémantique de l'interaction en cours.

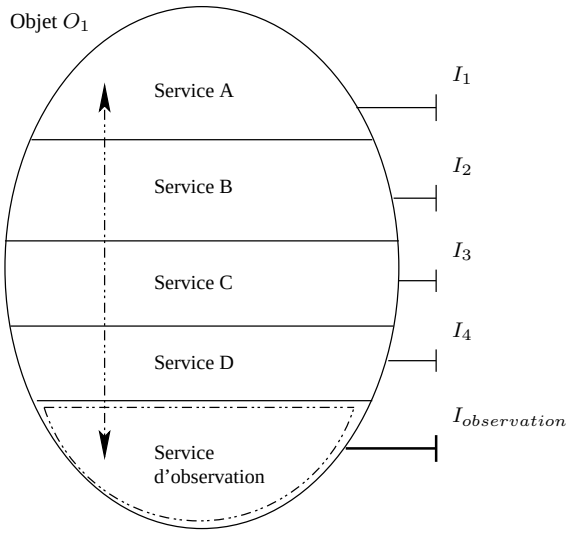


FIG. 2.4 – Représentation d'un objet et de ses interfaces.

Une interaction entre interfaces n'est possible que si un lien a été établi. Ici, un lien représente un chemin de communication entre interfaces [**Stefani, 1995**]. Une communication est définie comme un déplacement d'informations entre deux ou plusieurs objets. Ce déplacement résulte d'une ou plusieurs interactions qui peuvent faire intervenir des objets intermédiaires (voir figure 2.5).

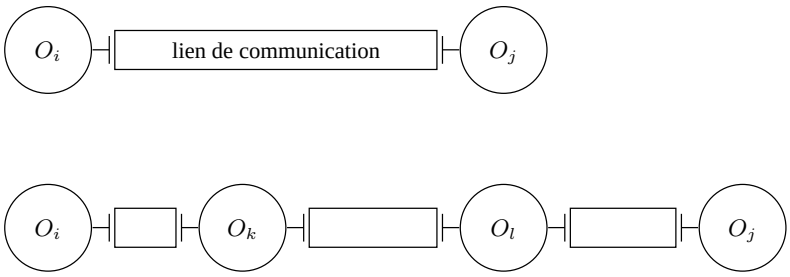


FIG. 2.5 – Lien entre interfaces pour le support d'une interaction.
Pour que deux objets O_i et O_j puissent interagir, il est nécessaire de créer au moins un lien de communication entre deux de leurs interfaces.

Pour établir ce lien, un mécanisme de désignation est nécessaire. Il nomme de façon unique les objets au sein d'un contexte de désignation qui met en relation un ensemble de noms bien formés avec un ensemble d'entités [**Iec, 1995a**]. Le lien peut être automatiquement créé par l'infrastructure de communication, ou manuellement via la création d'un objet de liaison ("*binding object*").

Dans ce cadre, une liaison représente une relation entre objets qui résulte de la définition d'un comportement de communication entre les différents protagonistes. L'implantation d'une liaison est réalisée à l'aide d'une usine à liaisons [**Stefani et al., 1998**, **Dang Tran et al., 1996**] (voir figure

2.6). Elle fabrique des instances de liaisons typées. Le type de la liaison est donc représentatif de son comportement. Par exemple, si l'on souhaite la présence d'un mécanisme de contrôle des erreurs sur les communications, une liaison de type TCP peut être établie. Si les objets à faire communiquer appartiennent au monde CORBA, les interactions sont alors de type IIOP (GIOP sur TCP/IP) [**Geib et al., 1999**].

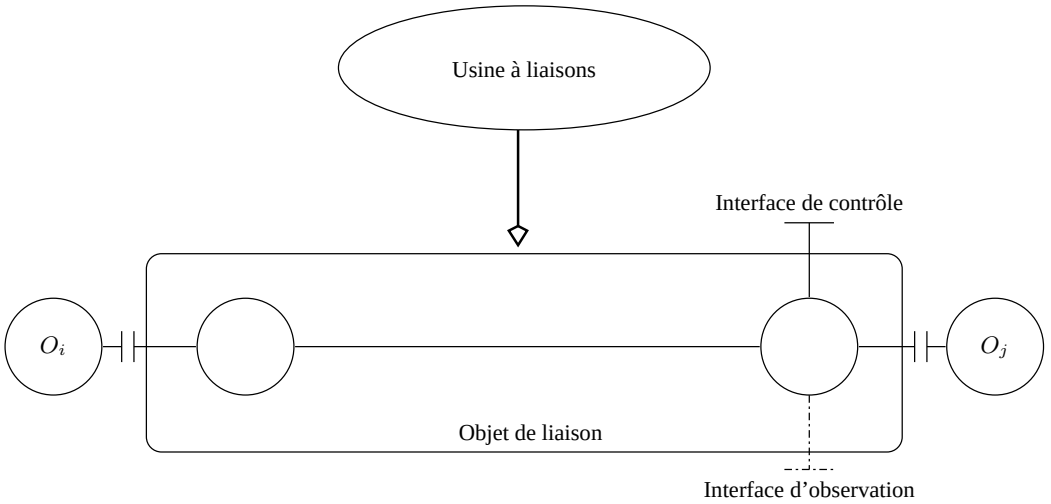


FIG. 2.6 – Création d'un objet de liaison typé.
Un objet de liaison représente une instance de communication typée. Il est créé par une usine à liaisons qui détermine sa sémantique et son comportement.

Comme à tout objet, il est possible d'associer des critères de qualité de service à l'objet de liaison. Il délimite de fait la portée des contraintes de QoS à respecter (i.e. toutes les liaisons ne doivent pas forcément répondre aux mêmes niveaux de QoS). L'objet de liaison procure une abstraction de bout en bout qui encapsule les différentes ressources et mécanismes utilisés. De cette façon la QoS n'est plus seulement prise en compte au niveau du réseau de communication, mais elle est aussi traitée au niveau du système servant de support aux objets qui communiquent.

Du point de vue de l'évaluation de performances, l'objet de liaison identifie la source d'information attachée aux interactions.

Observer le comportement et la qualité des interactions revient donc à observer les objets de liaison qui s'y rapportent.

Dans le modèle RM-ODP, les interactions entre objets sont essentiellement asynchrones [**Stefani, 1995**]. Elles prennent deux formes : opérationnelle (elle correspond à un appel de fonction sur une interface) et de flux (elle correspond à une abstraction représentant des séquences de données continues entre interfaces). Dans ce cas, le modèle d'interactions asynchrones est le reflet d'un ensemble d'hypothèses réalisées sur les communications. Ces hypothèses sont : pas d'horloge globale, pas d'accès à un état global, des durées d'exécution arbitrairement longues, des délais de communi-

cation non bornés, et des probabilités non nulles de défaillance. Ces deux modes d'interaction sont étudiés au travers de l'exemple présenté dans la section 6.5.

Nous avons vu que pour étudier le comportement et la qualité des interactions, il nous faut observer les objets de liaison associés. Ces objets utilisent des ressources pour s'exécuter. La section suivante présente le modèle de ressources qui leur est associé.

2.3 Modèles de ressource

> « *Nous appelons **ressource** tout objet pouvant être utilisé par un processus. A une ressource sont toujours associées des procédures d'accès, qui permettent de l'utiliser, et des règles d'utilisation qui constituent son "mode d'emploi".* » [**Krakowiak, 1985**]

Les ressources correspondent aux entités de base sur lesquelles repose l'exécution. Elles sont gérées et partagées, dans les environnements multiprogrammés, par un «allocateur de ressources». Le partage d'une ressource peut générer des situations pathologiques de privation (problèmes d'application de la politique d'allocation de ressources) ou d'interblocage (problèmes algorithmiques d'attente de libération de ressources) à l'origine de problèmes de performances.

Par l'observation des ressources nous jugeons de leur utilisation. Pour cela, un critère d'évaluation est défini. Il peut, par exemple, caractériser l'équilibre d'accès entre les différents prétendants. Dans ce cas, les observations portent sur la ressource mais aussi sur qui et pendant combien de temps elle est utilisée. Par contre, si l'on ne s'intéresse qu'à l'utilisation brute de la ressource, les informations liées au contexte ne sont pas prises en compte.

Les ressources sont classées en deux catégories : **les ressources de calcul** et **les ressources d'interaction**. Les deux sections suivantes proposent de définir chacune d'entre elles dans les niveaux d'abstraction matériel, système d'exploitation et applicatif retrouvés dans un système informatique.

2.3.1 Les ressources de calcul

Les ressources de calcul correspondent à l'ensemble des moyens mis en œuvre pour mener à bien les actions de traitement.

Au niveau matériel
Les ressources de calcul correspondent au processeur, "*Digital Signal Processing*" (DSP) et autres contrôleurs qui permettent la transformation des données.

Au niveau système d'exploitation
Les ressources de calcul sont des abstractions des ressources définies au niveau matériel. L'abs-

traction la plus directe correspond à une exécution séquentielle d'instructions par le processeur. Elle est nommée *activité* [**Krakowiak, 1985**]. Dans ce modèle, l'exécution d'un programme correspond à une suite d'activités.

Cette suite d'activités est aussi appelée processus. Le terme processus est défini dans [**Silberschatz et Galvin, 1994**] comme un programme en exécution. Il s'agit non seulement du code du programme, mais aussi de la valeur du compteur d'instructions ainsi que de l'état des registres utilisés. Il englobe généralement aussi la pile d'exécution qui contient les données temporaires et une section de données qui contient les variables globales.

> « *Un* "thread"*, quelquefois appelé* processus de poids léger *(en anglais* "Lightweight process (LWP)"*) ou fil d'exécution* [8]*, est une entité de base d'utilisation de l'unité centrale. Il est constitué d'un compteur d'instructions, un ensemble de registres et un espace de pile. Il partage avec des threads ressemblants, sa section de code et de données et les ressources du système d'exploitation comme les fichiers ouverts ou les signaux. Ils sont connus collectivement comme une* tâche*. Un processus traditionnel ou de* poids lourd *(en anglais* "heavyweight process"*) correspond à une tâche avec un seul* "thread"*.* » [**Silberschatz et Galvin, 1994**]

Cette définition peut aisément être étendue par le fait qu'un processus traditionnel correspond à une tâche avec au moins un "*thread*".
L'entrelacement des exécutions offre l'opportunité de recouvrir les temps de communication par du calcul. Ainsi, lorsqu'un processus, ou un "*thread*", est en attente de la fin d'une entrée/sortie ou d'une communication, un autre processus, ou un autre "*thread*", peut utiliser les ressources de calcul pour réaliser les traitements qui lui sont associés. Cette possibilité est d'autant plus intéressante dans le cas des "*threads*" que leur coût de commutation de contexte est faible par rapport à celui d'un processus.

Outre les "*threads*", il est intéressant d'associer aux ressources de calcul les abstractions liées aux mécanismes de synchronisation. Ils ne participent pas en tant que moteur d'exécution, mais en offrant la possibilité de sérialiser ou de suspendre sur condition une exécution.

Au niveau applicatif

Il s'agit une fois de plus d'abstraire les ressources du niveau inférieur pour en faciliter l'emploi. On y retrouve les abstractions de processus et "*threads*" ainsi que ceux liés aux mécanismes de synchronisation.

Il n'existe pas forcément de symétrie entre ressources du système d'exploitation et ressources applicatives. Par exemple, dans un environnement multi-programmé, si un processus possède

[8] Bien que "*thread*" soit traduit en français par «*fil d'exécution*» nous avons choisi de garder la dénomination anglophone pour le reste du document.

plus d'un "*thread*" pour s'exécuter il n'y aura pas automatiquement autant de "*threads*" système que de "*threads*" utilisateur.

Trois modèles d'implantation et d'exécution des "*threads*" utilisateur sont communément implantés [**IEEE, 1995**, **Carissimi, 1999**] (voir figure 2.7). Le modèle **1/1** dans lequel à chaque "*thread*" utilisateur correspond un "*thread*" système. Le modèle **N/1** dans lequel N "*threads*" utilisateur s'exécutent en parallèle sur le même "*thread*" système. Dans ce cas, il est nécessaire d'avoir un mécanisme d'ordonnancement des "*threads*" utilisateur indépendamment de l'ordonnancement des "*threads*" système. Enfin le modèle **N/M** dans lequel N "*threads*" utilisateur s'exécutent sur M "*threads*" système. Dans ce cas aussi, un mécanisme d'ordonnancement est nécessaire de façon à gérer l'exécution des "*threads*" utilisateur sur les "*threads*" système.

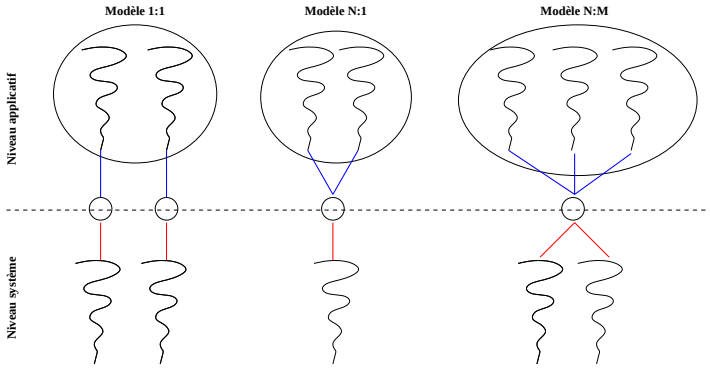


FIG. 2.7 – Modèles de "*threads*" communément implantés.

ReTINA [**Stefani et al., 1998**] définit la ressource "*thread*" comme processeur (séquentiel) logique. Il définit aussi des ressources correspondant à des primitives de synchronisation temps réel afin d'éviter des phénomènes de privation dus à l'emploi de primitives de synchronisation standard (tels que les *mutex* [**Dijkstra, 1965a**, **Dijkstra, 1965b**], les *sémaphores* [**Dijkstra, 1965a**, **Dijkstra, 1968**] ou les *moniteurs* [**Brinch Hansen, 1973**, **Hoare, 1974**]) et ainsi garantir l'ordre d'exécution ou l'accès exclusif aux ressources.

Le canevas défini par ReTINA comprend la notion de ressource "ordonnançable" et d'ordonnanceur. Une ressource "ordonnançable" est associée à un ordonnanceur qui met en œuvre une politique particulière de contrôle pour l'accès à la ressource par les "*threads*" (par exemple pour éviter les phénomènes d'inversion de priorité [**Sha et al., 1990**, **IEEE, 1995**]).

Dans ce modèle, l'observation des ressources de calcul correspond donc à l'observation de l'activité des "*threads*" (système et utilisateur) ainsi que l'utilisation des mécanismes de synchronisation (voir section 5.1).

2.3.2 Les ressources d'interaction

Du point de vue de l'objet, les données sont classées en deux catégories. Les données qui appartiennent à la capsule de l'objet et celles qui sont à l'extérieur. L'accès à une donnée à l'extérieur de la capsule n'est possible que par une interaction qui elle-même dépend de la création d'une liaison (voir section 2.2).

Suivant le modèle RM-ODP, *une liaison est modélisée par un objet de liaison*. Le modèle de ressource de RM-ODP [**Stefani, 1995**] et ReTINA [**Dang Tran et al., 1996**] définit le concept d'usine à liaisons ayant la responsabilité de construire les objets de liaison. Il s'agit d'objets typés dont la sémantique repose sur le type de communication choisi (un objet de liaison basé sur le protocole de communication TCP/IP réalise le contrôle de flux et d'erreur)[9] .

Dans ce cadre, l'observation des objets de liaison servant de support aux interactions est assimilable à l'observation des accès aux données à l'extérieur de la capsule de l'objet.

L'observation de l'accès aux données demande donc d'observer les mécanismes de gestion des données locales ainsi que les moyens de communication. L'utilisation de la ressource d'accès aux données est donc définissable dans les trois niveaux d'abstraction d'un système :

Au niveau matériel
Elle correspond à l'utilisation des différents niveaux de mémoire présents localement dans un système (i.e. registre, cache, RAM), les unités de stockage (i.e. disque dur, CD-ROM) ainsi que les liens de communication (i.e. filaire, optique, radio).

Au niveau système d'exploitation
Elle représente la mise en œuvre des mécanismes d'adressage, des protocoles de communication et des abstractions système (par exemple les "*sockets*") pour l'instanciation des liens de communications.

Au niveau applicatif
Elle correspond à l'utilisation de l'abstraction mémoire fournit par le système d'exploitation, ainsi que l'emploi des objets de liaison servant pour les interactions nécessaires lors de l'accès à des données hors de la capsule de l'objet.

Dans ce modèle, l'observation de la ressource d'interaction correspond donc à l'observation de la mémoire vive, des défauts de pages, des entrées/sorties locales et des abstractions de communication

[9] De même, la vision "*Engineering*" d'ODP [**Iec, 1995b**] définit la notion d'objet de protocole. Cet objet procure une abstraction des ressources de communication et des protocoles utilisés pour interconnecter des noeuds. Un nœud est défini comme une abstraction des ressources pour le traitement d'information au quel est associé un système d'exploitation minimal.

telles que les "*sockets*" (voir section 5.1).

2.4 Conclusion

Le contexte général dans lequel se place notre travail est donc l'observation d'applications à objets réparties. Pour cela nous avons défini un modèle de traitement d'informations qui représente l'architecture physique et fonctionnelle des systèmes servant de support à l'exécution.

Puis, nous avons présenté le modèle d'exécution de RM-ODP dans lequel *observer le comportement et la qualité des interactions revient à observer les objets de liaison qui s'y rapportent.*

Enfin, nous avons structuré un modèle de ressources d'exécution considérant des ressources liées au calcul et des ressources propres aux interactions. Ces deux types de ressources servent de moteur au déroulement des exécutions.

Il nous faut à présent traiter des méthodes et moyens d'observation dédiés aux applications à objets réparties.

Chapitre 3

Définition et conception d'une infrastructure d'observation

L'observation est une activité essentielle lorsque l'on souhaite obtenir des informations sur l'état ou l'exécution d'un système. Dans le cadre de ce travail, nous nous intéressons au "*monitoring*" de système à composants distribués. Le "*monitoring*" est défini par Joyce [**Joyce et al., 1987**] comme le processus de collecte, d'interprétation et de présentation d'informations se rapportant à un système observé. Il s'agit donc d'un processus passif, en opposition au contrôle qui agit sur l'état et l'exécution du système. Cette activité induit un certain nombre de problèmes parmi lesquels nous distinguerons :

La «fraîcheur» des données
: Des délais trop importants entre l'obtention de l'information et son traitement peuvent rendre difficile la construction d'une vue consistante du système.

Le nombre des composants observés
: Il peut être important et donc impliquer une grande quantité d'informations à traiter.

Le partage des ressources
: Elles sont partagées entre l'application et l'environnement d'observation, ce qui se traduit par une perturbation de l'exécution.

Afin de répondre à ces problèmes, nous avons choisi de construire une infrastructure d'observation dont on maîtrise le comportement. Comme il n'est pas possible de tout observer, cette infrastructure doit être paramétrable afin de limiter la quantité de données à collecter et en donnant des garanties sur leur qualité.

> « *Nous passons donc d'un concept géométrique, celui du corps continu, à un ensemble de données discontinues, de nature statistique et informelle.* » [**Perdijon, 1998**]

Cette transition nécessite la mise en place de points de mesures dans et/ou autour du système étudié. Ce sont les sources d'informations auxquelles sont associées une qualité et une sémantique propre à l'analyse menée.

3.1 L'approche multi-niveaux

L'approche multi-niveaux repose sur le découpage du système observé en niveaux d'abstraction. Il s'agit d'un découpage «arbitraire» ayant pour objectif de faciliter l'interprétation des phénomènes observés. Il est réalisé en fonction de ce que l'on cherche à observer, de l'ouverture et de la complexité du système.

Un système informatique est communément modélisé par la superposition verticale des niveaux : applicatif, système et matériel. De façon schématique, le niveau applicatif représente les applications de l'utilisateur, le niveau système les services nécessaire à leur exécution et le niveau matériel les ressources physiques mises à disposition par le niveau système (voir figure 3.1).

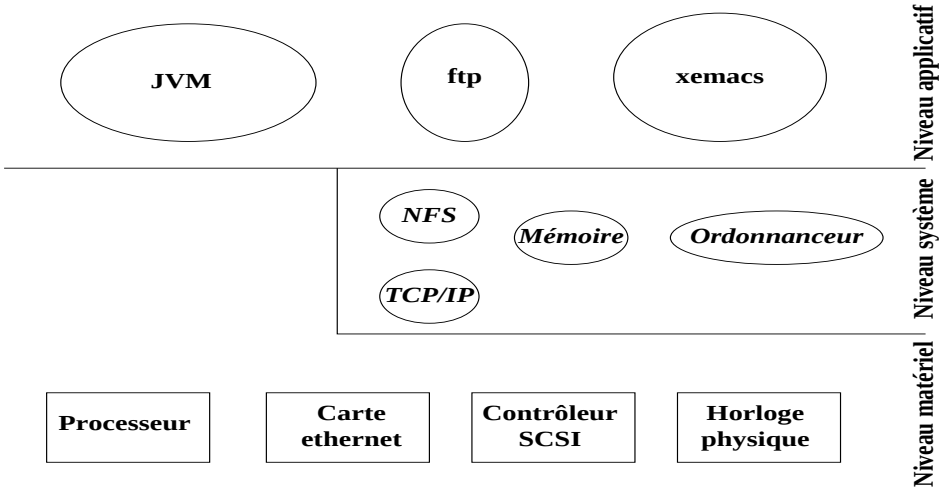


FIG. 3.1 – Exemple de modélisation du système en trois niveaux d'abstraction.
Il s'agit du modèle utilisé pour notre travail. Les Sections 3.3 et 3.4 présentent les techniques d'observation de ces différents niveaux.

La décomposition du système caractérise la portée des observations. Une observation n'est directement interprétable que dans son niveau d'origine. Ainsi, pour qu'elle ait un sens à travers l'ensemble des niveaux, il est nécessaire de construire des fonctions de projection. Ces fonctions doivent exprimer et préserver la sémantique et la qualité des informations projetées. Ainsi, si l'on prend comme exemple la réception d'un message, l'observation matérielle correspondante sera l'arrivée de paquets sur la carte réseau. Les fonctions de projection doivent être capable d'exprimer la relation entre l'événement de bas niveau : réception, de paquet et l'opération applicative : réception de message.

Pour permettre la construction de ces fonctions, il est nécessaire d'identifier de manière non ambiguë les observations ainsi que de définir une clé de projection commune à l'ensemble des niveaux. La clé qui a été choisie se base sur un système de datation globale, accessible par l'ensemble des niveaux d'abstraction et commun à la totalité du système étudié, et sur l'identification du "*thread*" système actif à la date où l'événement est enregistré. Grâce à l'association d'un système de datation et d'une désignation univoque, il nous est possible de construire une relation d'ordre causal sur les observations provenant des différents niveaux d'abstraction.

Les informations que nous traitons proviennent des composants qui constituent l'application (voir section 2.2) et le système hôte (voir section 2.1.1). Ces composants interagissent de façon locale et distante selon le modèle applicatif mis en œuvre.

La section suivante présente les méthodes d'observation communément utilisées pour l'analyse de performances.

3.2 Observation d'un composant

Le concept de composant permet de modéliser les éléments manipulés dans un système distribué. Un composant est soit un élément primaire telle qu'une ressource, soit un élément fabriqué par composition avec d'autres composants. Tout comme un objet, un composant est une structure dynamique qui encapsule un état et un comportement. Dans la suite du document, les termes «objet» et «composant» ne seront plus distingués et feront référence à la même notion.

3.2.1 Le "*monitoring*"

Le "*monitoring*" s'applique à un composant ou un ensemble de composants reliés par un sens commun ("*monitoring*" d'un domaine) [**Mansouri-Samani et Sloman, 1992**]. A chaque composant est associé un état et des événements qui correspondent à des changements d'états. Du point de vue de l'observation, le comportement du composant est caractérisé par la succession de ses états observés.

> « *L'état d'un composant est une mesure de son comportement à un instant donné et est représenté par un ensemble de* variables d'états *contenu dans un* vecteur d'état. » [**Oates, 1995**]

Un événement est défini comme une entité atomique qui est le reflet d'un changement d'état du composant [**Mansouri-Samani et Sloman, 1992**]. L'état d'un composant possède donc une portée temporelle définie par la durée entre l'occurrence de deux événements.

Selon la complexité du système, seul un sous-ensemble des états sera considéré comme intéressant à observer. Cette réduction permet de définir l'ensemble des *états d'intérêt*. Il s'agit d'une opération de filtrage ayant pour but de limiter l'intrusion et les perturbations liées à l'observation.

3.2.2 Construction d'une trace d'observation

Nous cherchons donc à construire une représentation des états et des événements associés à un composant ou un groupe de composants. L'ensemble des données collectées forme la **trace d'observation.** Il existe plusieurs techniques d'observation pour constituer cette trace :

Périodique

La technique d'échantillonnage ("*Time Driving Monitoring*") correspond à une observation périodique du système tel que le réalise «***gprof***» [**Graham et al., 1982**]. Elle représente une discrétisation de l'exécution. Les états du système sont observés indépendamment des changements d'état, ce qui implique qu'un même état peut être observé un nombre indéfini de fois.

Cette méthode d'observation permet de construire des «instantanés» de l'état du système. La nature des données obtenues se prête aux analyses statistiques qui rendent compte de la façon dont s'est déroulée l'exécution sans prendre en considération l'ordre des observations. Par contre, elle est moins adaptée à la reconstruction de la dynamique de l'exécution dont la qualité est fortement liée au choix du pas d'échantillonnage. En effet, des phénomènes, dont la durée est inférieure au pas d'échantillonnage, peuvent ne pas être observés.

Sur demande

L'observation est déclenchée à la demande de l'environnement d'observation [**Dasgupta, 1986**, **Sheehan et al., 1999**]. Il peut s'agir d'une demande périodique ou générée lors de la satisfaction de contraintes.

Événementielle

L'observation est réalisée lorsqu'un *événement d'intérêt* est capté par l'environnement d'observation ("*Event Driving Monitoring*") [**Miller et al., 1994a**, **Ottogalli et al., 2001**] (voir section 5.1). Cette technique est utilisée lorsque l'on cherche à reconstruire la dynamique de l'exécution de l'application.

A chaque occurrence d'un événement, un enregistrement est réalisé. La notion «d'occurrence d'un événement» est définie par la satisfaction d'un ensemble de conditions. Ces conditions sont liées à l'évaluation de l'état du système. Il s'agit de construire des abstractions par composition d'états et d'événements. La base de ces abstractions repose sur les *événements primaires* correspondant aux événements de plus petite granularité observables dans le système.

Un des problèmes majeur de cette approche est l'impossibilité de prévoir la quantité d'événements à observer. En conséquence de quoi, il est difficile d'anticiper sur les perturbations liées à l'observation et même de garantir le bon fonctionnement de l'application (voir sections 4.2 et 6.3). Il est en effet possible d'aboutir à une situation où la collecte des données paralyse

entièrement le système. Pour éviter cela, des techniques d'adaptation de la granularité de l'observation sont mises en œuvre telles que celles présentées dans les travaux de [**Boutrous Saab, 2000**].

Ces trois techniques sont utilisables quel que soit le niveau d'abstraction considéré.

Dans le cadre de ce travail, nous nous intéresserons plus particulièrement aux traces événementielles. Elles ont le double avantage de permettre la reconstruction de la dynamique d'exécution ainsi que les informations statistiques telles que celles qu'obtenues dans le cadre d'une observation périodique. Les observations traitées proviennent des niveaux d'abstraction système et applicatif. Les deux sections suivantes abordent les techniques employées pour leur obtention.

3.3 Observations du niveau système

Le «niveau système», auquel il est fait référence dans ce chapitre, représente l'observation de l'activité du noyau. Il s'agit de réaliser des observations à un bas degré d'abstraction afin d'identifier les mécanismes de base employés lors de l'exécution. De plus, les mécanismes d'observation du niveau système peuvent permettre d'accéder, à faible coût, aux informations statistiques construites par certains noyaux. A titre d'exemple, le noyau Linux associe une structure de données à chacun des processus créés. Cette structure contient, entre autres informations, la durée d'utilisation de la ressource processeur en mode utilisateur ou la quantité de mémoire vive consommée [**Torvalds et al., 2001**] (voir section 5.1.2). Cette structure est accessible à tout processus qui s'exécute en mode système et est réifiée au travers du système de fichiers virtuel `/proc/`.

Un des projets les plus avancé dans ce domaine ce nomme «***KernInst***» [**Tamches et Miller, 1999**]. Il offre la possibilité d'insérer dynamiquement du code dans le noyau et ce en cours d'exécution. L'insertion est réalisée à la demande et peut être retirée de la même façon.

Dans «KernInst», un point d'instrumentation correspond à l'insertion de code en remplaçant l'instruction qui aurait du être exécutée par un point de branchement (voir figure 3.2). Ce point de branchement pointe sur le code d'instrumentation qui encapsule l'instruction initiale, ainsi qu'un mécanisme de continuation de l'exécution.

Dans «**Paradyn**» [**Miller et al., 1994a**], les points d'observation sont placés en début et fin de fonction ou lors d'un appel de fonction. Par contre, dans «KernInst», ils peuvent être insérés de façon arbitraire dans l'application [**Miller et al., 1994a**]. De plus, l'insertion est réalisée avec un surcoût minimal puisque l'exécution n'est pas interrompue pour réaliser des vérifications comme c'est le cas dans Paradyn. L'intégrité est maintenue en ne remplaçant qu'une seule instruction à la fois, et en passant par une zone mémoire tampon pour obtenir l'adresse de retour.

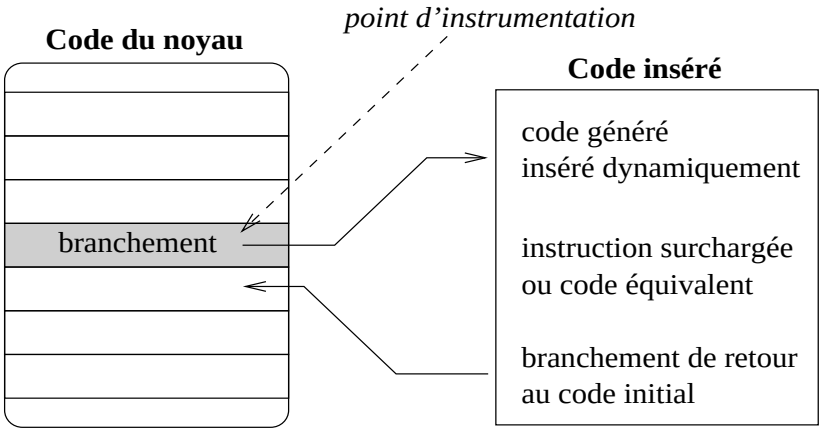


FIG. 3.2 – «KernInst» : insertion de code dans le noyau.

Un exemple d'utilisation est présenté dans [**Tamches et Miller, 1999**]. Du code compilé est inséré à la demande dans le noyau par «KernInst». Le lien avec l'application de l'utilisateur est réalisé par le démon *kerninstd*. Le coût d'insertion mesuré, dans le cas le plus défavorable, est de l'ordre de $100\mu s$[1]. En ce qui concerne l'empreinte mémoire, elle est calculée en additionnant la taille du code à ajouter, le nombre d'instructions nécessaires à l'insertion du code, la taille de l'instruction initiale et le nombre d'instructions pour le retour à l'instruction suivante. L'ordre de grandeur est ici le kilo octet.

Il existe d'autres projets tel que le développement des bibliothèques «**PAPI**» [**Browne et al., 2000**] ou «**PCL**» [**Berrendorf et Ziegler, 1998**] qui proposent des interfaces d'accès aux compteurs de performance du niveau matériel. Ces bibliothèques donnent la possibilité d'obtenir des informations d'une granularité extrêmement fine. Elles sont toutefois limitées dans le sens où elles ne reflètent que l'utilisation des composants possédant des registres de performances. Cette voie n'a actuellement pas encore été explorée.

Pour ce qui nous concerne, les observations du niveau système portent sur les appels système. Leur observation nous informe sur l'émission ou la réception de messages au travers de l'abstraction "*socket*" (voir section 5.1.2). Comme notre plate-forme d'expérimentation est composée de stations Linux, nous profitons de l'observation des appels système pour lire des indices de consommation de ressources d'exécution tenus à jour par le noyau (voir section 5.1.2).

3.4 Observations du niveau applicatif

L'observation du niveau applicatif correspond à une volonté d'étudier le comportement du système du point de vue de l'utilisateur. On se place à un niveau d'abstraction représentatif de la séman-

[1] Le coût d'insertion de code est compris entre 392 μs et 600 μs sur une station UltraSPARC 1 à 167 MHz.

tique de l'application.

L'instrumentation de l'application prend typiquement les formes suivantes (voir figure 3.3) :

L'insertion de code

Il peut avoir lieu manuellement [**Lange et al., 1992**, **Shende et al., 2001**], ou de façon automatique ou semi-automatique [**Rose et al., 1998**]. Le code initial de l'application est modifié avant ou durant la compilation. Cette technique offre la possibilité de placer les points d'instrumentation n'importe où dans l'application. Elle permet aussi d'accéder facilement à l'état des variables internes.

Cette solution demande l'accès au code source de l'application ainsi qu'une phase de compilation à chaque modification de l'instrumentation.

L'utilisation de bibliothèques instrumentées

Elles remplacent les bibliothèques standards. Cette technique est utilisée par «**SvPablo**» [**Rose et al., 1998**] ou «**VAMPIR**» [**Nagel et al., 1996**]. Les bibliothèque sont liées à la compilation ce qui a pour avantage de ne pas modifier le code applicatif original et donc ne demande pas accès au code source. Toutefois, une phase de compilation est nécessaire.

L'instanciation d'une interface d'observation

Une interface fonctionnelle est mise à disposition afin de réaliser des opérations d'observation à l'instar de la JVMPI [**JVMPI, 1999**] proposée par Sun pour sa JVM. Dans ce cas, le code d'instrumentation se présente sous la forme d'une librairie qui est chargée dynamiquement lors de l'exécution [**Kazi et al., 2000**].

Cette technique à l'inconvénient de limiter les points d'observation à ceux fournis par l'interface, mais permet l'observation d'application sans en posséder le code source et sans avoir à la recompiler.

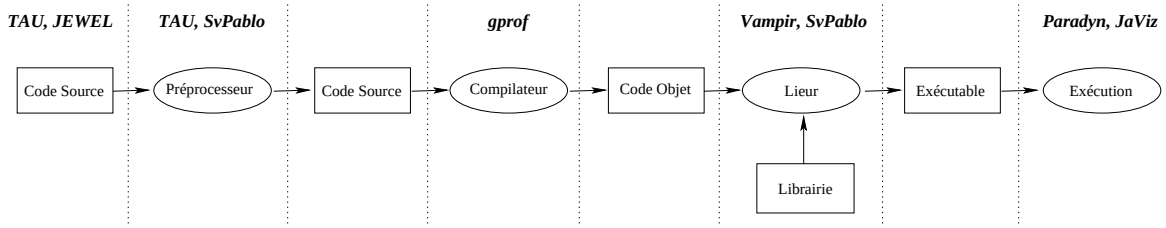


FIG. 3.3 – Synthèse des techniques d'insertion de code d'instrumentation dans les différents environnements de traçage.

Notre étude ayant pour ambition l'observation du méta ORB Jonathan, nous nous sommes orientés vers l'instanciation de l'interface d'observation de la JVM de SUN : la JVMPI 5.1.1. Il nous a ainsi été possible d'observer l'exécution des JVM. Les observations obtenues mettent en évidence les appels de méthodes, l'entrelacement des "*threads*" leur servant de support exécutif ainsi que l'utili-

sation des objets de synchronisation. Ces informations du niveau applicatif sont à mettre en relation avec celles obtenues au niveau système.

3.5 Mise en correspondance des observations multi-niveaux

Afin de relier les observations réalisées dans les différents niveaux d'abstraction, il nous faut les projeter dans un référentiel de sens commun. Pour cela, nous avons choisi de mettre en place un système de datation globale qui servira de «base de projection» pour l'ensemble des niveaux et du système étudié.

Le système de datation globale se décompose en deux parties :
– un système de datation local présent sur chacun des sites.
– le calcul des paramètres de correction des dates locales en dates globales.

3.5.1 Définition d'un référentiel de sens commun

La définition d'un référentiel de sens commun a pour but de reconstruire la dynamique globale de l'exécution. Pour cela, il nous faut définir une relation d'ordre sur les observations. Deux types d'ordre seront considérés : l'ordre logique et l'ordre temporel qui est une extension linéaire de l'ordre logique. L'ordre logique permet de retrouver la causalité entre les événements et donc de reconstruire l'enchaînement des événements. Grâce à cet ordre, il est possible de répondre à la question *comment* mais pas *combien* de temps. Pour répondre à cette seconde question, il nous faut quantifier la consommation des ressources d'exécution associées aux événements. En ce qui concerne la consommation de la ressource de calculs, ou bien d'accès aux données distantes, la question du *combien* fait principalement référence à la durée d'utilisation de la ressource. Il est possible de répondre à cette question dès lors que l'on possède un système de mesure du temps : une **horloge**. Cette horloge permettra de mesurer des durées qui serviront de métrique aux indices considérés. Mais la présence d'une horloge peut faire plus que mesurer des durées. Elle permet aussi d'établir un ordre sur l'occurrence des événements [**Rabenseifner, 1997**].

Le modèle d'exécution considéré se base sur l'emploi de "*threads*" [**Carissimi, 1999**]. Or, l'exécution sur un "*thread*" est purement séquentielle. Cela signifie qu'en associant à chaque événement sa provenance et sa date, il est possible de reconstruire l'ordre causal de l'exécution au sens des horloges de Lamport [**Lamport, 1978**] ou de ses extensions. A l'aide de cet ordre, et de l'identification en nom et lieu des événements, nous pouvons reconstruire le chemin de causalité entre observations provenant de différents niveaux d'abstraction (voir figure 3.4). De ce fait, un événement provenant d'un niveau d'abstraction i devient interprétable dans un niveau d'abstraction j en analysant son contexte événementiel.

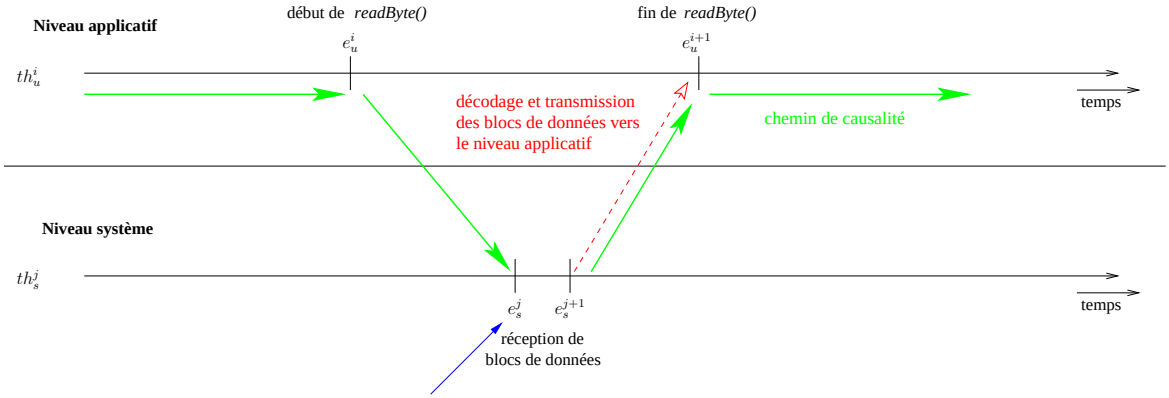


FIG. 3.4 – Construction du chemin de causalité entre les niveaux d'abstraction.
En datant les événements de début (e_u^i) et de fin (e_u^{i+1}) de méthode, ainsi que le début (e_s^j) et la fin (e_s^{j+1}) de réception de blocs de données, il devient possible de reconstruire le chemin de causalité entre des observations du niveau système et applicatif.

Ainsi, si l'on prend l'exemple de la réception d'un message, celui-ci se traduit au niveau applicatif par une attente sur un objet représentant un port de communication. Dans le cas de Java, on observe, par exemple, l'exécution d'une méthode bloquante `readByte()` sur un objet représentant une "*socket*" de communication. Au niveau système, l'arrivée d'un message se traduira par la réception de blocs de données sur l'abstraction "*socket*" du noyau. Les blocs de données seront interprétés et remontés jusqu'au niveau applicatif. Lorsque le noyau transmet les blocs reçus à l'application, la méthode en attente se débloque et réceptionne les données. Dans cette situation, si d'une part on identifie le "*thread*" qui exécute la méthode `readByte()`, le "*thread*" noyau qui réceptionne les blocs de données, et que d'autre part on date les événements de début et fin de méthode ainsi que la réception des données, on est alors capable de reconstruire le chemin de causalité entre la réception de blocs de données au niveau système et l'exécution d'une méthode du niveau applicatif (voir figure 3.4).

Il nous faut donc définir une fonction de projection dans le référentiel de sens commun.

