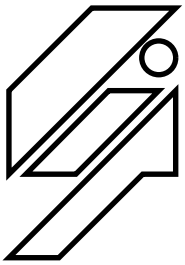
Protocole de communication utilisé dans PIMSy

Xavier-François Vigouroux



Laboratoire de l'Informatique du Parallélisme

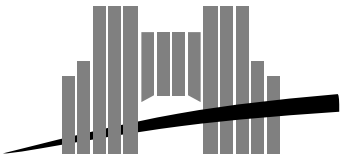
Ecole Normale Supérieure de Lyon
Unité de recherche associée au CNRS n°1398

Protocole de communication utilisé dans PIMSy

Xavier-François Vigouroux

21 octobre 1994

Technical Report N^{O} 94.03



Ecole Normale Supérieure de Lyon
46 Allée d'Italie, 69364 Lyon Cedex 07, France
Téléphone : (+33) 72.72.80.00 Télécopieur : (+33) 72.72.80.80
Adresse électronique : lip@lip.ens−lyon.fr

Protocole de communication utilisé dans PIMSy

Xavier-François Vigouroux

21 octobre 1994

Abstract

PIMSy (stands for Parallel Implementation of a Monitoring System) is a tool to analyze distributed trace files. For all that, the generating phase, instead of generating one trace file while the program is executed, uses the different disks (of the parallel machine) to store different parts of a trace files. This way, we can read the trace files concurrently and avoid the bottleneck of the massive parallel monitoring.

This report is dedicated to the people that will work on the PIMSy project. It consists in describing the protocol used in this tool : the messages name, the order in which they have to be exchanged ...

Keywords: Monitoring, protocol

Résumé

PIMSy (acronyme de Parallel Implementation of a Monitoring System) est un outil d'analyse de fichiers de traces distribués. Pour cela, la phase de génération, au lieu de générer un unique fichier de traces pendant l'exécution d'un programme, utilise les différents disques (de la machine parallèle) pour stocker les différentes parties des fichiers de traces. Ainsi, nous pouvons lire les fichiers de trace d'une manière concurrente et éviter le goulot d'étranglement du monitoring massivement parallèle.

Ce rapport est dédié aux personnes devant travailler sur le projet PIMSy. Il consiste en la description du protocole utilisé par cet outil : le nom des messages, l'ordre dans lequel ils doivent être échangés, ...

Mots-clés: Monitoring, protocole

Table des matières

Table des figures

Chapitre 1

Introduction et Motivations de PIMSy

1.1 Introduction

La recherche de la puissance des ordinateurs parallèles passe par 3 paramètres : la puissance de chaque processeur, le nombre de ces processeurs, et enfin l'utilisation de ces processeurs. En effet si l'un de ces paramètres est multiplié par un facteur k, le temps d'une exécution sera divisé par un facteur k (ou presque, car le réseau de communication entre en compte dans le temps global, même lors de recouvrement calcul/communication).

Alors que les deux premiers paramètres sont à la charge des constructeurs de machines, le troisième concerne la partie logiciel de l'exécution. Pour savoir si l'utilisation de la machine parallèle est correct, le programmeur est dans l'obligation d'analyser son programme. Il peut le faire à la main, utiliser des « profiler », ou utiliser le « monitoring ».

1.2 Le Monitoring

Le monitoring consiste à stocker dans un « fichier de traces » les informations concernant les événements ayant lieu dans la machine parallèle lors d'une exécution. Un modèle décrivant le monitoring est dit « à trois phases » :

- **la génération** peut être faite de trois manières :
  - d'une manière logicielle en insérant dans le source des parties gérant la génération et le stockage des événements,
  - d'une manière matérielle en plaçant des capteurs dans la machine utilisée,
  - d'une manière hybride en utilisant les deux méthodes.

  La génération doit permettre le stockage des informations mais aussi ouvrir sur la phase de transport de cette information. En effet, c'est lors de la génération des événements que l'on décide de vider ou non les tampons. Bien sûr, le déclenchement du déplacement des événements vers une zone d'analyse ne concerne que la génération du type logicielle ;
- **le transport** a pour but de faire le lien entre la phase précédente et l'analyse de l'information générée. Son occurence, la périodicité de son occurence ainsi que son instant sont déterminant pour la non-perturbation du programme observé. En effet l'information stockée utilise la mémoire des processeurs et donc risque d'affecter les allocations faites par le programme

observé. D'un autre côte, le déchargement des buffers se fait par l'intermédiaire du réseau de communication déjà utilisé par le programme. Donc l'utilisation du réseau risque de différer des communications. Il faut par conséquent doser finement la taille à partir de laquelle on vide la mémoire les processeurs : si on les vide tous en même temps, si on attend que le réseau soit peu occupé (on peut le faire en examinant les tampons de trace en cours de remplissage), etc. Il est à noter que, de toute façon, le fait d'espionner une exécution va créer des modifications dans le comportement de ce même programme ;

- **l'analyse** permet de donner à l'utilisateur les informations générée mais sous différentes formes. Par exemple, lors de cette phase, l'utilisateur peut visualiser les communications ou voir l'état d'un des processus.

Les événements que l'on observe sont généralement de deux types : internes et externes :

- les événements internes concernent un seul processus : celui qui les génère. Un exemple est l'affectation d'une variable (`x:= 3`). Cet événement, en aucun cas, ne concernera un autre processus (hormis dans le cas de mémoire partagée). Ce type d'événement est intéressant pour suivre le déroulement d'un processus : valeur d'une condition, numéro d'itération d'une boucle, etc. Cela permet notamment de mettre en évidence les différentes phases d'un algorithme, et ainsi, extraire les désynchronisations entre processus,

- les événements externes qui, dans le modèle du passage de messages, concernent les communications. Un événement externe est, ainsi, divisible en deux parties : la partie ayant lieu dans le processus source et la seconde dans les processus destination.

