

Rapport de Stage de Fin de Tronc Commun
CADILHAC Michaël (cadilh_m - 17018)
12 janvier 2006

Étude comparative de méthodes de programmation au sein du passage de message tolérant aux pannes

Résumé : Ce rapport présentera le stage effectué en fin d'enseignement de tronc commun du cursus de l'ÉPITA. Il y sera détaillé les problématiques liées au calcul distribué tolérant aux pannes et celles inhérentes au développement d'application dans ce cadre.

Table des matières

Introduction	3
1 Présentation du cadre du stage	7
1.1 Le LRI	7
1.2 Le service et l'équipe	10
2 Cahier des charges	11
2.1 Présentation du projet	11
2.2 Principes de tolérance aux pannes	13
2.3 Principe de MPICH-V	14
2.4 Travail demandé	16
2.4.1 Multi-threading	16
2.4.2 Les <i>threads</i> au sein de MPICH-V	17
2.4.3 Résultats attendus	19
3 Compte-rendu d'activité	21
3.1 Spécifications techniques du protocole de MPICH-V	21
3.2 Développement du client	23
3.3 Développement de la version <i>mono-threadée</i> du serveur	24
3.4 Développement de la version un <i>thread</i> par client	26
3.5 Développement de la version généralisant la première	27
3.6 Développement de la version avec <i>threads</i> spécialisés	28
3.7 Mise au point et expériences	30
4 Résultats	31
4.1 Conformité au cahier des charges	31
4.2 De l'apprentissage grâce au stage	33

Conclusion	35
Liste des acronymes	36
Glossaire	37
Table des figures	39
Bibliographie	40
Annexes	1
ChangeLog du projet	1
Message Relaying Techniques for Computational Grids and their Relations to Fault Tolerant Message Passing for the Grid . . .	6
Introduction au rapport d'activité 2000-2004 du LRI	29

Introduction

Ce document présente dans le détail le stage concluant l'enseignement de tronc commun du cursus de l'ÉPITA¹ (École pour l'Informatique et les Techniques Avancées). Dans ce cadre a été effectué durant les trois derniers mois de 2005 un travail au LRI² (Laboratoire de Recherche en Informatique) au sein d'une équipe de recherche de l'INRIA FUTURS³ : GRAND LARGE⁴.

Les problématiques de distribution dans les communications réseaux connaissent un intérêt et un engouement rare dans ces dernières années. Considérées comme un point essentiel pour la progression de l'informatique scientifique, son étude est aussi bien menée par des recherches théoriques que pratiques.

Le but est de faire mener à un ensemble de machine un calcul ou un processus de manière distribuée. Il s'agit, *in fine*, d'obtenir de \mathcal{N} ordinateurs en réseau, \mathcal{N} fois la puissance d'un ordinateur. C'est un défi en ce sens où, outre le caractère hétéroclite des ordinateurs, il y a la possibilité que l'un d'eux s'arrête, envoie des données erronées ou ralentisse ses pairs.

Le travail présenté ici porte sur une branche précise de ce domaine : la tolérance aux pannes au sein d'un réseau d'ordinateur désirant communiquer sans perte de donnée. Tout un pan de la théorie des systèmes distribués s'appuie sur la supposition qu'un ensemble de machine sera *infaillible* ; afin de donner une existence réelle à ces travaux, il faut créer des mécanismes permettant à cette supposition d'être valide.

¹<http://www.epita.fr>

²<http://www.lri.fr>

³<http://www-futurs.inria.fr>

⁴<http://www.inria.fr/recherche/equipes/grand-large.fr.html>

Lorsqu'une machine d'un système distribué s'arrête, le système est dit tolérant aux pannes, dans notre cas, si :

1. Les autres machines ne sont pas affectées,
2. Le calcul ou le processus sera repris sans erreur.

D'un point de vue technique, la tolérance aux pannes, quelque soit sa conception, peut être implémentée à des *couches réseaux* très différentes. Par *couches réseaux*, il est fait référence aux multiples niveaux de communication employés. La couche la plus basse représente l'envoi d'impulsions électriques *via* la carte réseau, c'est à dire la partie physique de la communication, tandis que la couche la plus haute peut par exemple représenter l'envoi d'une donnée complexe et formatée, avec la certitude que le message sera reçu si il est envoyé.

Ainsi l'équipe GRAND LARGE, dont un des principaux buts est d'introduire la tolérance aux pannes dans les communications entre *grilles* d'ordinateur, se base sur le protocole MPI. MPI – pour Message Passing Interface – est un standard de communication entre des noeuds (des machines) exécutant un programme *parallèle* sur un système de *mémoire distribuée*. Pour plus de détail, se reporter à [3].