3.5.2 Définition d'une fonction de projection

L'objet de cette fonction est de corréler les observations provenant de différents niveaux d'abstraction. La section 3.5.1 décrit comment établir une relation d'ordre causal entre les événements. Mais pour pouvoir l'appliquer dans le cadre d'une analyse multi-niveaux, il nous manque l'information nécessaire pour relier les événements des différents niveaux. Dans l'exemple présenté par la figure 3.4, il nous manque une partie de l'information pour corréler la réception de blocs de données par le "*thread*" th_s^j et l'exécution de la méthode `readByte()`.

Si l'on se trouve dans un système mono-utilisateur, l'association est automatique. Par contre, dans

un système multi-utilisateurs à base de "*threads*", la corrélation n'est plus aussi évidente. Il faut définir une clé de corrélation entre les activités systèmes et l'exécution du niveau applicatif. Cette clé est construite en fonction du modèle de "*thread*" employé. Trois modèles se dégagent [**IEEE, 1995**, **Carissimi, 1999**] (voir figure 2.7) :

le modèle **1/1** dans lequel à chaque "*thread*" utilisateur correspond un "*thread*" système.

le modèle **N/1** où N "*threads*" utilisateur s'exécutent sur le même "*thread*" système.

le modèle **N/M** où N "*threads*" utilisateur s'exécutent sur M "*threads*" système.

Le cas **1/1** est le plus favorable. L'exécution des "*threads*" utilisateurs est directement reliée à celle des "*threads*" système. Dans l'exemple présenté Figure 3.4, le "*thread*" système associé au "*thread*" utilisateur th_u^i est le même que celui qui va réceptionner les blocs de données (th_s^j). Pour réaliser les corrélations entre événements systèmes et applicatifs, il suffit d'identifier la correspondance entre "*threads*" utilisateur et "*threads*" système.

Dans le cas **N/1** le principe est globalement équivalent. Il faut tout d'abord réaliser l'association entre le "*thread*" système et les "*threads*" applicatifs. A la suite de quoi, la séquence d'ordonnancement des "*threads*" utilisateur doit être identifiée afin de déterminer quel "*thread*" s'exécutait lors des événements $e_{s,i}^c$ et $e_{s,i}^d$. Ainsi, l'observation de l'événement applicatif $e_{u,j}^{a+1}$ est corrélée à l'événement système $e_{s,i}^c$ et l'événement $e_{u,k}^{b+1}$ à $e_{s,i}^d$ (voir figure 3.5).

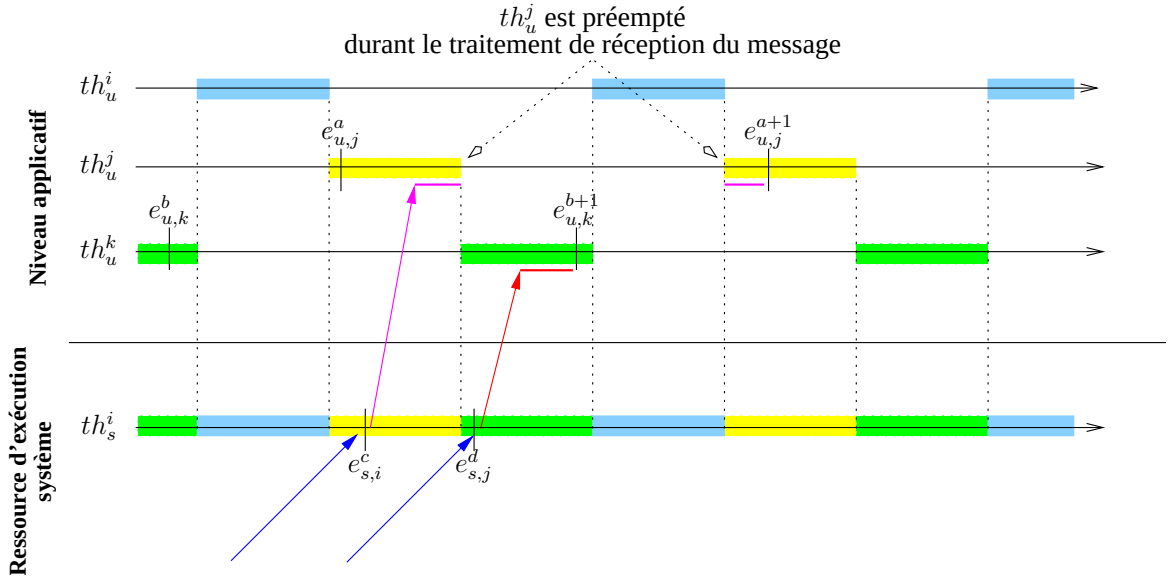


FIG. 3.5 – Réception de messages dans un modèle de "*threads*" **N/1**.
L'ordonnancement des "*thread*" utilisateurs est représenté par l'enchaînement des barres de couleurs sur le "*thread*" système.

Le problème est similaire avec le modèle **N/M**. La première étape consiste à identifier les "*threads*"

système employés par les "*threads*" utilisateurs. Puis chaque message doit être estampillé d'une date et de l'identifiant du "*thread*" système qui l'a traité. Ainsi, en enregistrant la séquence d'ordonnancement des "*threads*" utilisateur sur les "*threads*" système, il est possible de déterminer quel "*thread*" utilisateur s'exécute lors de la réception des messages. Dans l'exemple présenté par la Figure 3.6, nous pouvons rattacher l'événement utilisateur de fin de réception de message ($e^{a+1}_{u,j}$) par le "*thread*" th^j_u à la réception du message (événement système $e^c_{s,i}$) par le "*thread*" système th^i_s.

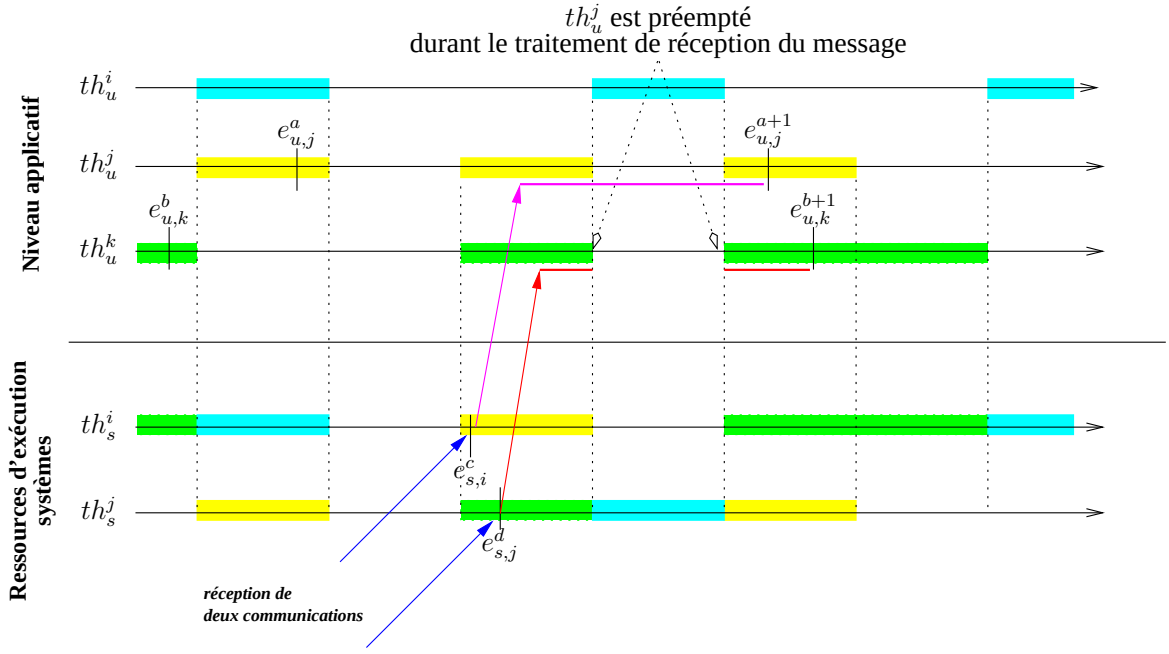


FIG. 3.6 – Réception de messages dans un modèle de "*threads*" **N/M**.
C'est le "*thread*" système th^j_s qui sert de support d'exécution lors de l'observation de l'événement utilisateur $e^{a+1}_{u,j}$. C'est le "*thread*" système th^i_s qui traite la réception du premier message. Malgré tout ces deux événements sont corrélés grâce à l'identification de la séquence d'ordonnancement des "*threads*" utilisateurs sur les "*threads*" système.

La mise en correspondance des "*threads*" utilisateur et des "*threads*" système, qui exécutent réellement l'application[2], fournit donc une clé de projection commune à l'ensemble des niveaux d'abstraction. Selon le modèle d'exécution des "*threads*" employés, une information supplémentaire sur l'ordonnancement des "*threads*" utilisateurs est nécessaire. Mais cette clé demeure valide et est employée dans la mise en œuvre de notre infrastructure d'observation et d'analyse (voir section 5.2).

3.6 Conclusion

Ce chapitre a abordé les problèmes liés à la construction d'une infrastructure d'observation multi-niveaux. Nous avons introduit le modèle de système sous-tendu ainsi que différentes techniques de mise en œuvre de l'observation.

[2]C'est les "*threads*" système qui ont réellement accès aux ressources physiques du site.

Nous avons de même abordé la problématique de la corrélation des observations multi-niveaux. Il ressort que la mise en place d'un système de datation globale et d'une fonction de projection permet d'interpréter les événements des différents niveaux d'abstraction.

La fonction de projection doit être construire en répondant aux impératifs de nommage des événements, de création d'une relation d'ordre, et d'identification d'une clé commune à l'ensemble des niveaux d'abstraction.

Chapitre 4

Interprétation des observations

Indépendamment de la collecte des données, traitée dans le chapitre précédent, l'interprétation de ces dernières est une phase délicate à plus d'un égard. Aussi, une étape préliminaire de prétraitement est nécessaire. Elle permet d'homogénéiser la qualité et la sémantique des données brutes afin de les rendre interprétables les unes par rapport aux autres. Ce prétraitement prend la forme, dans le cas qui nous concerne, de la projection des dates d'observation dans un référentiel de temps global et de la prise en compte des perturbations liées à l'observation. La trace ainsi obtenue est dite «corrigée».

Une fois ce prétraitement effectué, l'analyse des observations corrigées devient possible. Deux types d'analyse sont présentés. Une analyse statique, qui synthétise le comportement global sans tenir compte de l'ordre dans lequel les observations ont été réalisées. Une analyse dynamique, qui s'attache à reconstruire la manière dont l'exécution a eu lieu en terme de chemins d'exécution et d'interdépendance entre ceux-ci.

Nous allons tout d'abord aborder le problème de la construction d'un référentiel de temps commun, premier élément du prétraitement.

4.1 Construction post-mortem d'un système de datation globale

Comme il a été montré dans la section 3.5.1, un système de datation globale est nécessaire à l'interprétation d'observations multi-niveaux. Cette nécessité est d'autant plus grande lorsque les observations proviennent d'une application distribuée. Ce système de datation se doit d'être localement commun à l'ensemble des niveaux d'abstraction, et commun à l'ensemble des sites présents dans le système.

La majorité des systèmes d'exploitation fournissent un service de datation locale. Il est typiquement basé sur une horloge matérielle ou bien sur un registre du processeur qui compte le nombre

de cycles depuis son initialisation (voir section 4.1.4). La date fournie par cette horloge locale possède la même sémantique quel que soit le niveau d'accès. Seul le coût d'accès change. Si l'on prend l'exemple d'un système d'exploitation Linux, le coût d'obtention de la date est de l'ordre de la microseconde depuis le niveau utilisateur, alors qu'il est de l'ordre du dixième de microseconde depuis le niveau système (voir section 4.2).

Par contre, pour ce qui est de la mise en œuvre d'un système de datation globale, le problème devient plus délicat à résoudre. Deux grandes catégories de solutions existent : celles basées sur un équipement dédié et celles utilisant des techniques logicielles.

Les systèmes de datation matérielle reposent typiquement sur une horloge de grande finesse et un média de synchronisation dédié. Le système d'observation «**ZM4**» [**Hofmann et al., 1994**] possède un tel système. La granularité des horloges utilisées est de $100ns$. Elles sont tenues synchronisées à une horloge de référence à l'aide d'un canal de communication (le "*tick channel*").

IBM propose un système de datation globale matérielle pour ses machines parallèles SP. Le système est implanté au niveau du "*switch*" d'interconnexion des nœuds de la machine [**Wu et al., 2000**]. Chaque nœud possède une horloge locale. Le "*switch*" en possède également une qui sert de référence pour synchroniser l'ensemble du système. Afin de ne pas avoir à modifier les outils de prise de trace, les enregistrements restent datés avec l'horloge locale. A intervalles de temps réguliers, un n-uplet constitué des dates de chacune des horloges locales et du "*switch*" est enregistré. L'erreur entre les dates locales et la date de référence permet de calculer la correction à appliquer afin de construire un référentiel de temps commun.

Ces exemples de datation matérielle permettent d'obtenir un résultat de très bonne qualité, mais restent lourds et coûteux à mettre en place. Une approche logicielle offre des alternatives satisfaisantes dans bon nombre de cas.

Un des projets le plus largement répandu est le "*Network Time Protocol*" (NTP) [**Mills, 1991**]. Il s'agit d'un système de synchronisation utilisant le réseau de communication classique (local (LAN) et même global (WAN)) pour communiquer. Le système repose sur un modèle client/serveur. Les serveurs sont organisés de façon hiérarchique. Un serveur de niveau n se synchronise sur un serveur de niveau $(n-1)$. Plus on s'éloigne du serveur primaire et moins la synchronisation est précise. La précision obtenue est de l'ordre de la dizaine de milliseconde [**Mills, 1989**]. Si l'on adopte une solution hybride, en couplant l'approche NTP sur un réseau local avec du matériel de synchronisation (par exemple un GPS), la précision peut atteindre la nanoseconde [**Mills et Kamp, 1999**]. Il apparaît donc que cette solution reste délicate à mettre en place lorsque l'on souhaite synchroniser les différentes horloges à la μs.

Une autre solution entièrement logicielle est proposée dans les travaux de [**Duda et al., 1987**, **Haddad, 1988**] et [**Maillet, 1996**]. Elle propose de calculer des paramètres de correction des dates locales afin de les projeter dans un référentiel de temps commun. C'est la solution qui a été adoptée

car peu coûteuse en matériel et portable car indépendante de l'architecture matérielle du système.

La mise en œuvre d'un système de datation globale logiciel est présenté dans la section suivante.

4.1.1 Modèle linéaire d'horloge

La solution choisie pour construire un système de datation globale repose sur le travail de thèse de Eric Maillet [**Maillet, 1996**]. Elle consiste à estimer des paramètres de dérive et décalage à l'origine des horloges présentes sur chacun des sites [1] afin de projeter les dates enregistrées dans un système de datation de commun : l'horloge globale (HG).

Chaque site prenant part à l'exécution est donc muni d'une horloge physique locale $hp_i, i \in 1..p$. Notons $d_i(t)$ la date lue par hp_i en t la date absolue [2]. La granularité des dates ainsi obtenues est en $\mathcal{O}(\mu s)$ (voir section 4.1.4). D'après [**Maillet, 1996**], les dates fournies par l'horloge physique des ordinateurs peuvent être modélisées par une équation linéaire de la forme :

$$lt_i(t) = \alpha_i.t + \beta_i + \delta_i, i \in \{1..p\}; \tag{4.1}$$

où t est le temps absolu, la constante α_i (proche de 1) la dérive de l'horloge, la constante β_i le décalage à l'origine ($t = 0$) et δ_i une modélisation de la résolution r de l'horloge et des perturbations aléatoires du site i. Nous ferons l'hypothèse que δ_i est de moyenne nulle. Ce modèle de $lt_i(t)$ n'est correct que si l'on suppose que les paramètres physiques de l'environnement comme la température, la pression et le voltage restent stables, et que t est suffisamment petit pour négliger le vieillissement du cristal [3]. Hors ces conditions, les paramètres α_i et β_i pourraient ne plus être constants.

En éliminant t de l'équation 4.1 écrite pour i et j, nous obtenons la dépendance linéaire entre lt_i et lt_j :

$$lt_j(t) = \alpha_{i,j}.t + \beta_{i,j} + \delta_{i,j}, i, j \in \{1..p\}; \tag{4.2}$$

$$\begin{aligned} \alpha_{i,j} &= \frac{\alpha_j}{\alpha_i}; \\ \beta_{i,j} &= \beta_j - \alpha_{i,j}\beta_i; \\ \delta_{i,j} &= \delta_j - \alpha_{i,j}\delta_i; \end{aligned}$$

$\delta_{i,j}$ est une variable aléatoire de moyenne nulle. Cette dépendance nous permet de choisir un site de référence $réf$, et de convertir les dates obtenues sur tout sites j en une date du site $réf$. Donc, pour

[1] Un site est défini comme une infrastructure matérielle qui possèdent des ressources de calcul et d'accès aux données, ainsi qu'un système de datation local.

[2] La date absolue correspond à la date qui serait donnée si l'on possédait un système de datation commun à l'ensemble du système.

[3] Les oscillateurs à cristal ont une sensibilité à la température de l'ordre de $10^{-6}s/\ ^\circ C$, et un taux de vieillissement de l'ordre de $10^{-7}s/jour$

tout site $j \neq réf$, il nous faut calculer les paramètres $\alpha_{réf,j}$ et $\beta_{réf,j}$ de façon à calculer la date globale (DG) :

$$DG_j(t) = \frac{lt_j(t) - \alpha_{réf,j}}{\beta_{réf,j}} \simeq lt_{réf}(t), j \in \{1..p\}; \tag{4.3}$$

$DG_j(t)$ est la date globale vue par le site j à l'instant t.

4.1.2 Les échantillons de données

Pour pouvoir estimer les paramètres $\alpha_{réf,j}$ et $\beta_{réf,j}$ ($j \in \{1..p\} \setminus \{réf\}$), nous effectuons une série de lectures d'horloges sur les sites j et $réf$. Le protocole expérimental mis en œuvre comprend une phase de communication durant laquelle les sites j et $réf$ vont lire la date fournie par leur horloge locale. Une phase de communication est composée de $n > 200$ cycles. Un cycle est décomposé en :

- la lecture de la date sur le site j : $lt_j(t) = t_j^k$;
- l'envoi d'un message au site de référence ;
- la lecture de la date sur le site $réf$: $lt_{réf}(t) = t_{réf}^k$;
- l'envoi d'un message au site j ;
- la lecture de la date sur le site j : $lt_j(t) = t_j^{k+1}$;

En posant comme hypothèse que les durées de communications entre sites sont symétriques, nous obtenons un ensemble de couples $(\widehat{t_{réf,j}^k}, t_{réf}^k), k \in [1..n]$ où $\widehat{t_{réf,j}^k}$ est une estimation sur le site j de la date $t_{réf}^k$ (voir figure 4.1). Dans notre cas : $\widehat{t_{réf,j}^k} = t_j^k + \frac{\delta}{2}, \delta = t_j^{k+1} - t_j^k$

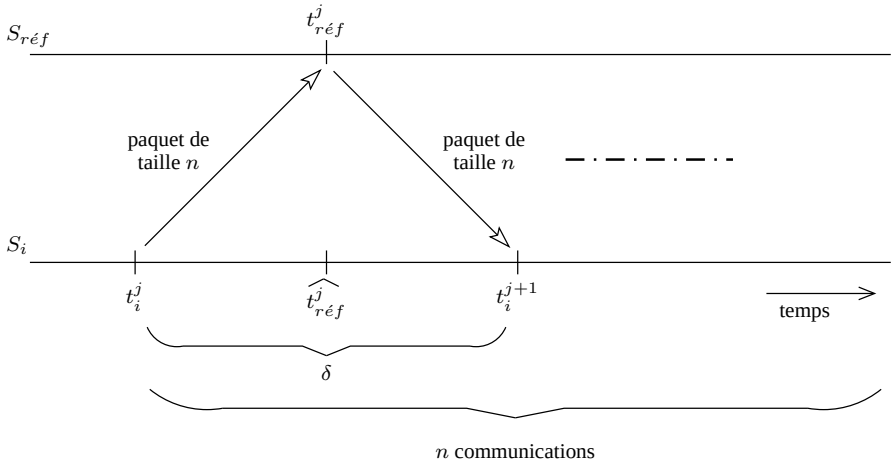


FIG. 4.1 – Schéma de communication pour construire les couples $(\widehat{t_{rf,i}^j}, t_{rf}^j)$.
$\widehat{t_{rf,j}^k}$ est une estimation de la date t_{rf}^k.

Les valeurs des paramètres $\alpha_{réf,j}$ et $\beta_{réf,j}$ sont estimées à partir de ces échantillons.

4.1.3 Estimation des paramètres

Les paramètres $\alpha_{réf,j}$ et $\beta_{réf,j}$, notés $\widehat{\alpha_j}$ et $\widehat{\beta_j}$, sont estimés en utilisant un calcul de type régression linéaire [4] sur un échantillon de points collectés comme décrit dans la section 4.1.2.

Nous posons ici comme hypothèse que les communications sont symétriques. En reprenant le modèle d'horloge de la section 4.1.1, nous obtenons l'équation :

$$f(t_j^k) = \widehat{t_{réf,j}^k} = \widehat{\alpha_j}.t_j^k + \widehat{\beta_j} \tag{4.4}$$

Le calcul de la droite sur l'ensemble des couples $(\widehat{t_{réf,i}^j}, t_{réf}^j)$ [**Jain, 1991**] nous fournit les estimateurs des paramètres $\alpha_{réf,j}$ et $\beta_{réf,j}$ (voir figure 4.2). La qualité de ces estimateurs est qualifiée par l'écart-type de l'erreur quadratique associé à la régression linéaire, et aux intervalles de confiance associés aux paramètres $\widehat{\alpha_j}$ et $\widehat{\beta_j}$.

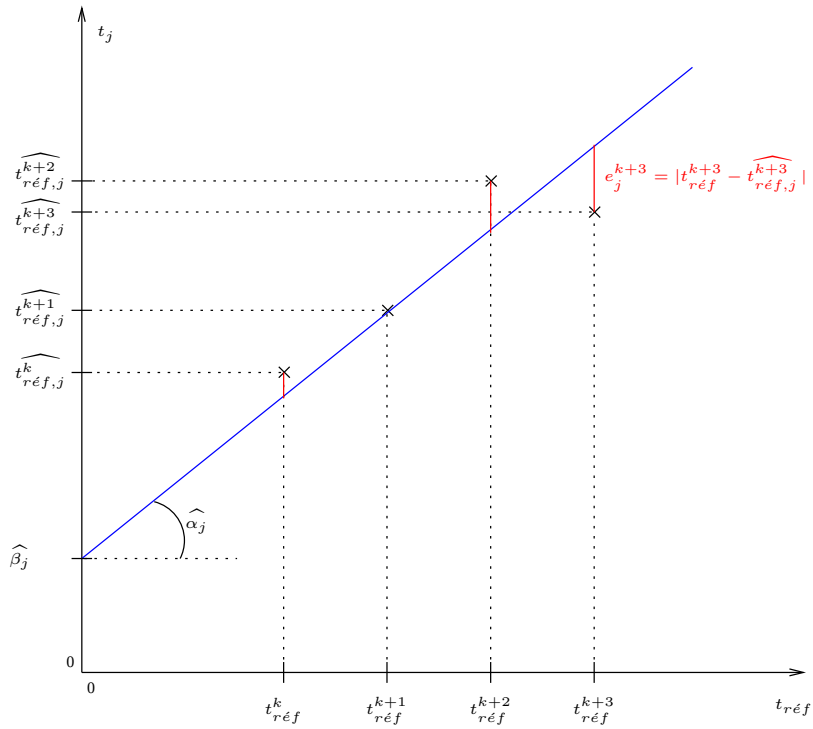


FIG. 4.2 – Calcul de la régression linéaire sur l'ensemble des couples $(\widehat{t_{réf,j}^k}, t_{réf}^k)$.
Les paramètres $\widehat{\alpha_j}$ et $\widehat{\beta_j}$ sont calculée par minimisation de l'erreur quadratique : $|t_{réf}^k - \widehat{t_{réf,j}^k}|^2$

Cette méthodologie a été employée pour estimer les paramètres de correction entre trois sites. Les résultats sont présentés dans la section 6.4 page 97 .

Les travaux de E. Maillet [**Maillet et Tron, 1995**, **Maillet, 1995**] ont été réalisés dans le cadre de systèmes parallèles. Ces systèmes sont fortement homogènes. Chaque nœud comprend la même

[4] Les paramètres sont calculés par minimisation de l'erreur quadratique.

quantité et la même qualité de ressources. Dans notre cas, nous étudions des systèmes répartis dans lesquels les quantités et les qualités des ressources varient d'un site à l'autre. Seule l'hypothèse de symétrie des communications est maintenue.

Dans le cas où l'hypothèse de symétrie dans les communications ne peut être vérifiée, un problème se pose pour l'estimation du décalage à l'origine. En effet, l'estimation de la dérive s'apparente à un calcul de débit[5]. Ce calcul est identique que les communications soient symétriques ou pas. Par contre, pour ce qui est du décalage à l'origine, une dissymétrie des communications entraînera une erreur dans l'estimation de β le décalage à l'origine (voir figure 4.3).

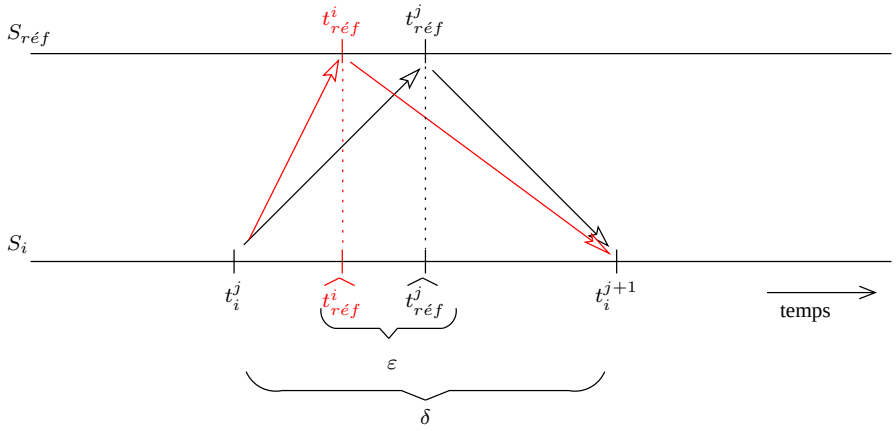


FIG. 4.3 – Erreur de l'estimation du décalage à l'origine due à une dissymétrie dans les communications.
Le protocole de mesure estime la valeur $\beta - \varepsilon$ au lieu de β.

Nous avons pu constater de grandes imprécisions dans le calcul de l'estimation du décalage à l'origine des horloges. Pour remédier à ce problème nous proposons une technique de synchronisation des horloges avant l'estimation des paramètres de régression linéaire (voir section 4.1.4).

4.1.4 Synchronisation des horloges sous Linux

La construction de la date dans un système Linux dépend de l'architecture du processeur de la machine. Si le processeur possède un registre "*Time Stamp Counter*" [6], le noyau se base sur la valeur de ce dernier pour mettre à jour la date du système :

```
static inline unsigned long do_fast_gettimeoffset(void)
```

Dans le cas où le processeur ne possède pas ce registre, la date est tenue à jour à l'aide de l'horloge matérielle présente sur la machine. Dans ce cas, c'est le nombre de cycles de l'horloge qui sert de référence pour calculer le temps écoulé et non le nombre de cycles du processeur :

[5] Il nous faut mesurer quelle quantité de données ont été transmises dans un intervalle de temps défini.

[6] Ce compteur contient le nombre de cycles que le processeur a effectué. En divisant le nombre de cycles par la fréquence du processeur, on obtient une date dont l'origine est la date de mise en activité de la machine.

```
static unsigned long do_slow_gettimeoffset(void)
```

Dans les deux cas, la date tenue à jour par le système est supposée être à une granularité de la microseconde. L'emploi de la fonction `do_fast_gettimeoffset()` permet d'envisager une telle précision. La fréquence des processeurs étant en méga-hertz, la granularité du pas d'incrémentation du "*Time Stamp Counter*" est inférieure à la nanoseconde. Par contre, lors de l'emploi de `do_slow_gettimeoffset()`, une telle précision est difficilement atteinte. Elle dépend directement de la qualité de l'horloge utilisée [7].

Dans notre cas, nous utilisons des machines qui possèdent un registre "*Time Stamp Counter*". Les dates que nous obtenons possèdent donc une granularité de l'ordre de la microseconde. Cette granularité est bornée par la durée moyenne d'exécution de la fonction `gettimeofday` qui, en mode utilisateur, est de 1 μs et, en mode noyau, de 0,2 μs (voir section 4.2).

Pour à accéder cette date, le noyau Linux fournit donc la fonction :

```
void do_gettimeofday(struct timeval *tv)
```

La structure `struct timeval` possède deux champs : `long tv_sec` qui fournit la partie seconde de la date, et `long tv_usec` qui fournit la partie microseconde de la date. Cette fonction n'est accessible qu'en mode noyau. Elle est appelée, depuis le mode utilisateur, par la fonction :

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

La structure `struct timezone` n'est plus utilisée, sa présence est justifiée par un souci de compatibilité avec le code antérieur.

De la même façon qu'il est possible d'obtenir la date du système, il est aussi possible de la corriger. La noyau Linux fournit pour cela la fonction :

```
void do_settimeofday(struct timeval *tv)
```

Le paramètre `struct timeval *tv` représente la date à laquelle on veut mettre le système. Cette fonction permet donc de corriger la date du système avec une précision de la microseconde.

Le protocole expérimental mis en place pour synchroniser les systèmes de datation se décompose en deux phases (voir figure 4.4). Une première phase durant laquelle chaque site communique avec un site de référence. Cette première phase de communication sert à déterminer la durée moyenne de communication d'une date avec le site de référence (voir figure 4.5). Elle est suivie d'une seconde phase durant laquelle le site de référence envoie sa date locale au site qui le sollicite. Chaque site client $j, j \neq réf$ mesure la durée de communication de la date de référence. Si cette durée est conforme au résultat obtenu durant la première phase, la site j recale son horloge avec la date reçue ; sinon, il initie une nouvelle sollicitation tant que la durée de communication n'est pas valide. De cette façon,

[7] Il est à remarquer que celle communément rencontrée dans les stations de travail ou les ordinateurs personnels ne permet pas d'atteindre la précision souhaitée.

il est possible d'estimer la date sur le site de référence en rajoutant le temps de communication à la date reçue.

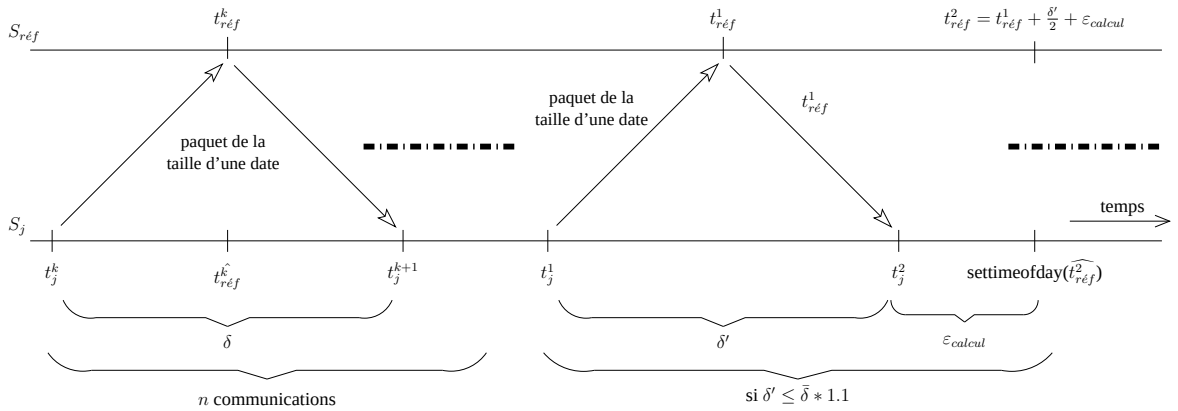


FIG. 4.4 – Synchronisation des systèmes de datation sur un site de référence.
Le protocole mis en place se décompose en deux phases. Une première phase de communication afin d'estimer la durée de communication d'une date. Et une seconde durant laquelle le site de référence envoie sa date locale sur laquelle se recale le site client.

4.2 Prise en compte des perturbations

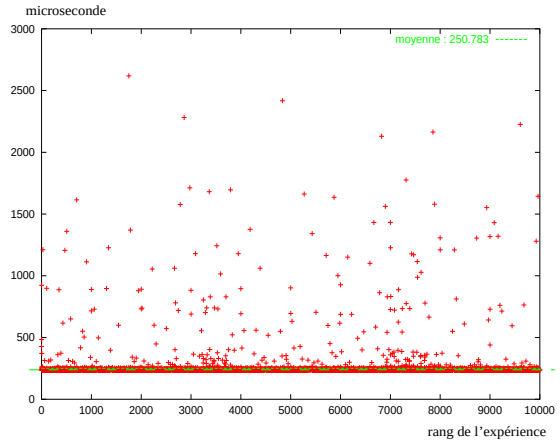
« *Toute mesure est une interaction qui déforme l'objet"* » [**Perdijon, 1998**]

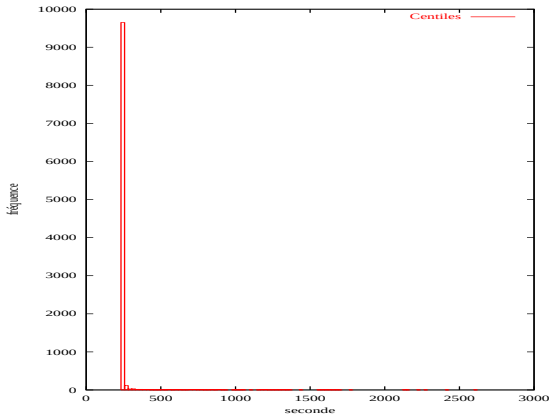
Comme l'écrit Brillouin : "Le fait nouveau, c'est la considération du prix, du coût d'une observation. Le physicien classique croyait pouvoir passer à la limite et considérer ce qui se produit lorsque les erreurs sont réduites à zéro. Nous savons maintenant que ce passage à la limite n'est pas concevable, car il coûterait un prix infini".

Un des problèmes fondamentaux lié à l'observation est celui de la perturbation par l'instrumentation [**Jain et al., 1996**]. Les perturbations induites par l'observation se divisent en deux classes [**Maillet, 1996**, **Maillet et Tron, 1995**] :

Les perturbations directes

Elles correspondent au coût en temps et en ressources pour enregistrer un événement. Ce coût est localisé autour du point d'instrumentation et résulte directement de l'exécution des instructions supplémentaires d'instrumentation. Que ce soit en termes de temps ou de consommation de ressources, ce coût peut être évalué pour chaque classe d'événements tracés. Il est fonction de la taille de l'enregistrement et de la bande passante accessible sur le système de sortie (mémoire, disque ou réseau selon l'outil de trace utilisé).





Taille de l'échantillon	=	10000
Moyenne	=	250.783
Min	=	233.00
Max	=	2619.00
Ecart-type	=	100.167
IC 90 %	:	] 249.136 .. 252.431 [

9 déciles de l'echantillon	=	9000
Moyenne	=	239.140
Min	=	233.0
Max	=	244.0
Ecart-type	=	2.807
IC 90 %	:	] 239.092 .. 239.189 [

FIG. 4.5 – Durée de communication entre le site de référence et un site client.
Le site du client est connecté au site de référence par un lien à 100 Mbit/s. L'analyse de l'échantillon nous indique que les 10% de mesures de poids forts biaisent la valeur de la moyenne. En effet la valeur de l'écart-type est du même ordre de grandeur que la moyenne. En éliminant ces 10%, nous obtenons un écart-type de deux ordres inférieurs à la moyenne. Nous pouvons donc en conclure que la moyenne des durées de communication appartient à l'intervalle] 239.092 .. 239.189 [avec un niveau de confiance de 90%.

Les perturbations indirectes

Elles correspondent aux effets de bord provoqués par l'observation. Elles peuvent affecter l'exécution de sections de codes, d'applications ou de processus non instrumentés. Il s'agit par exemple de l'inhibition des optimisations de compilation, de l'augmentation du nombre de changement de contexte, ou de la modification du comportement d'un programme qui communique par messages asynchrones (modification de l'ordre d'arrivée des messages).

Il est concevable d'estimer les perturbations indirectes par l'analyse de traces du niveau système (nombre de changements de contexte, défauts de pages, entrées/sorties...). Malgré tout, elles restent difficiles à identifier et à corriger. C'est pourquoi les environnements de traçage cherchent à les minimiser au maximum. Pour ce qui est des perturbations directes, leur identification étant plus aisée, elles peuvent dans certains cas être corrigées par un traitement *post mortem* de la trace.

En effet, tant que l'exécution est de type déterministe, la correction des perturbations peut se limiter à la soustraction de leur coût. Par contre, dès que l'application introduit de l'indéterminisme, par exemple la réception de communications, la soustraction du coût de la perturbation ne suffit plus à corriger l'effet de l'observation. C'est tout le chemin d'exécution de l'application qui peut être altéré

(voir figure 4.6).

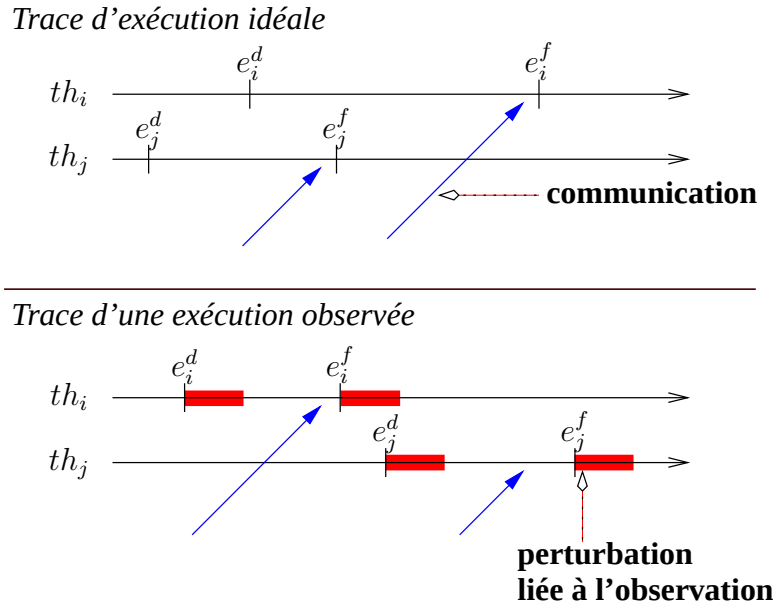


FIG. 4.6 – Modification du chemin d'exécution de l'application due à l'observation.
Les perturbations liées à l'observation peuvent modifier l'ordonnancement des exécutions dans un modèle d'exécution à base de "*threads*". Dans des situations d'indéterminisme, comme la mise en place d'un ensemble de "*threads*" en attente de réception de communications, le chemin d'exécution de l'application peut être totalement modifié.

Que l'on se trouve dans le cadre d'observations événementielles ou par échantillonnage, il est possible d'identifier des classes d'enregistrements auxquelles sont associées leurs perturbations. Dans le cas de l'échantillonnage, cette perturbation est prévisible. Il est donc possible de la borner. Par contre, pour ce qui est de l'observation événementielle, le niveau de perturbation est directement lié à la dynamique de l'application. Il est difficile de prévoir son impact. Afin de se prévenir de situations où les perturbations étoufferaient l'exécution, des mécanismes de sécurité peuvent être mise en place.

Ainsi, la plate-forme d'observation PHONIX propose un mécanisme de sonde auto-adaptable [**Boutrous Saab, 2000**]. L'adaptation à la dynamique de l'observation est réalisée selon deux axes : la granularité des observations et la quantité des sources d'information. La granularité est définie par le biais de règles de filtrage qui sont appliquées aux observations. L'architecture du système d'observation repose sur le paradigme de sonde. Ce système inclut un mécanisme d'insertion et de retrait à chaud de ces dernières. En jouant sur ces deux dimensions, nous pouvons limiter les perturbations liées à l'observation et éviter des situations de blocages ou de dégradation trop importantes des performances du système.

4.3 Analyse statique

Une fois les opérations de prétraitement réalisées, nous pouvons passer à l'analyse des données transformées.

Nous parlons d'analyse statique lorsque celle-ci ne prend pas en compte l'ordre des observations dans le traitement des données. Ce type d'analyse fait typiquement référence aux traitements statistiques classiques qui sont réalisés par des outils de "*profiling*" comme *gprof* [**Graham et al., 1982**].

Les traitements statistiques sont réalisés, par exemple, sur la durée d'exécution des méthodes ou le nombre de leurs invocations. Les informations fournies permettent de mettre en évidence **où** le temps est passé durant l'exécution. Il est ainsi possible de discriminer les méthodes selon leur fréquence d'appel (voir tableau 4.1), ou leur durée d'exécution (voir tableau 4.2). En cas de problèmes de performances, ces discriminations permettent de faire ressortir les méthodes sur lesquelles un travail d'optimisation doit être mené.