Dans le cas de l'augmentation du nombre de processeurs, le nombre d'événements générés va être affecté: si on double le nombre de processeurs d'une machine parallèle, le nombre d'événements internes va au moins doubler (le nombre de processus s'exécutant étant double) et le nombre d'événements externes risque de suivre les mêmes variations (le nombre de destinataires étant double aussi). Quoi qu'il en soit, le passage au parallélisme massif pose un problème important au niveau de la gestion de cette quantité énorme d'information.

De plus, l'augmentation de la quantité d'information, pose aussi des problèmes lors de l'exécution elle-même car la place mémoire occupée devient non négligeable.

PIMSy [PTV93] est une solution pour continuer à pouvoir utiliser le monitoring dans les machines massivement parallèles. L'idée est d'utiliser cette même machine parallèle pour l'exécution et l'analyse. Ainsi, la puissance de l'analyse suivra la même évolution que la puissance de l'exécution. De plus, en utilisant les différents lieux de stockage d'une machine parallèle, on supprime la phase de transport.

L'idée est de conserver le parallélisme inhérent aux fichiers de traces en partitionnant la machine parallèle suivant ses capacités de stockage. Ainsi, au lieu de lire un seul fichier de trace, on en lit plusieurs et de manière concurrente, chaque processus lecteur connaissant une partie de l'information.

Chapitre 2

Description Simplifiée de PIMSy

2.1 Composants

Les ensembles sont notés en majuscule et les éléments en minuscule.

$\mathcal{C}$ représente l'ensemble des processus de visualisation respectant le protocole de communication.

$\mathcal{TS}$ représente l'ensemble des « Trace Server ». Ceux-ci communiquent entre eux pour répondre aux requêtes des clients. Un client est géré par un unique $ts \in \mathcal{TS}$, et un ts peut gérer plusieurs clients.

TS Front (tsf) : Ce processus est l'interface entre l'utilisateur et PIMSy, il permet de créer des clients ou des $\mathcal{TS}$, etc.

La figure 2.1 montre les liens de communication entre les différents éléments de chacun de ces ensembles.

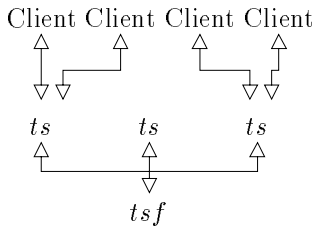


FIG. 2.1 - *Architecture du logiciel*

Dans la suite, des éléments appartenant à ces ensembles vont avoir des rôles particuliers. Pour un souci de simplicité nous les caractérisons ici :

Client Source ($cs \in \mathcal{C}$) : C'est un client qui émet une requête. Pour simplicité on ne considère qu'une seule requête à la fois.

Clients Destination ($\mathcal{CD} \subset \mathcal{C}$) : Clients qui recevront le résultat d'une requête fait par le cs.

TS Source ($tss \in \mathcal{TS}$) : c'est le ts qui gère le cs.

2.1.1 Processus serveur de traces

Le serveur de traces est le processus central de PIMSy. Il gère les fichiers de traces et les clients qui veulent les lire. Cette couche, non seulement, met en relation les deux parties, mais en plus filtre les informations pour minimiser le travail fait par les clients. En effet, lorsqu'un client demande de l'information sur un fichier de trace, il ne demande pas la totalité mais seulement une partie (un intervalle de temps, un sous-ensemble des processeurs, ...) ; or cette opération de filtrage est exécutée par les serveurs de traces qui, ainsi, décharge les clients du travail. Ce filtrage est bien sûr distribué sur l'ensemble $\mathcal{TS}$ qui ensuite fusionnent l'information filtrée, avant de l'envoyer au client qui la demande.

2.1.2 Processus TSF

Ce processus est le frontal de PIMSy, c'est lui qui permet d'ajouter des serveurs de traces, de les détruire. C'est aussi lui qui lors de la création d'un client indique qui va le gérer pour minimiser la charge. C'est l'interface homme-machine du système.

2.2 Répertoires et fichiers

La structure des fichiers est en concordance avec celle de PIMSy : certains fichiers sont distribués et d'autres sont communs. La description qui suit est celle qui est implantée actuellement, elle permet de simuler plusieurs disques. Bien sûr, ce n'est pas la version définitive.

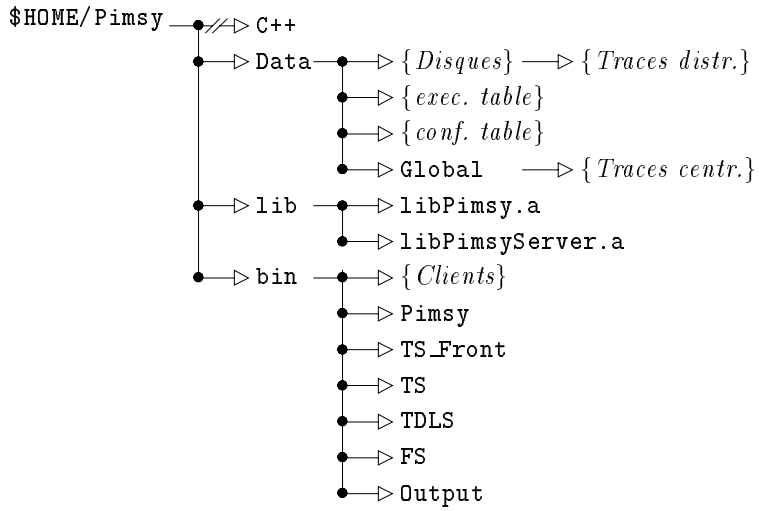


FIG. 2.2 - *Arborescence utilisée actuellement pour PIMSy*

Les fichiers concernant PIMSy sont tous situés à partir du répertoire `$HOME/Pimsy`. L'arborescence est celle décrite dans la figure 2.2. Le premier niveau est divisé en quatre répertoires :

`C++` ce répertoire est un lien symbolique vers les fichiers sources pour permettre la compilation de nouveaux clients. En effet, les clients doivent utiliser les mêmes définitions que les serveurs,

`Data` ce répertoire est celui dans lequel les differents *ts* vont lire pour répondre aux requêtes des clients. Il contient des fichiers de configuration (foir figure 2.1) qui permetent de savoir où un nœud a écrit lors de son exécution pour que les *ts* lisent les traces lui correspondant,

`lib` dans ce répertoire se trouvent les bibliothèques permettant de construire de nouveaux clients en ayant un accès simplifié aux fichiers de trace et à leur manipulation,

`bin` contient tous les exécutables faisant partie du « package » PIMSy, que ce soit les *ts*, les clients, ou la fenêtre de sortie `output`.