De nombreuses implémentations de MPI existent, nous pouvons nommer parmi elle le très notable Open MPI⁵, un de ses prédécesseurs LAM/MPI⁶, ou encore MPICH⁷. C'est sur ce dernier que se base MPICH-V⁸, une implémentation pour *ressources volatiles*, comprendre pour noeuds pouvant subir une défaillance, développée par l'équipe GRAND LARGE.

Un des avantages qui est donné par l'utilisation du protocole MPI et de ses implémentations, est qu'il se situe lui même au dessus du protocole TCP/IP. Ce standard de communication entre ordinateurs a la particularité de vérifier l'intégrité des messages reçus. Ainsi, le point 1. évoqué plus haut dans les critères pour être tolérant aux fautes nous est acquis : tout ce que

⁵<http://www.open-mpi.org>

⁶<http://www.lam-mpi.org>

⁷<http://www-unix.mcs.anl.gov/mpi/mpich/>

⁸<http://www.mpich-v.net>

les autres machines *recevront* aura été réellement *envoyé* et donc *calculé* par l'ordinateur subissant une défaillance.

Le principe original de MPICH-V se situe dans l'utilisation d'une machine, dans un premier temps réputée fiable, par laquelle passeront toutes les communications. Elle sauvegardera ainsi les messages et permettra à terme une reprise du calcul. L'application exécutée par cette machine est appelé le «canal mémoire» (ci-après nommé le CM), jouant le rôle double de *pipeline* (faisant suivre les messages) et de mémoire pour les transferts. Il est à noté que le principe sera généralisé afin que l'on puisse faire fonctionner ce *protocole* avec plusieurs CM.

Le travail qui devait être effectué est une étude comparative sur différentes *méthodes* d'implémentation et de programmation du CM. Il s'agit d'évaluer quel est l'impact de certains choix sur les performances globales du système de communication tolérant aux pannes. Ces choix se portent principalement sur l'utilisation faite des fils d'exécution légers des processus, plus communément appelés *threads*. Un *thread* est une exécution particulière d'un processus sur une machine partageant les zones mémoires qu'il utilise avec les autres *threads*. Des détails plus techniques éclaircissant la réelle problématique de la programmation à l'aide de *threads* seront donnés par la suite.

Ce stage requérait des compétences en programmation système et réseau relativement bas niveau, le tout dans le langage de programmation natif aux systèmes UNIX, à savoir le C. Une bonne connaissance des systèmes UNIX en général, et du système GNU/LINUX en particulier, ainsi que de ses spécifications techniques était un réel plus. Il va de soit, ensuite, que les problèmes de la programmation utilisant des *threads* devaient être connus, et de préférence pratiqués. Parmi ces besoins, quelques uns me faisaient défaut, et il sera vu plus loin si et comment les lacunes furent comblées.

Outre l'opportunité qui m'était offerte de renforcer mes connaissances dans les domaines que j'avais déjà approchés, la possibilité d'apprendre et de comprendre ce qui passionnait nombre de gens dans le réseau m'a profondément séduite. Car, il faut bien le dire, non seulement je ne m'y connaissais que peu en ce domaine mais en plus je ne l'aimais pas.

Pour autant, le point qui m'intéressait le plus et qui m'a fait opter pour

ce stage plutôt qu'un autre, était le cadre. Il s'agissait de s'éloigner de ce «monde de l'entreprise» qui ne m'intéresse que fort peu, pour se rapprocher du milieu plus universitaire, plus vivant et, à mon sens bien évidemment, plus dynamique qu'est celui de la recherche. Les mots LRI et INRIA⁹ furent des arguments chocs face aux autres possibilités que j'avais.

Le reste de ce document s'attachera à présenter aussi bien ledit cadre que le travail effectué, au fil des semaines. Seront donc abordés les aspects techniques et humains de mon stage au travers de mon installation dans les locaux, ma formation à MPI et l'apprentissage des problématiques de tolérance aux pannes, le travail proprement dit et l'établissement des résultats.

⁹<http://www.inria.fr>

Chapitre 1

Présentation du cadre du stage

1.1 Le LRI

Le laboratoire de Recherche en Informatique (LRI), créé il y a plus de 30 ans à l'Université Paris-Sud, est l'un des plus grands et des plus prestigieux laboratoires français de recherche en informatique. Au 1^{er} janvier 2004, le laboratoire comptait 177 membres : 78 chercheurs et enseignants-chercheurs permanents, 63 doctorants, 22 personnels non permanents et 14 personnels techniques et administratifs.