Id	nom de la classe	méthode
501	org/objectweb/david/libs/presentation/portable/PortableMarshallerFactory.PortableUnMarshaller	readByte()
364	org/objectweb/david/libs/presentation/portable/CDRMarshallerFactory.CDRMarshallerL	writePadding()
377	org/objectweb/david/libs/presentation/portable/PortableMarshallerFactory.PortableMarshallerL	writeInt()
362	org/objectweb/david/libs/presentation/portable/CDRMarshallerFactory.CDRMarshallerL	writeInt()
490	org/objectweb/david/libs/presentation/portable/CDRMarshallerFactory.CDRUnMarshaller	skipPadding()
507	org/objectweb/david/libs/presentation/portable/PortableMarshallerFactory.PortableUnMarshaller	readInt()
482	org/objectweb/david/libs/presentation/portable/CDRMarshallerFactory.CDRUnMarshaller	readInt()
407	org/objectweb/david/libs/presentation/portable/PortableMarshallerFactory.PortableMarshaller	writeByteArray()
136	org/objectweb/jonathan/apis/resources/Chunk	release()
133	org/objectweb/jonathan/apis/resources/Chunk	<init>

Id	méthode	nbr	durée	moyenne	écart-type	IC 90%	
501	readByte()	1672	12.455831	0.007450	0.153329	]0.001281...0.013618[	réception
364	writePadding()	1584	0.012939	0.000008	0.000004	]0.000008...0.000008[	envoi
377	writeInt()	1583	0.012861	0.000008	0.000004	]0.000008...0.000008[	envoi
362	writeInt()	1583	0.063606	0.000040	0.000032	]0.000039...0.000041[	envoi
490	skipPadding()	1059	0.008963	0.000008	0.000005	]0.000008...0.000009[	réception
507	readInt()	1056	0.010742	0.000010	0.000035	]0.000008...0.000012[	réception
482	readInt()	1056	0.045704	0.000043	0.000059	]0.000040...0.000046[	réception
407	writeByteArray()	518	0.015552	0.000030	0.000007	]0.000030...0.000031[	envoi
136	release()	442	0.003788	0.000009	0.000007	]0.000008...0.000009[	réception
133	<init>	442	0.004077	0.000009	0.000007	]0.000009...0.000010[	envoi

TAB. 4.1 – Les 10 méthodes le plus fréquemment appelées.
Ces méthodes sont toutes employées dans le cadre de l'envoi ou de la réception de messages. Elles ont toutes des durées d'exécution relativement faibles (de l'ordre de la μs), sauf pour la méthode 501 `readByte()` qui est bloquante.

Id	nom de la classe	méthode
753	org/objectweb/jonathan/libs/protocols/tcpip/TcpIpProtocol.Session	run()
325	org/objectweb/david/libs/protocols/giop/GIOPProtocol.ClientSession_Low	send()
501	org/objectweb/david/libs/presentation/portable/PortableMarshallerFactory.PortableUnMarshaller	readByte()
781	org/objectweb/david/libs/presentation/portable/CDRMarshallerFactory.CDRUnMarshallerD	prepare()
775	org/objectweb/jonathan/libs/protocols/tcpip/TcpIpChunkProvider	prepare()
282	org/objectweb/jonathan/libs/resources/tcpip/JConnectionMgr.Connection	receive()
764	org/objectweb/jonathan/libs/resources/tcpip/IPv4ConnectionFactory.Connection	receive()
662	org/objectweb/david/libs/stub_factories/std/JStubFactory.ClientDelegate	invoke()
661	org/objectweb/david/libs/stub_factories/std/JStubFactory.ClientDelegate	invoke()
731	org/objectweb/david/libs/protocols/giop/GIOPProtocol.GIOPSession_High	send()

Id	méthode	nbr	durée	moyenne	écart-type	IC 90%	
753	run()	208	12.648012	0.060808	0.432118	]0.011520...0.110095[	envoi
325	send()	208	12.611865	0.060634	0.432120	]0.011346...0.109922[	envoi
501	readByte()	1672	12.455831	0.007450	0.153329	]0.001281...0.013618[	réception
781	prepare()	418	12.438120	0.029756	0.305846	]0.005148...0.054364[	réception
775	prepare()	418	12.424137	0.029723	0.305843	]0.005115...0.054331[	réception
282	receive()	418	12.358310	0.029565	0.305845	]0.004957...0.054173[	réception
764	receive()	418	12.350261	0.029546	0.305845	]0.004938...0.054154[	réception
662	invoke()	209	0.878108	0.004201	0.003884	]0.003760...0.004643[	envoi
661	invoke()	209	0.802866	0.003841	0.003505	]0.003443...0.004240[	envoi
731	send()	209	0.797699	0.003817	0.003485	]0.003420...0.004213[	envoi

TAB. 4.2 – Les 10 méthodes dont l'exécution est la plus longue.
Les méthodes dont les durées sont les plus importantes sont typiquement bloquantes. Elles sont employées dans le cadre des interactions entre JVM.

Le calcul de l'écart-type de la moyenne nous informe sur la qualité de l'exécution. Un écart-type important indique la présence de mesures aberrantes ou d'instabilités dans les différentes durées d'exécution.

Il est alors souhaitable de calculer la fonction de répartition des fréquences de durée d'exécution.

Bien que les informations fournies soient suffisantes pour identifier les plus gros problèmes de performances [8], elles demeurent inappropriées pour analyser des problèmes algorithmiques tels que les interblocages ou la mauvaise utilisation d'un "*pool*" de "*threads*". L'«analyse dynamique» de l'exécution peut permettre de parer ce manque.

4.4 Analyse dynamique

Contrairement à l'analyse statique, l'analyse dynamique d'une exécution prend en compte l'ordre dans lequel les événements ont eu lieu. On cherche à reconstruire les chemins d'exécution et plus particulièrement les interdépendances entre ces chemins. Pour ce faire, il est nécessaire d'établir un

[8] Il s'agit surtout, dans ce cas, d'identifier des problèmes de programmation.

ordre sur les événements. Si l'analyse a pour objet d'identifier le comportement logique/algorithmique de l'exécution, l'ordre causal suffit. Si par contre on cherche à identifier, par exemple, le poids des communications par rapport au calcul, il est alors nécessaire d'utiliser un système de datation globale tel que présenté dans la section 4.1.

L'analyse dynamique demande la prise en considération d'une quantité importante de données. Elle doit pouvoir rendre compte de multiples points de vues et de multiples niveaux de granularité. La visualisation est la technique la plus communément utilisée dans ce cas. Pour être performant, un outil de visualisation doit répondre à certains critères [**Miller, 1993**]. Parmi les plus importants nous pouvons citer [**de Oliveira Stein, 1999**, **Chassin de Kergommeaux, 2000**] :

La possibilité de passer à l'échelle (*"scalability"*)
Il représente la capacité à traiter de grandes quantités de données sur le plan de la durée d'exécution mais aussi sur celui de la granularité des observations.

L'interactivité
Il fait référence aux capacités d'interaction de l'utilisateur avec l'outil de visualisation. Il s'agit, par exemple, de pouvoir choisir ce que l'on souhaite voir et comment.

L'extensibilité
Il correspond à l'adaptation de la visualisation à de nouveaux types de données ou de nouveaux modèles de programmation.

«**Vampir**» [**Nagel et al., 1996**] fait partie des outils de visualisation susceptibles de répondre à ces critères. Il offre la possibilité d'appliquer des filtres durant l'observation ou la visualisation, ce qui lui permet de répondre au critère de passage à l'échelle. L'aspect interactif ressort par la possibilité de créer de nouvelles vues de l'exécution, ainsi que de faire le lien entre la visualisation et le code source. Mais étant un produit commercial, il n'est pas facile de juger de son extensibilité.

«**Paradyn**» [**Miller et al., 1994a**] répond lui aussi aux critères cités ci-dessus. Il a été initialement conçu pour rechercher automatiquement les situations de manque de performances dans des traces d'exécution de longue durée comportant un grand nombre d'éléments d'activité. Bien que l'analyse soit réalisée à partir d'événements observés durant l'exécution, les indices de performance fournis offrent une vue statique de l'exécution. Elle ne permet pas de reconstruire la dynamique de l'exécution en terme de successions d'états observés. Ce problème a été traité par [**Karavanic et al., 1997**] en couplant Paradyn avec «**Devise**», un outil permettant d'intégrer plusieurs flux de données et de les représenter sous de multiples formes.

Enfin, «**Pajé**» [**de Oliveira Stein, 1999**] peut être considéré comme la solution la plus adaptée à notre contexte. Le passage à l'échelle est supporté par des fonctions de filtrage ou de zoom appliquées à une trace événementielle. L'interactivité repose sur la notion d'inspection qui permet d'obtenir des informations sur les objets représentés, ou par la mise en évidence d'éléments lors de visualisation.

Pour ce qui est de l'extensibilité, l'architecture même de "Pajé" est conçue à base de modules génériques interchangeables. De plus, la trace événementielle interprétée est auto-descriptive, ce qui permet d'intégrer la description du modèle d'exécution à la trace elle-même.

C'est l'outil de visualisation qui a été retenu pour mener nos analyses dynamiques. Ce choix est motivé par la flexibilité et l'ouverture dont fait preuve "Pajé". Il a été utilisé avec succès pour la visualisation d'exécutions parallèles basées sur des "*thread*" communiquants [**Bernard et al., 1999**, **Blayo et al., 1999**] ou encore l'analyse d'applications Java distribuées [**Ottogalli et al., 2001**]. De plus, le code source de l'application est disponible ce qui donne la possibilité de l'adapter au plus près à nos besoins.

4.5 "Pajé" : un outil de visualisation flexible et générique

"Pajé" est développé dans le cadre du projet Athapascan [**Briat et al., 1997**] développé au laboratoire Informatique et Distribution à Grenoble [**Pajé, 2001**]. Il a été initialement conçu pour la visualisation "post mortem" de traces événementielles d'applications parallèles. Le modèle de programmation considéré repose sur l'exécution concurrente de "*threads*" communiquant par envoi de messages. Les traces applicatives sont visualisées sous la forme d'un diagramme espace-temps.

"Pajé" organise, en interne, les données à visualiser de façon hiérarchique. Cette hiérarchie de données est représentative du modèle de programmation. Elle est définie dans l'en-tête de la trace événementielle. La prise en compte par "Pajé" d'un nouveau modèle de programmation signifie la définition d'une nouvelle hiérarchie de données construite à partir des notions d'entités et de conteneurs [9].

Afin d'éviter les confusions avec le terme «objet», les objets manipulés par "Pajé" sont nommés entités et conteneurs. Une entité est donc une donnée élémentaire tels qu'un événement, l'état d'un "*thread*", une communication ou la valeur d'un indice de performance. Un conteneur, quant à lui, représente un ensemble d'entités ou de conteneurs. Il s'agit donc d'un objet de plus haut niveau permettant de construire la hiérarchie des objets visuels. Le modèle d'exécution est donc représenté dans "Pajé" par un arbre dont les branches correspondent à la hiérarchie des conteneurs et les feuilles aux entités (voir figure 4.7).

Une attention toute particulière est portée sur le caractère interactif de "Pajé". Ainsi, la visualisation ne défile pas plus ou moins rapidement à l'écran, mais au contraire c'est l'utilisateur qui se déplace dans le temps pour visualiser la section de l'exécution qui l'intéresse (voir figure 4.8). Cette navigation temporelle permet de remonter ou d'avancer dans le temps, mais aussi d'effectuer un zoom avant ou arrière afin d'obtenir une vue plus globale ou plus précise de l'exécution. De même, lorsque

[9] Il s'agit ici du caractère extensible de "Pajé" qui ne lie pas à un seul modèle de programmation. "Pajé" a ainsi aisément pu être étendu afin d'interpréter des traces événements d'applications à objets réparties.

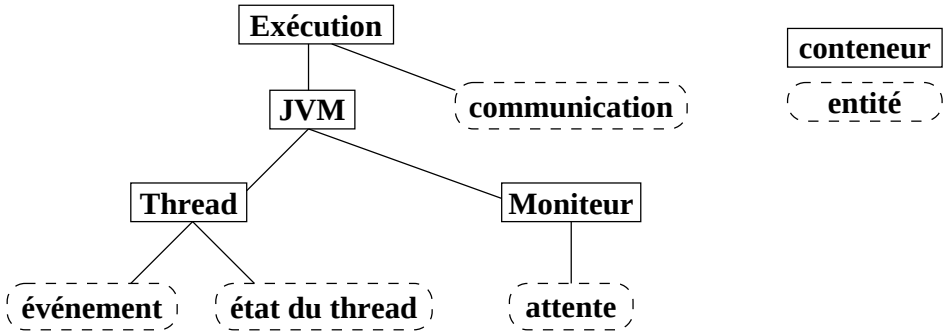


FIG. 4.7 – Exemple de hiérarchie de données interprétée par "Pajé".

l'on déplace la souris sur le diagramme espace-temps, des informations contextuelles apparaissent dans la barre supérieure de l'outil. Ces informations indiquent la date ainsi qu'un commentaire sur l'entité qui est sélectionnée. Outre ce premier niveau d'information, il est possible de sélectionner une entité, pour la faire ressortir de la visualisation, et inspecter son contenu pour obtenir toutes les informations qui s'y rapportent. On peut, par exemple, sélectionner une entité représentant l'exécution d'une méthode et obtenir son nom et la durée de son exécution, ou bien inspecter une entité représentant une communication et obtenir la source, la destination et la taille du message communiqué.

L'aspect "scalabilité" [10] de la visualisation est tout particulièrement pris en compte lors du développement de "Pajé". Pour cela, "Pajé" permet la création et l'utilisation de filtres. Ces filtres permettent, par exemple, de regrouper ou éclater des entités afin de construire une représentation synthétique, respectivement étendue, d'un même jeu de données. L'application de filtres offre la possibilité de simuler des effets de zoom. Ainsi, une exécution est, par exemple, visualisée comme un seul "*thread*" représentant l'activité de tous les "*threads*" observés. Il s'agit ici d'observer l'exécution dans sa globalité, à un haut niveau d'abstraction, afin d'en déterminer le comportement général. A la suite de quoi, il est possible de se focaliser sur une période de temps donnée et de dégrouper, ou regrouper différemment, les "*threads*" afin d'en observer le comportement.

"Pajé" est employé dans le cadre d'analyses multi-niveaux d'exécutions parallèles [**Ottogalli et Vincent, 1999**]. Il permet de corréler des observations de consommation de ressources système (l'exécution des "*threads*" en mode système ou utilisateur) à une trace applicative (l'exécution du support d'exécution parallèle Athapascan) et ainsi de valider les choix algorithmiques réalisés.

Un premier travail, portant sur l'utilisation de "Pajé" pour la visualisation d'applications réparties, est présenté dans [**Ottogalli et al., 2001**]. Il nous a permis de valider le choix de "Pajé" pour l'analyse multi-niveaux d'application à objets répartis.

[10] Il s'agit de la capacité à passer à l'échelle.

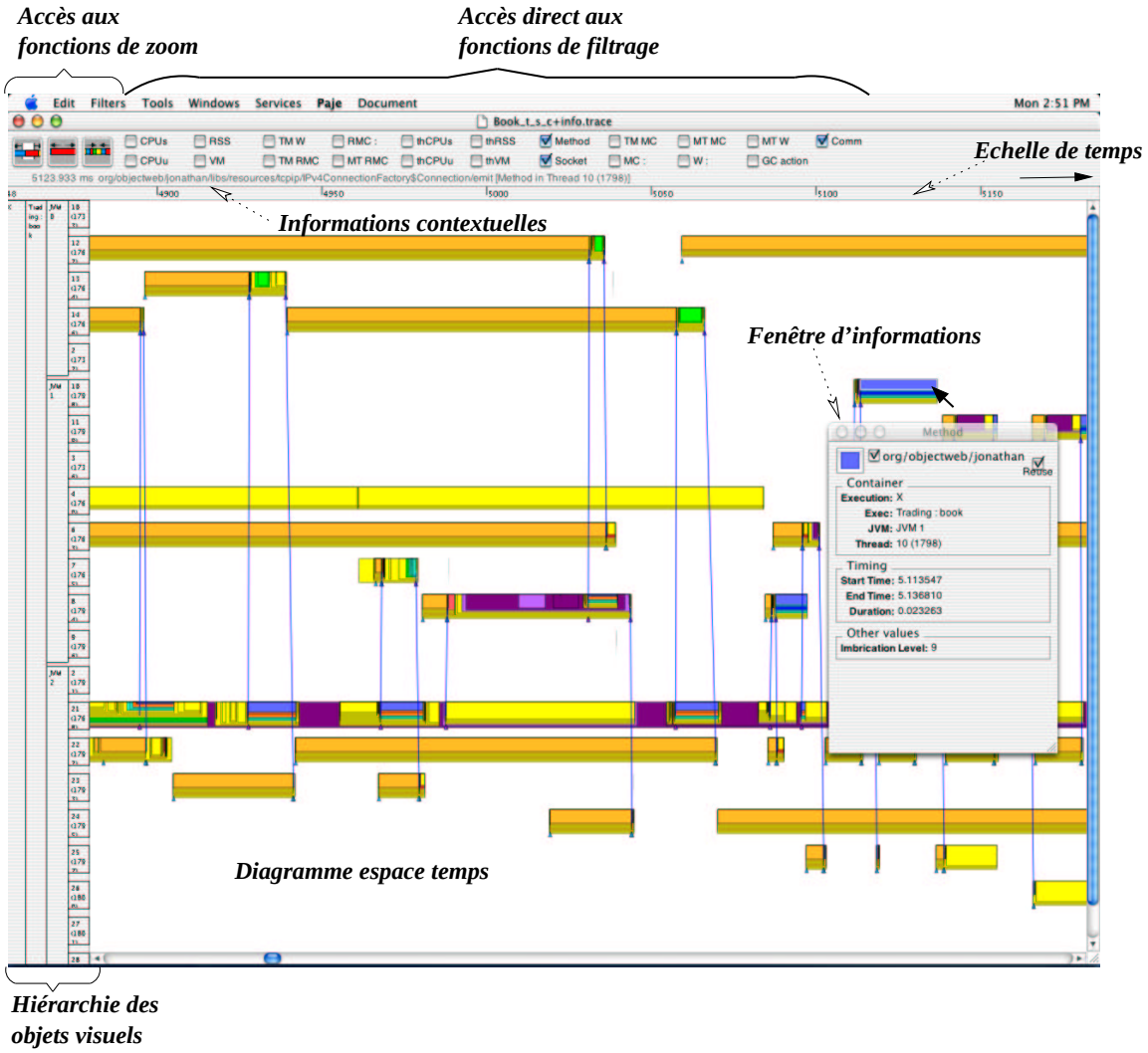


FIG. 4.8 – Visualisation dans “Pajé” d’une trace événementielle par un diagramme espace temps.
La fenêtre de visualisation du diagramme espace temps se divise en deux parties. La barre supérieure qui rassemble les principales fonctionnalités de “Pajé”, et la zone inférieure qui représente la trace événementielle. La sélection d’une entité permet d’obtenir une nouvelle fenêtre contenant la totalité des informations qui s’y rapportent.

4.6 Conclusion

Dans ce chapitre nous avons abordé les problèmes liés à l’interprétation des observations multi-niveaux collectées dans un système réparti. Puis, nous avons traité de la construction d’un système de datation globale dont la sémantique est maintenue au travers des niveaux d’abstraction et sur la totalité du système. L’implantation d’un tel système est présentée dans le chapitre 6 (voir section 6.4). Enfin, nous avons présenté le problème des perturbations liées à l’observation qui doivent être prises en compte afin de limiter ou corriger les écarts à l’exécution sans observations.

Il ressort que l’analyse d’une trace événementielle devient possible une fois les prétraitements

réalisés. L'analyse est statique si l'on cherche à dégager le comportement global de l'application, sans se soucier de l'ordre des événements. Elle devient dynamique si l'on souhaite reconstruire les chemins d'exécution et identifier leurs interdépendances. "Pajé" a été choisi comme outil de visualisation pour mener à bien ce type d'analyse.

La chapitre suivant traitera de la mise en œuvre d'une plate-forme d'observation pour application Java distribuée. C'est sur ce type d'application que nous avons validé notre infrastructure d'observation et d'analyse.

Chapitre 5

Mise en œuvre de la plate-forme d'observation

Ce chapitre présente l'implantation d'une infrastructure d'observation multi-niveaux pour application distribuée. Cette infrastructure a été développée dans le cadre du projet Jonathan[1]. Les observations portent donc sur l'exécution d'applications Java.

Une application Java s'exécute au sein d'une "*Java Virtual Machine*" (JVM). Les JVM utilisent les abstractions de communication fournies par le système d'exploitation hôte pour interagir avec l'extérieur.

Nous allons donc décrire les techniques et moyens mis en œuvre afin d'obtenir une trace événementielle de l'exécution d'une application Java répartie.

5.1 L'observation multi-niveaux

Le système observé est modélisé comme la superposition des niveaux d'abstraction applicatif et système[2] (voir section 3.1). Nous allons commencer par décrire les moyens mis en œuvre pour réaliser les observations du niveau applicatif.

5.1.1 Observations du niveau applicatif

Les observations du niveau applicatif sont obtenues au travers de la "*Java Virtual Machine Profiling Interface*" (JVMPI). Il s'agit d'une interface fonctionnelle permettant d'observer l'occurrence

[1] Projet développé au sein du département DTL/ASR de France Télécom R&D

[2] Le niveau d'abstraction matériel ne sera pas traité dans ce travail.

d'événements du niveau applicatif et de réaliser des opérations de contrôle sur la JVM (voir figure 5.1).

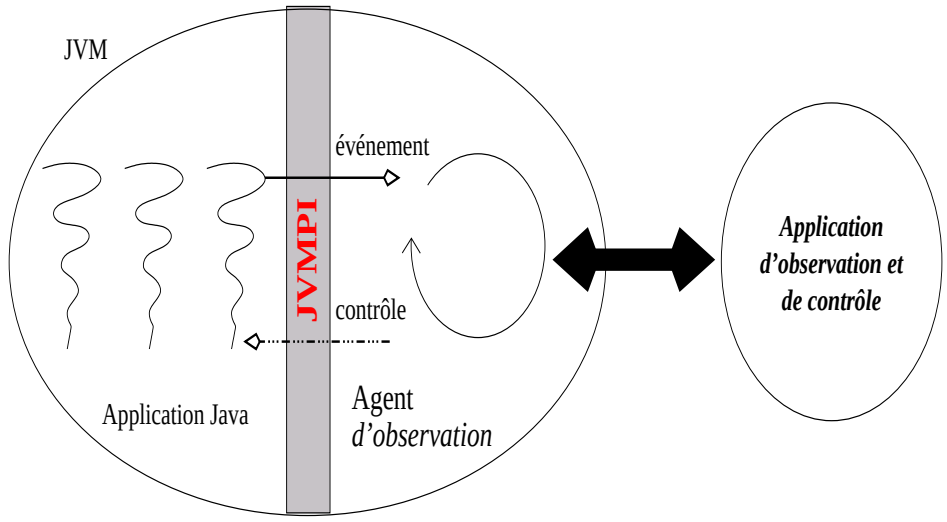


FIG. 5.1 – Modèle général de la JVMPI.
La JVMPI définit une interface fonctionnelle d'observation et de contrôle, ainsi que la possibilité de connecter la JVM à une application extérieure.

Il est ainsi possible d'observer des événements relatifs à l'exécution de l'application Java mais aussi des événements se rapportant à l'exécution de la JVM. Les événements applicatifs correspondent, par exemple, au chargement d'une classe, au début ou à la fin d'exécution d'une méthode. Les événements se rapportant à l'exécution de la JVM correspondent, par exemple, à l'exécution du ramasse-miettes, à la création ou la terminaison d'un "*thread*" Java.

La JVMPI ouvre une porte entre la JVM et son contexte d'exécution. Elle permet de charger le code applicatif d'un agent d'observation ("*Profiling Agent*"), qui implante la réaction que l'on souhaite adopter lors de l'observation d'un événement, ou lorsqu'une sollicitation de contrôle est générée.

L'agent d'observation est basé sur un modèle d'exécution de type événement/action (comme c'est le cas dans OMIS (voir section 1.4.1)). Il est chargé par la JVM lors de son initialisation. Il instancie la fonction `void notifyEvent(JVMPI_Event *event)` qui définit les actions à réaliser lors de l'observation d'un événement ou la sollicitation d'une action de contrôle. Cette fonction possède comme paramètre le contexte dans lequel les événements prennent place. Il se présente sous la forme d'une union de structures définissant le type de l'événement observé ainsi que des informations circonstancielles (voir figure 5.2).

Par défaut, l'agent d'observation ne prend en compte aucun événement. Les événements sont observés par type. Un type correspond, par exemple, au début ou fin d'exécution de méthode. Pour qu'un événement d'un type défini soit observé, il doit être enregistré par le biais de la fonction de contrôle

```
typedef struct {
    jint event_type;                    /* event_type */
    JNIEnv *env_id;                     /* env where this event occurred */

    union {
        struct {
            char *class_name;           /* class name */
            char *source_name;          /* name of source file */
            jint num_interfaces;        /* number of interfaces implemented */
            jint num_methods;           /* number of methods in the class */
            JVMPI_Method *methods;      /* methods */
            jint num_static_fields;     /* number of static fields */
            JVMPI_Field *statics;       /* static fields */
            jint num_instance_fields;   /* number of instance fields */
            JVMPI_Field *instances;     /* instance fields */
            jobjectID class_id;         /* id of the class object */
        } class_load;

        struct {
            jobjectID class_id;         /* id of the class object */
        } class_unload;

        ... /* Refer to the section on JVMPI events for a full listing */
    } u;
} JVMPI_Event;
```

FIG. 5.2 – Structure de données de la JVMPI.
La structure `JVMPI_Event` possède deux attributs communs à l'ensemble des événements : l'identifiant de l'événement (`jint event_type`) et un pointeur sur le "*thread*" courant (`JNIEnv *env_id`). Le reste de la structure est composé d'une union de structures propre à chaque événement.

`jint (*EnableEvent)(jint event_type, void *arg)`. Une fois l'enregistrement effectué, tous les événements de début ou fin d'exécution de méthode sont remontés à l'agent d'observation, sans aucune distinction. Le mécanisme inverse est aussi prévu (`jint (*DisableEvent)(jint event_type, void *arg)`) afin de stopper l'observation.

Ainsi, la JVMPI donne accès à l'observation d'événements génériques tels que le début ou la fin d'exécution d'une méthode. Par contre, aucun mécanisme n'est prévu pour filtrer les événements. Il n'est ainsi pas possible de n'observer que les événements se rapportant à une méthode donnée. Les fonctions de filtrage doivent être implantées à l'intérieur de l'agent d'observation.

Dans notre cas, le filtrage a lieu sur l'identifiant des méthodes à observer. Lorsqu'une classe est chargée, une fonction de filtrage indique si les méthodes définies dans cette classe doivent être observées. Le filtre peut éliminer toutes les méthodes d'une classe, ou seulement un sous-ensemble de celles-ci. Dans le premier cas, la fonction de filtrage se base sur le nom de la classe pour savoir si les méthodes sont à observer. Dans le second cas, chaque méthode est traitée séparément. La fonction

de filtrage peut agir par affirmation (on définit les méthodes à observer) ou par négation (on définit les méthodes à ne pas observer). L'approche est généralisable en définissant une fonction d'évaluation, de complexité arbitraire, pour le filtrage. De cette façon, l'agent d'observation choisit les classes d'événements, et même plus précisément les événements qu'il veut observer. Son implantation définit les actions déclenchées pour chaque type d'événement.

Dans le cadre de notre travail, les actions à réaliser correspondent à l'enregistrement des événements permettant de reconstruire la dynamique d'exécution de l'application. Pour cela, les événements sont datés par le système de datation locale (voir section 4.1) et une trace typée leur est associée. Elle correspond à :

La création et la destruction des *"threads"*

L'événement Java capté lors de la création d'un "*thread*" est `JVMPI_EVENT_THREAD_START`. La trace enregistrée est constituée de l'identifiant du "*thread*" au sein de la JVMPI (`jobjectID thread_id`) et son identifiant au sein du système d'exécution (i.e. l'identifiant du processus ou "*thread*" système [3] qui lui sert de support d'exécution).

Ces informations servent à identifier le «lieu d'exécution» des méthodes et ainsi à réaliser la correspondance avec les consommations de ressources système (voir section 3.1).

Lors de la destruction d'un "*thread*", un événement `JVMPI_EVENT_THREAD_END` est observé. Il définit la portée du "*thread*" qui l'a généré. La trace générée est formée de l'identifiant JVMPI de ce dernier.

Le chargement d'une classe

L'événement Java capté est `JVMPI_EVENT_CLASS_LOAD`. La structure de données qui lui est associée possède entre autres attributs le nombre de méthodes définies dans la classe et un pointeur sur une liste de type `JVMPI_Method *`. Il permet donc de définir les méthodes à observer.

Si la classe n'est pas filtrée, les traces générées correspondent à l'identification de la classe et la description de ses méthodes. La trace associée aux méthodes non filtrées comprend le nom de la méthode, les numéros de lignes de début et fin du code associé, et de l'identifiant JVMPI `jmethodID method_id`. Cet identifiant est placé dans une table de hachage qui sert à déterminer les méthodes à observer durant l'exécution.

Le début et la fin d'exécution des méthodes

L'événement Java capté lors d'un début de méthode est `JVMPI_EVENT_METHOD_ENTRY2`. La structure de données associée fournit l'identifiant JVMPI de la méthode ainsi que l'identi-

[3] Dans la suite de ce chapitre, le terme processus englobera les notions de processus lourd et "*thread*", sauf mention explicite.

fiant de l'objet sur lequel la méthode est appelée.

L'identifiant JVMPI est recherché dans le table de hachage construite lors du chargement des classes. S'il fait partie des éléments de la table une trace est générée, sinon, l'événement est ignoré.

La trace est constituée de l'identifiant du contexte d'exécution (le "*thread*" Java qui exécute la méthode) ainsi que celui de la méthode et de l'objet.

L'événement Java capté lors d'une fin de méthode est `JVMPI_EVENT_METHOD_EXIT`. L'information attachée à cet événement est l'identifiant JVMPI de la méthode qui se termine. Là encore, une recherche est réalisée dans la table de hachage afin de déterminer si la méthode est à observer ou non. Si oui, une trace formée de l'identifiant du contexte d'exécution et de l'identifiant de la méthode est produite. Il est toutefois à signaler que la date associée à la terminaison d'une méthode est à prendre avec précaution. Il n'est en effet pas garanti que le "*thread*" système ne soit pas "dé-ordonnancé" entre l'instant où la méthode se termine et celui où l'événement est observé (voir figure 6.15).

Ainsi, l'observation des événements de début et fin d'exécution identifient les méthodes exécutées, le nombre de fois qu'elles sont appelées, par qui, et pour quelle durée [4].

L'utilisation des objets de synchronisation

Ils permettent de détecter les situations d'attente et de blocage provoquées par l'utilisation des objets de synchronisation.

A chaque objet Java est associé un verrou [**JVMSpec, 1999**]. Il n'existe pas d'opération explicite de prise ou de relâche du verrou. Les opérations sont réalisées implicitement en accord avec le sémantique du langage. Toutefois, les mécanismes internes de la JVM réalisent explicitement ces opérations [5]. Une méthode «synchronisée» (`synchronized`) réalise une opération de prise de verrou avant de s'exécuter. Son exécution ne commence qu'après l'acquisition du verrou. Si la méthode invoquée est une méthode d'instance, le verrou utilisé est celui de l'instance de classe. Si la méthode est statique, le verrou utilisé est celui de l'objet `Class` qui représente la classe dans laquelle la méthode est définie. Le verrou est relâché lors de la terminaison de la méthode.

Une mise en attente sur un objet de synchronisation est signalée par l'événement `JVMPI_EVENT_MONITOR_WAIT`, et la fin de l'attente par l'événement `JVMPI_EVENT_MONITOR_`

[4] Comme il n'est pas possible de garantir la qualité de l'événement de terminaison de méthode, les durées calculées sont à analyser en tant que majorant de la durée réelle.

[5] Elles apparaissent sous la forme des opérations `monitorenter` et `monitorexit`.

WAITED. La structure de données associée à ces événements procure l'identification du contexte d'exécution, de l'objet de synchronisation employé (`jobjectID object`), et du délai maximum d'attente fixé (`jlong timeout`). Ce sont les informations contenues dans la trace générée.

Des situations de blocage sont provoquées par l'utilisation de verrous. Ils se classent en deux catégories, les verrous du niveau JAVA (`MONITOR_CONTENDED`) et les verrous du niveau système (`RAW_MONITOR_CONTENDED`). Trois événements leurs sont associés :

– l'événement qui identifie le "*thread*" d'où provient la demande d'acquisition du verrou : `JVMPI_EVENT_MONITOR_CONTENDED_ENTER` et `JVMPI_EVENT_RAW_MONITOR_CONTENDED_ENTER`;

– l'événement qui identifie le "*thread*" ayant obtenu le verrou : `JVMPI_EVENT_MONITOR_CONTENDED_ENTERED` et `JVMPI_EVENT_RAW_MONITOR_CONTENDED_ENTERED`;

– l'événement qui identifie le "*thread*" qui est à l'origine de la libération du verrou : `JVMPI_EVENT_MONITOR_CONTENDED_EXIT` et `JVMPI_EVENT_RAW_MONITOR_CONTENDED_EXIT`.

Les traces produites sont constituées de l'identifiant du contexte d'exécution et de l'identifiant de l'objet de synchronisation (`jobjectID object`).

Elles permettent de reconstruire les relations de dépendances algorithmiques (sérialisation) et structurales (utilisation de verrous par le système hôte) associées à une exécution.

Le début et la fin d'exécution de ramasse-miettes

L'événement de début est `JVMPI_EVENT_GC_START` et celui de fin `JVMPI_EVENT_GC_END`. L'événement de début sert à identifier quel "*thread*" est à l'origine de cette action. Celui de fin indique la quantité d'objets et de cellules mémoires qui ont été libérés.

Les opérations de contrôle gèrent l'exécution de la JVM et son comportement vis-à-vis de l'agent d'observation. Elles peuvent, par exemple, stopper un "*thread*" ou initier l'exécution du ramasse-miettes. C'est au travers de ces opérations que l'agent d'observation enregistre les types événements à observer. Dans un souci de limiter au maximum les perturbations liées à l'observation, c'est la seule situation pour laquelle nous utilisons les opérations de contrôle.

Les informations obtenues au travers de la JVMPI nous offrent la possibilité de reconstruire la dynamique d'exécution au sein des JVM. Il nous faut à présent obtenir des informations du niveau système pour relier l'exécution des différentes JVM et évaluer leur consommation en ressources d'exécution.

5.1.2 Observations du niveau système

Les informations obtenues au niveau applicatif permettent de reconstruire l'exécution au sein d'une JVM. Par contre, il est difficile de reconstruire ses interactions avec d'autres JVM. Si l'on prend l'exemple des communications, comme les paramètres d'appels de méthodes ne font pas partie du contexte obtenu lors de l'observation d'un début de méthode, il n'est pas possible de savoir avec qui l'interaction prend part ni quelle quantité d'information est transmise. Pour cela, nous avons choisi d'observer les mécanismes mis à contribution au niveau du système d'exploitation pour les communications entre JVM.

Les observations du niveau système [6]sont obtenues en déroutant des appels système définis dans le noyau Linux. Chacun de ces appels est référencé par un pointeur de fonction. L'ensemble des pointeurs est rassemblé dans un vecteur de pointeurs de fonctions (`sys_call_table`). Chaque appel système est identifié par un numéro correspondant à son rang dans le vecteur. Dans ce cadre, dérouter un appel système signifie remplacer le pointeur de la fonction initiale par un pointeur sur sa propre fonction (voir figure 5.3).

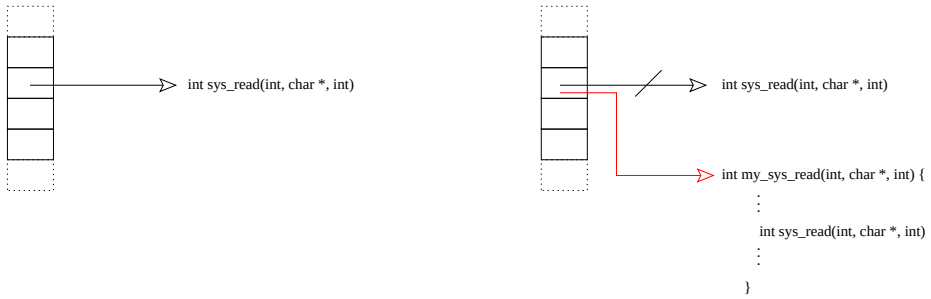


FIG. 5.3 – Schéma général de déroutage d'un appel système.

De cette façon, il est possible de modifier le comportement voire même la sémantique d'un appel système. Cette technique est utilisée pour insérer le code d'observation dans le noyau. L'insertion est réalisée à chaud, par substitution de la fonction originale par du code compilé contenant l'appel à la fonction initiale et le mécanisme d'observation. Il s'agit globalement de la même technique que celle utilisée par Paradyn [**Tamches et Miller, 1999**], si ce n'est que la modification du code original est réalisée par insertion d'un module.

L'architecture des noyaux Unix a évolué d'un modèle monolithique vers un modèle à modules [**Tanenbaum, 1994**]. Dans ce nouveau modèle, le noyau est composé d'une partie "principale" et de modules contenant du code spécialisé inséré sur demande. De cette manière, il est possible de limiter la taille et les fonctionnalités du noyau en n'y intégrant que le minimum pour l'amorçage du système [**Card et al., 1998**]. Si l'on prend l'exemple du noyau Linux (à partir de la version 2.x), il est possible

[6]Dans la suite de cette section le terme *système* sera employé sans distinction avec *système d'exploitation* sauf mention explicite.

de définir un noyau minimal qui est étendu par des modules correspondant par exemple aux pilotes de carte réseau ou à des services tel que NFS ("*Network File System*"). Dans ce cas, le module NFS n'est chargé que si ce type de système de fichier est utilisé et il est déchargé lorsqu'il ne l'est plus. Ce modèle à modules permet donc de rajouter ou de modifier dynamiquement des fonctionnalités du noyau. C'est la solution qui a été retenue pour observer les communications entre JVM.

Lorsqu'une JVM utilise un mécanisme de "*socket*" pour communiquer, cela se traduit par une série d'appels système sur un descripteur de fichier typé (voir figure 5.4). Ainsi, les lectures sur une "*socket*" Java correspondent à l'exécution d'appels système `read()`. Les écritures sur une "*socket*" Java correspondent à l'exécution d'appels système `socketcall()`. A chaque "*socket*" Java est associé un descripteur de fichier représentant une "*socket*" système. Ce descripteur permet d'identifier le lien de communication point à point [7] associé à la "*socket*". Les ports de communications et les adresses IP des machines offrent une clé d'identification univoque des participants à la communication.

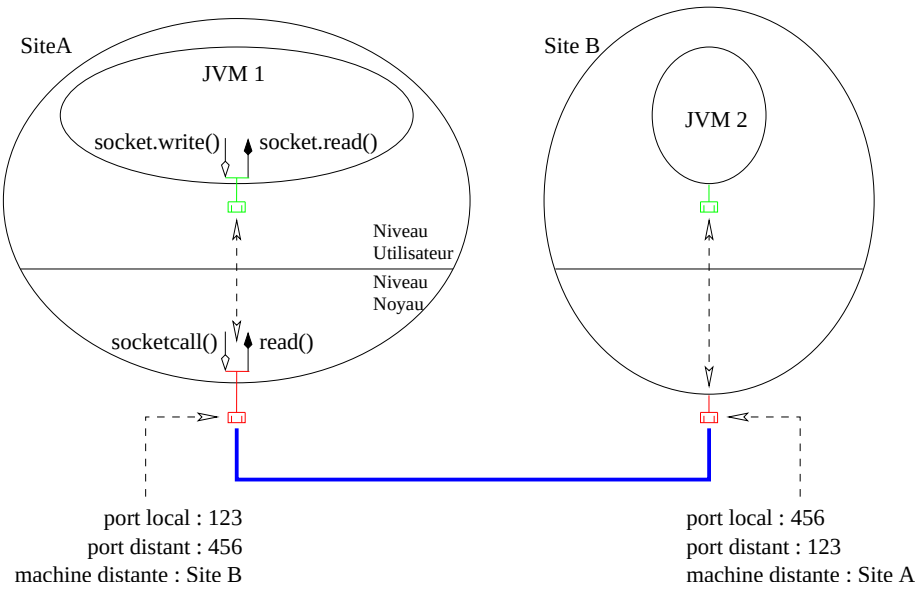


FIG. 5.4 – Implantation des "*socket*" Java.

Ainsi, l'observation des communications est réalisée au travers des appels système `read` et `socketcall` sur les `sockets` qui leurs sont associés. Pour cela, les deux appels système sont déroutés afin d'être remplacés par nos propres appels système (voir figure 5.5). Cependant, ces appels système sont utilisés par l'ensemble des processus présents dans le système. Il est donc nécessaire de mettre en œuvre une fonction de filtrage afin de n'observer que les appels provenant des JVM observées.