#Disques			$= \mid \mathcal{TS} \mid$
Disque_0	#nœuds	liste des nœuds	
Disque_1	#nœuds	liste des nœuds	
...	...	...	
$\text{Disque}_{\#Disque}$	#nœuds	liste des nœuds	

TAB. 2.1 - *Description d'un fichier de configuration (`.map`)*

2.3 Variables d'environnement de PIMSy

La seule variable d'environnement utilisée par PIMSy est, pour l'instant : `PIMSY_MASTER_HOST` qui permet de savoir par défaut devant quelle machine se trouve l'utilisateur. Par exemple, l'affichage des clients se fera sur l'écran de cette machine. Bien sûr cet écran peut changer durant une execution pour permettre d'utiliser PIMSy sur plusieurs écrans.

2.4 Construction de PIMSy

Remarque : la liste de tous les messages se trouvent à la fin de ce document. Toutes les abbréviations y sont expliquées.

2.4.1 Ajout d'un *ts*

La structure de base de l'ensemble $\mathcal{TS}$ est construite par ajout successif de chacun des *ts* par le *tsf*.

Lorsque l'on ajoute un nouveau ts_i à $\mathcal{TS}$, le protocole est celui représenté par la figure 2.3 dont voici les étapes :

1. le *tsf* crée la nouvelle tache (ts_i),
2. le *tsf* envoie un [MIS] – « Message Init Server » – à ts_i les différents éléments de la configuration (écran, répertoire contenant les fichier de traces, ...) de PIMSy,
3. le *tsf* envoie un [MCE] – « Message Change Environment » – à ts_i la liste de tous les serveurs de traces de la configuration (ts_i n'y étant pas encore),
4. le *tsf* ajoute ts_i à cette même configuration

5. le tsf envoie un [MCE] à tous les ts l'apparition de ce nouvel arrivant. En effet, dans la suite, les ts ont besoin de la liste des clients existants pour permettre la gestion des requête.

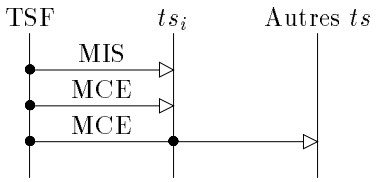


FIG. 2.3 - *Protocole suivi lors de l'ajout d'un nouveau serveur de traces*

2.4.2 Ajout d'un client

L'idée centrale du protocole est de créer le client par le tsf et d'en donner la gestion à un ts le plus vite possible. Comme cela, le choix du ts est centralisé et permet de gérer la charge.

Dans la suite on considère que l'on crée un nouveau client c_j dont la gestion sera attribué à ts_i. Voici le détail de la figure 2.4 :

1. le tsf crée la nouvelle tache (c_j),
2. le tsf envoie un [MAC] – « Message Add Client » – à ts_i l'identificateur de c_j,
3. le ts_i envoie un [MIC] – « Message Init Client » – au c_j pour l'initialiser et indiquer que c_j communique avec lui.

Remarque : la gestion de la charge est pour l'instant très simple, c'est-à-dire, que le tsf compte le nombre de clients de chacun des ts et attribue le nouveau client au moins chargé. La suppression d'un des client entraîne, bien sûr, un décompte sur le serveur de traces concerné.

Remarque : on voit que la gestion des clients n'est pas distribuée, c'est-à-dire, que chaque ts ne gère que la liste des clients qu'il a sous sa tutelle et non pas la totalité. En effet, seul ts_i reçoit un avis de la création du nouveau client. Il faut réfléchir sur ce fait pour savoir si c'est la meilleure solution. Nous verrons dans la suite quelles sont les conséquences d'un tel choix.

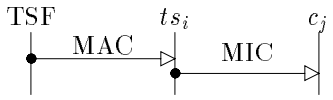


FIG. 2.4 - *Protocole suivi lors de l'ajout d'un nouveau client*

2.5 Configuration des clients

2.5.1 Association d'un filtre

Un client peut filtrer les événements par lesquels il est intéressé. Ce filtrage se fait suivant trois axes : le type de l'événement, le moment de son occurence, et le lieu de son occurence. Nous avons remarqué qu'un client reste intéressé par le même type d'événement tout le temps de son exécution, c'est-à-dire qu'un client visualisant les communications sera toujours intéressé par les communications. Partant de cette constatation, nous avons divisé les trois axes de filtrage en deux parties : la partie fixe (Quoi et Où), la partie variable (Quand). Un client peut donc définir la partie fixe de ce filtre dès son initialisation grâce à l'envoi d'un message de type [MDC] – « Message define Client » – vers son *ts*. Il n'y a pas de retour de message de la part du serveur de traces (voir figure 2.5). Les champs de ce message sont : le nombre et la liste des processus à conserver, le nombre et la liste des types d'événements à conserver.

Remarque : la gestion de la liste de processus à conserver doit être assez fine pour permettre la notion d'intervalle ou de propriété à vérifier par les processus (les numéros pairs, ...). Dans la version actuelle, cette gestion n'est pas faite, mais elle sera ajoutée au système.

FIG. 2.5 - *Protocole suivi pour la définition de la partie fixe du filtre d'un client*

2.5.2 Demande de la liste des exécutions

Après avoir définit un filtre associé, un client demande (généralement) la liste des fichiers de trace auxquels il a accès. Cette liste est simplement consitutée des fichiers de traces stockés sur les disques. Le protocole consiste à envoyer une demande [MAFEL] (« Message Ask For Execution List » – puis recevoir une réponse [MRFEL] – « Message reply for execution List » – voir figure 2.6).

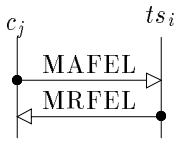


FIG. 2.6 - *Protocole suivi pour la demande de la liste des exécutions par un client*

Nous préférons choisir *liste des exécutions* à *exécutable* car un même *exécutable* peut avoir été exécuté plusieurs fois pour comparer les différents comportements. Une telle exécution est définie par la table 2.2 qui est appelée *table d'exécution*.

le nom de l'exécutable	
la date	
le nom du fichier	décrit la machine lors de l'exécution (fichier `.map`)
l'auteur	
le nombre de *ts*	correspondant avec le nombre de disques durs utilisés
le nombre de noeuds	ce qui permet de configurer les clients
le nombre d'événements	
le premier instant	estampillage du premier événement
le dernier instant	estampillage du dernier événement
la fréquence d'horloge	le nombre d'estampilles différents par seconde

TAB. 2.2 - *Description d'une exécution*

Remarque : la vérification pour savoir si le nombre de serveurs de traces est suffisant pour reconstituer la trace complète ne se fait pas à ce niveau mais bien plus tard, c'est-à-dire, lors de la demande d'information concernant un fichier de traces. La solution de n'envoyer que la liste des fichiers de traces valides n'est pas satisfaisante car le nombre de *ts* peut évoluer dans le temps, et par conséquent, cette liste aussi.

2.5.3 Demande d'information sur une exécution

Un utilisateur peut extraire de la liste des exécutions vue ci-dessus, le nom de celle par laquel il est intéressé. Cette demande ne consiste qu'à envoyer un message [MAFET] – « Message Ask for Execution Table » – afin de recevoir un message [MRFET] – « Message Reply For Execution Table » – contenant les information de la table 2.2. La figure 2.7 décrit la suite des messages.

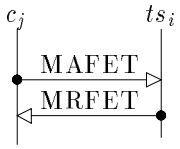


FIG. 2.7 - *Protocole suivi pour la demande de la table d'exécution par un client*

2.6 Destruction du système

2.6.1 Destruction d'un *ts*

Pour la destruction d'un serveur, tout passe par un unique type de message [MCS] – « Message Close Server » – dont le sens de transit détermine l'action à faire. Un accusé de réception peut être retourné s'il l'est demandé. Supposons que l'on détruise ts_i, alors les actions suivantes sont effectuées :

1. le *tsf* envoie un [MCS] à ts_i

2. le ts_i émet vers tous ces clients un message de terminaison [MCC] et attend le retour des réponses.

3. Si le tsf attend un retour alors, le ts_i envoie un nouveau [MCS] au tsf pour dire que la destruction est en cours.

4. le tsf supprime le ts_i des tables.

5. le tsf attend la prochaine phase de remise à jour pour envoyer aux autres ts les modifications de l'environnement. Cela permet d'éviter de surcharger le réseau [La version suivante enverra tout de suite les modifications d'environnement, car elles sont assez rares pour considérer que leur effet est négligeable].

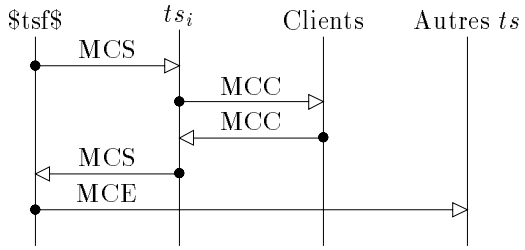


FIG. 2.8 - *Protocole suivi lors de la suppression d'un serveur de traces*

2.6.2 Destruction d'un client

L'idée est la même que pour la destruction d'un serveur de traces mais, cette fois-ci, les échanges se font entre le client et le ts qui le détruit (voir figure 2.9). De plus, le client peut être l'initateur de sa propre fermeture. Enfin, pour permettre la gestion de la charge par le tsf, un message [MCC] – « Message Close Client » – est envoyé en direction de celui-ci.

FIG. 2.9 - *Protocole suivi lors de la suppression d'un client*

Chapitre 3

Gestion des requêtes

3.1 Qu'est-ce qu'une requête ?

Le message servant à diffuser une requête est constitué de trois parties :

- **le message** permet d'encapsuler la requête pour qu'elle puisse être envoyée à travers le réseau. Cette encapsulation permet donc de faire des « send », « receive », « pack », ...
- **la requête** constitue la partie variable du filtre à effectuer pour obtenir un morceau de trace. Cette partie est décrite par la table 3.1,
- **la publication** qui permet de spécifier la liste des destinataires à la requête (i. e. l'ensemble $\mathcal{CD}$).

l'instant de départ	
l'instant final	
le nom de l'exécution	
le nom de l'expéditeur	
le numéro de la requête	ce qui permet d'en gérer plusieurs à la fois pour faire des comparaisons, ou être un client centralisateur
un instant précis	permet une synchronisation fine avec les autres clients

TAB. 3.1 - *Description d'une requête*

3.2 Requête simple ($\mathcal{CD} = \{cs\}$)

Pour le client, nous avons essayer de minimiser le nombre de messages qu'il voyait concernant la gestion de sa requête. Par conséquent, au cas où il ne demande de l'information que pour lui, il ne verra comme échange que l'envoi de la requête et la réponse à celle-ci. Bien sûr, entre les différentes parties du système sont échangés un grand nombre de messages. Voici la description générale de la figure 3.1 :

1. *cs* envoie grâce à un [MAFI] – « Message Ask For Info » – sa requête au serveur de traces qui le gère,

2. *tss* fusionne le filtre de la requête avec la partie fixe qu'il a reçu préalablement ([MDC]),

3. *tss* envoie [TS_MAFI] – « Trace Server Message Ask For Info » – la requête rehaussée de différentes informations aux autres serveurs de traces.