La fonction de filtrage est basée sur l'identification du processus qui est à l'origine de l'appel système. Lors du déroutage, on accède au contexte d'exécution, c'est-à-dire à la structure `task_struct`

[7] Dans la mesure où le lien de communication ne correspond pas à une diffusion ou une diffusion par groupe.

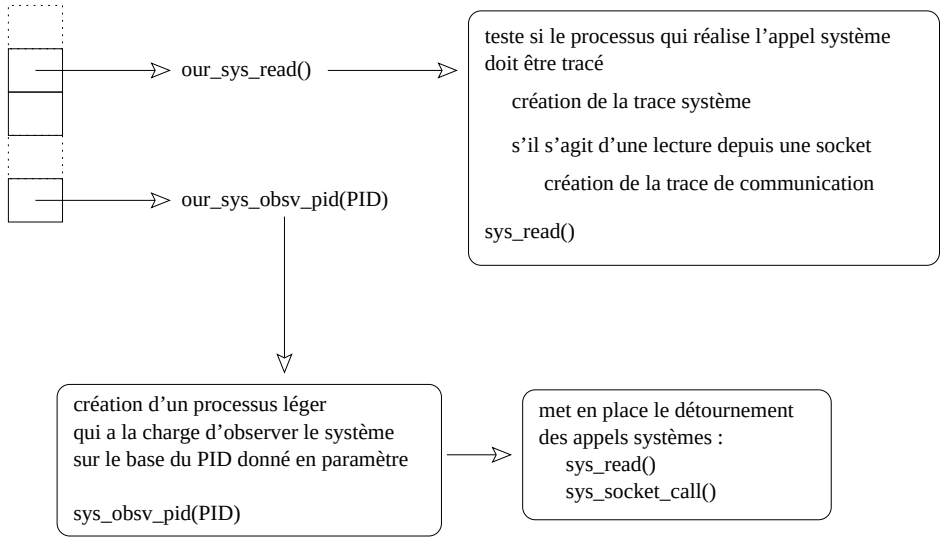


FIG. 5.5 – Déroutage des appels système utilisés dans les communications entre JVM.
L'observation des communications est réalisée en déroutant les deux appels système : `read()` et `socketcall()`. Ils possèdent comme paramètres un descripteur de fichier, qui identifie les participants à la communication, ainsi que la quantité de données transférées.

associée au processus actif. Cette structure contient entre autres données, l'identifiant du processus concerné. Pour savoir si le PID obtenu représente un processus à observer, nous le comparons avec le PID du premier processus qui a initialisé la JVM : le PID_0. Si le PID du processus actif (PID_i) est égal au PID_0, ou s'il s'agit d'un de ses fils [8] , le code d'observation est exécuté.

Dans le cas d'une lecture

Le descripteur de fichier associé à la lecture fait référence à un nœud d'information système (`inode`)[9]. S'il s'agit d'une "*socket*", une trace de communication est générée. Elle consiste en deux points de mesures : un premier avant l'appel effectif de `read()`, et second après sa terminaison ; ce qui nous permet de mesurer la durée d'exécution de l'appel système. Les informations collectées sont :

- la date courante ;
- le PID du processus ;
- le numéro du port de communication local et distant ;
- l'adresse IP de la machine distante ;
- la taille du tampon allouée, et la quantité de données lues.

[8] Lorsqu'un processus crée un processus ou un "*thread*", ce dernier est considéré comme étant un de ses fils. Cette relation de parenté est transitive. Il est donc possible de remonter l'arbre des ancêtres jusqu'au processus `init` dont le PID vaut 1.

[9] Tous les fichiers présents dans le système possèdent un `inode` . Un `inode` est typé, c'est-à-dire que la structure de données qui lui est associée permet de déterminer s'il s'agit d'un fichier, d'un tube ou d'une "*socket*".

Dans le cas d'une écriture

Le descripteur de fichier réfère toujours une "*socket*". Par contre, il sert de point d'entrée pour plusieurs opérations sur les "*socket*" système [**Stevens, 1990**, **Card et al., 1998**]. Parmi les opérations accessibles par `socketcall()`, nous nous intéressons uniquement à `SYS_SEND` qui est employé lors de l'envoi de paquet. Les informations collectées sont :

- la date courante ;
- le PID du processus ;
- le numéro du port de communication local et distant ;
- l'adresse IP de la machine distante ;
- la taille du tampon alloué,et la quantité de données écrites.

Le déroutage des appels système est réalisé sous un contexte d'exécution noyau [10]. Il a ainsi accès à la structure de contrôle attachée à tout processus (`task_struct`) qui est tenue à jours par le système Linux dans notre cas. Cette structure est représentative du contexte d'exécution dans lequel se trouve le processus. Elle contient, entre autres informations, des indices de consommation de ressources d'exécution tels que : la durée d'exécution effective (en mode utilisateur et noyau), la quantité de mémoire utilisée (taille virtuelle, taille en mémoire, taille de l'exécutable...), ou le nombre de défauts de page (mineurs et majeurs)[11].

Ces indices sont significatifs du comportement de l'application en tant que telle, mais aussi de la façon dont elle s'exécute dans le système.

La durée d'exécution effective

Elle est représentative de l'utilisation de la ressource d'exécution processeur. Grâce à cet indice, il est possible d'évaluer le degré de concurrence dans le système. En effet, l'ordonnanceur standard de Linux tente de répartir équitablement la ressource processeur entre les différents processus. [**Bouchi et al., 2001**] propose comme indice de performance le temps nécessaire à exécuter une boucle d'invocations à la fonction `yield()` [12] par un "*thread*" Java. Cet indice permet d'évaluer la disponibilité de la ressource processeur. Toutefois, il est à mettre en regard avec les attentes sur entrées/sorties qui ne consomment pas ce type de ressource.

[10] Afin d'implanter un modèle de protection deux modes d'exécutions sont possibles : utilisateur et noyau. Le mode utilisateur limite les droits du processus et ne lui donne accès qu'à son propre espace d'adressage et aux segments de mémoire partagée. C'est le mode dans lequel s'exécute les JVM. Au contraire, en mode noyau, le processus possède tous les droits d'exécution et d'accès à n'importe quelle adresse mémoire. Il a aussi accès à l'ensemble des informations statistiques et de contrôle tenues à jour par le système.

[11] La notion de «défaut de page» est directement liée aux mécanismes de gestion de la mémoire du noyau hôte. Il existe deux types de défaut de page : mineur et majeur. Un défaut de page mineur correspond à la situation où le processus à besoin d'une nouvelle page mémoire. Un défaut de page majeur est provoquée lorsque le système de gestion de la mémoire accède à la zone de permutation.

[12] Cette fonction permet à un "*thread*" d'être "dé-ordonnancé" avant la fin de son quantum d'exécution.

De même, l'utilisation de la ressource processeur est réalisée dans deux modes d'exécution : utilisateur et système. Un processus s'exécute en mode système lorsqu'il utilise les fonctionnalités de bas niveau du système. Ces fonctionnalités correspondent, par exemple, à la réalisation effective de lectures ou écritures dans le système [13]. Il s'agit d'un mode d'exécution plus prioritaire et plus « dangereux » puisqu'il donne accès à la totalité de l'espace d'adressage du système.

La quantité de mémoire utilisée et le nombre de défauts de pages

Elle est représentatif de l'utilisation de la ressource d'accès aux données locales. Le modèle général d'exécution d'une application fait intervenir deux types de ressources : les ressources d'exécution et les ressources d'accès aux données. Dans ce modèle, la ressource d'accès aux données locales correspond à l'espace d'adressage utilisé par l'application. Cet espace est fourni et géré par le système. Si l'on considère l'exemple du système d'exploitation Linux, un espace d'adressage virtuel est associé aux processus. Le système fait la correspondance entre cet espace virtuel et ses mécanismes de gestion de la mémoire. Dans ce cas précis, la mémoire est gérée sous forme de pages de 4 ko qui sont réparties entre la mémoire centrale et une zone de permutation sur un média de sauvegarde de type disque dur. Ainsi, lorsqu'un processus demande à accéder à une adresse de l'espace d'adressage virtuel, le système fait la correspondance avec la ou les pages de mémoires correspondantes.

L'étude de ces indices joue sur deux points de vues : le placement des données et l'utilisation des médias de stockage secondaires [14]. En effet, un placement inapproprié des données provoque de nombreux défauts de pages. Lorsque celles-ci sont de type «majeur», un accès bloquant au média de stockage secondaire est réalisé. Or les temps d'accès à ce type de média sont communément de l'ordre de la milliseconde. Il s'agit donc d'un facteur important de dégradation des performances.

De plus, dans le cas particulier des exécutions Java, l'utilisation de mémoire est aussi gérée par la JVM. Des actions de «ramasse-miettes» sont initiées dans le but de libérer la mémoire qui n'est plus utilisée. Ces actions ont lieu soit sur demande, soit lorsque la mémoire virtuelle a atteint une taille prédéfinie. Or, ces actions sont généralement très coûteuses. Dans le cas de la JVM de Sun, le comportement standard du ramasse-miettes est de stopper l'ensemble des "*threads*" Java afin de vérifier la totalité des références sur les objets présents dans le tas. Les objets ne possédant plus de référence sur eux sont «libérés». Ce mécanisme permet donc de limiter l'utilisation de la ressource physique mémoire, mais induit un surcoût d'exécution important lors de sa mise en œuvre.

[13] Le niveau utilisateur exporte des abstractions de haut niveau de ces opérations de base. Le programme utilise ces abstractions qui lui permettront d'accéder aux fonctions système. L'intérêt de cette méthode et de mettre en place un modèle de protection afin de limiter la portée du processus, et de réaliser des actions de contrôle avant de lui permettre de s'exécuter en mode privilégié (noyau)

[14] les disques durs, CD-ROM, DVD, lecteur de bandes...

La quantité de données écrites et/ou lues sur les "*socket*"

Elle est représentative de l'utilisation de la ressource d'accès aux données distantes. La problématique de performance est du même ordre que celle du placement des données. Tout accès à une ressource distante est au minimum de l'ordre de la milliseconde. Lorsque l'infrastructure de communication est du type TCP/IP, les durées de communication ne sont pas bornées. Il est ainsi tout à fait possible qu'une application reste bloquée en attente de réception de données.
En analysant la quantité de données transmises et les durées de communication, il est possible de faire apparaître le ratio communication sur calcul. Un placement judicieux des données et de services nécessaires à l'exécution, peut permettre de minimiser l'impact des attentes dues aux communications en les recouvrant par du calcul [**Galilée et al., 1998**, **Cavalheiro, 1999**].

Ainsi, grâce aux informations obtenues au niveau système, il devient possible de tracer les communications réalisées entre JVM. Ces traces sont représentatives des appels de méthode distante réalisés durant l'exécution. De plus, nous avons la possibilité d'observer la consommation des ressources nécessaires à l'exécution.

Il nous faut à présent définir les modalités de conversion pour passer d'observations brutes à des informations propres à l'analyse.

5.2 Traitement des données

Nous venons de décrire les mécanismes mis en œuvre afin d'observer une application Java distribuée. Le résultat de cette observation est une trace événementielle multi-niveaux. Toutefois, cette trace n'est pas analysable en l'état. Elle doit être convertie en un format lié à l'analyse que l'on souhaite mener.

Dans le cadre de l'analyse de cette trace par un outil de visualisation, le format cible est celui de "Pajé". Nous allons à présent détailler la construction de la fonction de conversion développée.

5.2.1 Le langage de commande générique de "Pajé"

"Pajé" possède un composant de simulation générique contrôlé par des événements spéciaux contenus dans le fichier de trace. Ce fichier est divisé en deux parties :

Un en-tête

Elle contient l'ensemble des entités et conteneurs qui sont interprétés par le simulateur, ainsi que leur signature.

Le corps de la trace

Il représente les observations traduites en fonction du contenu de l'en-tête.

C'est dans ce fichier qu'est défini et déclaré le type des objets visuels qui sont employés. Une fois la fichier de trace construit, le simulateur crée des instances de ces objets et leurs applique les opérations propres à leur sémantique [**de Oliveira Stein, 1999**].

Ainsi, si l'on considère l'exemple d'un modèle d'exécution basé sur des objets s'exécutant sur des "*threads*", la visualisation des appels de méthodes nécessite (voir figure 5.6) :

- de définir un conteneur représentant l'exécution ;
- de définir un conteneur JVM ;
- de définir un conteneur "*thread*" ;
- de définir les états associés au conteneur "*thread*" (l'ensemble des méthodes observées) ;
- de créer une instance de JVM ;
- de créer les instances de "*thread*" ;
- de simuler les débuts et fins d'appel de méthode qui sont représentés sur l'instance de "*thread*" correspondante.

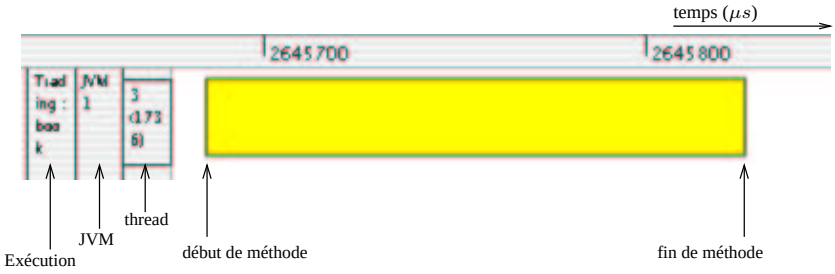


Figure 5.6: Représentation d'un appel de méthode par l'environnement de visualisation "Pajé".

On voit ici apparaître une hiérarchie d'objets visuels représentative du modèle d'exécution de l'application. Cette hiérarchie s'exprime en termes de conteneurs et d'entités (voir section 4.5). Les entités sont de quatre types : état, événement, lien, valeur.

Une entité de type "état"

Elle représente l'état d'un conteneur. Ce qui peut correspondre, par exemple, à l'exécution

d'une méthode par un "*thread*". Elle est représentée sous la forme d'une boîte. Les boîtes peuvent être imbriquées et ainsi représenter la pile d'appel de méthode associée à un "*thread*" (voir figure 5.7).

Une entité de type "événement"

Elle isole une date particulière à laquelle est reliée une observation, par exemple le début de réception d'une communication. Elle est représentée par un triangle dont la pointe supérieure identifie la date de l'événement.

Une entité de type "lien"

Elle relie deux conteneurs quelconques entre une date de début t_d et une date de fin t_f, par exemple le début et la fin d'une communication. Elle est représentée par un trait qui relie les deux conteneurs.

Une entité de type "valeur"

Elle associe une valeur numérique à un intervalle de temps. Elle est utilisée, par exemple, pour visualiser la quantité de mémoire utilisée par un "*thread*". Elle est représentée par un ensemble de segments correspondant aux valeurs observées.

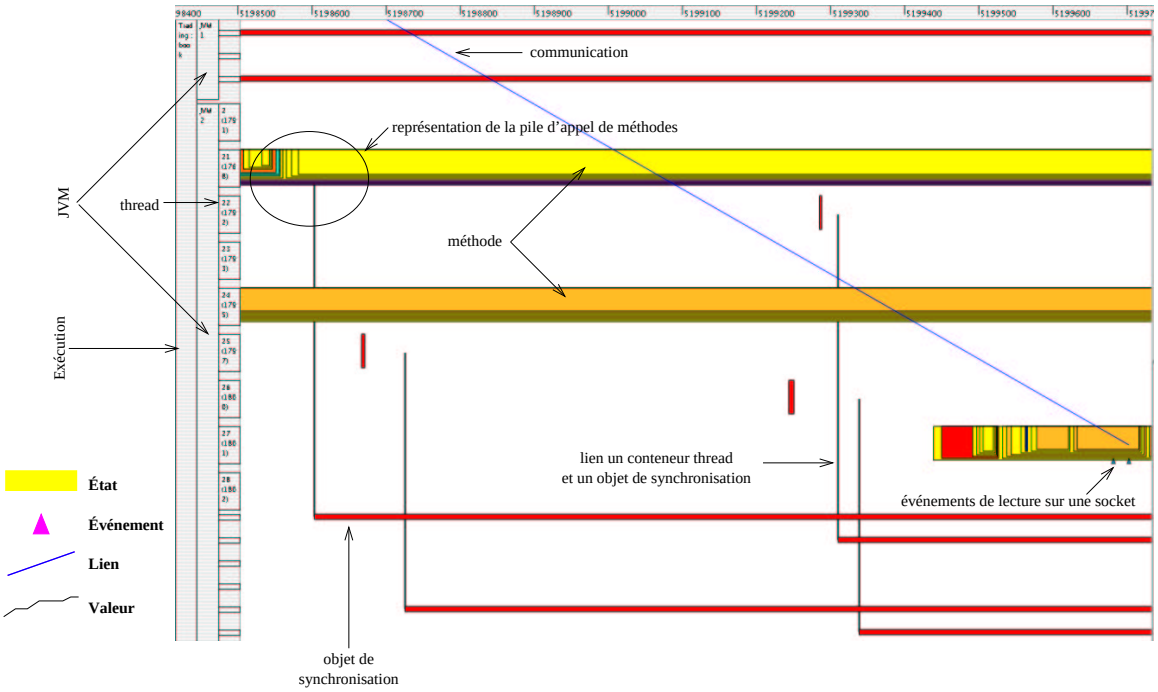


Figure 5.7: Sémantique des objets visuels utilisés dans "Pajé".
La figure 5.6 représente de façon détaillée la règle latérale et supérieure.

Les conteneurs servent à structurer, sous la forme d'un arbre, la hiérarchie d'objets visuels. Ils peuvent s'emboîter afin de représenter les différents niveaux d'abstraction du modèle d'exécution. Du point de vue de "Pajé", on obtient une hiérarchie de types d'objets visuels qui est calculée au fur et à mesure de la lecture de la trace par le simulateur. Elle est ensuite fournie aux autres modules de "Pajé".

5.2.2 Construction d'une trace "Pajé"

Nous nous plaçons dans la cadre de l'**analyse post-mortem** d'une trace d'exécution. Il nous est ainsi possible de disjoindre totalement l'activité de collecte des données de l'interprétation et de la visualisation. *Les transformations opérées pour passer de données brutes, issues de l'observation, à des informations interprétables définissent une fonction de conversion.* Dans le cas présent, les données proviennent de traces réalisées au niveau applicatif et système. Elles doivent être transformées en une trace interprétable par "Pajé". Le modèle de programmation et la sémantique attachée aux données s'expriment conjointement dans la construction de l'en-tête de la trace "Pajé".

L'en-tête de la trace est propre à un modèle d'exécution. Le corps de la trace est formé de lignes qui correspondent à la traduction des événements applicatifs et systèmes en événements "Pajé" [**de Oliveira Stein, 1999**]. Les deux premières colonnes des lignes identifient le type de l'événement "Pajé" et sa date d'occurrence. Les dates sont strictement croissantes. Les deux sections suivantes décrivent la conversion des observations en une trace "Pajé".

5.2.2.1 Construction à partir de la trace applicative

Les observations du niveau applicatif sont le reflet de l'activité des JVM en termes :

d'exécution de l'application au travers des appels de méthodes ou de l'utilisation des objets de synchronisation.

d'exécution de la JVM [15] par le biais des événements liés à la gestion des "*threads*" (création, destruction), ou à la gestion de mémoire (exécution du ramasse-miettes).

Le modèle d'exécution proposé repose sur la notion de "*threads*" servant de support à l'exécution de méthodes. Dans la hiérarchie de type d'objets visuels "Pajé" (voir figure 5.8) le conteneur de plus haut niveau représente l'exécution, il inclut les conteneurs JVM qui incluent eux mêmes les conteneurs "*threads*". Aux conteneurs "*threads*" on associe l'entité de type état «*méthode*». Les

[15] Qui, du point de vue du niveau applicatif, représente l'infrastructure d'exécution.

valeurs possibles de cette entité correspondent à l'ensemble des méthodes observées[16].

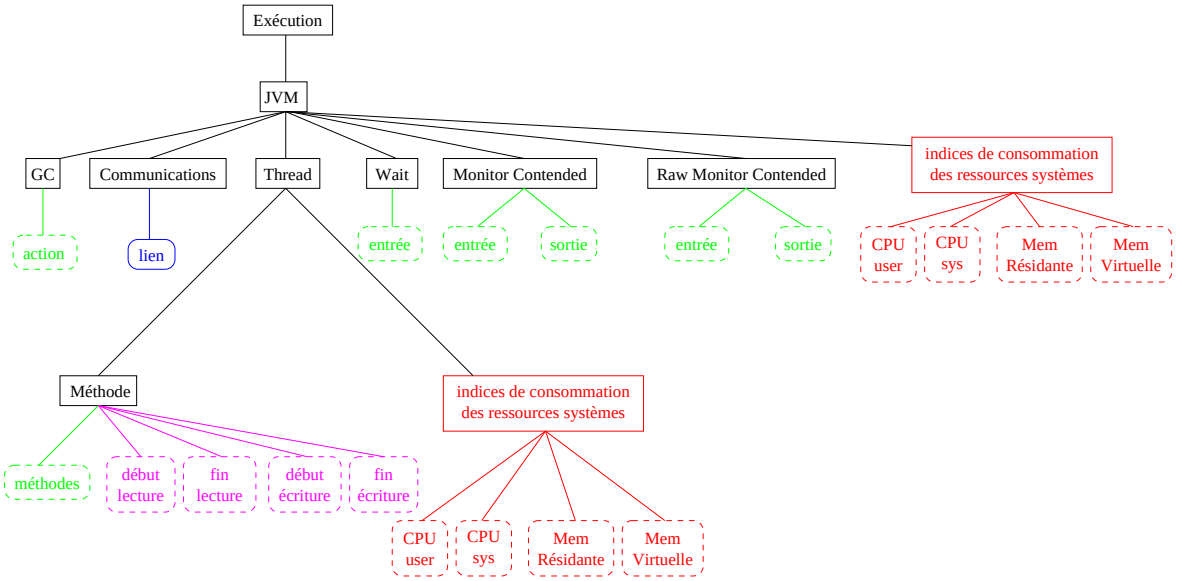


Figure 5.8: Hiérarchie de type des objets à visualiser dans "Pajé".
La hiérarchie de type des objets visuels est représentative de la sémantique donnée aux observations. Elle permet de mettre en correspondance des données provenant de différents niveaux d'observation. Il est ainsi possible de corréler les indices de performances à l'exécution de l'application : globalement pour chacune des JVM, ou "*thread*" par "*thread*".

Les conteneurs associés aux objets de synchronisation sont au même niveau que ceux des "*threads*". Les entités qui leurs sont associées sont aussi de type état. Elles représentent l'attente, l'acquisition et la libération de ces objets.

L'en-tête de la trace "Pajé" est propre à un modèle d'exécution. L'annexe C présente celle qui a été construite pour l'étude du serveur multimédia (voir section 6.5). Elle contient la hiérarchie des objets visuels présentée par la figure 5.8 ainsi que la création des conteneurs représentant l'exécution, trois JVM auxquelles on associe un conteneur pour le ramasse-miettes.

A partir de cet en-tête, la fonction de conversion que nous avons construite va interpréter les événements de la trace applicative comme suit :

L'événement de création d'un "*thread*" Java

Il va générer une trace "Pajé" de la forme :

id	date	nouveau cont.	type cont.	conteneur	commentaire
7	2.456053	TH_1_3	TH	JVM_1	"3 (1736)"

qui correspond à la création d'un conteneur "*thread*". Le premier élément de la ligne identifie l'événement "Pajé" interprété par le simulateur RefAn C. Le deuxième correspond à la date

[16] Il s'agit de l'ensemble des méthodes qui n'ont pas été filtrées.

d'occurrence de l'observation. Ces deux premiers éléments sont présents dans toutes les lignes du corps de la trace "Pajé". Ce conteneur sera identifié au sein de "Pajé" par la chaîne de caractères `TH_1_3`. Il s'agit ici de la création du quatrième "*thread*" sur la deuxième JVM, qui est associé au "*thread*" système de PID 1736.

L'événement de chargement de classe

Il ne sert qu'à identifier le nom de la classe. Par contre, chacune des méthodes qu'elle définit produira une trace "Pajé" de la forme :

id	date	nouvelle valeur	type cont.	commentaire
6	2.616716	1_1	MTH	"Book/main"

Il s'agit ici d'une trace qui représente un événement "Pajé" de type `PajeDefineEntityValue`. Il définit un nouvel état (`"Book/main"`) qui pourra être pris par une entité méthode (`MTH`). Cet état est identifié au sein de "Pajé" par la chaîne de caractères `1_1`. Il doit être unique.

Ainsi, lors du chargement d'une classe, une trace "Pajé" est générée pour chaque méthode définie, et un identifiant unique lui est associé. C'est cet identifiant qui sera utilisé lors des changements d'état d'une entité méthode.

L'événement de début de méthode

Il sera traduit comme un changement d'état de l'entité méthode. Le contexte d'exécution associé à cet événement identifie la JVM et le "*thread*" qui lui servent de support d'exécution. La trace "Pajé" est de la forme :

id	date	type cont.	conteneur	valeur
11	2.640720	MTH	TH_1_3	1_1

Celle-ci correspond au début d'exécution de la méthode identifie au sein de "Pajé" par `1_1` sur le "*thread*" identifié par `TH_1_3`. Il s'agit donc de la méthode `"Book/main"` qui commence son exécution sur le "*thread*" système `1736`.

L'événement de fin de méthode

Il sera aussi traduit comme un changement d'état de l'entité méthode. De la même façon que pour un événement de début de méthode, le contexte d'exécution associé identifie la JVM et le "*thread*" qui lui sont associés.

En mémorisant la pile d'appel de méthodes de chacun des "*threads*", il nous est possible de vérifier la cohérence des observations traitées. En effet, puisque l'exécution sur un "*thread*" est toujours séquentielle, la première observation de fin de méthode doit correspondre à la dernière observation de début de méthode. Dans le cas contraire, la conversion est arrêtée et un message

d'erreur est généré.

Si les observations sont cohérentes, la trace "Pajé" est de la forme :

id	date	type cont.	conteneur
12	2.675072	MTH	TH_1_3

ce qui correspond à la terminaison de la dernière méthode associée au "*thread*" `TH_1_3`.

Les événements de synchronisation

Ils sont traduits comme un changement d'état sur l'entité correspondante. D'une façon générale, l'observation d'une synchronisation est représentée par une entité de type état et deux entités de type lien (voir section 6.5.2.3). L'entité de type état représente l'objet sur lequel la synchronisation est réalisée. Les entités de type lien symbolisent la relation de dépendance entre l'objet de synchronisation et le "*thread*" dont l'exécution est bloquée.

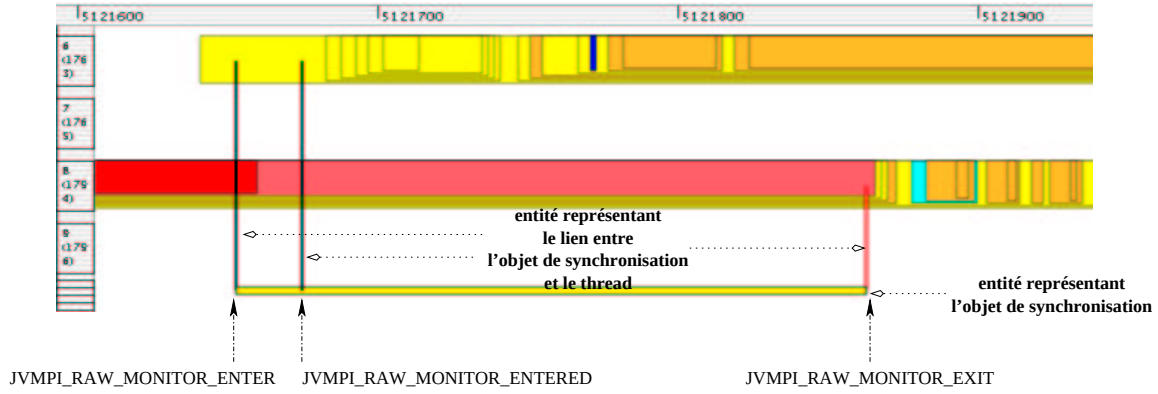


FIG. 5.9 – Représentation d'un objet de synchronisation de type `RAW_MONITOR_CONTENDED` par "Pajé".

Le "*thread*" 6 se bloque sur un objet de synchronisation de type `RAW_MONITOR_CONTENDED` qui passe dans l'état `RAW_MONITOR_CONTENDED_ENTER`. La réception de l'événement `RAW_MONITOR_CONTENDED_ENTERED` indique que le "*thread*" est débloqué. L'événement `RAW_MONITOR_CONTENDED_EXIT` indique que le "*thread*" 8 a débloqué le "*thread*" 6.

Les mécanismes de synchronisation sont représentés via l'occurrence des événements JVMPI de type `RAW_MONITOR_CONTENDED`, `MONITOR_CONTENDED` et `MONITOR_WAIT`. Pour les deux premiers types d'événements trois observations leur sont associées[17] (voir figure 5.9) :

`RAW_MONITOR_CONTENDED_ENTER`

Il est capté lorsqu'un "*thread*" cherche à entrer dans une section du programme protégée par un `raw monitor` déjà acquis par un autre "*thread*". La trace "Pajé" correspondante prend

[17] Seuls les événements rattachés à `RAW_MONITOR_CONTENDED` seront détaillés. Le cas de `MONITOR_CONTENDED` est identique modulo l'utilisation des conteneurs consacrés et le nommage.

la forme :

id	date	type cont.	conteneur		paramètres	
112	0.848866	RMC_ACT	RMC_0_2	RMC_ENTER	"enter (2)"	
16	0.848866	NOT_TM_RMC	JVM_0	NOT_TM_RMC_ENTER	TH_0_2	NOT_TM_RMC_2-2
17	0.848866	NOT_TM_RMC	JVM_0	NOT_TM_RMC_ENTER	RMC_0_2	NOT_TM_RMC_2-2

Elle correspond au changement d'état de l'entité `RMC_0_2`, qui représente le troisième `RAW_MONITOR_CONTENDED` de la première JVM, signifiant qu'un "*thread*" se place en attente sur l'objet de synchronisation. Les deux dernières lignes de la trace indiquent que c'est le "*thread*" `TH_0_2` qui se bloque. Elles correspondent à la définition d'une entité lien entre l'entité "*thread*" et l'entité `RAW_MONITOR_CONTENDED`.

`RAW_MONITOR_CONTENDED_ENTERED`

Il est capté lorsqu'un "*thread*" rentre dans une section protégée par un objet de synchronisation `RAW_MONITOR_CONTENDED`. La trace "Pajé" correspondante prend la forme :

id	date	type cont.	conteneur		paramètres	
121	0.850971	RMC_ACT	RMC_0_2	RMC_ENTER	"enter (2)"	
16	0.850971	NOT_MT_RMC	JVM_0	NOT_MT_RMC_ENTER	RMC_0_2	NOT_MT_RMC_2-2
17	0.850971	NOT_MT_RMC	JVM_0	NOT_MT_RMC_ENTER	TH_0_2	NOT_MT_RMC_2-2
112	0.850971	RMC_ACT	RMC_0_2	RMC_ENTERED	"entered (2)"	

Les trois premières lignes de la trace représentent la terminaison de l'état `RMC_ENTER` et la création d'un lien faisant le lien entre l'objet de synchronisation et le "*thread*" concerné. La dernière ligne indique que l'entité passe dans un état `ENTERED`. Cet état signifie que le "*thread*" a acquis l'objet de synchronisation, mais on ne possède pas l'information définissant qui lui a permis de l'acquérir.

`RAW_MONITOR_CONTENDED_EXIT`

Il est capté lorsqu'un "*thread*" relâche un `RAW_ MONITOR_CONTENDED` sur lequel un autre "*thread*" est bloqué. C'est l'événement qui termine l'état `ENTERED`. La trace "Pajé" correspondante prend la forme :

id	date	type cont.	conteneur		paramètres	
121	0.851057	RMC_ACT	RMC_0_2	RMC_ENTERED	"exit (2)"	
16	0.851057	NOT_MT_RMC	JVM_0	NOT_MT_RMC_EXIT	RMC_0_2	NOT_MT_RMC_2-2
17	0.851057	NOT_MT_RMC	JVM_0	NOT_MT_RMC_EXIT	TH_0_2	NOT_MT_RMC_2-2

Pour les événements de type `MONITOR_WAIT` seules deux observations sont à interpréter :

`MONITOR_WAIT`

Il est capté lorsqu'un "*thread*" se met en attente sur un objet de synchronisation. La trace "Pajé" est du même type que celle de `RAW_MONITOR_CONTENDED_ENTER`, c'est-à-dire un changement d'état de l'entité correspondante et un lien entre le "*thread*" qui se bloque et cette entité.

`MONITOR_WAITED`
Il est capté lorsqu'un "*thread*" est débloqué. La trace "Pajé" est du même type que celle de `RAW_MONITOR_CONTENDED_EXIT`.

Après avoir décrit le traitement associé aux observations du niveau applicatif nous allons à présent nous intéresser à celles du niveau système.

5.2.2.2 Construction à partir de la trace système

Les enregistrements système sont de deux types. Le premier correspond aux observations liées aux communications : écritures et lectures sur les "*sockets*". Elles symbolisent l'accès aux données distantes. Le deuxième se rapporte aux informations de consommation de ressources d'exécution (utilisation de la ressource de calcul, accès aux données locales).

Écritures et lectures sur les "*sockets*"
Dans le cadre des écritures et lectures sur les "*sockets*", la traduction dans le langage de commande "Pajé" est réalisée à l'aide d'entités de type lien et événement. L'entité de type lien représente les parties prenantes à l'interaction. Les entités de type événement identifient les dates auxquelles les appels système ont été déroutés (voir figure 5.10).

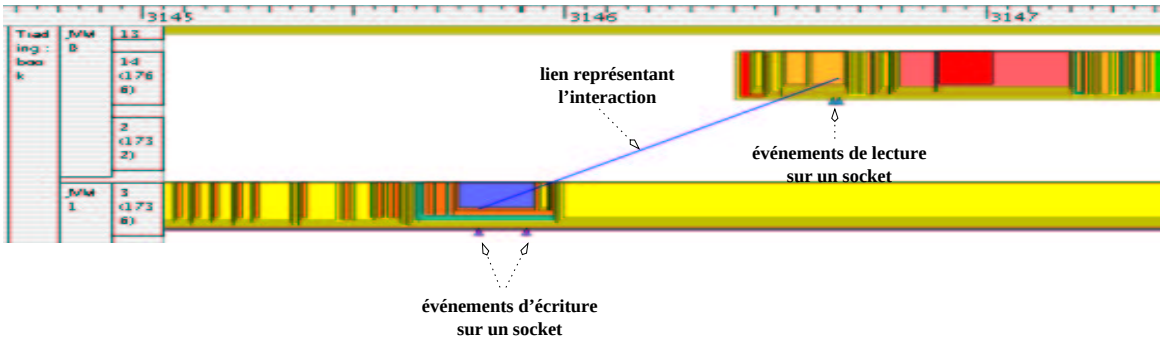


Figure 5.10: Représentation d'une communication par "Pajé".

Si l'on se place à un niveau applicatif, à chaque envoi de message correspond une ou plusieurs écritures "*socket*" et à chaque réception correspond une ou plusieurs lectures "*socket*". Si l'émission génère plusieurs écritures, à chacune est associée une entité de type événement. Si la réception demande plusieurs lectures, à chacune est aussi associée une entité de type événement. Le lien, quant à lui, est défini entre l'événement correspondant à la première écriture et à la dernière lecture.

C'est le cas, par exemple, de la mise en œuvre d'un modèle de programmation où un "*thread*" est placé en attente sur un port de communication. A chaque réception de message est associée au moins deux lectures. La première correspond à la détection de l'arrivée de message. Les suivantes représentent la réception du message (voir figure 5.11).

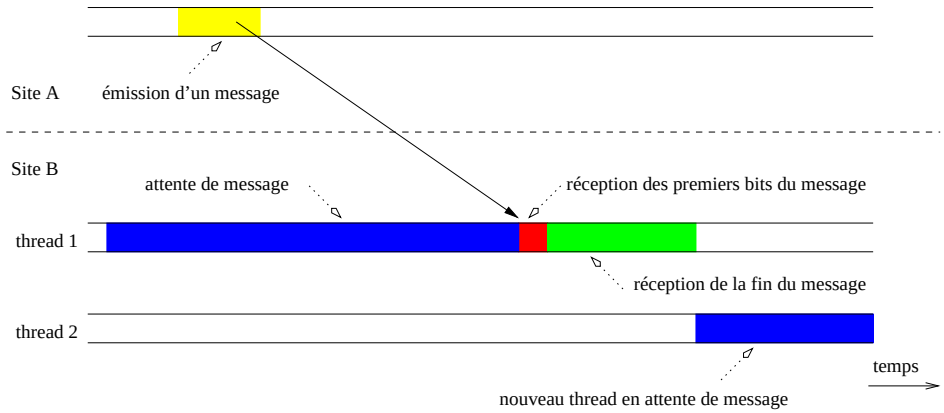


Figure 5.11: Modèle de "*pool*" de "*threads*" en attente de réception de message.
L'utilisation des techniques de multi-programmation pour le réception de message, se traduit par la création d'un "*pool*" de "*threads*" ayant en charge le traitement des communications entrantes. Un "*thread*" est mis en attente sur un port donné. Lorsqu'un message arrive, la lecture des premiers bits déclenche le mécanisme de traitement d'un nouveau message. Le "*thread*" qui a lu les premiers bits finit la réception du message, et un autre "*thread*" du "*pool*" se met en attente d'une nouvelle réception.

Ainsi, la construction de la trace "Pajé" correspondant à l'envoi d'un message sur une "*socket*" est de la forme :

id	date	type cont.	conteneur	paramètres		ports	taille	durée
161	3.012016	COMM	CUR	COMM_0	TH_1_3	9600-53511	100	
91	3.012016	SOCKET	TH_1_3	SOCKET_WRITE		9600-53511	100	0.000090
91	3.012106	SOCKET	TH_1_3	SOCKET_WRITE		9600-53511	100	0.000090

La première ligne identifie le début d'un lien, dont l'origine est le "*thread*" `TH_1_3`. Le début de l'écriture sur la "*socket*" est représenté par la seconde ligne, et la fin par la troisième. Ces deux événements symbolisent l'envoi d'un message d'une taille de 100 octets en 90 μs.

La construction de la trace "Pajé" correspondant à la réception d'un message sur une "*socket*" est de la forme :

id	date	type cont.	conteneur	paramètres		ports	taille	durée
91	2.973146	SOCKET	TH_0_12	SOCKET_READ		9600-53511	1	0.039018
91	3.012164	SOCKET	TH_0_12	SOCKET_READ		9600-53511	1	0.039018
91	3.012450	SOCKET	TH_0_12	SOCKET_READ		9600-53511	99	0.000012
91	3.012462	SOCKET	TH_0_12	SOCKET_READ		9600-53511	99	0.000012
171	3.012462	COMM	CUR	COMM_0	TH_0_12	9600-53511	100	

Les deux premières lignes représentent l'attente de début de message. Un seul octet est lu. Il déclenche le mécanisme de réception d'un message (voir section 6.5.3) qui correspond aux deux lignes suivantes. Enfin la dernière ligne matérialise la fin du lien sur le "*thread*" `TH_0_12`.

Consommations des ressources d'exécution

Elles sont représentées par des entités de type valeur. "Pajé" leur associe trois opérations : une première pour leur donner une valeur `PajeSetVariable`, une seconde pour leur ajouter une valeur `PajeAddVariable`, et une troisième pour leur soustraire une valeur `PajeSubVariable` .

Dans notre cas, nous n'utilisons que la première fonction étant donné que les informations à convertir portent sur des quantités et non des variations. La trace "Pajé" est de la forme :

id	date	type de l'entité	conteneur	valeur
13	0.791913	CPU_S	TH_0_9	3
13	0.791913	CPU_U	TH_0_9	7
13	0.791913	RSS	TH_0_9	438
13	0.791913	TOTAL_VM	TH_0_9	20196
13	0.791913	JVM_CPU_S	JVM_0	3
13	0.791913	JVM_CPU_U	JVM_0	7
13	0.791913	JVM_RSS	JVM_0	438
13	0.791913	JVM_TOTAL_VM	JVM_0	20196

Les quatre premières lignes correspondent aux consommations, par les "*thread*" `TH_0_9`, de la ressource processeur en mode système (`CPU_S`) et en mode utilisateur (`CPU_U`), ainsi que la ressource mémoire vive (`RSS`) et mémoire virtuelle (`TOTAL_VM`). Les quatre dernières représentent la consommation de ces mêmes ressources mais pour l'ensemble des "*threads*" présents dans la JVM `JVM_0`.

5.3 Conclusion

Nous avons développé une infrastructure d'observation multi-niveaux pour étudier des applications à objets réparties. Cette infrastructure a été mise en œuvre dans le cadre du projet Jonathan pour l'observation d'applications Java réparties.