4. tous les serveurs de traces échangent alors des messages de type [TS_MTIE] – « Trace Server Message Trace Info Exchange » –pour obtenir l'information globale. L'algorithme suivi est un « odd even » afin que chacun des *ts* obtienne une partie de l'information qu'il ne reste qu'à fusionner pour obtenir l'information finale.

5. pour finir, une fusion est réalisée par *tss* lors des réceptions successives de [TS_MRFI] – « Trace Server Message Reply For Info » .

6. *cs* reçoit un [MRFI] – « Message Reply For Info » – résultat de cette fusion.

On voit que dans ce protocole, seuls le premier et le dernier point concernent le client ; tout le reste demeure interne au système.

Remarque : Dans la version actuelle, ce protocole, qui représente la partie centrale du traitement, est simplifié. Dans les versions future l'algorithme de « odd even » sera implanté.

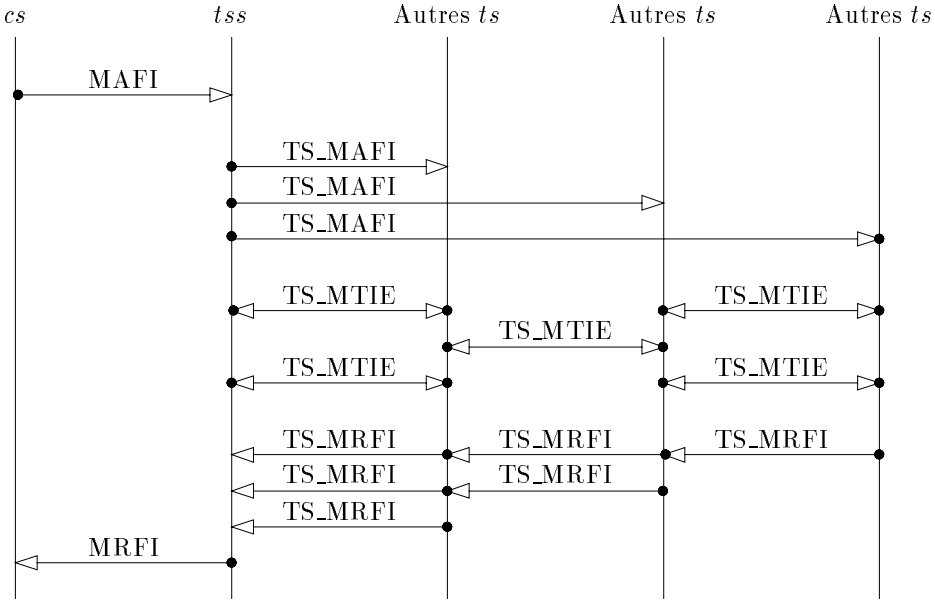


FIG. 3.1 - *Protocole suivi lors de la gestion d'un requête simple*

3.3 Requête à destinataires multiples ($\mathcal{CD} \neq \{cs\}$)

3.3.1 Introduction

Les requêtes à destinataires multiples sont le moyen que nous avons trouvé pour synchroniser différents clients au-dessus de PIMSy. En effet, pour que plusieurs vues travaillent sur la même

partie d'un fichier de trace, un client unique demande l'information à diffuser et chacun des clients à synchroniser sont alors positionner sur la même partie du même fichier de trace. Pour ensuite limiter la vitesse des clients à celui qui est le plus lent, on ajoute au protocole un message de fin de travail ([MEOW] – « Message End Of Work »).

3.3.2 Description

Dans le cas de requête à destinataire multiple, le traitement est plus compliqué que le cas présenté en 3.2. Toute la partie décrite alors reste valide mais une gestion est ajoutée pour manipuler les destinataires multiples.

Grâce à la partie fixe des filtres de chacun des clients ([MDC]), les destinataires recevront pour une même requête des résultats différents. Pour décrire l'ensemble $\mathcal{CD}$, nous avons cherché la simplicité plutôt que l'efficacité. En effet, les traitements faits pour construire chacune des réponses se font les uns à la suite des autres sans ce soucier (pour l'instant) des optimisations possibles.

Une fois que chacun des clients de $\mathcal{CD}$ ont reçu leur réponse, il faut, pour synchroniser l'ensemble $\mathcal{CD}$, gérer un message de fin de travail [MEOW]. Le protocole suivi est simple (voir figure 3.2):

1. succession de traitements pour chacun des destinataires,
2. chacun des destinataires finit son travail avec la tranche de fichier de trace reçu,
3. chacun des destinataires envoie un [MEOW] à son gestionnaire, qui fusionne les réponses,
4. chacun des serveurs de traces envoie sa réponse [MEOW] au serveur de traces source (i. e. *tss*) qui fusionne,
5. une fin de travail généralisée est envoyée au *cs* qui peut alors envoyer une nouvelle requête.

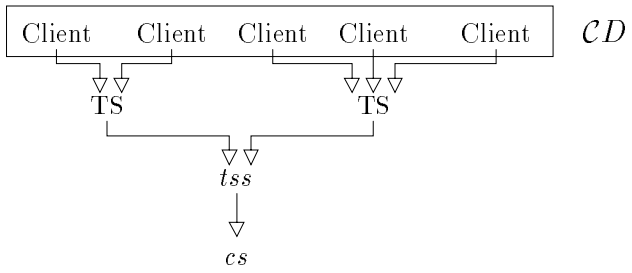


FIG. 3.2 - *Protocole suivi pour la fin du traitement d'une requête*

Chapitre 4

Description complète du protocole

En réalité, la présentation du chapitre 2 n'est pas complète. Nous avons volontairement réduit le nombre de processus différents. Dans ce chapitre, nous allons justifier l'existence d'autres types de processus, les décrire et énumérer les nouvelles parties du protocole.

4.1 Processus Output

Ce processus sert à afficher des messages de texte venant de n'importe quel processus : clients, *ts* ou *tsf*. La gestion est simple (voir figure 4.1) :

1. initialisation du processus par le *tsf* de la même façon qu'un client usuel [MIC],
2. tout processus peut lui envoyer un message à afficher [MT] – « Message Text » ,
3. sortie du processus lors de la réception d'un message de type [MCC].

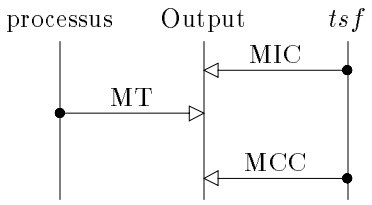


FIG. 4.1 - *cycle de vie du processus output*

4.2 Division d'un *ts*

Lorsqu'on analyse les taches que doit réaliser un serveur de traces, on remarque que l'une concerne la gestion des clients et l'autres la gestion des fichiers de traces. Ces deux parties sont assez indépendantes et leur interface est fine. Nous avons donc spécialisé le processus de serveurs de traces en deux nouvelles tâches : la première est appelée *tdls* (pour « Trace Disk Less Server ») et la seconde *fs* (pour « File Server »).

4.3 Processus *tdls*

Le *tdls* constitue la partie du serveur de traces gérant les clients. Ceci implique par exemple qu'il ne possède pas d'informations sur les fichiers de traces (liste, fichier de configuration, description d'un fichier de trace, ...). Lorsqu'un client demande de l'information concernant l'un des fichiers, le *tdls* prolonge la demande jusqu'à un *fs* auquel il est attaché. Donc lors de la réception d'un [TS_MAFI], un tel processus ne fait rien. En cas de réception d'un [MAFI], il stocke cette demande, puis la distribue aux différents *ts* qui, eux, ont l'information. Pour savoir quels sont ces *ts*, ils utilisent une récupère une table decrivant l'exécution.

Remarque : Ces processus devaient servir de mémoire cache pour éviter des lectures trop fréquentes sur les disques locaux. Mais, pour l'instant, cette notion n'est pas encore implémentée.

4.3.1 Initialisation

l'initialisation d'un *tdls* est faite par le *tsf* qui lui envoie un message de type [MIDLS] – « Message Init DiskLess Server » . Ce message contient différents éléments de configuration ainsi que l'identificateur du *fs* auquel il doit s'attacher. La figure 4.2 représente les étapes de cette initialisation.

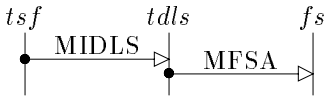


FIG. 4.2 - *Protocole suivi lors de l'initialisation du tdls*

4.3.2 Destruction

La destruction d'un *tdls* se fait soit directement du *tsf*, soit par l'intermédiaire des *fs*. Dans le premier cas, la réception d'un [MCS] entraîne le détachement du *tdls* (en utilisant un [MFSD] – « Message File Server Detach »). Dans le second cas, un message du type [MCFS] est envoyé au *fs* par le *tsf*. Ce message entraîne une demande de destruction des *tdls* auxquels il est attaché. La figure 4.4 montre ce second comportement.

4.4 Processus *fs*

Ce processus ne gère pas de client mais lit des informations sur les fichiers de trace comme le ferait un serveur de trace classique (voir 2.1.1).

4.4.1 Initialisation

L'initialisation d'un serveur de fichiers se fait de la même façon qu'un *ts* grâce à l'envoi d'un message contenant des données de configuration. Le message contenant cette information est un [MIFS] et le comportement est décrit par la figure 4.3.

FIG. 4.3 - *Protocole suivi pour l'initialisation d'un fs*

4.4.2 Destruction

La destruction d'un fs est cohérente avec les autres destructions, mis à part le détachement automatique du *tdls* attaché. La figure 4.4 montre ce comportement.

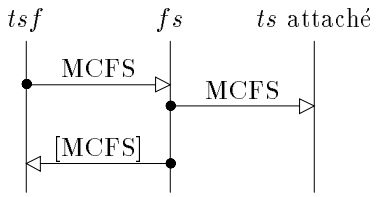


FIG. 4.4 - *Protocole suivi pour la destruction d'un fs*

4.5 Liaisons entre *tdls* et fs

Les liens sont simples : comme chacune des parties représente une fonctionnalité initialement dans les *ts*, les liens sont les mêmes que dans le protocole simple. Mais le fait qu'il s'agit d'un unique processus simplifie la gestion. Certaines informations globales aux *ts* sont, maintenant, à échanger entre les *tdls* et les fs. Pour mettre en place cette correspondance, on utilise le message [MFSA] qui transite du *tdls* vers le fs (voir figure 4.5). En fin d'exécution un message du type [MFSD] est émis dans le même sens pour mettre fin à la liaison.

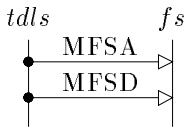


FIG. 4.5 - *Protocole suivi pour l'attachement et le détachement d'un fs et tdls*

4.5.1 Demande de la liste des exécutions par un client

De telles demandes ne font, en fait, que transitées par le *tdls* qui les envoie directement sur le fs avec lequel il est attaché. Ensuite le retour de la réponse se fait directement du fs au client. La figure 4.6 montre la demande de la liste des exécutions.

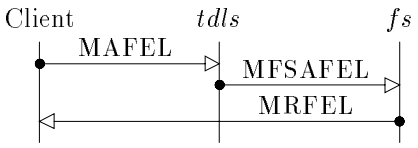


FIG. 4.6 - *Protocole suivi lors de la demande de la liste des exécutions avec passage par un tdls et fs*

4.5.2 Demande d'une table d'exécution par un client

Comme en 2.5.3, un client peut, une fois le [MRFEL] reçu, demander des informations concernant une exécution. La méthode est la même que dans la section précédente et est décrite figure 4.7.