Nous avons montré que les informations du niveau applicatif, obtenues au travers de la JVMPI, permettent de reconstruire la dynamique de l'exécution d'une JVM. Nous avons aussi montré que l'observation des écritures et des lectures effectuées par les "*threads*" des JVM faisant partie de l'application répartie permettent de reconstruire les interactions distantes. Nous avons ainsi pu reconstruire une représentation de la dynamique d'exécution d'une application à objets répartie.

De plus, grâce à l'observation des appels système liés aux "*sockets*" nous avons pu collecter des informations de consommation de ressources système nécessaires à l'exécution. Ces informations sont mises en correspondance avec les traces applicatives et systèmes afin de fournir une représentation multimodales de l'exécution.

Le chapitre suivant présente la validation de notre infrastructure par l'observation d'un serveur multimédia.

Chapter 6

Validation de l'approche

Ce chapitre a pour objet de valider la mise en œuvre de l'infrastructure d'observation et d'analyse d'application à objets répartis présentée au Chapitre 5. Pour ce faire, elle a été testée afin d'étudier une application représentant un serveur multimédia distribué donnant accès à des livres et des morceaux de musique. Cette application est intéressante car elle est représentative de deux types d'interactions :

asynchrone lorsque l'on cherche à accéder à un livre.

synchrone lorsque l'on souhaite écouter un morceau de musique.

Cette étude doit permettre de prouver «l'utilisabilité» de l'infrastructure en termes d'exécution et d'analyse. En effet, il nous faut prouver que l'infrastructure d'observation ne perturbe pas de façon excessive l'exécution, et ainsi que les données collectées sont significatives du comportement observé. De plus, il nous faut montrer l'intérêt et le potentiel de nos observations. Nous souhaitons analyser le comportement dynamique global de l'application et ainsi faire apparaître les liens entre les différentes exécutions distantes. Cette analyse doit nous permettre d'identifier les interdépendances et les problèmes de performances tels que les goulots d'étranglement ou les interblocages.

Nous avons développé une analyse complète sur un exemple d'exécution représentant l'obtention du résumé d'un livre présent sur le serveur. Notre analyse porte sur la représentation sous la forme d'un diagramme espace-temps par "Pajé" de l'exécution. Ce diagramme fait apparaître les méthodes employées ainsi que leur répartition sur les "*thread*" Java mis à contribution. De plus, afin de faire ressortir les consommations des différents types de ressources d'exécution, l'utilisation du processeur, des objets de synchronisation, de la mémoire et des communications inter JVM sont représentés dans ce diagramme.

Bien que notre analyse porte sur la totalité de l'exécution, elle concerne plus particulièrement trois phases significatives. De ces trois phases sont identifiés et extraits des schémas d'exécution types.

6.1 Exemple d'implantation

Les résultats correspondent à des traces réalisées sur des stations Linux (noyau 2.4), sur lesquelles est installée la JDK 1.2.2 RC4 de Sun. Trois machines ont été utilisées :

la machine A (g-anache) : un ordinateur portable DELL INSPIRON 7500, dont le processeur est un Intel Céléron 466 Mhz avec 192 Mo de mémoire, un disque dur IDE UDMA 33 Mhz de 10 Go et une carte réseau PCMCIA Xircom CEM56-100 à 100 Mbit/s.

la machine B (g-pcstag5) : une station de travail GATEWAY GP7-500, dont le processeur est un Intel Pentium III 500 Mhz avec 128 Mo de mémoire, un disque dur IDE UDMA 33 Mhz de 20Go et une carte réseau 3Com PCI 3c905C Tornado à 100 Mbit/s.

la machine C (g-eant) : une station de travail DELL Precision 610, dont le processeur est un Intel Pentium II 455 Mhz avec 256 Mo de mémoire, un disque dur SCSI Ultra2 Wide 80 Mo/s et une carte réseau 3Com PCI 3c905B Cyclone 100baseTx à 100 Mbit/s.

Les sites A, B et C font références aux machines A, B et C présentées ci-dessus.

Les observations du niveau des JVM sont réalisées à l'aide de la librairie C `libetp.so`, dont la taille est de 26424 octets ($\simeq 26$ ko). Elle correspond à un code source de 2400 lignes.

Les observations du niveau noyau sont réalisées à l'aide des modules

`syscall_socket.o`
Il réalise le déroutage des appels système ainsi que la trace d'observation. La taille du module est de 9196 octets qui correspondent à 863 lignes de code.

`unsyscall_socket.o`
Il permet de retrouver les appels système d'origine. La taille du module est de 2568 octets qui correspondent à 91 lignes de code.

`obsv_pid.o`
Il réalise le déroutage de l'appel système `obsv_pid` rajouté au noyau pour réaliser le lien entre le processus à observer et le module dans le noyau. La taille du module est de 1532 octets qui correspondent à 60 lignes de code.

`obsv_flush.o`
Il réalise le déroutage de l'appel système `obsv_flush` rajouté au noyau pour vider le tampon de trace du module `syscall_socket.o`. La taille du module est de 1400 octets qui correspondent à 60 lignes de code.

L'empreinte totale de l'environnement d'observation s'élève donc à environ 14 ko.

6.2 Utilisation

La construction du référentiel de temps commun demande deux phases de communication avant le début de l'exécution. La première a pour objectif de réaliser une synchronisation «forte» des systèmes de datations présents sur les différents sites (voir section 4.1.4). La seconde correspond à la création de l'échantillon qui sert à calculer les dérives et les biais d'horloge par rapport à un site de référence (voir section 4.1.2).

Indépendamment de la construction du référentiel de temps commun, les modules d'observation du noyau doivent être insérés avant le commencement de l'exécution. Les traces système sont vidées dans un fichier binaire sur le disque dur local dans trois cas : a) la taille maximale du tampon est atteinte; b) une action de «ramasse-miettes» est réalisée; c) les modules sont "dé-insérés" du noyau.

Une fois les modules insérés et les phases de communication achevées, les JVM sont lancées avec en paramètre la librairie d'observation et ses propres paramètres (ex.: `java -Xrunetp:monit=y Music`). La librairie possède trois paramètres : `filter_file=file_name` pour indiquer un fichier textuel contenant le préfixe des classes des méthodes à exclure (voir section 5.1.1) ; `monit=y|n` pour générer une trace d'utilisation des objets de synchronisation ; et `perturb=y|n` afin d'enregistrer les perturbations liées à l'observation (voir section 4.2). Les traces applicatives sont vidées dans un fichier binaires sur le disque dur local dans trois cas : a) la taille maximale du tampon est atteinte ; b) une action de «ramasse-miettes» est réalisée ; c) la JVM termine son exécution.

Une fois l'exécution terminée, les modules sont "dé-insérés" et les différentes traces sont rassemblées pour traitement. Il s'agit ici de réaliser les opérations de prétraitement nécessaires avant toute analyse (voir section 4).

6.3 Coût de l'observation

Avant toute projection des observations dans un référentiel de sens commun, il nous faut déterminer la représentativité des observations. L'estimation des perturbations directes (voir section 4.2) nous permet de juger de l'écart entre une exécution classique et observée.

6.3.1 Coût de la datation

Les observations réalisées au niveau utilisateur sont datées à l'aide de la fonction :

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

Cette fonction renvoie la date tenue à jour par le noyau Linux par le biais de la structure `struct timeval *tv`. Elle est constituée d'une partie seconde (`long tv_sec`), et d'une partie microseconde (`long tv_usec`) [1].

L'algorithme utilisé par le noyau pour tenir à jour cette date diffère selon l'architecture du processeur utilisé. Lorsque le processeur possède un registre de type "*Time Stamp Counter*", la date est mise à jour à partir de la valeur de ce registre [2]. Elle correspond au nombre de cycles effectués par le processeur. Dans ce cas, le système de datation matérielle de la machine repose sur le processeur et sa précision est directement liée à sa cadence. La cadence du processeur est mesurée à l'initialisation du noyau dans la fonction :

```
static unsigned long __init calibrate_tsc(void)
```

Pour ce faire, une alarme est armée pour une durée de $1\mu s$. Puis une boucle infinie est démarrée. Sa seule fonction est d'incrémenter un compteur. A la fin de la microseconde, la boucle est cassée. La valeur du compteur indique la fréquence du processeur mesurée par le noyau[3].

Dans le cas où ce registre n'existe pas, la date est mise à jour à partir d'un circuit spécialisé qui procure lui aussi un nombre de cycles écoulés.

Dans notre cas, nous utilisons des machines qui possèdent le registre "*Time Stamp Counter*". Nous avons donc accès à la version «rapide» de `gettimeofday()`. Le coût moyen d'utilisation de cette fonction est de $\simeq 1\mu s$ (voir figure 6.1). Ce coût est principalement dû aux changements de contexte liés au passage du mode d'exécution utilisateur au mode noyau.

La fonction noyau sous-jacente est :

```
void do_gettimeofday(struct timeval *tv)
```

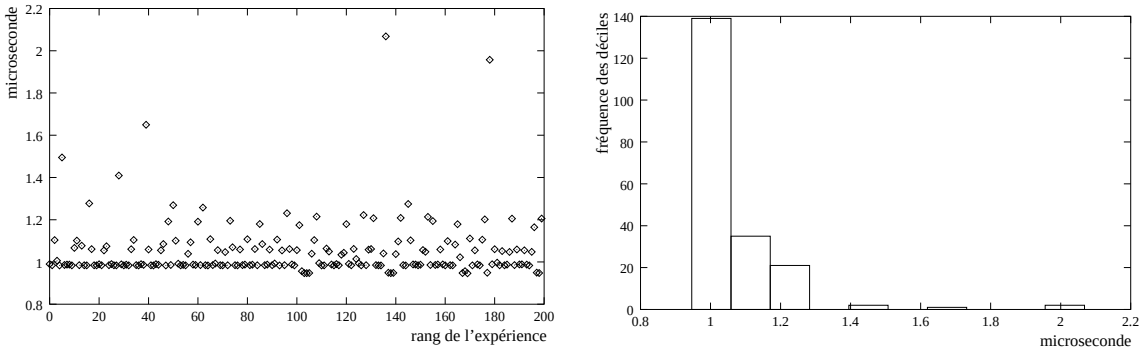
Sa durée moyenne d'exécution est de $\simeq 0,2\mu s$ (voir figure 6.2). C'est la fonction qui est utilisée pour dater les observations du niveau système.

Dans les deux cas, la date est enregistrée dans une structure `struct timeval *tv` dont la taille est de 8 octets.

[1] La date 0.0 correspond au 01.01.1970

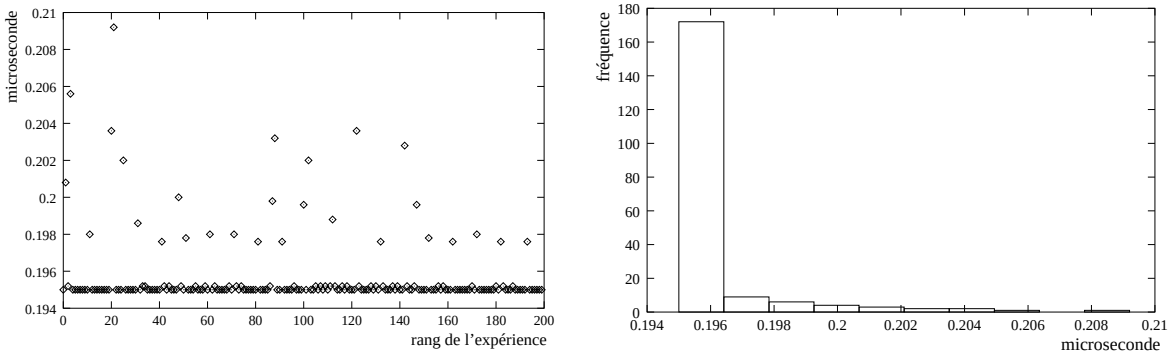
[2] La valeur du registre est placée dans une variable de 32 bits de type `unsigned long`

[3] Il s'agit en fait d'un estimateur de fréquence réelle



Taille de l'échantillon	=	200	IC 90%	:	] 1.033 .. 1.065 [
Min	=	0.946	IC 95%	:	] 1.030 .. 1.068 [
Max	=	2.068	IC 99%	:	] 1.024 .. 1.074 [
Moyenne	=	1.049			
Ecart-type	=	0.137			

FIG. 6.1 – Durée d'exécution de la fonction utilisateur `gettimeofday()`.
Les observations réalisées au niveau utilisateur sont datées à l'aide de la fonction `gettimeofday()`. La granularité des dates obtenues est la microseconde.



Taille de l'échantillon	=	200	IC 90%	:	] 0.195 .. 0.195 [
Min	=	0.195	IC 95%	:	] 0.195 .. 0.196 [
Max	=	0.209	IC 99%	:	] 0.195 .. 0.196 [
Moyenne	=	0.195			
Ecart-type	=	0.002			

FIG. 6.2 – Durée d'exécution de la fonction noyau `do_gettimeofday()`.
Les observations réalisées au niveau du noyau sont datées à l'aide de la fonction `do_gettimeofday()`. La granularité des dates obtenues est la microseconde.

6.3.2 Coût de l'observation du niveau applicatif

La fonction `gettimeofday()` est donc utilisée lors de la datation des événements du niveau utilisateur. Elle induit une perturbation directe de $\simeq 1\mu s$ par événement enregistré.

La majeure partie des observations du niveau applicatif se rapporte à l'exécution des méthodes.

Pour ce qui est de la trace qui nous sert d'exemple, 137232 événements de début et de fin de méthodes ont été traités par l'agent d'observation, pour un total de 138123 événements observés soit plus de 99% des observations (voir tableau 6.1). Toutefois, seuls les événements non filtrés sont à l'origine d'une trace (voir section 5.1.1). La fonction de filtrage correspond à la recherche de l'identifiant JVMPI de la méthode dans une table de hachage construite lors des chargements de classe. Le coût de cette recherche est en moyenne de $2\mu s$. Le coût moyen d'observation d'un événement de début ou fin de méthode est de $3\mu s$ (voir tableau 6.1). Les valeurs maximales correspondent à des situations où le "*thread*" est préempté.

Nous avons observé 138123 événements sur une durée d'exécution totale de $\simeq 8s$. La durée totale d'observation des événements est de $0.460756s$, soit un surcoût de $3\mu s$ par événement pris en compte. Dans ce cas, la perturbation directe induite par l'observation s'élève à $\simeq 6\%$ de la durée totale d'exécution. Il apparaît donc que la trace obtenue peut tout à fait être considérée comme représentative de l'exécution de l'application.

Du point de vue de l'empreinte mémoire, la taille des éléments de la trace est de 60 octets par observation. Cette taille diffère lors du chargement d'une classe où la trace est de 324 octet plus 64 octets par méthode définie. Dans le cas de l'exemple présenté dans les tableaux 6.1 et 6.2, la taille de la trace des événements est de 3584040 octets ($\simeq 3.5$ Mo) et celle des classes de 96748 octets ($\simeq 100$ ko).

EVT	NBR	MIN (μs)	MAX (μs)	DUREE (μs)	MOYENNE (μs)
METHOD_ENTRY2	68637	2	1470	186061	3
METHOD_EXIT	68595	2	1100	176280	3
THREAD_START	11	4	87	144	13
THREAD_END	2	6	7	13	6
CLASS_LOAD	334	3	290	8885	27
JVM_INIT_DONE	1	7	7	7	7
JVM_SHUT_DOWN	1	11967	11967	11967	11967
RAW_MONITOR_CONTENDED_ENTER	1	5	5	5	5
RAW_MONITOR_CONTENDED_ENTERED	1	3	3	3	3
RAW_MONITOR_CONTENDED_EXIT	1	5	5	5	5
MONITOR_CONTENDED_ENTER	35	2	28	137	4
MONITOR_CONTENDED_ENTERED	35	2	27	124	4
MONITOR_CONTENDED_EXIT	35	2	5	109	3
MONITOR_WAIT	217	2	22	695	3
MONITOR_WAITED	213	2	27	677	3
MONITOR_DUMP	1	5	5	5	5
dump	3	11920	32472	75639	25213

TAB. 6.1 – Surcoût induit par l'observation au travers de la JVMPI
Le surcoût est mesuré en calculant la durée d'exécution de la fonction `void notifyEvent(JVMPI_Event *event)` de la librairie d'observation (voir section 5.1.1).

Début de méthode en μs

Échantillon complet			Les 9 premiers déciles		
Taille de l'échantillon	=	33740	Taille de l'échantillon	=	30366
Moyenne	=	2.591	Moyenne	=	2.238
Min	=	2.00	Min	=	2.00
Max	=	1900.00	Max	=	3.00
Ecart-type	=	15.449	Ecart-type	=	0.426
IC 90%	:	] 2.452 .. 2.729 [	IC 90%	:	] 2.234 .. 2.242 [

Fin de méthode en μs

Échantillon complet			Les 9 premiers déciles		
Taille de l'échantillon	=	33700	Taille de l'échantillon	=	30330
Moyenne	=	2.366	Moyenne	=	2.153
Min	=	2.00	Min	=	2.00
Max	=	1131.00	Max	=	3.00
Ecart-type	=	8.502	Ecart-type	=	0.360
IC 90%	:	] 2.290 .. 2.442 [	IC 90%	:	] 2.150 .. 2.156 [
IC 95%	:	] 2.275 .. 2.457 [	IC 95%	:	] 2.149 .. 2.157 [
IC 99%	:	] 2.247 .. 2.485 [	IC 99%	:	] 2.148 .. 2.158 [

Chargement de classe en μs

Échantillon complet			Les 9 premiers déciles		
Taille de l'échantillon	=	328	Taille de l'échantillon	=	295
Moyenne	=	28.408	Moyenne	=	16.135
Min	=	2.00	Min	=	2.00
Max	=	553.00	Max	=	76.00
Ecart-type	=	51.945	Ecart-type	=	19.903
IC 90%	:	] 23.690 .. 33.126 [	IC 90%	:	] 14.229 .. 18.041 [

TAB. 6.2 – Analyse statistique du surcoût lié aux événements de début de méthode, fin de méthode et chargement de classe.

6.3.3 Coût de l'observation du niveau système

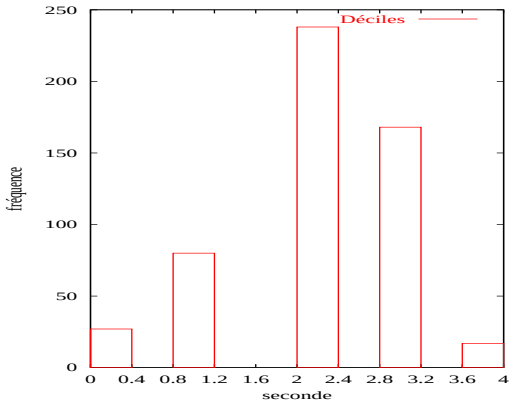
Nous allons à présent analyser le surcoût de déroutage des appels système liés aux écritures et lectures sur les "*sockets*" (voir figure 6.3). Il est estimé en mesurant le temps d'exécution du code de déroutage des appels système, ainsi que le temps d'enregistrement des informations de consommations des ressources système.

Le surcoût moyen lié aux 530 écritures est de $2.13\mu s$. Celui des 837 lectures est en moyenne de $1.55\mu s$ (voir figure 6.3). On peut noter une forte variabilité de l'échantillon des écritures, puisque l'écart-type calculé est du même ordre que la moyenne.

La durée totale des écritures est de $34816 * \mu s$, et celle de lectures est de $24316567\mu s$. La perturbation totale liée aux écritures est de $1128\mu s$, et celle aux lectures à $1303\mu s$. La perturbation directe s'élève donc à 3% pour les écritures et 0.005% pour les lectures.

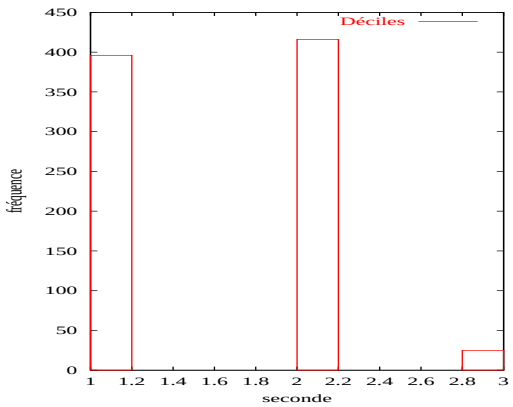
Dans ce cas encore, il apparaît donc que la trace obtenue peut tout à fait être considérée comme représentative de l'exécution observée.

Répartition en fréquences des écritures sur une "socket"



Taille de l'échantillon	=	530
Moyenne	=	2.128301
Min	=	0.00000
Max	=	4.00000
Ecart-type	=	0.886013
IC 90%	:	] 2.064992 .. 2.191611 [

Répartition en fréquences des lectures sur une "socket"



Taille de l'échantillon	=	837
Moyenne	=	1.556750
Min	=	1.00000
Max	=	3.00000
Ecart-type	=	0.553970
IC 90%	:	] 1.525251 .. 1.588248 [

FIG. 6.3 – Surcoût de déroutage des appels système d'écriture et de lecture sur une "*socket*".

Chaque observation génère deux traces d'une taille de 36 octets. La taille totale de trace est donc de $\simeq 44ko$. La taille d'une trace de consommation de ressources système est de 32 octets.La taille totale de trace est donc de $\simeq 40ko$.

Un résumé du surcoût de l'observation du niveau applicatif et système est présenté dans le tableau 6.3.

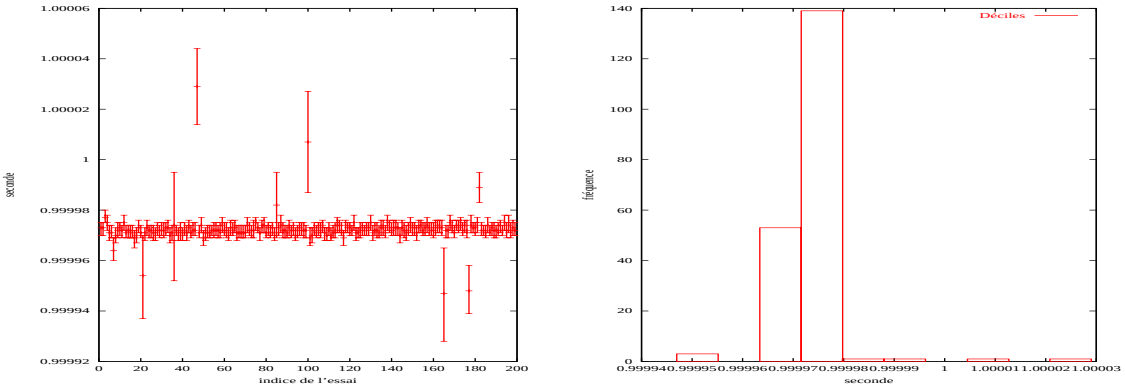
	durée moyenne (μs)	taille unitaire (octet)
`gettimeofday(...)`	1.049	8
`do_gettimeofday(...)`	0.195	8
événement JVMPI	3	60
écriture sur une "*socket*"	2.13	$36 + 32$
lecture sur une "*socket*"	1.55	$36 + 32$

TAB. 6.3 – Résumé du surcoût lié à l'observation.

6.4 Construction d'un référentiel de temps commun

Nous avons réalisé une série de 200 phases de communication pour le calcul des paramètres de correction d'horloge (voir section 4.1) afin de qualifier la qualité des estimateurs. Les communications ont lieu entre deux couples de sites : (A,B) et (A, C).

Chaque phase comporte 10000 communication de type «PingPong» (voir section 4.1.2). Pour chacun des échantillons, nous avons calculé la dérive et le décalage à l'origine de l'horloge du site B et C par rapport à l'horloge du site A. Les résultats obtenus pour le couple (A,B) sont présentés dans le suite de ce chapitre (voir figure 6.4, 6.6). Les résultats du couple (A,C) sont présentés dans l'annexe A.



Taille de l'échantillon	=	200
Moyenne	=	$1 - 28.10^{-6}$
Min	=	$1 - 53.10^{-6}$
Max	=	$1 + 29.10^{-6}$
Ecart-type	=	27.10^{-6}
IC 90%	:	] 0.999969 .. 0.999975 [

FIG. 6.4 – Paramètres de dérive de l'horloge entre les sites A et B.

L'analyse statistique du paramètre de dérive d'horloges montre une bonne stabilitée de la qualité de l'estimateur. Dans le cas du couple de sites (A, B), la longueur de l'intervalle de confiance à 90%, attaché aux 200 expériences, est de 6.10^{-6}. Nous pouvons donc conclure que la valeur moyenne calculée est représentative de la valeur théorique estimée. De la même façon, pour le couple de site (A, C), la longueur de l'intervalle de confiance à 90% est de 3.10^{-6}. Là encore, nous pouvons donc conclure que la valeur moyenne calculée est représentative de la valeur théorique estimée.

Pour ce qui est de l'analyse du paramètre de décalage à l'origine, la qualité de l'estimateur est fortement dépendante de la distance à la date d'origine. En effet, bien que l'erreur sur le calcul du paramètre de dérive d'horloge soit faible, il joue en facteur multiplicatif sur le calcul du décalage à l'origine (voir figure 6.5). Ce qui explique la forme en «cône» des résultats présentés par la figure 6.6.

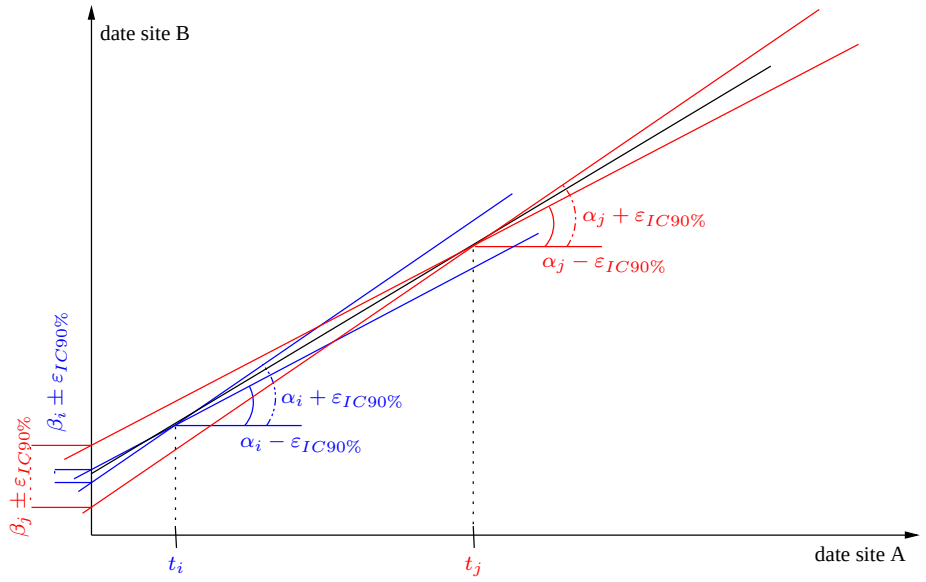
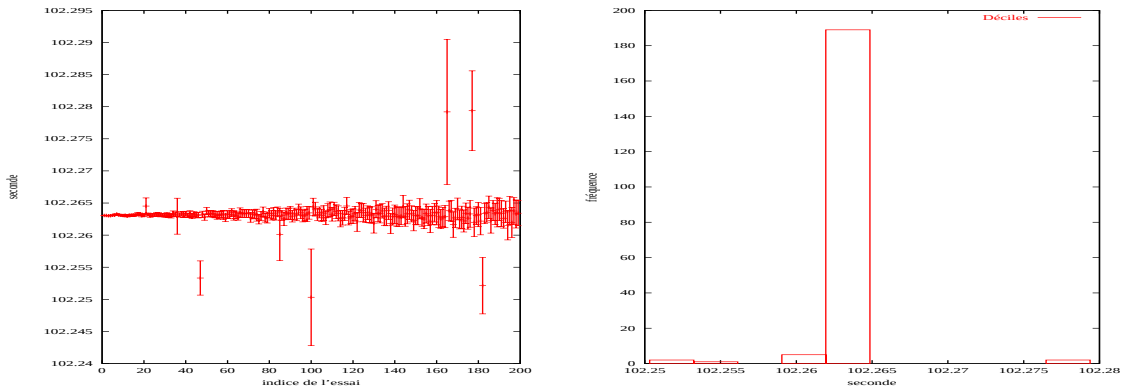


FIG. 6.5 – Relation entre l'intervalle de confiance à 90% de α et β.



Échantillon complet

Taille de l'échantillon	=	200
Moyenne	=	102.26315
Min	=	102.25032
Max	=	102.27938
Ecart-type	=	0.003838
IC 90%	:	] 102.26270 .. 102.26360 [

FIG. 6.6 – Paramètres de biais de l'horloge de la machine g-pcstag5.

La qualité de l'estimation du décalage à l'origine peut être problématique au regard de la granularité des communications. La figure 6.7 représente la longueur des intervalles de confiance à 90% associés aux paramètres de décalage d'horloge pour le couple (A,B). En première analyse, il est possible de modéliser la relation entre la distance à la date d'origine et la longueur de l'intervalle de confiance par l'équation linéaire :

$$y = 59.10^{-7}x + 898.10^{-7}.$$

Ainsi, à partir de $\simeq 2s$ la longueur de l'intervalle de confiance est de $0.0001s$ soit la même granularité que la durée des communications. Au bout de $\simeq 27s$ elle dépasse les $0.000256s$ et devient supérieure à la durée moyenne de communication calculée lors de la constitution des échantillons.

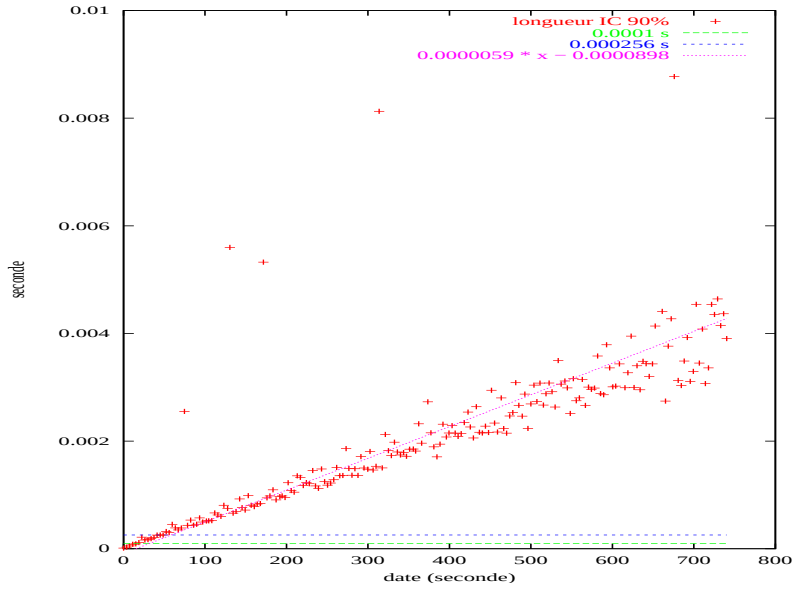
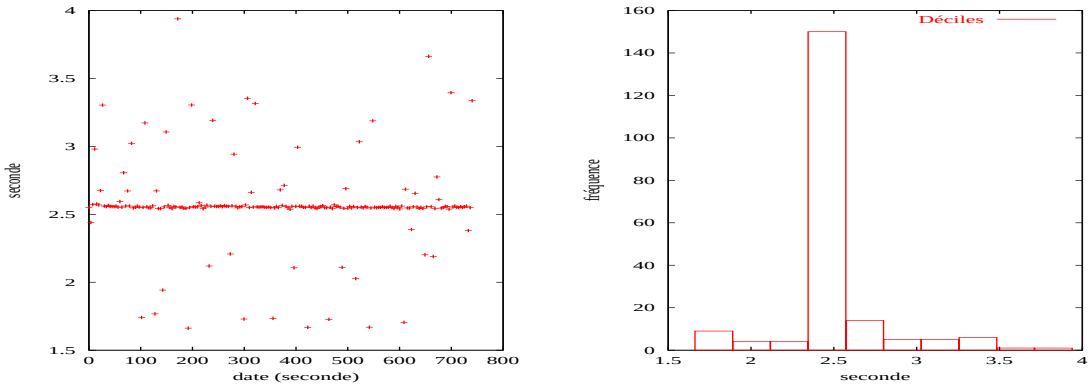


FIG. 6.7 – Longueur de l'intervalle de confiance à 90% du décalage à l'origine pour la correction d'horloge entre les sites A et B.

Nous avons réalisé une analyse statistique sur les durées de communications liées aux échantillons. Aucun phénomène temporel flagrant n'apparait (voir figure 6.8). Il serait donc souhaitable d'effectuer une analyse de Fourier afin de déterminer s'il existe des périodes qui pourraient expliquer cette rapide dégradation.

La méthode de correction mise en œuvre ne permet donc pas d'estimer de façon fiable les durées des communications inter-sites. Elle nous permet toutefois de construire une trace cohérente du point de vue de l'ordre causal. La trace ainsi corrigée est malgré tout susceptible de faire ressortir des dissymétries, ou des durées de communications «anormales», dès lors qu'elle dépasse la granularité de l'erreur sur le décalage à l'origine.

La rapide dégradation de la qualité de l'estimateur du décalage à l'origine ne permet donc pas de corriger des traces sur de longues durées. Toutefois, au regard de la durée d'exécution des exemples présentés, elle reste suffisamment fiable dès lors que l'on synchronise les machines en début d'expérience comme décrit dans le Section 4.1.4.



Taille de l'échantillon	=	200
Moyenne	=	2.563823
Min	=	1.66204
Max	=	3.93855
Ecart-type	=	0.298259
IC 90%	:	] 2.529042 .. 2.598603 [

FIG. 6.8 – Durée de communication mesurée lors de la constitution des 200 échantillons.

6.5 Analyse et visualisation du serveur multimédia

Afin de démontrer la validité de notre approche, nous avons choisi de la tester sur une application répartie représentant un serveur multimédia. L'observation a porté sur le serveur, mais aussi sur les clients qui y accèdent.

Une analyse complète est réalisée sur un exemple représentant l'accès à un résumé de livre. Il permet de faire apparaître trois phases dans l'exécution : I) la mise en place du serveur, II) l'initialisation du client, III) la réalisation d'une requête.

Le même type d'analyse a été réalisée dans le cas où un client souhaite obtenir à un morceau de musique. Il s'agit d'une situation où l'on crée un flux de données, par opposition à l'accès à un livre où les requêtes sont de type asynchrones. Un exemple est présenté dans l'annexe B.

6.5.1 Description de l'application observée

L'application observée représente un serveur multimédia de livres et de musique au format MP3. L'architecture générale de l'application est constituée d'un «courtier» (“*trader*”), d'un serveur de livre, d'un serveur de musique et de clients (voir figure 6.9).

Le courtier a pour fonction de référencer les ressources ou services disponibles, ainsi que de les rendre accessibles via des requêtes portant sur des propriétés [**Tra, 1997**]. Pour cela, chaque serveur de livre, ou de musique, enregistre les livres, ou les morceaux de musique, ainsi que leurs propriétés.

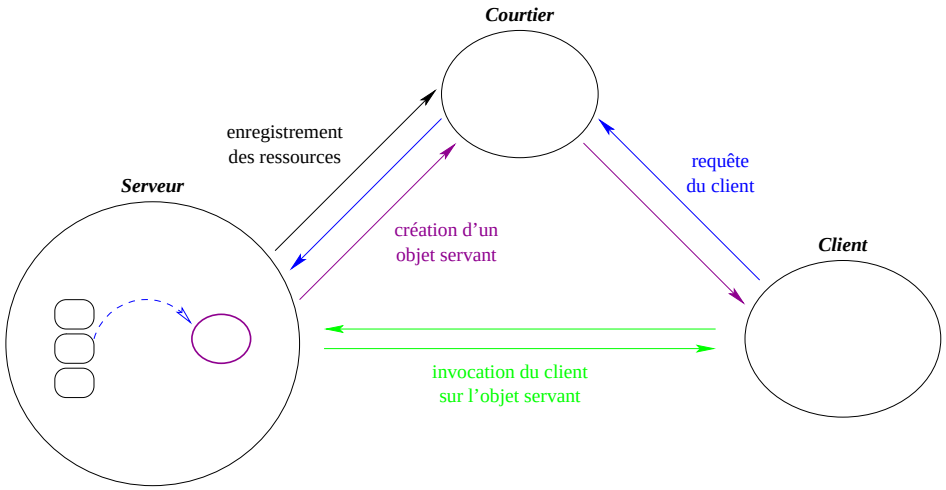


FIG. 6.9 – Architecture générale de l'application.

Prenons l'exemple où un client souhaite consulter un livre. Il réalise une requête auprès du courtier à partir des propriétés que doit remplir le livre. Le courtier lui renvoie une référence sur un serveur de livre pouvant répondre à sa demande. Une fois cette référence acquise, il pourra effectuer les opérations permises : obtention de la couverture, du résumé ou du contenu du livre. Le mécanisme est similaire lorsqu'il s'agit d'un morceau de musique.

L'expérimentation repose sur l'infrastructure de communication Jonathan. Elle est composée d'au moins trois JVM : une pour le courtier, une pour un serveur, et une par le client. Les JVM sont indifféremment réparties sur un nombre de sites variant de un au nombre total de JVM. Elles s'exécutent avec l'option «"*threads*" natifs», ce qui signifie que chaque "*thread*" Java s'exécute sur un "*thread*" système (modèle d'exécution **1/1**, voir section 2.3.1 et figure 6.10).

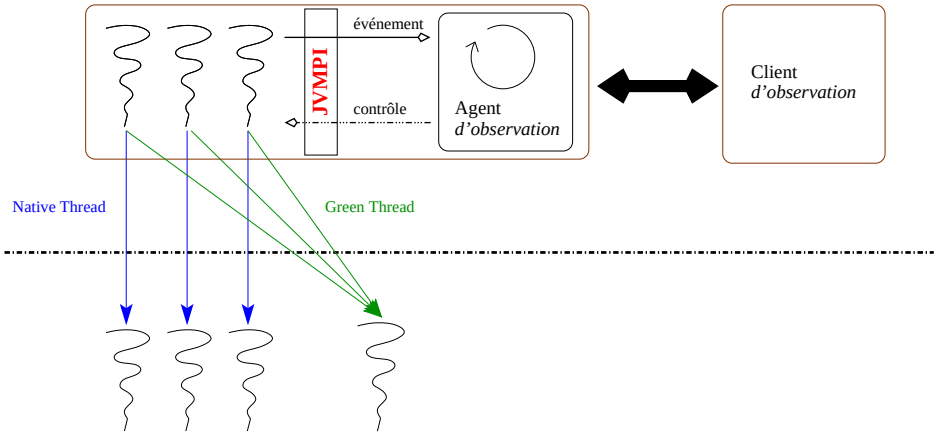


FIG. 6.10 – Les JVM créent des "*threads*" Java de type «natif» ("*Native Thread*").

Les observations réalisées au niveau applicatif doivent nous permettre d'identifier l'action et le comportement des différentes couches logicielles mises à participation :

La couche applicative

Elle représente l'exécution d'un serveur, des ressources livres ou morceaux de musique ainsi que les clients.

La couche de courtage

Elle représente les mécanismes nécessaires à la mise en œuvre de cette fonctionnalité.

La couche d'infrastructure

Elle représente l'utilisation du méta ORB Jonathan lors des appels de méthodes distantes.

En plus de cela, nous souhaitons mettre en évidence le rôle et l'impact des objets de synchronisation, des communications, ainsi que la consommation des ressources système dans la vue dynamique de l'exécution (voir figure 6.11). Cette vision nous permettra d'estimer le coût des communications par rapport au calcul et ainsi faire apparaître d'éventuels problèmes algorithmiques ou des goulots d'étranglement. L'étude statique devra faire apparaître les méthodes les plus fréquemments appelées ainsi que celles qui s'exécutent le plus longtemps.

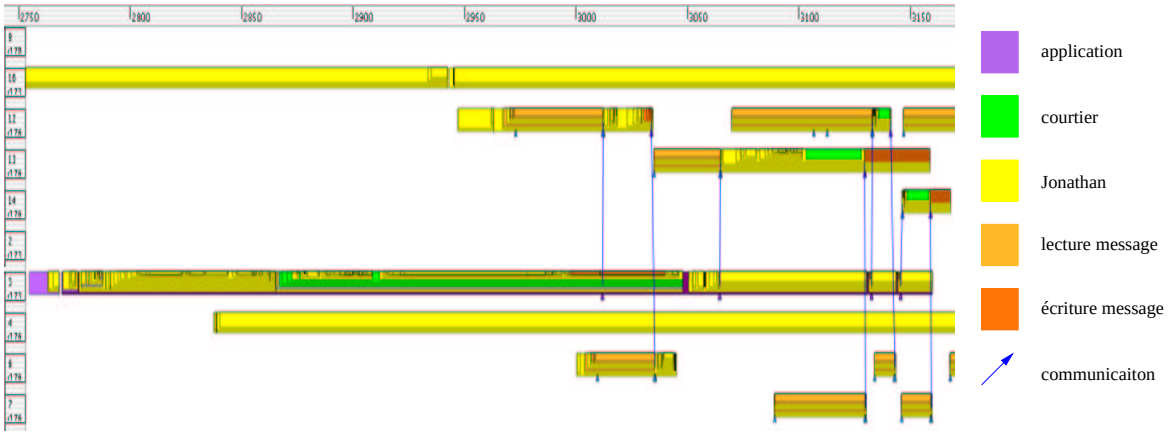


FIG. 6.11 – Sémantique des couleurs de méthode dans "Pajé".
L'exécution des méthodes propres à l'application apparaît en violet. Celle du courtiers apparaît en vert et celle de Jonathan en jaune. Les méthodes en gris font ressortir la création d'un nouveau lien de communication. Les méthodes en orange foncé correspondent aux envois de messages, et celles en orange claire au réceptions de ces derniers.

6.5.2 Exemple du serveur de livres

L'exemple représente la requête d'un client qui souhaite obtenir le résumé d'un livre présent sur un des serveurs. Notre analyse commence par l'étude de la dynamique d'exécution du courtier, du serveur et du client à l'aide du diagramme espace temps fourni par "Pajé" (voir section 6.5.2.1). Ensuite, nous analysons les consommations en ressources d'exécution au travers des durées de communication, de la consommation de la ressource processeur et de la taille prise en mémoire centrale (voir section 6.5.2.2). Puis, notre attention se porte sur l'utilisation des objets de synchronisation `wait` employés pour éviter les attentes actives (voir section 6.5.2.3). Enfin, nous réalisons une étude

statistique sur les méthodes dont la durée d'exécution est la plus longue (voir section 6.5.2.4).

6.5.2.1 Analyse du diagramme espace-temps

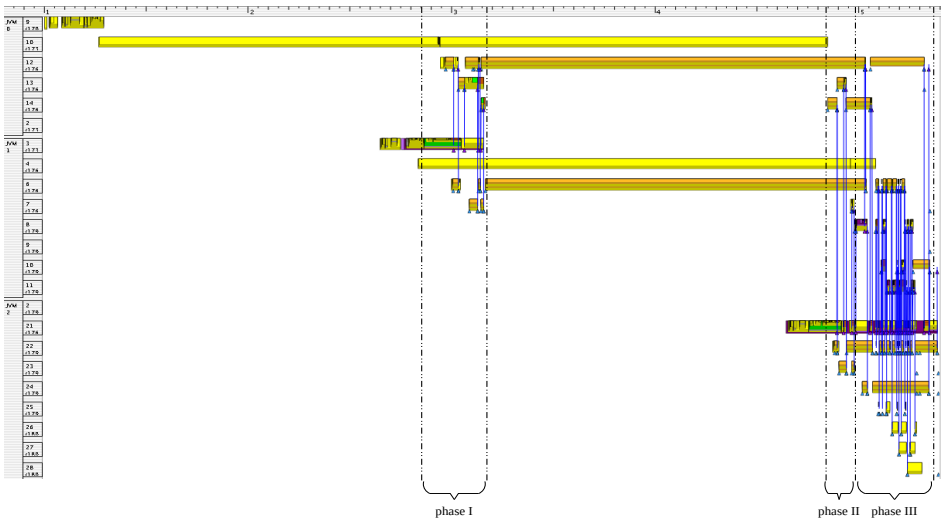


FIG. 6.12 – Exemple d'accès à une ressource «Livre».
Vue générale d'une exécution représentant l'accès à une ressource «Livre», suivie de l'obtention de son résumé. Les trois phases identifiées sont analysées par les figures 6.13, 6.14 et 6.15.

Une exécution requiert donc le lancement d'au moins un courtier [4] (voir figure 6.12) . A la suite de quoi, il nous faut lancer au moins un serveur de livre. Chaque serveur récupère une référence sur le courtier et enregistre les livres qu'il possède (voir figure 6.13). L'enregistrement de chaque livre provoque l'exécution de la méthode `put()` propre à la couche de courtage. Une fois cette phase d'initialisation terminée, un client peut être démarré.

Le client commence par obtenir une référence sur le courtier. Une fois cette référence acquise, il lui demande celle d'un serveur pouvant lui fournir une ressource répondant à un ensemble de propriétés (ce qui correspond à l'exécution d'une méthode `get()` de la couche de courtage). Le courtier lui renverra une référence pour chaque serveur susceptible de répondre à sa requête. Le client contacte le premier serveur afin qu'il lui renvoie une référence sur la ressource. Si le serveur peut répondre au client, il enregistre auprès du courtier un nouveau objet servant qui représente la ressource souhaitée par client (voir figure 6.14). Une fois l'enregistrement terminé, le client interroge le courtier pour obtenir la référence de l'objet servant. A partir de cet instant, le client peut réaliser des opérations permises par la ressource.

Dans le cas de l'accès à une ressource de type livre, les opérations possibles sont : l'obtention de la couverture, du résumé ou du contenu du livre. La figure 6.15 représente la situation où c'est

[4] Il est possible de lancer plusieurs courtiers qui coopérent pour répondre aux requêtes des clients. Dans nos exemples, un seul courtier est utilisé.

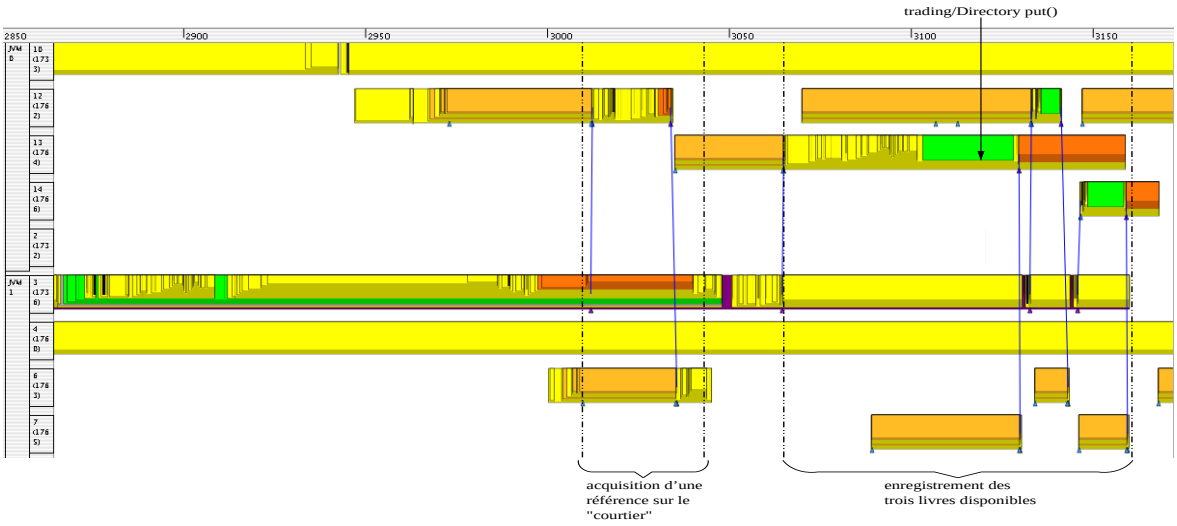


FIG. 6.13 – Initialisation d'un serveur de livre : phase I.
Après avoir récupéré une référence sur le courtier, le serveur de livre enregistre chacun des ouvrages qu'il possède.

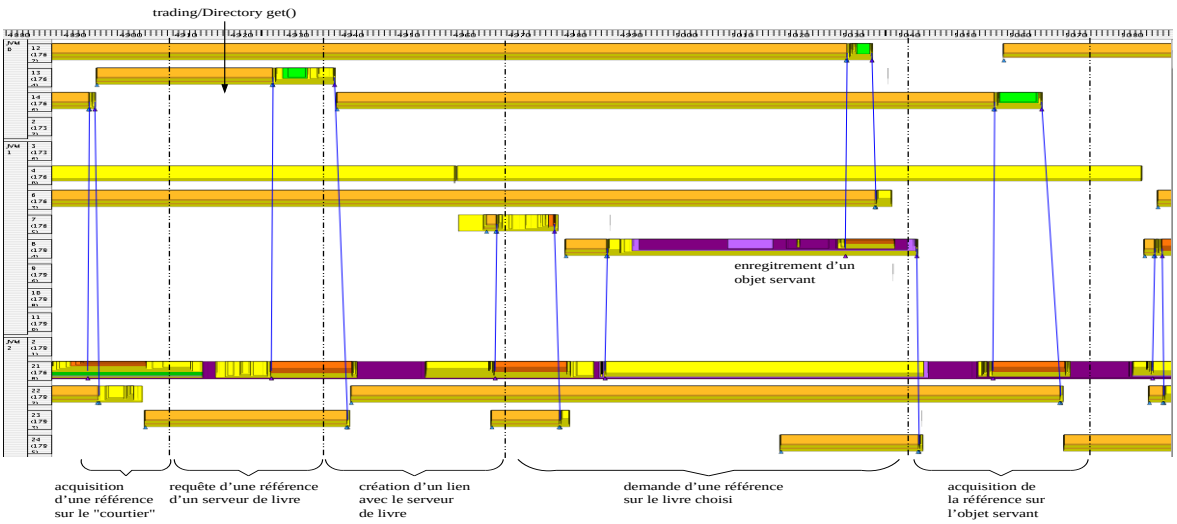


FIG. 6.14 – Initialisation d'un client : phase II.
Après avoir récupéré une référence sur le courtier, le client obtient une référence sur un objet servant nouvellement enregistré par le serveur. A chaque requête, le serveur crée un objet servant qui représente la ressource à laquelle le client veut accéder.

le résumé du livre qui est demandé. Celui-ci correspond à un fichier de 22 lignes qui sont envoyées une à une au client. Nous observons donc 22 requêtes formulées par le "*thread*" numéro 3 du client, auxquelles vont répondre les "*threads*" 6, 8, 10, 11 et 12 du serveur. Les réponses sont réceptionnées par les "*threads*" 4 et 7 du client. Cet exemple nous permet de visualiser la politique de "*pool*" de "*threads*" mis en œuvre dans Jonathan lors des l'exécution d'invocations de méthodes distantes.

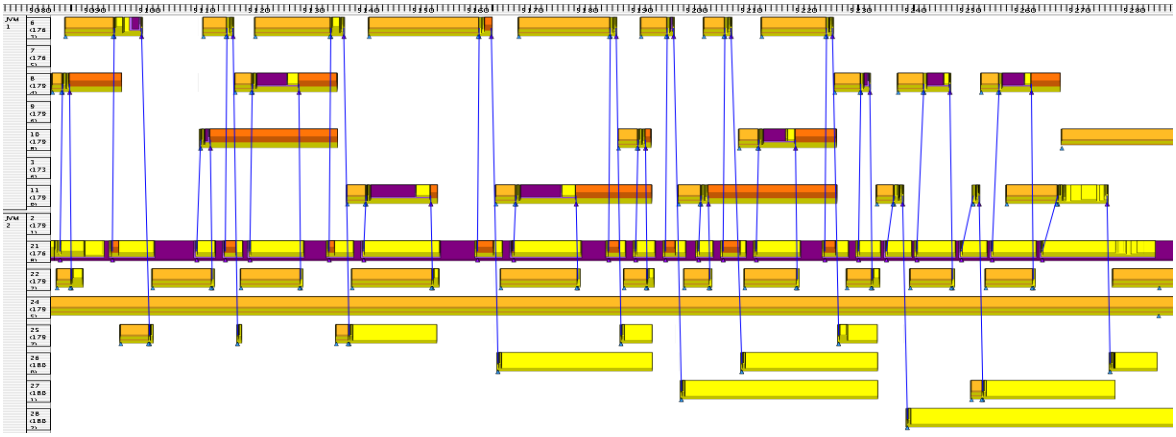
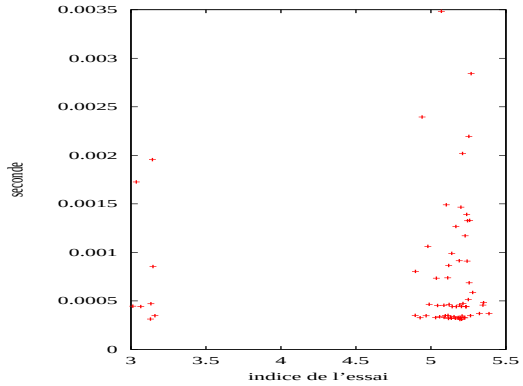
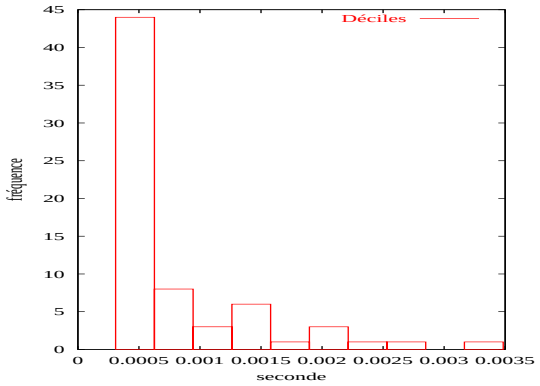


FIG. 6.15 – Lecture du résumé d'un livre : phase III.
Le client invoque une méthode distante sur l'objet servant afin qu'il lui fournisse le résumé du livre. Ce résumé correspond à un fichier de 22 lignes qui seront envoyées une à une au client.

6.5.2.2 Analyse de la consommation de ressource système

Du point de vue de l'analyse de performances, nous allons à présent étudier la façon dont l'application utilise les ressources d'exécution mises à sa disposition. Il apparaît clairement que les communications ne représentent pas un frein à l'exécution (voir tableau 6.4). Leur durée moyenne est inférieure à la milliseconde. Elles sont peut nombreuses, et leur durée totale est de $0.051686s$ sur $5.096047s$ d'exécution ($\simeq 1\%$).





Taille de l'échantillon	=	68
Moyenne	=	0.000760
Min	=	0.000309
Max	=	0.003484
Ecart-type	=	0.000663
IC 90%	:	] 0.000627 .. 0.000892 [

TAB. 6.4 – Analyse des communications pour l'exemple du serveur de livres.

Nous allons à présent nous intéresser à la consommation des ressources processeur et mémoire (voir figure 6.16).

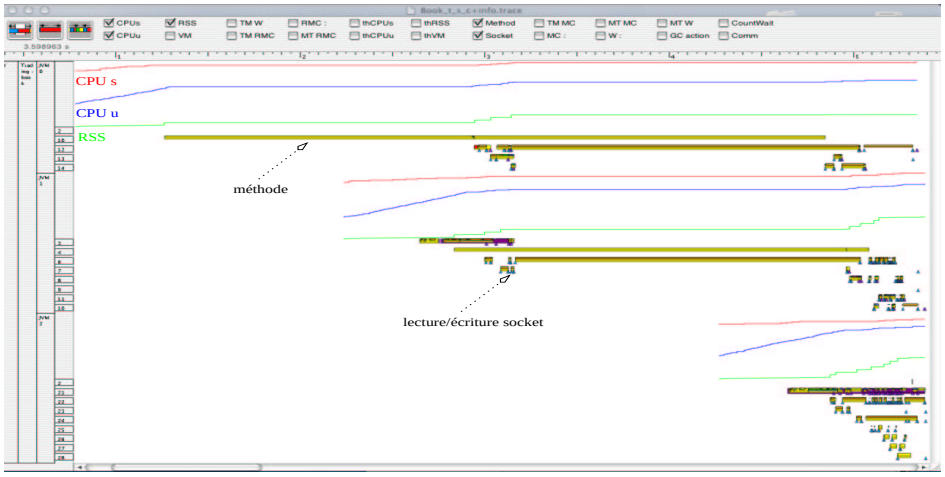


FIG. 6.16 – Exemple du «Livre» : consommation des ressources d'exécution.

Cette visualisation met en corrélation les consommations de la ressource processeur en mode utilisateur (CPU u), en mode système (CPU s) et la taille en mémoire occupée par le JVM (RSS) avec l'exécution de l'application. Les mesures réalisées sont en centième de seconde (la granularité de l'ordonnancement des processus). Elles sont ici exprimées en milliseconde.

La ressource processeur est utilisée en deux modes : utilisateur et système. Le mode utilisateur correspond à l'utilisation du processeur pour exécuter du code applicatif. Le mode système représente l'exécution des opérations de bas niveau telles que les entrées/sorties à la charge du noyau.

L'analyse de ces indices montre que la ressource processeur est principalement consommée lors de l'initialisation des JVM (voir figure 6.17). Par la suite, elle est de nouveau sollicitée lors des opérations applicatives d'enregistrement de livres, d'acquisition de la référence par le client, et de réalisation de la requête même dans une moindre proportion.

Le processeur est majoritairement utilisé en mode utilisateur. La ressource est donc principalement utilisée pour des activités du niveau applicatif. On voit de même qu'elle est consommée de façon régulière. Aucun signe de pénurie ou d'appropriation par une seule JVM n'apparaît.

La consommation de la ressource mémoire est analysée par la taille[5] occupée par les JVM en mémoire principale (RSS). Dans ce cas, le coût de l'initialisation est beaucoup moins important que pour la ressource processeur. Toutefois, les trois phases de l'exécution sont aussi le lieu privilégié de consommation de la mémoire locale (voir figure 6.18). On peut voir que la mémoire est consommée par blocs, ce qui apparaît par une courbe en forme en de créneaux.

Durant la phase d'acquisition de l'objet servant, on voit apparaître la consommation de la ressource mémoire par le client lorsqu'il initie la liaison avec le serveur. Du côté du serveur, un phéno-

[5]La taille est indiquée en nombre de pages. La taille d'une page est 4 ko.

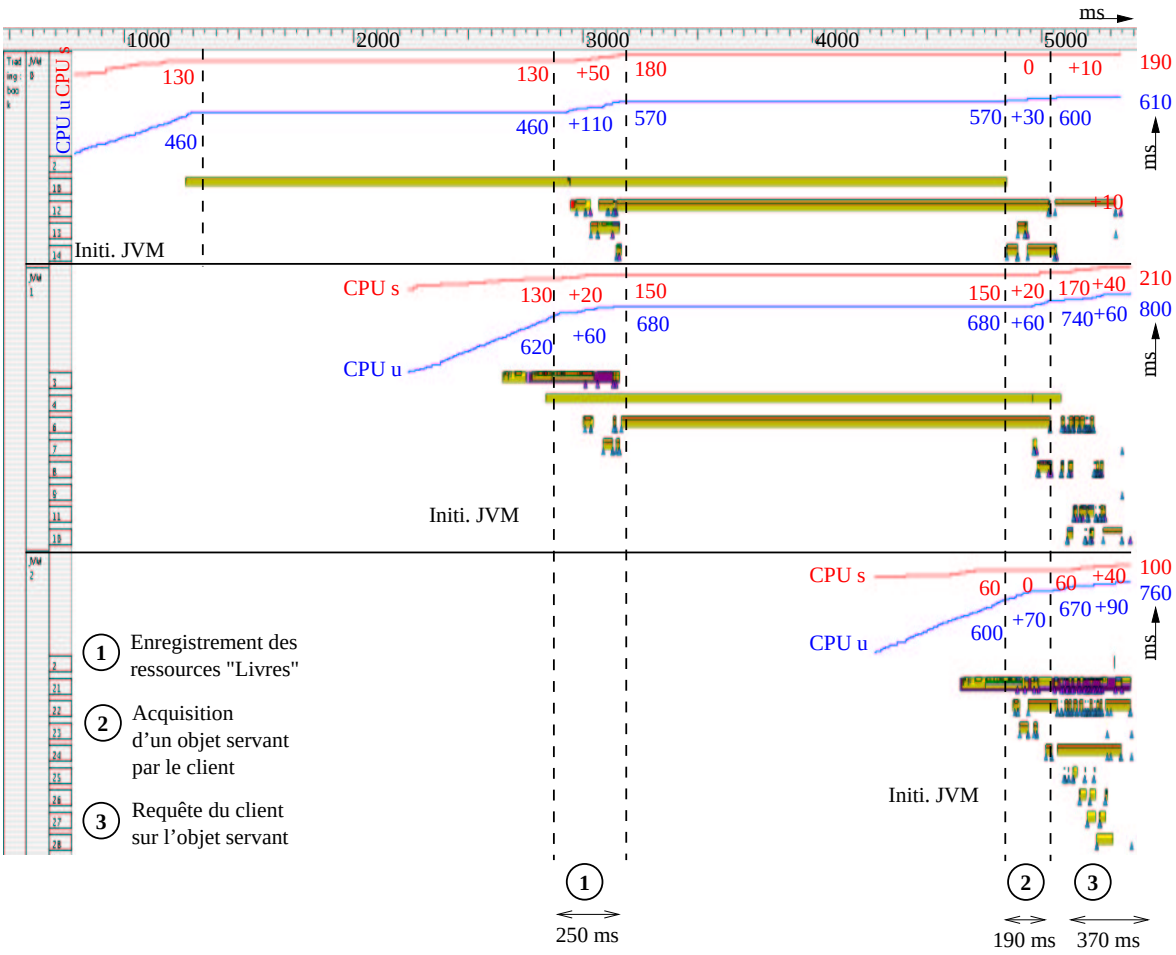


FIG. 6.17 – Exemple du «Livre» : utilisation de la ressource processeur.
Consommation de la ressource processeur en mode utilisateur (CPU u) et système (CPU s). Les mesures réalisées sont en centième de seconde (la granularité de l'ordonnancement des processus). Elles sont ici exprimées en milliseconde.

mène équivalent est détectable lors de la création et l'enregistrement de l'objet qui servira les requêtes du client.

Pour ce qui est de la phase de requête, le serveur consomme de la mémoire durant les premières invocations. Cette consommation est à rattacher à la réaction du serveur afin de répondre aux requêtes du client. Le client, quant à lui, consomme de la ressource mémoire de façon régulière au fur et à mesure de l'exécution des requêtes.

Mais les ressources processeur et mémoire ne sont pas les seules à jouer un rôle important dans l'analyse des performances. Un emploi erroné des objets de synchronisation peut aboutir à des situation de blocage aux conséquences catastrophiques pour l'exécution.

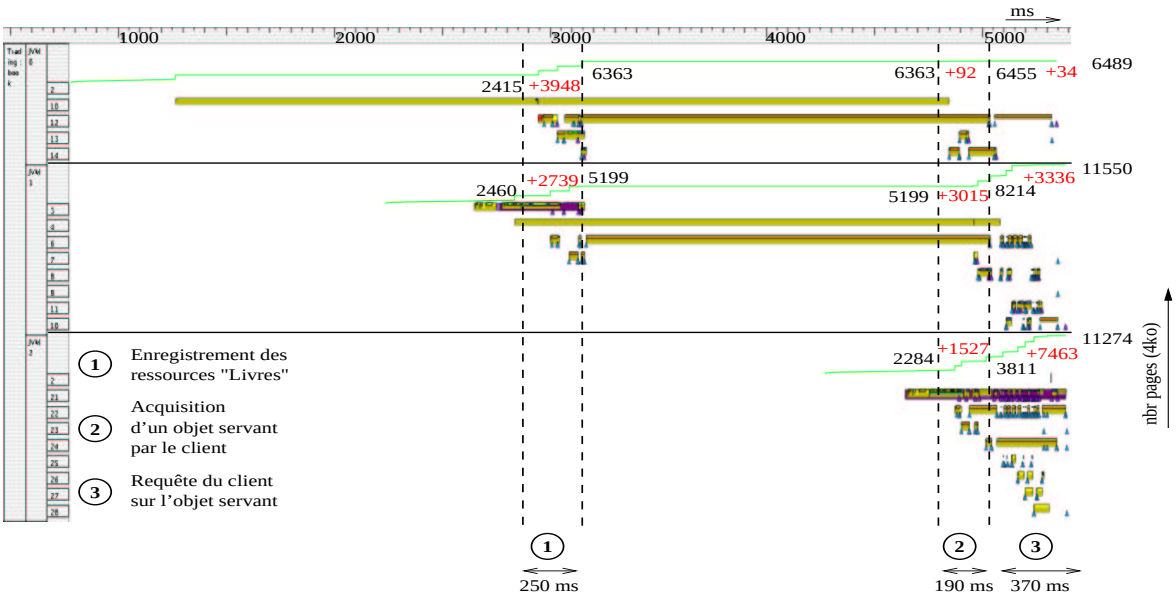


FIG. 6.18 – Exemple du «Livre» : utilisation de la ressource mémoire «vive».
Les données visualisées correspondent au nombre de pages associées aux JVM présentes dans la mémoire vive du site (“*Resident Set Size*”). La taille d'une page est de 4 ko.

6.5.2.3 Analyse de l'objet de synchronisation `wait`

Les attentes sur des objets de synchronisation sont provoquées par deux situations :

le blocage volontaire dû à l'exécution d'un appel bloquant telle que l'attente sur une “*socket*”, ou l'invocation de la méthode `wait()` ;

la sérialisation des exécutions provoquées par la présence de sections de code protégées par `synchronized`.

Nous allons à présent détailler un emploi typique de `wait` dans la mise en œuvre d'un “*pool*” de “*threads*” pour la réception de messages. Une vue globale est présentée dans la figure 6.19. Le “*thread*” 21 émet une requête et se met en attente sur un objet de synchronisation. Le “*thread*” 22, quant à lui, est positionné en attente de réception d'une communication.

Lorsque la JVM reçoit un bloc de données, c'est le “*thread*” 22 qui le réceptionne (voir figure 6.20). Le déballage des données permet d'identifier la nature du message (dans le cas présent le type de la requête). Une fois cette opération terminée le “*thread*” exécute la méthode `jonathan/libs/resources/JScheduler.JJob()` qui a pour effet de débloquer un autre “*thread*” (le “*thread*” 27) pour qu'il se place en attente d'un nouveau message (voir figure 6.21). Une fois la totalité des données déballées, le “*thread*” exécute la méthode `JScheduler.notify()` qui débloque le “*thread*” 21 qui avait envoyé le message. Le message reçu correspond donc à la réponse de la requête initiale.

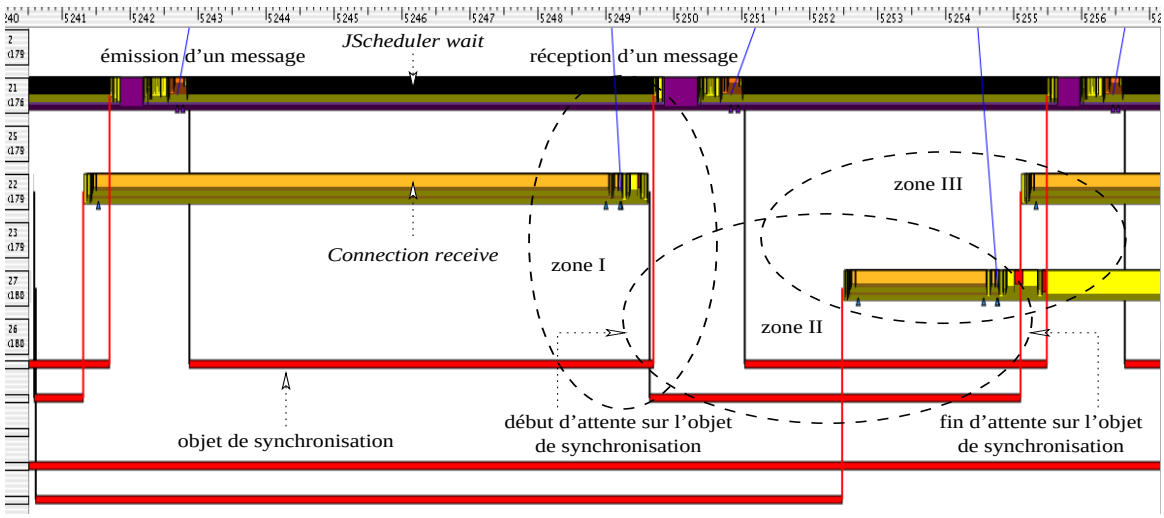


FIG. 6.19 – Utilisation des objets de synchronisation lors de la réception et l'émission de messages.
Analyse des mécanismes de synchronisation sur les objets `wait` lors de l'émission et la réception de messages. Un zoom est réalisé sur les zones I, II et III dans les figures 6.20, 6.21 et 6.22.

A la suite de quoi le "*thread*" 22 se met en attente (`JScheduler.register()`) avec les autres "*threads*" du "*pool*".

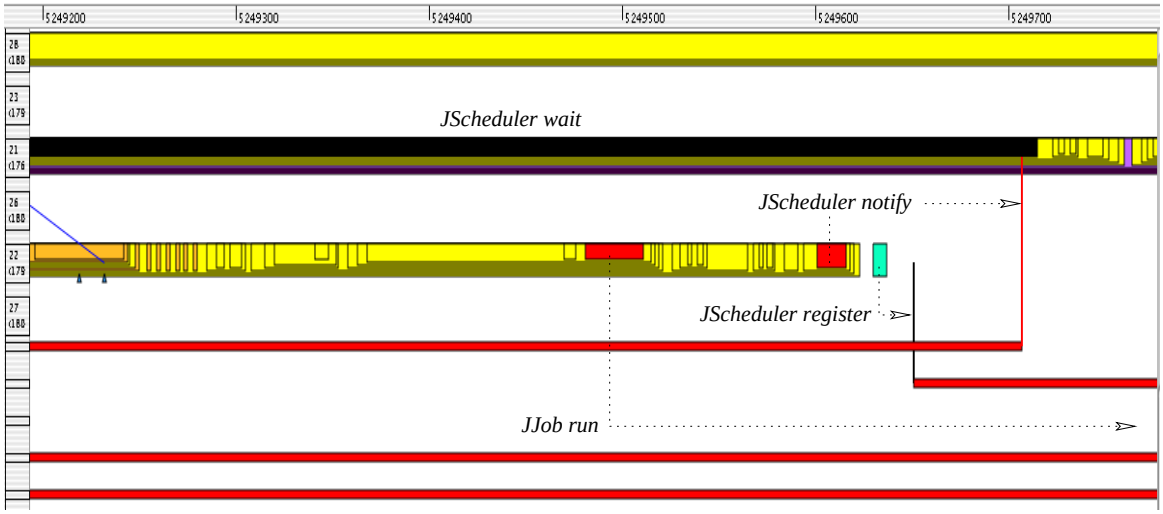


FIG. 6.20 – Zoom de la zone I (voir figure 6.19).
Le "*thread*" 22 qui réceptionne le message génère deux notifications qui débloquent des attentes sur des objets de synchronisation `wait()`. La première correspond à l'exécution de la méthode `run()` de la classe `jonathan/libs/resources/JScheduler.JJob()`. Elle provoque la mise en attente d'un message par un nouveau "*thread*". La deuxième correspond à l'exécution de la méthode `notify` de la classe `jonathan/libs/resources/JScheduler.JScheduler`. Elle débloque le "*thread*" 21 qui s'était mis en attente par le méthode `wait` de la même classe.

La figure 6.22 représente la réception d'un message par le "*thread*" 27 qui va à son tour débloquer les "*threads*" 22 et 21 avant de se remettre en attente.

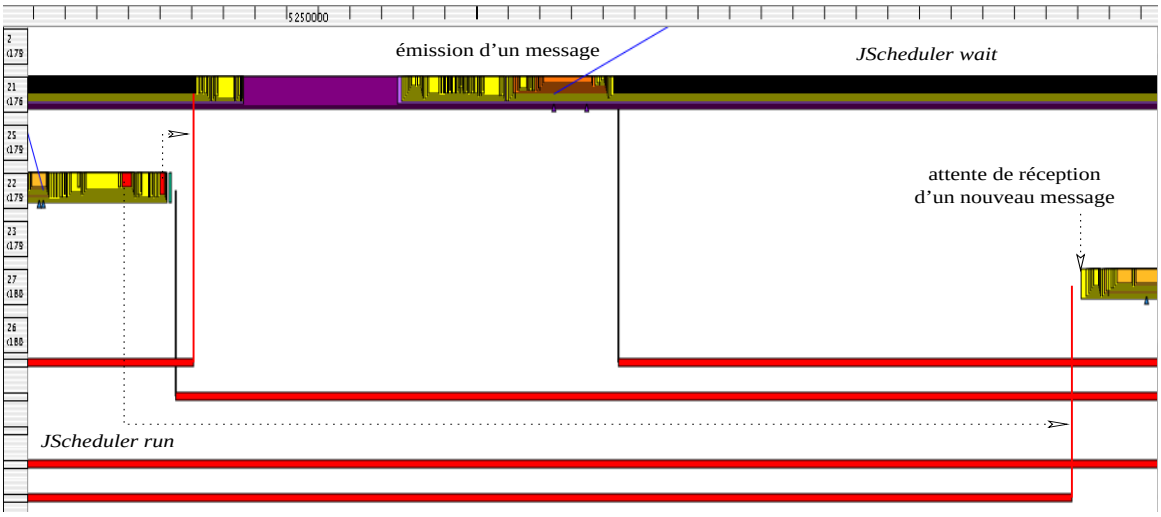


FIG. 6.21 – Zoom de la zone II (voir figure 6.19).
Le "*thread*" 22 invoque la méthode `run` de la classe `jonathan/libs/resources/JScheduler.JJob` qui débloque le "*thread*" 27. Celui-ci se met en attente de réception d'un nouveau message dès lors que le "*thread*" est ordonnancé.

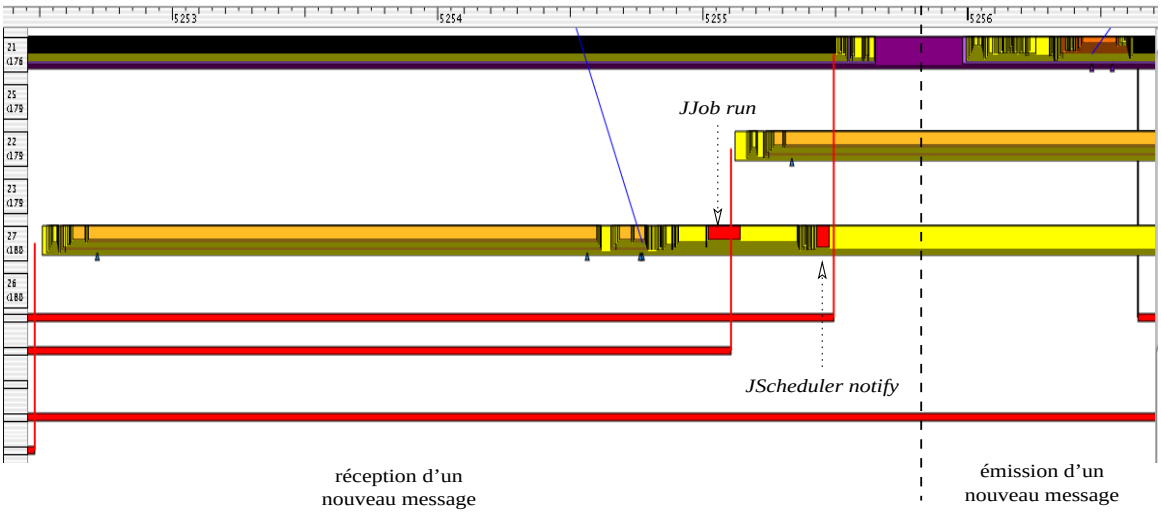


FIG. 6.22 – Zoom de la zone III (voir figure 6.19).
Le "*thread*" 27 réceptionne un nouveau message. Il invoque les méthodes `run` et `notify` qui débloquent les "*threads*" 22 puis 21. Le "*thread*" 22 se met en attente d'un nouveau message puis redonne la main au "*thread*" 28 qui débloque le "*thread*" 21.

6.5.2.4 Analyse des durées d'exécution

Nous allons à présent nous intéresser d'une façon plus fine à l'exécution des méthodes. Les outils d'analyse de performances classiques procurent des informations statistiques sur l'exécution en termes de nombre d'appels et de durée d'exécution des méthodes. Ces deux aspects sont étudiés dans le suite de cette section.

Les méthodes sont numérotées selon l'ordre de chargement des classes, et à l'intérieur d'une classe par ordre de définition.

Pour ce qui est du courtier, il y a 284 différentes méthodes appelées pour un total de 2082 appels. L'analyse en fréquence du nombre d'appel nous indique que les méthodes sont principalement appelées un faible nombre de fois (moins de 8 fois en moyenne) (voir figure 6.23). En ce qui concerne les durées d'exécution des méthodes, les dix méthodes s'exécutant le plus longtemps correspondent à 92% de la durée d'exécution totale. Elles sont représentées dans le tableau ci-dessous :

Nom de la classe	Méthode	Nbr. Appels	Durée (s)
JConnectionMgr.SrvConnectionFactory	newSrvConnection	2	3.577483
SrvConnectionFactory	newSrvConnection	2	3.577186
IPv4ConnectionFactory.SrvConnectionFactory	newSrvConnection	2	3.574003
PortableMarshallerFactory.PortableUnMarshaller	readByte	72	2.482514
CDRMarshallerFactory.CDRUnMarshallerD	prepare	17	2.480840
TcpIpChunkProvider	prepare	17	2.479265
GIOPSrvConnection	receive	17	2.476452
IPv4ConnectionFactory.Connection	receive	17	2.475828
TcpIpProtocol.Session	run	8	2.411148
GIOPProtocol.ServerSession_Low	send	8	2.390453

Hormis ces dix méthodes, la durée moyenne d'exécution des 274 autres méthodes est de $\simeq 9ms$. Si l'on considère 90% de l'échantillon total (trié par ordre croissant), la durée moyenne d'exécution descend à $\simeq 4ms$.

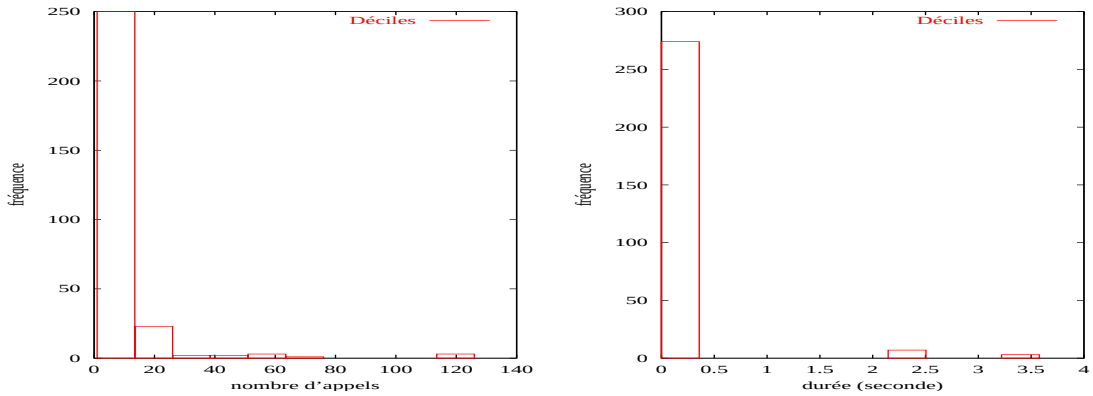


FIG. 6.23 – Déciles du nombre d'appels et de la durée d'exécution des méthodes par le courtier.

Pour ce qui est de serveur de livres, il y a 357 différentes méthodes appelées pour un total de 5336 appels. L'analyse en fréquence du nombre d'appels nous indique que les méthodes sont principalement appelées un faible nombre de fois (moins de 15 fois en moyenne) (voir figure 6.24). En

ce qui concerne les durées d'exécution des méthodes, les dix méthodes s'exécutant le plus longtemps correspondent à 77% de la durée d'exécution totale. Nous les présentons dans le tableau suivant :

Nom de la classe	Méthode	Nbr. Appels	Durée (s)
TcpIpProtocol.Session	run	29	2.376184
JConnectionMgr.SrvConnectionFactory	newSrvConnection	2	2.246312
SrvConnectionFactory	newSrvConnection	2	2.245969
IPv4ConnectionFactory.SrvConnectionFactory	newSrvConnection	2	2.243452
PortableMarshallerFactory.PortableUnMarshaller	readByte	240	2.190739
CDRMarshallerFactory.CDRUnMarshallerD	prepare	58	2.188640
TcpIpChunkProvider	prepare	58	2.186448
IPv4ConnectionFactory.Connection	receive	58	2.178939
GIOPProtocol.ClientSession_Low	send	5	1.977156
JConnectionMgr.Connection	receive	10	1.958149

Hormis ces dix méthodes, la durée moyenne d'exécution des 347 autres méthodes est de $\simeq 18ms$. Si l'on considère 90% de l'échantillon total (trié par ordre croissant), la durée moyenne d'exécution descend à $\simeq 7ms$.

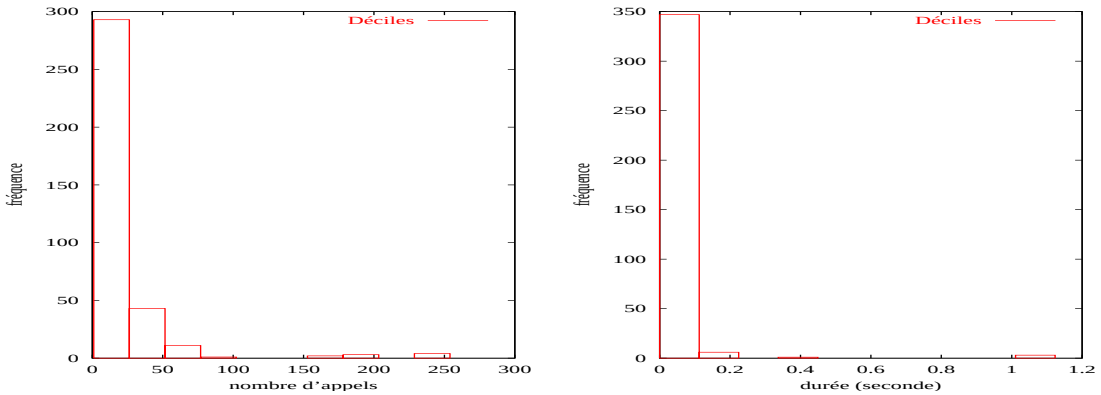


FIG. 6.24 – Déciles du nombre d'appels et de la durée d'exécution des méthodes par le serveur de livre.