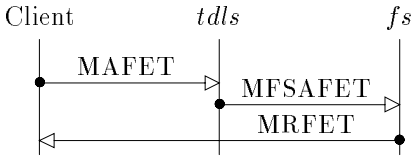


FIG. 4.7 - *Protocole suivi lors de la demande d'information sur une exécution avec passage par un tdls et fs*

4.5.3 Demande d'informations par un client sur un *tdls*

Comme expliqué plus haut, lorsqu'un client demande de l'information à un *tdls* via un [MAFI], ce dernier se comporte comme un *ts* en envoyant la demande aux autres *ts* (grâce à un [TS_MAFI]), en laissant ceux-ci échanger des [TS_MTIE], puis en attendant la réponse dans un message [TS_MRFI] (voir figure 4.8 où le *ts* devient *tdls*). Cette réponse est ensuite envoyée au client demandeur par un [MRFI].

Une différence est tout de même à noter : lors de la réception de la demande, le *tdls* doit vérifier que les *ts* sont suffisants pour pouvoir lire la totalité de l'information. Or, cette information est stockée sur un disque dur et un *tdls* n'a pas accès à un disque dur. Par conséquent, il doit envoyer un message au *fs* auquel il est attaché pour connaître cette table de configuration.

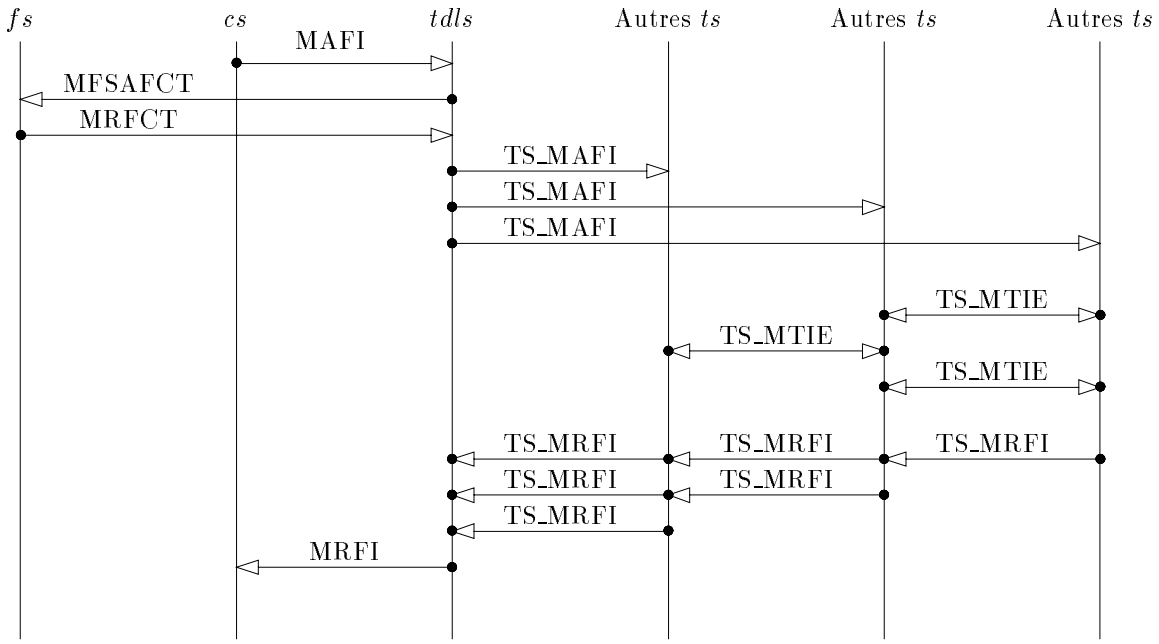


FIG. 4.8 - *Protocole suivi lors de la demande d'information avec passage par un tdls et fs*

Annexe A

Liste des Messages

MAC

signification : « Message Add Client »

description : permet au tsf d'indiquer au destinataire le nom d'un nouveau client qu'il aura à gérer.

source = $\{tsf\}$

destination = $\mathcal{TS} \cup \mathcal{TDLS}$

MAFEL

signification : « Message Ask For Execution List »

description : permet au client émetteur de demander la liste des exécutions dont il peut avoir le fichier de trace. La réponse est au format de [MRFEL].

Si la destination est un $tdls$ alors celui-ci ne sert que de relais vers le fs auquel il est attaché.

source = $\mathcal{C}$

destination = $\mathcal{TS} \cup \mathcal{TDLS}$

MAFET

signification : « Message Ask For Execution Table »

description : permet au client émetteur de demander des information concernant un fichier de trace. Pour connaître un fichier de trace valide, il utilise un message de type [MAFEL].

source = $\mathcal{C}$

destination = $\mathcal{TS} \cup \mathcal{TDLS}$

MAFI

signification : « Message Ask For Info »

description : représente la requête d'un client vers son gestionnaire. Il entraîne un gros traitement ([TS_MAFI], [TS_MTIE], [TS_MRFI]) puis le retour d'un [MRFI].

source $= \mathcal{C}$

destination $= \mathcal{TS} \cup \mathcal{TDLS}$

MCC

signification : « Message Close Client »

description : permet à *ts* source de demander la destruction d'un client. Un message de même est retourné si cela est demandé.

source $= \mathcal{C} \cup \mathcal{TS} \cup \mathcal{TDLS}$

destination $= \mathcal{C} \cup \mathcal{TS} \cup \mathcal{TDLS} \cup \{tsf\}$

MCE

signification : « Message Change Environment »

description : permet aux *ts* qui reçoivent ce message de mettre à jour la configuration de PIMSy, en nombre de serveurs de traces et en nombre de clients.

source $= \{tsf\}$

destination $= \mathcal{TS} \cup \mathcal{TDLS}$

MCS

signification : « Message Close Server »

description : est identique au [MCC] mais entre le tsf et un ts.

source $= \{tsf\} \cup \mathcal{TS} \cup \mathcal{TDLS}$

destination $= \{tsf\} \cup \mathcal{TS} \cup \mathcal{TDLS}$

MDC

signification : « Message Define Client »

description : permet à un client de définir la partie fixe du filtre à appliquer sur les réponses à ses requêtes : quoi et où.