Pour ce qui est du client, il y a 269 différentes méthodes appelées pour un total de 4845 appels. L'analyse en fréquence du nombre d'appels nous indique que les méthodes sont principalement appelées un faible nombre de fois (moins de 18 fois en moyenne) (voir figure 6.25). En ce qui concerne les durées d'exécution des méthodes, les dix méthodes s'exécutant le plus longtemps correspondent à 61% de la durée d'exécution totale. Il s'agit des méthodes citées dans le tableau ci-après :

Nom de la classe	Méthode	Nbr. Appels	Durée (s)
TcpIpProtocol.Session	run	29	0.960310
GIOPProtocol.ClientSession_Low	send	29	0.954643
ClientLeg	main	1	0.743033
PortableMarshallerFactory.PortableUnMarshaller	readByte	232	0.712047
CDRMarshallerFactory.CDRUnMarshallerD	prepare	50	0.710010
TcpIpChunkProvider	prepare	50	0.706757
JConnectionMgr.Connection	receive	50	0.701315
IPv4ConnectionFactory.Connection	receive	50	0.700426
Recepteur	readAbstract	1	0.333783
JStubFactory.ClientDelegate	invoke	29	0.331526

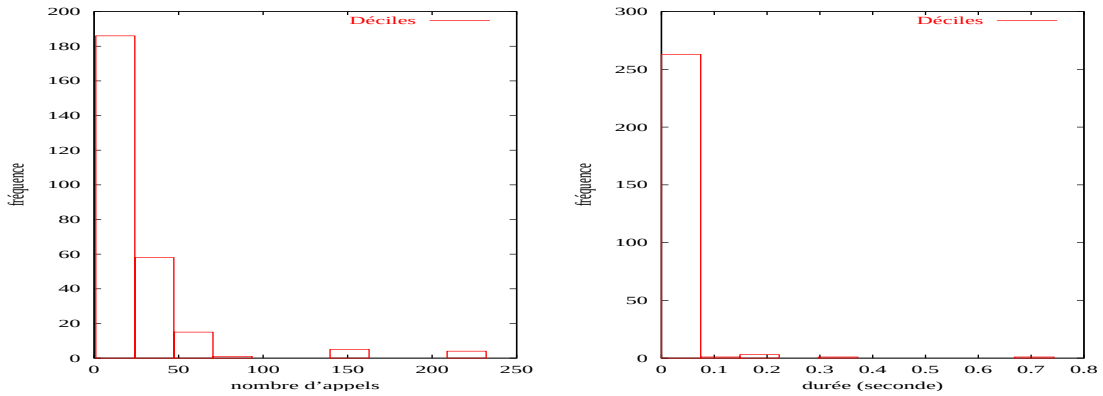


FIG. 6.25 – Déciles du nombre d'appels et de la durée d'exécution des méthodes par le serveur de livre.

Hormis ces dix méthodes, la durée moyenne d'exécution des 259 autres méthodes est de $\simeq 17ms$. Si l'on considère 90% de l'échantillon total (trié par ordre croissant), la durée moyenne d'exécution descend à $\simeq 7ms$.

Une analyse sémantique des méthodes présentées dans les tableaux ci-dessus montre une forte dépendance entre certaines de celles-ci. Il semble donc intéressant de faire ressortir des séquences types d'enchaînement de méthodes afin d'affiner nos analyses et aussi simplifier la trace à traiter.

6.5.3 Analyse d'une séquence type

Cette section a pour objet d'étudier la réception et le traitement d'une requête du point de vue du serveur. Pour ce faire, nous allons identifier quatre phases :

L'attente de requête
Le serveur attend la réception d'un message signifiant l'initialisation d'une requête.

La réception d'une requête
Le serveur reçoit un message et déballe son en-tête afin de déterminer le type de ce dernier.

Le traitement de la requête
Le serveur interprète la requête et réalise l'appel de méthode distante qui s'y rapporte.

Le retour de la requête
Le serveur prépare le retour de l'appel et renvoie le résultat.

Dans l'exemple présenté, nous analysons une situation où un client cherche à obtenir une référence sur un courtier. Pour cela, le client réalise une invocation de méthode distante qui génére une réponse de la part d'un courtier. Il s'agit ici d'un schéma classique de communication où le serveur est en attente de requête et répond à une sollicitation.

Le principe d'attente repose sur la création d'un objet de déballage de requête qui se met en attente d'un nouveau message. La mise en attente correspond à l'exécution de la méthode `readByte()` (voir figure 6.26). Cette méthode provoque l'instanciation d'un objet de déballage (`CDRUnMarshallerD.prepare()`) typé. Il s'agit du typage du protocole de communication (`Session.getProtocol()`), mais aussi de la méthode d'invocation de méthode distante attendue (`GIOPSrvConnection`). Une fois vérifié qu'aucun message n'est déjà en attente (`GIOPSrvConnection.available()`), un nouveau tampon de réception est créé (`JChunkFactory.newChunk()`) et le serveur se bloque en attente de lecture sur un objet `socket()` (`GIOPSrvConnection.receive()`).

Nous décomposons la réception d'une requête en trois phases : a) la réception d'un message ; b) l'identification de la nature de ce message ; c) le décodage du message reçu (voir figure 6.27). Dans le cas présent, le message reçu est de type IIOP [**OMG, 2001**, **Siegel, 1996**]. Il s'agit donc d'une invocation de méthode distante réalisée sur une liaison de type TCP/IP.

Le serveur est donc bloqué en attente de réception de message sur la méthode `GIOPSrvConnection.receive()`. Lorsqu'un message arrive, la réception du premier octet va terminer la première invocation de `receive` (voir figure 6.28). A la suite de quoi, le même "*thread*" va se re-

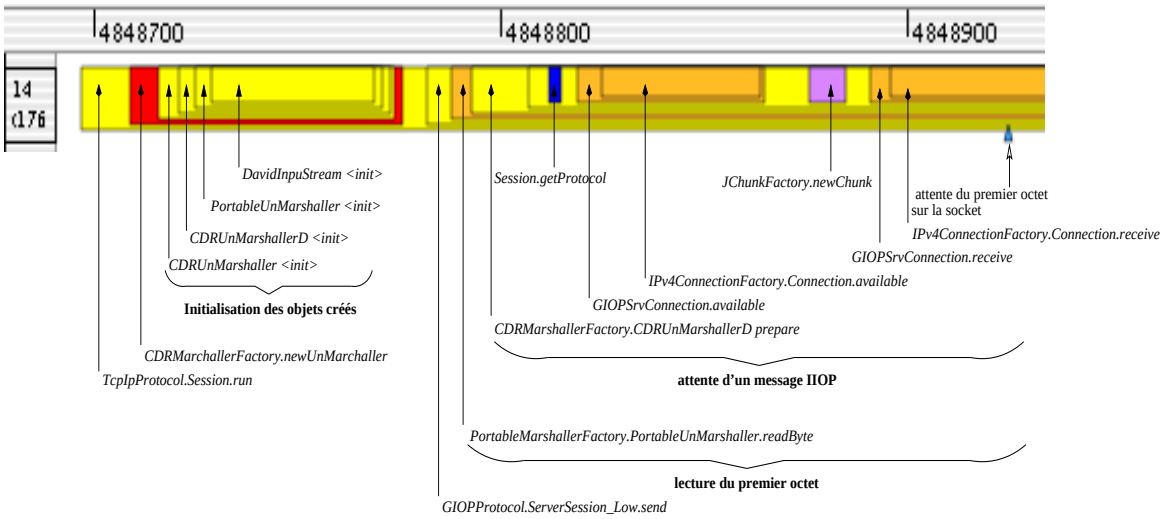


FIG. 6.26 – Attente de réception d'une nouvelle requête.

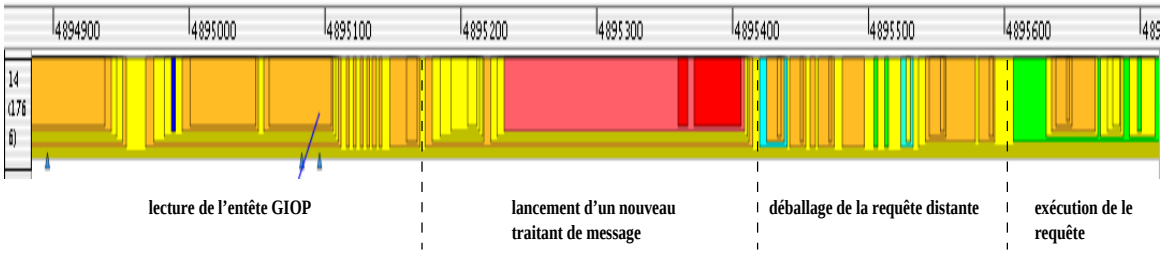


FIG. 6.27 – Réception d'une nouvelle requête.

mettre en attente afin de réceptionner la fin du message (voir figure 6.26). Le message réceptionné se trouve dans un objet de type `UnMarshaller` prêt à être déballé.

Le déballage commence l'interprétation de l'en-tête du message. Dans notre cas, les quatre premiers octets doivent correspondre aux lettres “G”, “I”, “O” et “P” ce qui identifie le type du message (voir figure 6.29). Une fois le type déterminé, le serveur déballe la suite de l'en-tête enfin de déterminer la nature et le format de la requête.

Après cela, l'objet de déballage est dupliqué afin d'être redimensionné en fonction de la taille lue dans l'en-tête (voir figure 6.30). L'ancien objet est «libéré» (`CDRMarshallerFactory.-CDRUnMarshallerD.close()`), et un nouveau “*thread*” est mis en attente de réception d'un nouveau message (`JScheduler.JJob.run()`).

L'en-tête de la requête GIOP est lu du nouvel objet de déballage. Cet en-tête contient des informations propres au contexte de la requête ainsi que la référence de l'objet distant (IOR) sur lequel

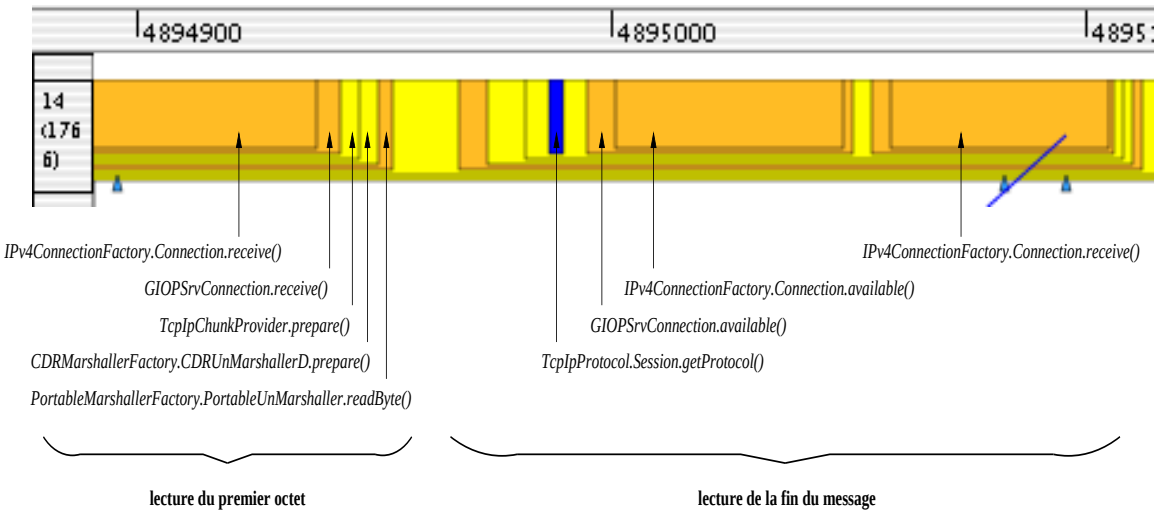


FIG. 6.28 – Lecture du premier octet et de la suite du message.

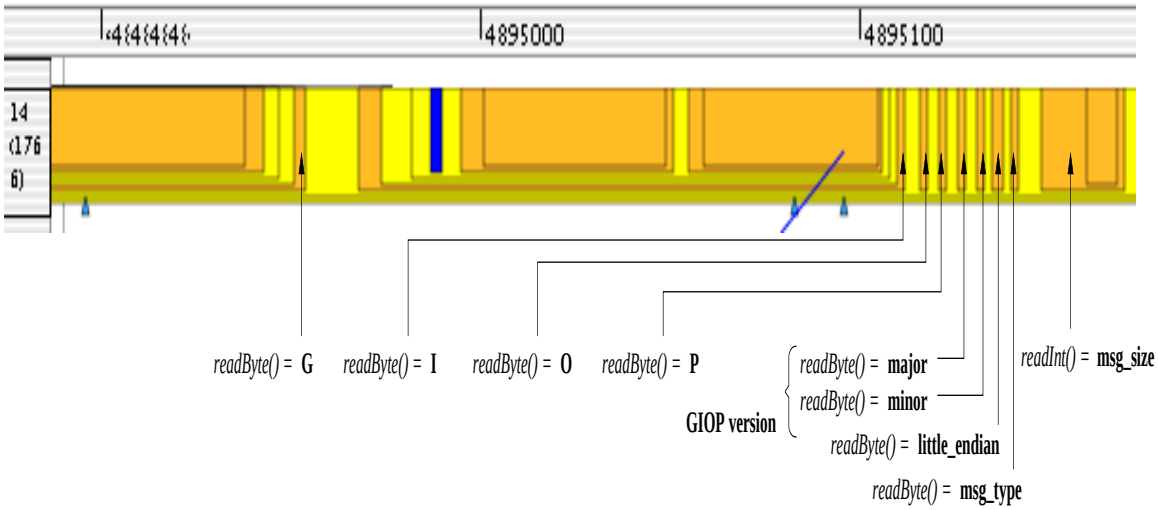


FIG. 6.29 – Déballage de l'en-tête GIOP.

l'appel de méthode porte. Une fois cette étape terminée, le serveur possède toutes les informations nécessaires pour réaliser l'opération demandée par le client.

L'invocation distante est décomposée en deux parties (voir figure 6.31). Tout d'abord, la référence de l'objet demandé est obtenue (`SingleOAdapter.SOAIdentifier.bind()`). Un nouvel objet de type `GIOPProtocol.ServerSession_High()` est créé. Il servira de support à l'invocation de la méthode ainsi qu'à l'envoi du résultat au client. Une fois l'objet créé, et la fin de l'en-tête déballé, l'invocation de méthode peut avoir lieu. Elle est supportée par la méthode `ServerDelegate.handleCorbaMethods()`. Les paramètres d'appel sont déballés, et l'objet de déballage est «libéré». Une invocation de méthode locale est réalisée (`_TraderServerImpl-Base._ids()`).

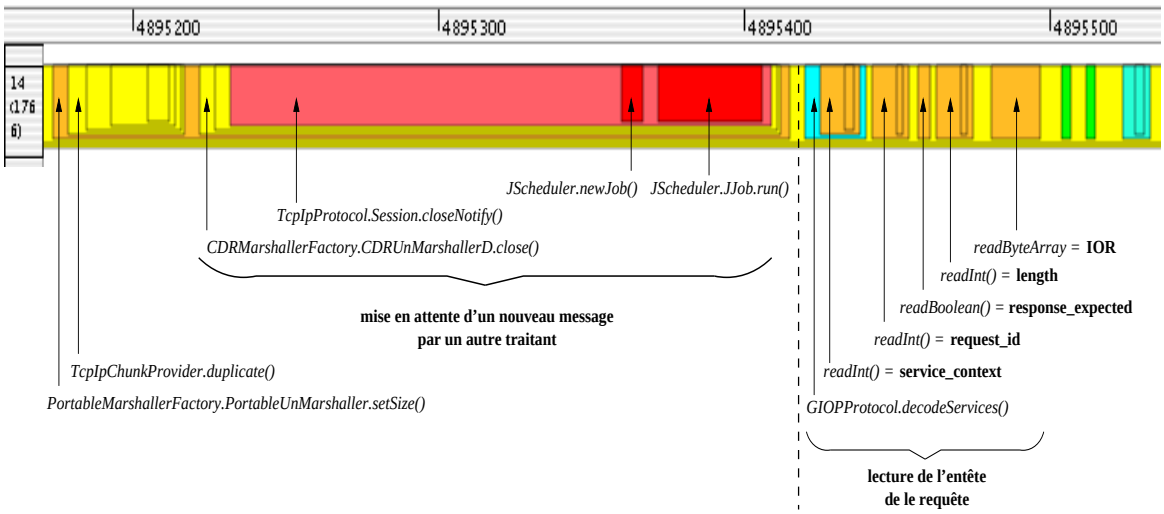


FIG. 6.30 – Mise en attente de messages d'un nouveau "*thread*" et déballage de la requête distante.

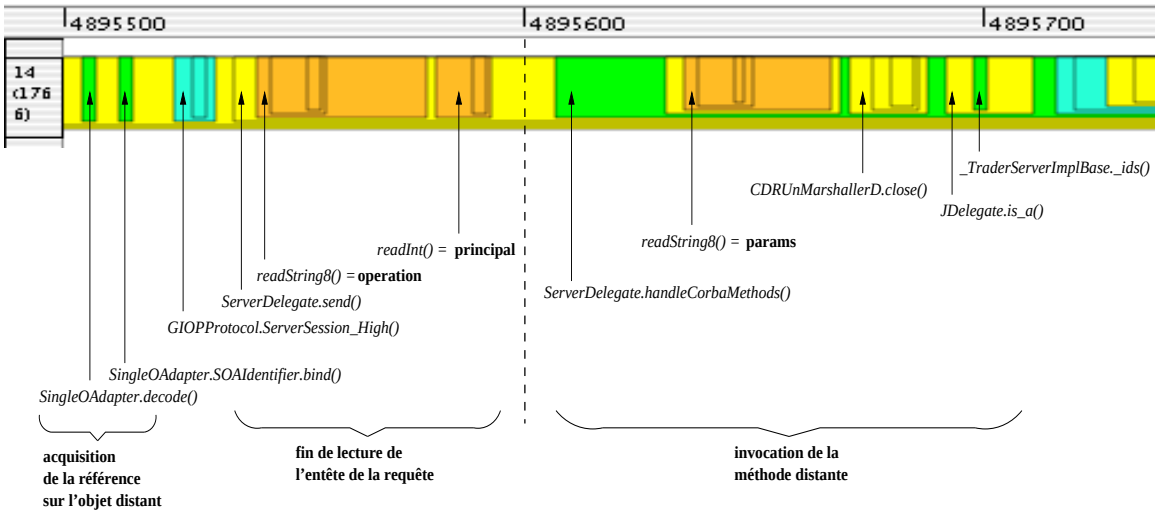


FIG. 6.31 – Traitement de la requête distante.

Une fois l'invocation terminée, la réponse est emballée pour être envoyée au client (`IPv4ConnectionFactory.Connection.emit()`) (voir figure 6.32). L'objet d'emballage qui avait été créé est «libéré» (`PortableMarshallerFactory.PortableMarshaller.close()`) et la session GIOP est fermée (`GIOPProtocol.GIOPSession_High.close()`). Le "*thread*" se met à la disposition de l'ordonnanceur pour recevoir un nouveau travail à effectuer (`JScheduler.register()`).

L'analyse de cet exemple fait apparaître les mécanismes de base d'invocation de méthode distante mise en œuvre dans Jonathan, c'est-à-dire :

– l'attente, par un "*thread*", de réception d'une requête.

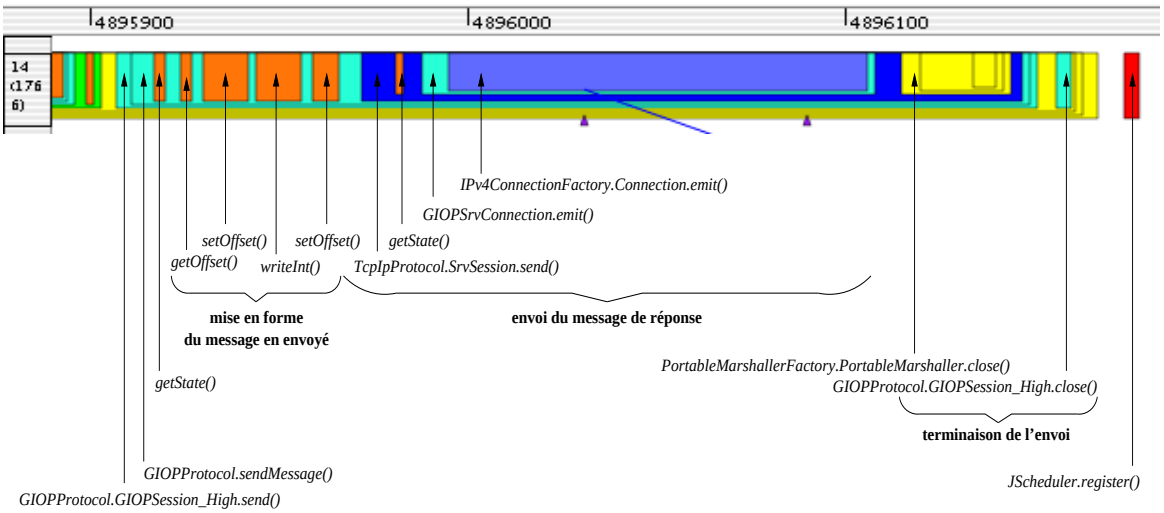


FIG. 6.32 – Retour de la requête vers le client.

– la réception d'un message qui provoque la mise en attente d'un nouveau "*thread*".

– l'interprétation du contenu du message.

– l'exécution du travail qui s'y rapporte.

– le retour du résultat à l'initiateur de la requête.

Elle met en évidence l'utilisation de "*pool*" de "*threads*" pour le traitement des requêtes ainsi que le cheminement du message au travers des couches de l'infrastructure d'invocation de méthode distante. Des séquences caractéristiques d'appel de méthode sont identifiées. Il serait intéressant de les utiliser afin de créer des représentations de plus haut niveau qui réduiraient et simplifieraient l'analyse des traces traitées.

6.6 Conclusion

Ce chapitre a établi «l'utilisabilité» de notre infrastructure en termes d'observation et d'analyse. Nous avons, en effet, montré que l'infrastructure d'observation ne perturbe pas de façon excessive l'exécution. Ainsi les données analysées sont significatives du comportement non observé.

De plus, nous avons validé l'intérêt de l'observation multi-niveaux pour, d'une part reconstruire la dynamique d'exécution d'une application répartie et d'autre part identifier les consommations de ressources système qu'elle génère.

Comme notre analyse porte sur une trace événementielle, il nous a de même été possible de réaliser une étude statistique sur l'exécution des méthodes et les communications observées.

Enfin, nous avons pu repérer des séquences d'appels de méthodes types tels que l'attente et le traitement d'une requête distante. L'identification de ces séquences permet de construire des représentations de plus haut niveau qui peuvent être utilisées pour simplifier l'analyse de la trace.

Chapitre 7

Conclusion et perspectives

7.1 Conclusion

Le travail présenté se place dans le contexte des applications à objets réparties. Ces applications se basent typiquement sur une couche intergicielle, par exemple un ORB de type CORBA, pour rendre transparents les problèmes liés à la répartition. L'exécution est alors modélisée comme un ensemble d'objets qui calculent et interagissent.

Notre travail a permis de traiter la problématique de la reconstruction post-mortem de la dynamique d'exécution afin de réaliser une analyse qualitative et quantitative multi-niveaux des ressources d'exécution consommées. Pour cela, **nous avons vérifié l'hypothèse que l'observation des interactions entre objets, par le prisme de l'observation multi-niveaux, procure les informations nécessaires à nos analyses.**

Cette problématique a notamment été appliquée dans le cadre du projet de méta-ORB écrit en Java, Jonathan, développé au sein du département DTL/ASR de France Télécom R&D. Cette étude a donné lieu au développement d'une infrastructure d'observation et d'analyse quantitative multi-niveaux pour application Java répartie.

7.2 Principales contributions techniques

La problématique de l'observation d'applications parallèles ou réparties est abordée par de nombreux projets (voir section 1.4). Toutefois, les solutions proposées ne répondent pas entièrement à la problématique traitée. Nous avons donc développé une infrastructure d'observation multi-niveaux fournissant des traces événementielles du niveau système et applicatif (voir section 5.1). Ces traces sont converties dans un format interprétable par l'environnement d'observation "Pajé" (voir sec-

tion 5.2.2), environnement dans lequel nous menons nos analyses dynamiques, qualitatives et quantitatives.

Les observations du niveau système sont obtenues par déroutage d'appels système du noyau Linux (voir section 5.1.2). Le déroutage a lieu par insertion dynamique d'un module. Les observations du niveau applicatif sont réalisées via la JVMPI exportée par la JVM de Sun (voir section 5.1.1). Cette approche a l'avantage de ne modifier ni l'application, ni le support d'interaction Jonathan, ni le noyau du système hôte.

Nous avons développé des fonctions de projection et conversion qui transforment les traces brutes en une trace interprétable par "Pajé" (voir section 5.2). C'est dans ces fonctions que l'on intègre le modèle d'exécution de l'application. Grâce à une projection dans un référentiel de sens commun, nous sommes capable de mettre en cohérence des observations de différents niveaux d'abstraction. Il nous est notamment possible d'associer des indices de consommation de ressources d'exécution à l'analyse de la dynamique de l'application et cela dans un contexte d'exécution répartie [1].

L'infrastructure d'observation et d'analyse quantitative multi-niveaux développée est testée et utilisée pour l'observation d'un serveur multimédia. Grâce à un surcoût d'observation limité (dans notre cas il est d'environ 7% de la durée totale d'exécution), la trace événementielle collectée peut être considérée comme significative de l'exécution observée (voir section 6.3).

L'infrastructure se présente sous la forme d'une bibliothèque, qui doit être chargée dynamiquement par les JVM, de modules à insérer dans le noyau et de fonctions de conversion pour la construction de la trace "Pajé". L'outil de visualisation "Pajé" est disponible sur le site internet du laboratoire I.D. [**Pajé, 2001**].

7.3 Limites et perspectives

Bien évidemment, le travail réalisé dans cette thèse n'est pas complet. Plusieurs travaux restent à mener et notamment pour répondre au problème de précision et de qualité du système de datation globale. Nous avons démontré, dans le section 6.4, qu'il est difficile d'estimer correctement le paramètre de décalage à l'origine. A cela nous voyons deux raisons possibles. Tout d'abord l'outil de mesure utilisé. Dans notre cas nous utilisons le système de datation local fourni par le noyau Linux. Nos machines possèdent un "*Time Stamp Counter*", mais comme nous l'avons analysé, il existe un vrai problème de fiabilité dans l'estimation de la fréquence du processeur lors de la phase d'initialisation (voir section 4.1.4). Ce problème ne semble pas pouvoir être résolu sans se baser sur une source de référence extérieure dont la qualité serait garantie. Dans un deuxième temps, il nous semble que le modèle d'horloge employé ne correspond pas au comportement des systèmes de datation présents

[1] Les indices de consommation sont locaux aux sites d'exécution, mais ils sont interprétables les uns par rapport aux autres grâce à l'analyse des interactions distantes (voir section 6.5.2.2).

dans les machines utilisées. Les travaux de Lethuillier [**Lethuillier, 2001**], réalisés dans le cadre d'un stage de maîtrise au laboratoire Informatique et Distribution de Grenoble, montrent que la température du processeur, et donc la charge du système, joue un rôle de toute importance dans l'estimation des paramètres de correction d'horloge. Or, à l'heure actuelle, la température du processeur ne rentre pas dans le modèle d'horloge utilisé.

D'autre part, il nous faudrait porter plus d'attention sur le format des enregistrements réalisés. Nous n'avons pas particulièrement cherché à optimiser la taille et la structure de ces derniers. Or, lors de l'observation d'un système fortement variable, ou si l'on s'intéresse à de longues durées d'exécution, la taille de la trace devient un problème critique du point de vue du stockage et du transport. Des techniques de compression de données devront être mises en œuvre pour répondre à cette critique.

De la même façon, bien que le niveau global de perturbation de l'exécution soit très acceptable (voir section 6.3), il serait important d'approfondir les techniques pour l'observation ainsi que celles de correction de l'intrusion. Cette action nous paraît tout particulièrement importante dans le cadre de l'observation du niveau système où le surcoût d'observation semble facilement optimisable.

Dans le cadre des perspectives à plus long terme, la piste de la reconnaissance des «patterns d'exécution» nous semble tout particulièrement intéressante. Les travaux de [**Mansouri-Samani et Sloman, 1997**], sur la définition d'un langage permettant de spécifier des abstractions de haut niveau en fonction d'événements de bas niveau et de contraintes logiques et temporelles, nous semblent être un bon point de départ. Ce type d'approche nous permettrait de simplifier la trace, par substitution de patterns par un équivalent de plus haut niveau, pour l'analyse mais aussi pour le traitement d'instances de plus grande taille. Toutefois, il nous faut auparavant étudier dans quelle mesure la sémantique des abstractions est équivalente à celle des observations. Si tel était le cas, il serait intéressant de voir si ces nouvelles traces «abstraites» peuvent permettre, à l'aide des techniques de preuve formelle, de prouver des propriétés de comportement de l'exécution. De même, serait il possible d'identifier, ou de définir, des comportements fonctionnels qui seront recherchés dans la trace d'observations ?

D'autre part, bien que notre modèle d'exécution semble assez générique pour ce qui est des exécutions d'applications à objets réparties, nous nous posons la question de sa généralisation à d'autres modèles d'exécution. Quelles seraient les limites de notre approche dans le cadre des modèles à base de processus communiquants, d'objets mobiles ou de composants ?

Il est de même important de s'interroger sur l'intégration de nouvelles sources et de nouvelles qualités d'information. Pour ce qui est des sources, la prise en compte d'informations provenant du routage des paquets et de la topologie des liens de communication nous semble être une démarche porteuse. Il est en effet intéressant de s'interroger sur la façon dont les communications se répartissent et utilisent les médiums de transport. En ce qui concerne la qualité de l'information, il nous semble pertinent de réfléchir sur les techniques d'observation et d'estimation du comportement dans des situations où une partie du système est opaque voire complètement obscure (on n'a accès qu'à une

partie de l'information, voir à aucune). Dans un tel cas, il nous faut modéliser l'exécution des parties non transparentes à partir des informations provenant des parties observables et du comportement théorique des parties qui ne le sont pas. Cette situation se retrouvera tout particulièrement lorsque l'on s'intéresse aux environnements embarqués mobiles.

Au titre des extensions de domaine, il nous semble intéressant d'aborder les problématiques liées à l'observation en ligne dans des systèmes de type «temps réel». Comment observer et quoi pour ne pas être submergé d'information ? Quel traitement doit-on réaliser sur les données ? Comment mettre en place un système de datation globale dans ce cas ?

Le débogage réparti [**Roos, 1994**] serait un champ d'investigation des plus pertinents. Le couplage avec l'outil de visualisation "Pajé" nous semble être particulièrement porteur dans cette optique. Il possède en effet un fort potentiel pour traiter ce genre de situation de par son extensibilité et l'interactivité dont il fait part.

Enfin, le domaine du contrôle est aussi de problématique connexe à la nôtre. Il demande, en effet, l'obtention d'informations sur l'exécution d'un système et plus particulièrement sur sa dynamique. Une approche multi-niveaux nous paraît être tout particulièrement indiquée pour construire une vue globale du contexte d'exécution, sur laquelle seront basées les opérations de contrôle. De plus, les travaux de [**Sekar, 1999**] démontrent qu'il est possible de détecter des attaques de système par la reconnaissance de motifs dans une trace événementielle. Cette application est en direct prolongation d'études sur la reconnaissance de patterns. L'acquisition de données sur l'exécution d'une application peut permettre la mise en place d'alarmes à partir desquelles il sera alors possible de réagir.

«O'Brien, en passant devant le télécran, parut frappé d'une idée. Il s'arrêta, se tourna et pressa un bouton sur le mur. Il y eut un bruit sec et aigu. La voix s'était arrêtée.

Julia laissa échapper un petit cri, une sorte de cri de surprise. Même dans sa panique, Winston fut trop abasourdi pour pouvoir tenir sa langue.

– *Vous pouvez le fermer ! s'exclama-t-il.*

– *Oui, répondit O'Brien. Nous pouvons le fermer. Nous avons ce privilège.*

(...)

– *Le dirai-je, ou voulez-vous le dire ? demanda-t-il.*

– *Je le dirai, répondit promptement Winston. Cette chose est-elle réellement fermée ?*

– *Oui. Tout est fermé. Nous sommes seuls.» **[Orwell, 1950]***

...fin des observations.

Bibliography

[**Aydt, 1994**]

AYDT, R. « The pablo self-defining data format ». Rapport Technique, Departement of Computer Science at the University of Illinois, Urbana-Champaign.

[**Bernard et al., 1999**]

BERNARD, P.-E., GAUTIER, T., et TRYSTRAM, D. « Large scale simulation of parallel molecular dynamics ». Dans *Proceedings of Second Merged Symposium IPPS/SPDP 13th International Parallel Processing Symposium and 10th Symposium on Parallel and Distributed Processing*, San Juan, Puerto Rico.

[**Berrendorf et Ziegler, 1998**]

BERRENDORF, R. et ZIEGLER, H. « Pcl - the performance counter library: A common interface to access hardware performance counters on microprocessors ». Rapport Technique, Forschungszentrum Jlich.

[**Bertalanffy, 1973**]

BERTALANFFY, L. Théorie générale des systèmes. Dunod, Paris.

[**Blayo et al., 1999**]

BLAYO, E., DEBREU, L., MOUNIÉ, G., et TRYSTRAM, D. « Topic 03 - dynamic load balancing for ocean circulation model with adaptive meshing ». Dans AMESTOY, P., BERGER, P., DAYDÉ, M., DUFF, I., FRAYSSÉ, V., GIRAUD, L., et RUIZ, D., éditeurs, *Euro-Par' 99 Parallel Processing - 5th International Euro-Par Conference*, numéro 1685 dans Lecture Notes in Computer Science, pages 303–312.

[**Blondeau et Latrive, 2000**]

BLONDEAU, O. et LATRIVE, F. « *Libres enfants du savoir numérique* ». Valensi, M., L'ÉCLAT édition.

[**Bouchi et al., 2001**]

BOUCHI, A., OLEJNIK, et R. LECOUFFE, P. « Un mécanisme d'observation de la charge des machines en java ». Dans *Actes des Rencontres francophones du parallélisme (RenPar 13)*, Paris, France.

[**Boutrous Saab, 2000**]

BOUTROUS SAAB, C. « *Conception et réalisation d'une plate-Forme d'observation adaptable à granularité variable pour systèmes et applications répartis* ». Thèse de doctorat en informatique, Université Pierre et Marie Curie.

[**Briat et al., 1997**]

BRIAT, J., GINZBURG, I., PASIN, M., et PLATEAU, B. « Athapascan runtime: Efficiency for irregular problems ». Dans *Proceedings of the Europar'97 Conference*, pages 590–599, Passau, Germany. Springer Verlag.

[**Brinch Hansen, 1973**]

BRINCH HANSEN, P. « *Operating System Principles* ». Prentice Hall, Englewood Cliffs.

[**Browne et al., 2000**]

BROWNE, S., DONGARRA, J., GARNER, N., HO, G., et MUCCI, P. « A portable programming interface for performance evaluation on modern processors ». *The International Journal of High Performance Computing Applications*, 14(3):189–204.

[**Brunst et al., 2001**]

BRUNST, H., WINKLER, M., NAGEL, W., et HOPPE, H.-C. « Performance optimization for large scale computing: The scalable vampir approach ». Dans ALEXANDROV, V., DONGARRA, J., JULIANO, B., RENNER, R., et KENNETH TAN, C., éditeurs, *ICCS'01: International Conference in Computational Science*, Lecture Notes in Computer Science 2074, pages 751–770, Berlin, Heidelberg. Springer.

[**Buchanan et al., 1998**]

BUCHANAN, B., NIEHAUS, D., DHANDAPANI, G., et MENON, R. « The data stream kernel interface ». Rapport Technique, University of Kansas
http://www.ittc.ku.edu/datastream/.

[**Buck et Hollingsworth, 2000**]

BUCK, B. et HOLLINGSWORTH, J. K. « An api for runtime code patching ». *The International Journal of High Performance Computing Applications*, 14(4):317–329.

[**Card et al., 1998**]

CARD, R., DUMAS, E., et MÉVEL, F. « *Programmation Linux 2.0 API système et fonctionnement du noyau* ». Eyrolles.

[**Carissimi, 1999**]

CARISSIMI, A. « *Le noyau exécutif Athapascan-0 et l'exploitation de la multiprogrammation légère sur les grappes de stations multiprocesseurs* ». Thèse de doctorat en informatique, Institut National Polytechnique de Grenoble, France.

[**Cavalheiro, 1999**]

CAVALHEIRO, G.-G.-H. « *Athapascan-1 : Interface générique pour l'ordonnancement dans un environnement d'exécution parallèle* ». Thèse de doctorat en informatique, Institut National Polytechnique de Grenoble, France.

[**Chassin de Kergommeaux, 2000**]

CHASSIN DE KERGOMMEAUX, J. « *Observation d'exécutions parallèles* ». Mémoire d'habilitation à diriger des recherches, Institut National Polytechnique de Grenoble.

[**Dang Tran et al., 1996**]

DANG TRAN, F., PEREBASKINE, V., STEFANI, J.-B., CRAWFORD, B., KRAMER, A., et OTWAY, D. « Binding and streams: the retina approach ». Dans *Proc. TINA '96*.

[**Dasgupta, 1986**]

DASGUPTA, P. « A probe-based fault tolerant scheme for the clouds operating system ». *TR*, GIT-ICS-86/05.

[**de Oliveira Stein, 1999**]

DE OLIVEIRA STEIN, B. « *Visualisation interactive et extensible de programmes parallèles à base de processus légers* ». Thèse de doctorat en informatique, Institut National Polytechnique de Grenoble, France.

[**Dijkstra, 1965a**]

DIJKSTRA, E. W. « Cooperating sequential processes ». Rapport Technique EWD-123, Technological University, Eindhoven, the Netherlands.

[**Dijkstra, 1965b**]

DIJKSTRA, E. W. « Solution of a problem in concurrent programming control ». *Communications of the ACM*, 8(9):569.

[**Dijkstra, 1968**]

DIJKSTRA, E. W. « The structure of the "THE" multiprogramming system ». *Communications of the ACM*, 11(5):341–346.

[**Duda et al., 1987**]

DUDA, A., HADDAD, Y., HARRUS, G., et BERNARD, G. « Estimating global time in distributed systems ». Dans *Proc. 7th International Conference On Distributed Computing Systems*, Berlin.

[**Dumant et al., 1998**]

DUMANT, B., DANG TRANG, F., HORN, F., et STEFANI, J.-B. « Jonathan: an open distributed processing environment in java ». Dans *Middleware'98: IFPI International Conference on Distributed Systems Platforms and Open Distributed Processing*.

[**Galilée et al., 1998**]

GALILÉE, F., ROCH, J.-L., CAVALHEIRO, G. H., et DOREILLE, M. « Athapascan-1: On-line building data flow graph in a parallel language ». Dans *Pact'98*, Paris, France.

[**Geib et al., 1999**]

GEIB, J.-M., GRANSART, C., et MERLE, P. « *CORBA: Des concepts à la pratique* ». Masson Editeur, Masson, France.

[**Goipnath et al., 1998**]

GOIPNATH, A., NIMMAGADDA, S., LIYANNARACHCHI, C., et NIEHAUS, D. « Performance measurement of corba endsystems ». Rapport Technique, University of Kansas.

[**Graham et al., 1982**]

GRAHAM, S. L., KESSLER, P. B., et MCKUSICK, M. K. « gprof: a call graph execution profiler ». Dans *Proceedings of the SIGPLAN '82 Symposium on Compiler Construction*, pages 120–126, Boston, MA.

[**Hac, 1997**]

« *Dictionnaire Universel Francophone* », Hachette édition.

[**Haddad, 1988**]

HADDAD, Y. « *Performance dans les systèmes répartis : des outils pour les mesures* ». Thèse de doctorat en informatique, Université Paris-Sud, Orsay.

[**Hoare, 1974**]

HOARE, C. A. R. « Monitors: An operating system structuring concept ». *Communications of the ACM*, 17(10):549–557.

[**Hofmann et al., 1994**]

HOFMANN, R., KLAR, R., MOHR, B., QUICK, A., et SIEGLE, M. « Distributed performance monitoring: Methods, tools, and applications ». *IEEE Transactions on Parallel and Distributed Systems*, 5(6):585–598.

[**Iec, 1995a**]

IEC, I. « Open distributed processing- reference model - part 1: Overview international standard 10746-1 itu-t recommendation x.902 »
http://www.iso.ch.

[**Iec, 1995b**]

IEC, I. « Open distributed processing- reference model - part 2: Foundations international standard 10746-2 itu-t recommendation x.902 »
http://www.iso.ch.

[**IEEE, 1995**]

IEEE. « *IEEE 1003.1c-1995: Information Technology — Portable Operating System Interface*

(POSIX) - System Application Program Interface (API) Amendment 2: Threads Extension (C Language) ». IEEE Computer Society Press, 1109 Spring Street, Suite 300, Silver Spring, MD 20910, USA.

[**Jain, 1991**]
JAIN, R. « *The Art of Computer Systems Performance Analysis.* ». John Wiley & Sons, Inc., Wiley édition.

[**Jain et al., 1996**]
JAIN, V., SPEZIALETTI, M., et GUPTA, R. « An approach for monitoring intrusion removal in real time systems ». Rapport Technique, University of Pittsburgh.

[**Joyce et al., 1987**]
JOYCE, J., LOMOW, G., SLIND, K., et UNGER, B. « Monitoring distributed systems ». *ACM Transactions on Computer Systems*, 5(2):121–150.

[**JVMPI, 1999**]
« *Java Virtual Machine Profiler Interface (JVMPI)* »
http://java.sun.com/products/jdk/1.2/docs/index.html.

[**JVMSpec, 1999**]
« *The Java Virtual Machine Specification* »
http://java.sun.com/products/jdk/1.2/docs/index.html.

[**Karavanic et al., 1997**]
KARAVANIC, K. L., MYLLYMAKI, J., LIVNY, M., et MILLER, B. P. « Integrated visualization of parallel program performance data ». *Parallel Computing*, 23(1–2):181–198.

[**Kazi et al., 2000**]
KAZI, I. H., JOSE, D. P., BEN-HAMIDA, B., HESCOTT, C. J., KWOK, C., KONSTAN, J. A., LILJA, D. J., et YEW, P.-C. « Javiz: A client/server java profiling tool ». *j-IBM-SYS-J*, 39(1):96–117.

[**Krakowiak, 1985**]
KRAKOWIAK, S. « *Principes des systèmes d'exploitation des ordinateurs* ». Dunod informatique, Paris.

[**Lamport, 1978**]
LAMPORT, L. « Time, clocks and the ordering of events in distributed systems ». *Communications of the ACM*, 21(7):558–564.

[**Lange et al., 1992**]
LANGE, F., KROEGER, R., et GERGELEIT, M. « Jewel: design and implementation of a distributed measurement system ». *IEEE Transactions on Parallel and Distributed Systems*, 3(6):657–671.

[**Lethuillier, 2001**]

LETHUILLIER, V. « Implémentation d'une technique de datation globale pour une grappe de pc ». Stage de Maîtrise, Laboratoire Informatique et Distribution, Grenoble.

[**Ludwig et al., 1996**]

LUDWIG, T., OBERHUBER, M., et WISMÜELLER, R. « An open monitoring system for parallel and distributed programs ». *Lecture Notes in Computer Science*, 1123.

[**Ludwig et Wismüeller, 1997**]

LUDWIG, T. et WISMÜELLER, R. « Omis 2.0 — universal interface for monitoring systems ». *Lecture Notes in Computer Science*, 1332(1332):267–276.

[**Ludwig et al., 1997**]

LUDWIG, T., WISMÜLLER, R., OBERHUBER, M., et BODE, A. « An open interface for the on-line monitoring of parallel and distributed programs ». *The International Journal of Supercomputer Applications and High Performance Computing*, 11(2):160–174.

[**Maillet, 1995**]

MAILLET, E. « Issues in performance tracing with tape-pvm ». Dans *Second European PVM User's Group Meeting*, pages 143–148. Hermès.

[**Maillet, 1996**]

MAILLET, E. « *Le traçage logiciel d'applications parallèles : conception et ajustement de qualité* ». Thèse de doctorat en informatique, Institut National Polytechnique de Grenoble, France.

[**Maillet et Tron, 1995**]

MAILLET, E. et TRON, C. « On efficiently implementing global time for performance evaluation on multiprocessor systems ». *Journal of Parallel and Distributed Computing*, 28(1):84–93.

[**Mansouri-Samani et Sloman, 1992**]

MANSOURI-SAMANI, M. et SLOMAN, M. « Monitoring distributed systems (a survey) ». Rapport Technique DOC92/23, Imperial College of Science, Technology and Medecine.

[**Mansouri-Samani et Sloman, 1997**]

MANSOURI-SAMANI, M. et SLOMAN, M. « Gem: A generalized event monitoring language for distributed systems ». *Distributed Systems Engineering*, 4(2):96–108.

[**Miller, 1993**]

MILLER, B. P. « What to draw? when to draw? an essay on parallel program visualization ». *Journal of Parallel and Distributed Computing*, 18(2):265–269.

[**Miller et al., 1994a**]

MILLER, B. P., CALLAGHAN, M. D., CARGILLE, J. M., HOLLINGSWORTH, J. K., IRVIN, R. B.,

KARAVANIC, K. L., KUNCHITHAPADAM, K., et NEWHALL, T. « The paradyn parallel performance measurement tools ». Rapport Technique, Department of Computer Sciences, University of Wisconsin.

[**Miller et al., 1994b**]

MILLER, B. P., HOLLINGSWORTH, J. K., et CALLAGHAN, M. D. « The paradyn parallel performance tools and PVM ». Rapport Technique, Department of Computer Sciences, University of Wisconsin.

[**Mills, 1991**]

MILLS, D. « Internet time synchronization: the network time protocol ». *IEEE Trans. Communications*, 10(COM-39):1482–1493.

[**Mills et Kamp, 1999**]

MILLS, D. et KAMP, P.-H. « The nanokernel ». Rapport Technique, University of Delaware.

[**Mills, 1989**]

MILLS, D. L. « Rfc 1128: Measured performance of the network time protocol in the internet system »
http://www.cis.ohio-state.edu/htbin/rfc/INDEX.rfc.html.

[**Mohr et al., 1996**]

MOHR, B., MALONY, A. D., et CUNY, J. E. Tau. Dans WILSON, G. V. et LU, P., éditeurs, *Parallel Programming in C++*, Scientific and Engineering Computation Series, pages 589–628. MIT Press, Cambridge, MA.

[**Nagel et al., 1996**]

NAGEL, W. E., ARNOLD, A., WEBER, M., HOPPE, H.-C., et SOLCHENBACH, K. « Vampir: Visualization and analysis of mpi resources ». *Supercomputer*, 12(1):69–80.

[**Newhall et Miller, 1998**]

NEWHALL, T. et MILLER, B. P. « Performance measurement of interpreted programs ». *Lecture Notes in Computer Science*, 1470:146–154.

[**Newhall et Miller, 2000**]

NEWHALL, T. et MILLER, B. P. « Performance measurement of dynamically compiled Java executions ». *Concurrency: Practice and Experience*, 12(6):343–362.

[**Niehaus et al., 1999**]

NIEHAUS, D., DINKEL, W., et HOUSE, S. « Effective real-time system implementation with kurt linux ». Dans *Real-Time Linux Workshop*, Vienne, Austria.

[**Nimmagadda et al., 1999**]

NIMMAGADDA, S., LIYANAARNCHCHI, C., GOPINATH, A., NIEHAUS, D., et KAUSHAL, A.

« Performance patterns: Automated scenario-based orb performance evaluation ». Dans *COOTS 99*, pages 15–28, San Diego, California.

[**Oates, 1995**]

OATES, T. « Fault identification in computer network A review and a new approach ». Technical Report UM-CS-1995-113, University of Massachusetts, Amherst, Computer Science.

[**OMG, 2001**]

« *The Common Object Request Broker: Architecture and Specification* ». Object Management Group, CORBA 2.4.2 édition.

[**Orwell, 1950**]

ORWELL, G. « *1984* ». Gallimard.

[**Ottogalli et al., 2001**]

OTTOGALLI, F.-G., LABBÉ, C., OLIVE, V., DE OLIVEIRA STEIN, B., CHASSIN DE KERGOMMEAUX, J., et VINCENT, J.-M. « Visualisation of distributed applications for performance debugging ». Dans ALEXANDROV, V., DONGARRA, J., JULIANO, B., RENNER, R., et KENNETH TAN, C., éditeurs, *ICCS'01: International Conference in Computational Science*, Lecture Notes in Computer Science 2074, pages 831–840, Berlin, Heidelberg. Springer.

[**Ottogalli et Vincent, 1999**]

OTTOGALLI, F.-G. et VINCENT, J.-M. « Mise en cohérence et analyse de traces logicielles ». *Calculateurs Parallèles, Réseaux et Systèmes répartis*, 11(2).

[**Pajé, 2001**]

« *WWW Pajé Home Page* »
http://www-apache.imag.fr/software/paje.

[**Perdijon, 1998**]

PERDIJON, J. « *La mesure* ». Flammarion.

[**Rabenseifner, 1997**]

RABENSEIFNER, R. « The controlled logical clock - a global time for trace based software monitoring of parallel applications in workstation clusters ». Dans *5th EUROMICRO Workshop on Parallel and Distributed Processing*, London UK.

[**Roos, 1994**]

ROOS, J.-F. « *Mise au point d'applications distribuées pour environnement de développement basé sur une technologie objet.* ». Thèse de doctorat en informatique, Laboratoire d'Informatique Fondamentale de l'Université de Lille-I.

[**Rose et al., 1998**]

ROSE, L. D., ZHANG, Y., et REED, D. A. « Svpablo: A multi-language performance analysis system ». *Lecture Notes in Computer Science*, 1469.

[**Sekar, 1999**]

SEKAR, R. « On preventing intrusions by processing behavior monitoring ». Dans *Workshop on Intrusion Detection and Network Monitoring*, Santa Clara, California, USA.

[**Sha et al., 1990**]

SHA, L., RAJKUMAR, R., et LEHOCZKY, J. P. « Priority inheritance protocols: An approach to real-time synchronization ». *IEEE Transactions on Computers*, 39(9):1175–1185.

[**Sheehan et al., 1999**]

SHEEHAN, T., MALONY, A., et SHENDE, S. « A runtime monitoring framework for the tau profiling system ». Dans *Third International Symposium on Computing in Object-Oriented Parallel Environments (ISCOPE'99)*, San Francisco, CA.

[**Shende, 1999**]

SHENDE, S. « Profiling and tracing in linux ». Dans *Extreme Linux Workshop 2*, Monterey CA. USENIX.

[**Shende et Malony, 2001**]

SHENDE, S. et MALONY, A. « Integration and application of the tau performance system in parallel java environments ». Dans *Joint ACM Java Grande - ISCOPE 2001 Conference*.

[**Shende et al., 1998**]

SHENDE, S., MALONY, A., CUNY, J., LINDLAN, K., BECKMAN, P., et KARMESIN, S. « Portable profiling and tracing for parallel, scientific applications using c++ ». Dans *Proceedings of the SIGMETRICS Symposium on Parallel and Distributed Tools (SPDT-98)*, pages 134–147, New York. ACM Press.

[**Shende et al., 2001**]

SHENDE, S., MALONY, D., et ANSELL-BELL, R. « Instrumentation and measurement strategies for flexible and portable empirical performance evaluation ». Dans *PDPTA'01: Tools and Techniques for Performance Evaluation Workshop*. C.S.R.E.A.

[**Siegel, 1996**]

SIEGEL, J. « *CORBA: Fundamentals and Programming* ». John Wiley & Sons Inc., New York, 1 édition.

[**Silberschatz et Galvin, 1994**]

SILBERSCHATZ, A. et GALVIN, P. B. « *Principes des systèmes d'exploitation* ». Addison-Wesley, Reading, 4° édition.

[**Stefani, 1995**]

STEFANI, J.-B. « Open distributed processing: An architecture basis for information networks ». *Computer Communications*, 18(11).

[**Stefani et al., 1998**]

STEFANI, J.-B., DUMANT, B., TRAN, F. D., et HORN, F. « The retina dpe kernel: A flexible, real-time orb framework ». *Lecture Notes in Computer Science*, 1430:286–296.

[**Stevens, 1990**]

STEVENS, R. W. « *UNIX Network Programming* ». Software Series. Prentice Hall PTR.

[**Tamches et Miller, 1999**]

TAMCHES, A. et MILLER, B. « Fine-grained dynamic instrumentation of commodity operating system kernels ». Dans *OSDI'99: 3rd Symposium on Operating Systems Design and Implementation*.

[**Tanenbaum, 1994**]

TANENBAUM, A. S. « *Modern Operating Systems* ». Prentice Hall, Englewood Cliffs.

[**Thom, 1980**]

THOM, R. « *Paraboles et catastrophes* ». Flammarion, Paris, Coll. Champs édition.

[**Torvalds et al., 2001**]

« *Linux Kernel Documentation* »
DocBook project.

[**Tra, 1997**]

« *Trading Object Service Specification* ». OMG.

[**Waheed et al., 1996**]

WAHEED, A., ROVER, D. T., et HOLLINGSWORTH, J. K. « Modeling, evaluation, and testing of paradyn instrumentation system ». Dans *Supercomputing '96 Conference Proceedings: November 17–22, Pittsburgh, PA*, New York, NY 10036, USA and 1109 Spring Street, Suite 300, Silver Spring, MD 20910, USA. ACM Press and IEEE Computer Society Press.

[**Wismüller et al., 1998**]

WISMÜLLER, R., TRINITIS, J., et LUDWIG, T. « Ocm–monitoring system for interoperable tools ». Dans *SPDT'98: SIGMETRICS Symposium on Parallel and Distributed Tools*, pages 1–9, Technishe Universitat Munchen, Oregon. ACM SIGMETRICS.

[**Wu et al., 2000**]

WU, C. E., BOLMARCICH, A., WOOTTON, M. S. D., PARPIA, F., CHAN, A., LUSK, E., et GROPP, W. « From trace generation to visualization: A performance framework for distributed parallel systems ». Dans *SC2000 : High Performance Networking and Computing. Dallas Convention Center, Dallas, TX, USA, November 4–10, 2000*, pages 69–70, New York, NY 10036, USA and 1109 Spring Street, Suite 300, Silver Spring, MD 20910, USA. ACM Press and IEEE Computer Society Press.

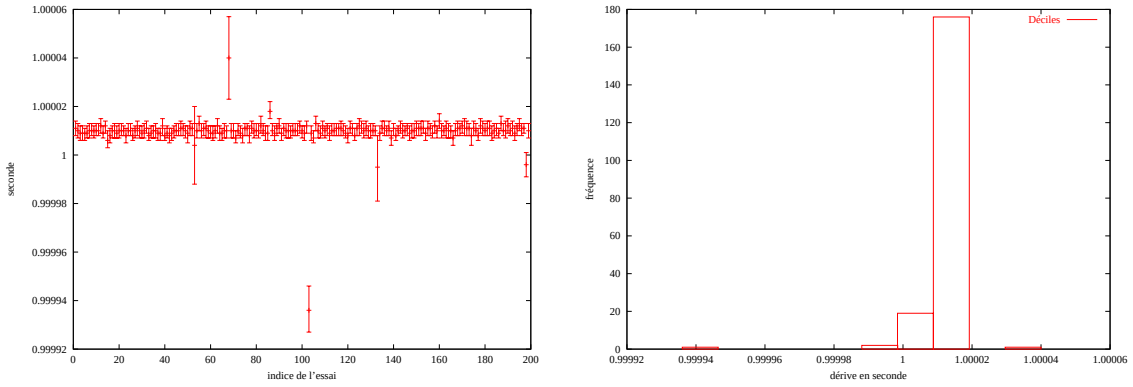
[**Zieliński et al., 1995**]

ZIELIŃSKI, K., LAURENTOWSKI, A., SZYMASZEK, J., et USZOK, A. « A tool for monitoring software-heterogeneous distributed object applications ». Dans *Proceedings of the 15th International Conference on Distributed Computing Systems (ICDCS'95)*, pages 11–18, Los Alamitos, CA, USA. IEEE Computer Society Press.

Annexe A

Paramètres de correction d'horloge G-eant

L'analyse statistique du paramètre de dérive d'horloges montre une bonne stabilitée de la qualité de l'estimateur (voir figure A.1). Le calcul de la longueur de l'intervalle de confiance à 90%, pour le couple de sites (A, C), est de 3.10^{-6} (il est de 6.10^{-6} pour le couple (A, B) : voir section 6.4). Nous pouvons donc conclure que la valeur moyenne calculée est représentative de la valeur théorique estimée.

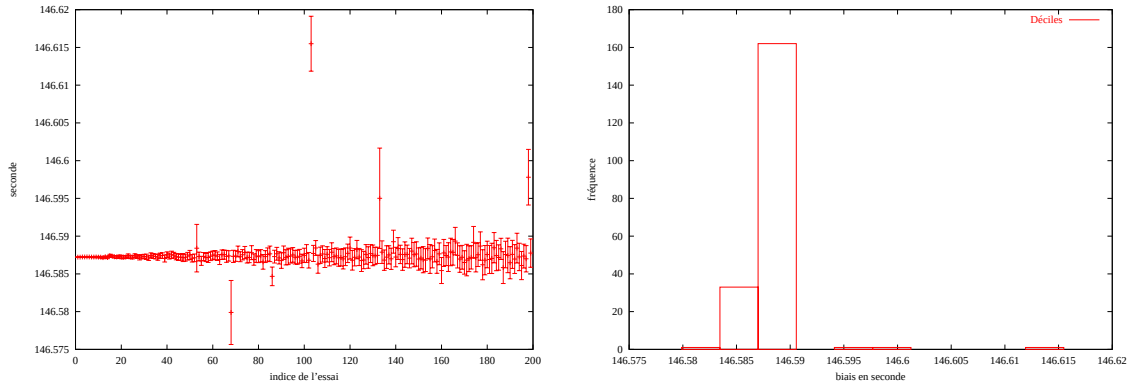


Échantillon complet

Taille de l'échantillon	=	200
Moyenne	=	1.000009
Min	=	0.999936
Max	=	1.00004
Ecart-Type	=	0.000009
IC 90%	:	] 1.000008 .. 1.000011 [

FIG. A.1 – Paramètres de dérive de l'horloge entre les sites A et C.

Pour ce qui est de l'analyse du paramètre de décalage à l'origine (voir figure A.2), la qualité de l'estimateur est fortement dépendante de la distance à la date d'origine, tout comme c'est le cas pour le couple (A,B) (voir section 6.4).



Échantillon complet

Taille de l'échantillon	=	200
Moyenne	=	146.58752
Min	=	146.57990
Max	=	146.61550
Ecart-Type	=	0.003023
IC 90%	:	] 146.58717 .. 146.58788 [

FIG. A.2 – Paramètres de biais de l'horloge de la machine g-eant.

Quoiqu'il en soit, les estimateurs calculés pour les deux couples de sites sont de même qualité et identifient le même problème de décalage à l'origine. Cette analyse est confirmée par d'autres échantillons réalisés avec les mêmes couples de machine ainsi qu'avec d'autres couples de machine. Une analyse de Fourier sur les échantillons peut être menée afin d'identifier des périodes qui peuvent expliquer cette rapide dégradation.

Annexe B

Exemple du serveur de musique

La trace présentée dans cette annexe correspond à une requête d'un client souhaitant écouter un morceau de musique. Nous retrouvons trois phases d'exécution : l'enregistrement des morceaux de musique auprès du courtier par le serveur ; l'acquisition de l'objet servant par le client ; l'exécution de la requête (voir figure B.1).

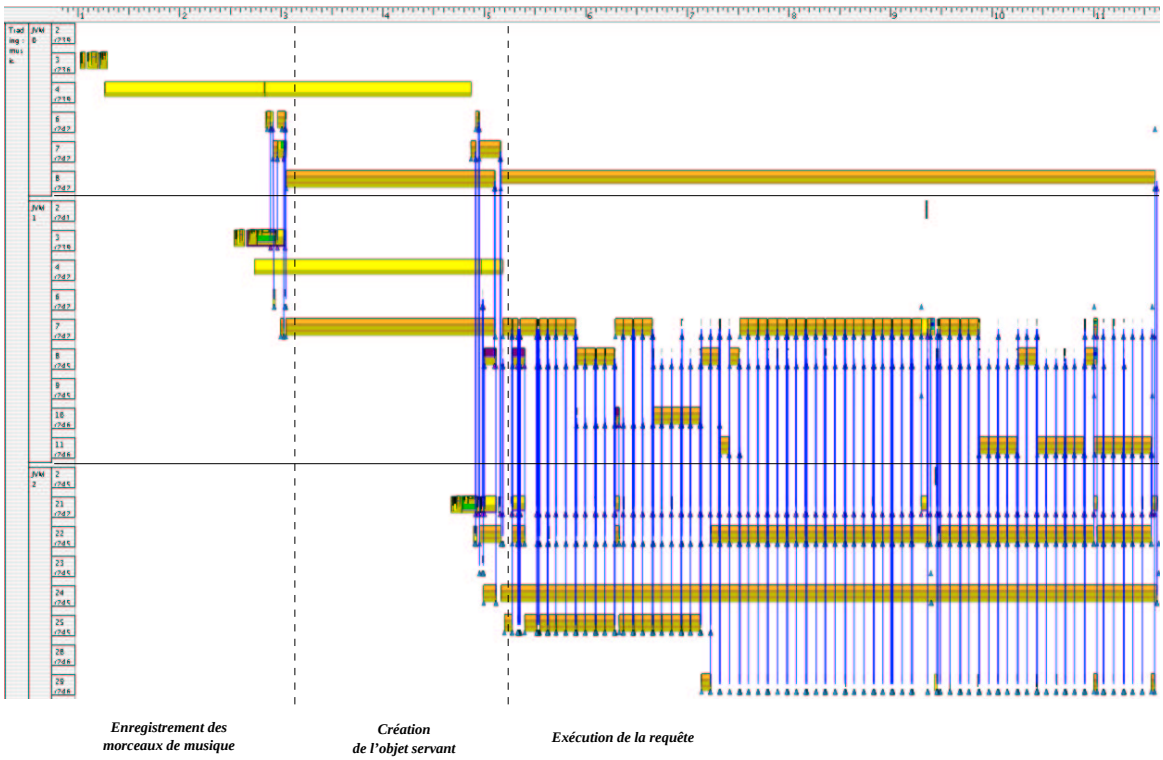


FIG. B.1 – Exemple d'accès à une ressource «Musique».

C'est la nature de la requête, de type «synchrone», qui différencie cet exemple de celui de la requête sur un livre (voir section 6.5.2). Dans ce cas, les interactions prennent une forme particu-

lière, puisqu'elles interviennent de façon régulière afin de remplir un tampon de données. Ce tampon représente une partie du morceau de musique qui est déplacée du serveur vers le client (voir figure B.2).

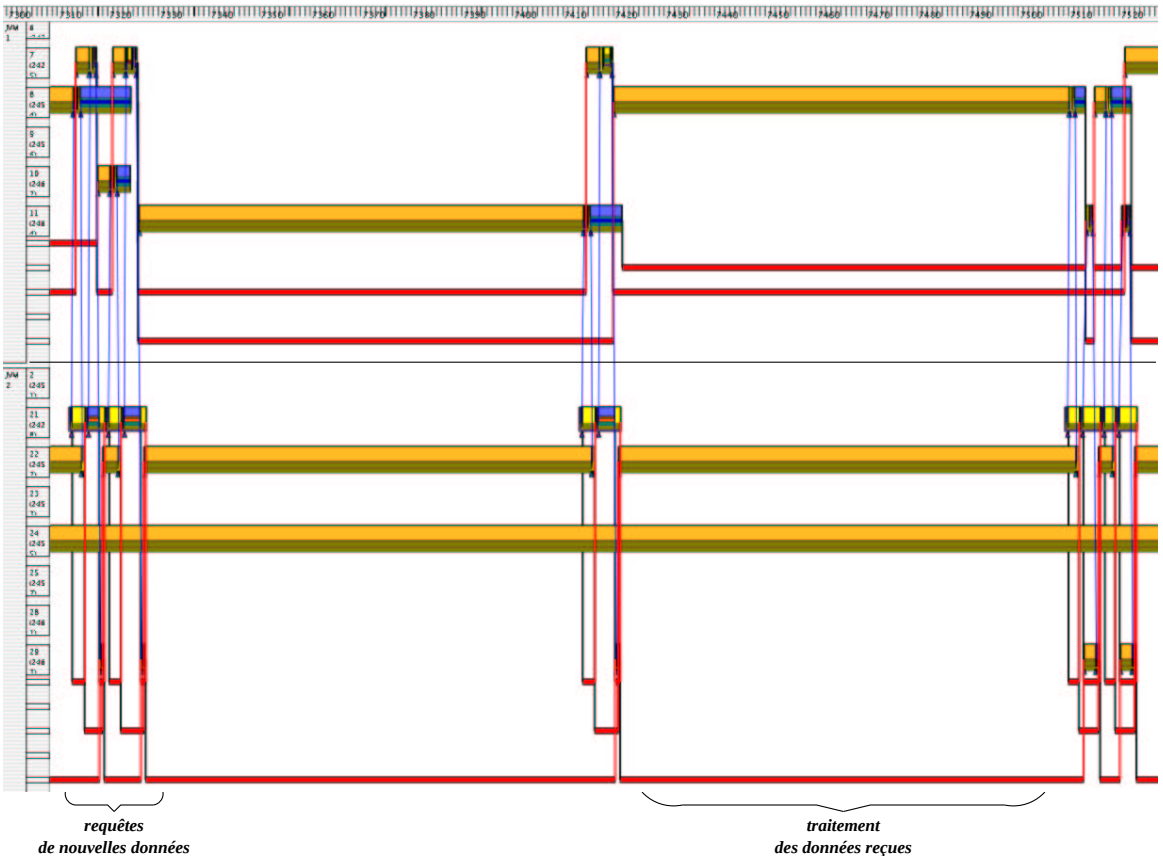


FIG. B.2 – Interactions entre le client et le serveur de musique.

Ces interactions doivent avoir lieu de façon régulière et en temps borné pour que le tampon ne soit jamais vide (voir figure B.3). Dans le cas contraire, le morceau de musique sera retranscrit d'une manière saccadée, ce qui peut être considéré comme un problème de performance. Les figures B.2 et B.3 représentent les objets de synchronisation utilisés dans la régulation du flux de données.

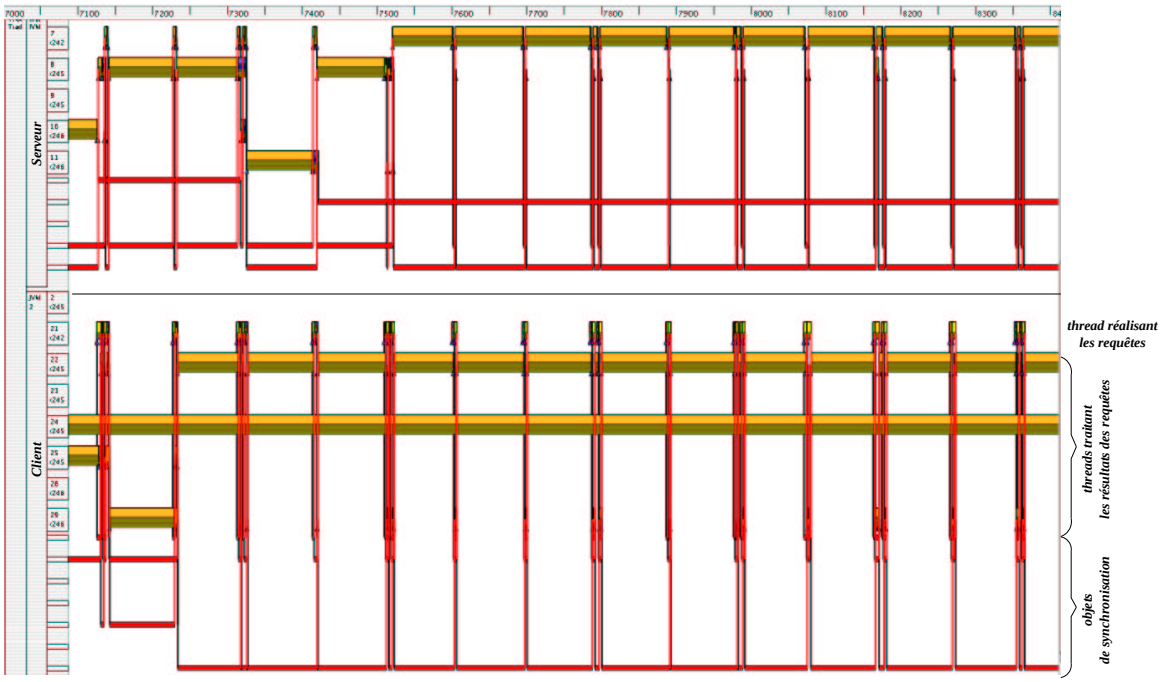


FIG. B.3 – Régulation du flux de données.

Pour ce qui est de l'analyse statistique des appels de méthode, elle est très proche de celle réalisée pour l'exemple du serveur de livres (voir section 6.5.2.4). A titre de comparaison, nous détaillons les résultats obtenus pour la trace du client.

Elle comprend 267 différents appels de méthode pour un total de 8995 appels. L'analyse en fréquence du nombre d'appels nous indique que les méthodes sont appelées en moyenne moins de 34 fois (voir figure B.4). En ce qui concerne les durées d'exécution, les dix méthodes s'exécutant le plus longtemps correspondent à 82% de la durée d'exécution totale. Elles sont représentées par le tableau ci-après :

Nom de la classe	Méthode	Nbr. Appels	Durée (s)
TcpIpProtocol.Session	run	59	2.833846
GIOPProtocol.ClientSession_Low	send	59	2.825851
PortableMarshallerFactory.PortableUnMarshaller	readByte	472	2.781494
CDRMarshallerFactory.CDRUnMarshallerD	prepare	115	2.778529
TcpIpChunkProvider	prepare	115	2.775461
JConnectionMgr.Connection	receive	115	2.764539
IPv4ConnectionFactory.Connection	receive	115	2.763089
JStubFactory.ClientDelegate	invoke	59	0.390000
JStubFactory.ClientDelegate	invoke	59	0.231851
GIOPProtocol.GIOPSession_High	send	59	0.218557

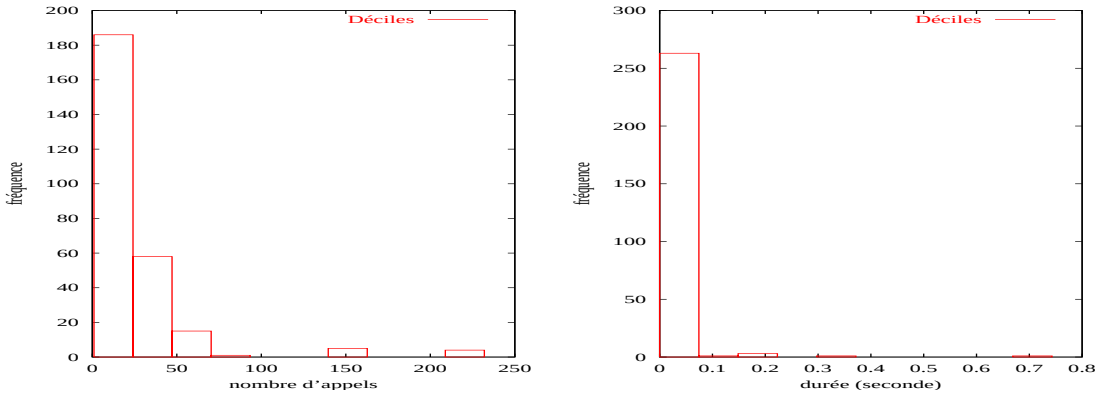


FIG. B.4 – Déciles du nombre d'appels et de la durée d'exécution des méthodes exécutées par le client.

Hormis ces dix méthodes, la durée moyenne d'exécution des 257 autres méthodes est de $\simeq 17ms$. Si l'on considère 90% de l'échantillon total (trié par ordre croissant), la durée moyenne d'exécution descend à $\simeq 7ms$.

Ces résultats sont similaires à ceux obtenus pour l'analyse de la requête sur le serveur de livres. Il en est de même pour la trace du courtier et du serveur de musique, ce qui prouve le caractère générique de notre approche.

Annexe C

Trace “Pajé”

L’en-tête de la trace “Pajé” est propre à un modèle d’exécution. Il se décompose en deux parties. La première partie contient les termes du langage de commande générique avec leur sémantique et leur syntaxe. Elle définit la richesse du langage. La deuxième partie correspond à la définition de la hiérarchie d’objets visuels associée à la trace. Cette hiérarchie exprime la sémantique attachée aux observations. Elle représente donc l’expressivité du langage.

La suite de l’annexe analyse la représentation de la hiérarchie de types construite pour représenter une exécution Java répartie. L’emploi de chacun des termes est commenté afin d’exprimer sa fonction. Chaque commentaire est composé d’un préfixe, décrivant à quel type d’objet le terme se rapporte (par exemple `Conteneur`), et un suffixe qui exprime la sémantique de ce dernier.

Simulation : définition de la durée de l’exécution

id	date	date de début	date de fin
0	0.000000	0.000000	5.0

Conteneur : définition des types

id	date	nouveau type	conteneur	nom
1	0.000001	EXEC	0	"Exec"
1	0.000001	JVM	EXEC	"JVM"
1	0.000001	TH	JVM	"Thread"
1	0.000001	GC	JVM	"GC"
1	0.000001	MONITOR_WAIT	JVM	"Monitor wait"
1	0.000001	MONITOR_CONTENDED	JVM	"Monitor Contended"
1	0.000001	RAW_MONITOR_CONTENDED	JVM	"Raw Monitor Contended"

Entité événement : définition d’un type

id	date	nouveau type	conteneur	nom
2	0.000002	SOCKET	TH	"Socket"

Entité état : définition des types

id	date	nouveau type	conteneur	nom
3	0.000003	MTH	TH	"Method"
3	0.000003	ACT	GC	"GC action"
3	0.000003	RMC_ACT	RAW_MONITOR_CONTENDED	"RMC : "
3	0.000003	MC_ACT	MONITOR_CONTENDED	"MC : "
3	0.000003	W_ACT	MONITOR_WAIT	"W : "

Entités valeur : définition des types (indices de consommation des ressources d’exécution)

id	date	nouveau type	conteneur	nom
4	0.000004	JVM_CPU_U	JVM	"CPUu"
4	0.000004	JVM_CPU_S	JVM	"CPUs"
4	0.000004	JVM_RSS	JVM	"RSS"
4	0.000004	JVM_TOTAL_VM	JVM	"VM"
4	0.000004	CPU_U	TH	"thCPUu"
4	0.000004	CPU_S	TH	"thCPUs"
4	0.000004	RSS	TH	"thRSS"
4	0.000004	TOTAL_VM	TH	"thVM"

Entité lien : définition des types

id	date	nouveau type	cont.	cont. source	cont. dest.	nom
5	0.000005	COMM	EXEC	TH	TH	"Comm"
5	0.000005	NOTIFY_TM_W	JVM	TH	MONITOR_WAIT	"TM W"
5	0.000005	NOTIFY_MT_W	JVM	MONITOR_WAIT	TH	"MT W"
5	0.000005	NOTIFY_TM_MC	JVM	TH	MONITOR_CONTENDED	"TM MC"
5	0.000005	NOTIFY_MT_MC	JVM	MONITOR_CONTENDED	TH	"MT MC"
5	0.000005	NOTIFY_TM_RMC	JVM	TH	RAW_MONITOR_CONTENDED	"TM RMC"
5	0.000005	NOTIFY_MT_RMC	JVM	RAW_MONITOR_CONTENDED	TH	"MT RMC"

Entité événement : définition des valeurs accessibles par les entités états

id	date	nouvelle valeur	entité	nom
6	0.000006	COMM_0	COMM	"inter JVM"
6	0.000006	SOCKET_READ	SOCKET	"receive message"
6	0.000006	SOCKET_WRITE	SOCKET	"send message"
6	0.000006	GC_P ACT	"processing"	
6	0.000006	RMC_ENTER	RMC_ACT	"enter RMC"
6	0.000006	RMC_ENTERED	RMC_ACT	"entered RMC"
6	0.000006	MC_ENTER	MC_ACT	"enter MC"
6	0.000006	MC_ENTERED	MC_ACT	"entered MC"
6	0.000006	MW_ENTER	W_ACT	"enter W"

Entité événement : définition des valeurs accessibles par les entités états (suite)

id	date	nouvelle valeur	entité	nom
6	0.000006	NOTIFY_TM_W_ENTER	NOTIFY_TM_W	"enter W"
6	0.000006	NOTIFY_MT_W_ENTERED	NOTIFY_MT_W	"entered W"
6	0.000006	NOTIFY_TM_MC_ENTER	NOTIFY_TM_MC	"enter MC"
6	0.000006	NOTIFY_MT_MC_ENTERED	NOTIFY_MT_MC	"entered MC"
6	0.000006	NOTIFY_TM_MC_EXIT	NOTIFY_TM_MC	"exit MC"
6	0.000006	NOTIFY_TM_RMC_ENTER	NOTIFY_TM_RMC	"enter RMC"
6	0.000006	NOTIFY_MT_RMC_ENTERED	NOTIFY_MT_RMC	"entered RMC"
6	0.000006	NOTIFY_TM_RMC_EXIT	NOTIFY_TM_RMC	"exitRMC"

Conteneurs : création des instances

id	date	nouveau cont.	type cont.	conteneur	commentaire
7	0.000007	CUR	EXEC	0	"Trading : book"
7	0.000007	JVM_0	JVM	CUR	"JVM 0"
7	0.000007	GC_0	GC	JVM_0	"GC 0"
7	0.000007	JVM_1	JVM	CUR	"JVM 1"
7	0.000007	GC_1	GC	JVM_1	"GC 1"
7	0.000007	JVM_2	JVM	CUR	"JVM 2"
7	0.000007	GC_2	GC	JVM_2	"GC 2"

RÉSUMÉ

Les moyens de traitement de l'information auxquels nous avons accès sont de plus en plus puissants, de plus en plus répartis. Des modèles de programmation, tels que la multiprogrammation légère ou la programmation par objets, leur sont appliqués afin de juguler l'accroissement de la complexité qui en découle. Mais qu'apportent ces modèles ? La mise en oeuvre de ces modèles permet elle d'employer au «mieux» les ressources disponibles ?

Pour répondre à cette question, nous nous plaçons dans une démarche de type «évaluation de performances». La problématique porte sur la reconstruction post-mortem, à partir de mesures, de la dynamique d'une exécution afin de réaliser une analyse qualitative et quantitative des ressources d'exécution consommées. L'hypothèse posée est que l'analyse des interactions entre objets, effectuée à différents niveaux d'abstraction, procure les informations suffisantes à nos études. Pour cela, nous réalisons des observations au niveau applicatif et système et cela sans modification du code exécutable original de l'application ou du noyau du système d'exploitation.

Une infrastructure d'observation multi-niveaux a été réalisée dans le cadre d'applications Java réparties. Elle a été appliquée à l'étude d'un serveur multimédia Java. Une analyse de l'algorithmique et des consommations de ressources systèmes a été menée. Pour cela, les observations du niveau applicatif sont effectuées au travers de la JVMPI. Le coût moyen d'observation est de $3\mu s$ par point de mesure. Celles du niveau système sont obtenues par insertion à chaud d'un module dans un noyau Linux. De cette façon, les appels systèmes attachés aux écritures et aux lectures dans une "*socket*" sont déroutés. Il nous est ainsi possible reconstruire les interactions distantes entre objets Java et d'obtenir des indices de consommation des ressources systèmes. Le coût moyen d'observation est ici de $2\mu s$ par point de mesure.

MOTS CLÉS : mesure et analyse de performances, observations multi-niveau, analyse post-mortem, dynamique d'exécution, systèmes répartis, applications Java réparties.

ABSTRACT

Nowadays computing systems are increasingly powerful and distributed. Programming methods, such as multithreading or object oriented programming, are applied to limit the ever-growing complexity of these systems. What are the benefits of using such methods ? Are they able to best leverage available resources ?

To answer that question, we studied the execution via a «debugging performance» method. The problematic concern lies in the post-mortem reconstruction, from measures, of the dynamic behavior of an execution. The goal is to achieve a qualitative and quantitative analysis of the consumed execution resources. The hypothesis is that the analysis of the interactions between objects, via the multi-layer observations, provides enough information to reach our goal. This is the reason why, we obtain information from the application and system layers of abstraction, without modifying the original code of the application or the operation system kernel.

We created a multi-layer observation infrastructure for distributed Java applications. This have been used to study a multimedia Java server. We proceeded to a algorithmic and a system resource consumption. The observations from the application layer are obtained from the JVMPI. The average cost of such observations is 3 microseconds for each measure. Observations from the system layer perspective are obtained by dynamically loading a module in a Linux kernel. Therefore, system calls used to write and read from a socket are re-routed to include our own code for observation. We are then able to reconstruct the remote interactions between Java objects and to obtain indexes on system resource consumption. The average cost of such observations is 2 microseconds for each measure.

KEYWORDS : measures and performance analysis, multi-layer observations, post-mortem analysis, execution patterns and behaviors, distributed systems, distributed Java applications.