source $= \mathcal{C}$

destination $= \mathcal{TS} \cup \mathcal{TDLS}$

MEOW

signification : « Message End Of Work »

description : permet d'indiquer qu'un destinataire d'un requête a fini son travail dessus. Ainsi, on peut attendre le plus long des clients destinataires. Une gestion par arbre est faite pour limiter le nombre de messages en transit : chaque serveur de traces de TSD fusionne les messages venant des clients qu'il gère ; lorsque le compte y est, il envoie ce même message au *tss* qui fusionne lui aussi avant de l'envoyer au *cs*.

source $= \mathcal{C} \cup \mathcal{TS} \cup \mathcal{TDLS}$

destination $= \{cs\} \cup \mathcal{TS} \cup \mathcal{TDLS}$

MFSA

signification : « Message File Server Attach »

description : ce message permet à un *tdls* de se lier avec un fs pour pouvoir avoir accès à des informations concernant le fichiers de trace.

source $= \mathcal{TDLS}$

destination $= \mathcal{FS}$

MFSAFEL

signification : « Message File Server Ask For Execution List »

description : ce message est émis par un *tdls* lorsqu'il reçoit d'un client une demande de type [MAFEL]. Ce message permet au *tdls* de demander au fs auquel il est attaché la liste des exécutions lisibles. La réponse [MRFEL] est renvoyée par le fs directement au client demandeur.

source $= \mathcal{TDLS}$

destination $= \mathcal{FS}$

MFSAFCT

signification : « Message File Server Ask For Configuration Table »

description : ce message est émis par un *tdls* pour vérifier si la configuration de PIMSy est suffisante pour lire un fichier trace demandé par un client.

source $= \mathcal{TDLS}$

destination $= \mathcal{FS}$

MFSAFET

signification : « Message File Server Ask For Execution Table »

description : ce message est émis par un *tdls* lorsqu'il reçoit d'un client un demande de type [MAFET]. Ce message permet au *tdls* de demander au fs auquel il est attaché de l'information concernant une exécution. La réponse [MRFET] est renvoyée par le fs directement au client demandeur.

source $= \mathcal{TDLS}$

destination $= \mathcal{FS}$

MFSD

signification : « Message File Server Dettach »

description : ce message permet à un *tdls* de se délier d'un fs en fin d'exécution

source $= \mathcal{TDLS}$

destination $= \mathcal{FS}$

MIC

signification : « Message Init Client »

description : permet à *ts* source l'initialiser la communication avec le client destinataire.

source $= \mathcal{TS} \cup \mathcal{TDLS}$

destination $= \mathcal{C}$

MIFS

signification : « Message Init File Server»

description : permet au tsf d'initialiser un serveur de fichiers qui vient juste d'être créé.

source $= \{tsf\}$

destination $= \mathcal{FS}$

MIS

signification : « Message Init Server »

description : permet au tsf d'indiquer au nouveau serveur de traces les différents éléments de la configuration (écran, répertoire contenant les fichier de traces, ...).

source = $\{tsf\}$

destination = $\mathcal{TS}$

MIDLS

signification : « Message Init Disk Less Server »

description : permet au tsf d'indiquer au nouveau $tdls$ le fs auquel il doit s'attacher lors de sa création.

source = $\{tsf\}$

destination = $\mathcal{TDLS}$

MRFCT

signification : « Message Reply For Configuration Table »

description : ce message est retourné par le fs vers un $tdls$ ayant préalablement émis un [MFSAFCT]. Ce message permet au $tdls$ de vérifier si PIMSy est configuré suffisamment pour lire toutes lies informations concernant une execution (i. e. savoir si à chaque disque est associé est lecteur).

source = $\mathcal{FS}$

destination = $\mathcal{TDLS}$

MRFEL

signification : « Message Reply For Execution List »

description : constitue la réponse à un [MAFEL] et est constitué de la liste des exécutions accessibles par PIMSy.

source = $\mathcal{TS} \cup \mathcal{FS}$

destination = $\mathcal{C}$

MRFET

signification : « Message Reply For Execution Table »

description : constitue la réponse à un [MAFET] concernant un fichier de trace et se compose des différentes informations de ce fichier : auteur, date, nombre d'événements, etc.

source $= \mathcal{TS} \cup \mathcal{FS}$

destination $= \mathcal{C}$

MRFI

signification : « Message Reply For Info »

description : ce message contient le morceau de fichier de trace demandé par un client ([MAFI]).

source $= \mathcal{TS} \cup \mathcal{TDLS}$

destination $= \mathcal{C}$

MT

signification : « Message Text »

description : ce message contient un texte que le processus output va afficher.

source $= all$

destination $= \{Output\}$

TS_MAFI

signification : « TS Message Ask For Info »

description : représente l'équivalent d'un [MAFI] mais pour la gestion interne d'une requête. Les messages commençant par TS sont des messages « internes ». Il est envoyé à tous les *ts* intéressés par une requête.

source $= \mathcal{TS} \cup \mathcal{TDLS}$

destination $= \mathcal{TS}$

TS_MRFI

signification : « TS Message Reply For Info »

description : est le message de conclusion de l'algorithme « odd even ». Il vient d'un *ts* participant au travail et va en direction du *ts* initiateur de la requête.

source = $\mathcal{TS}$

destination = $\mathcal{TS}$

TS_MTIE

signification : « TS Message Trace Info Exchange »

description : représente le message central de l'algorithme de « odd even » lors de la gestion d'une requête. Cette communication se fait entre deux *ts* et se situe entre la réception d'un [TS_MAFI] et l'envoi d'un [TS_MRFI].

source = $\mathcal{TS}$

destination = $\mathcal{TS}$

Index

Bibliographie

[PTV93] Jean-Yves Peterschmitt, Bernard Tourancheau, and Xavier-François Vigouroux. Monitoring the behavior of parallel programs: how to be scalable? Research Report 93.22, LIP, August 1993.