Le LRI est constitué de 10 équipes de recherche, assistées d'une équipe de support système et réseau et d'une équipe administrative. Le laboratoire occupe 4500m² dans le bâtiment 490 sur le campus d'Orsay. Ses principaux moyens de recherche sont un réseau haute performance, une grappe de calcul et une bibliothèque.

Le LRI est une unité mixte de recherche (UMR 8623) du CNRS et de l'Université ParisSud. Les personnels permanents sont issus de ces deux organismes (14 chercheurs et 10 ITAs CNRS, 64 enseignants-chercheurs et 4 IATOS Université Paris-Sud). Le budget annuel, hors salaires des permanents, est de l'ordre de 1.6M. Moins d'un tiers de ce budget provient des dotations de base du CNRS et de l'Université; le reste provient de contrats et subventions obtenus par les membres du laboratoire.

Les thèmes de recherche abordés par le LRI couvrent un large spectre

de l'informatique : algorithmique, complexité, calcul quantique, théorie des graphes, fondements des communications, micro-architecture, clusters et grilles de calcul, génie logiciel, programmation, interaction homme-machine, bases de données, systèmes d'inférence, fouille de données, apprentissage par machine, et bioinformatique.

Cette diversité est l'une des forces du laboratoire, car elle favorise les recherches aux frontières des thématiques, là où le potentiel d'innovation est le plus grand. Ainsi, il y a deux ans, des chercheurs de quatre équipes ont décidé de créer un axe transversal sur la bioinformatique. Cet axe est désormais une équipe à part entière qui développe une approche originale en combinant les concepts de différents domaines, comme l'apprentissage par machine et les algorithmes randomisés pour prédire l'information biologique pertinente dans les données génomiques.

L'introduction au rapport d'activité 2000-2004 est placée en annexe du présent document.

Le LRI est lui-même subdivisé en 10 équipes selon des orientations de recherche différentes :

- Théorie des graphes et fondements des communications,
- Algorithmique et complexité,
- Programmation,
- Bases de données,
- Preuves et programmes,
- Architectures parallèles.
- Parallélisme,
- Intelligence artificielle et systèmes d'inférence,
- Inférence et apprentissage,
- Bioinformatique.

L'organigramme suivant précise l'intégration des différentes équipes au sein même du LRI.

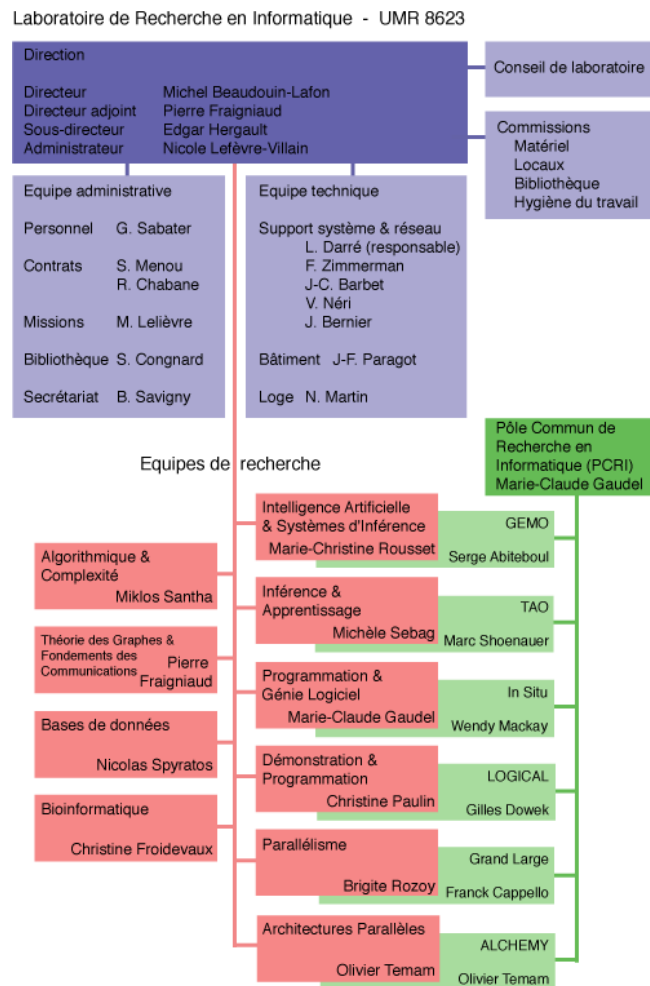


FIG. 1.1 – Organisation du LRI

1.2 Le service et l'équipe

Le travail effectué prenait place dans le service Parallélisme, au sein de l'équipe GRAND LARGE dirigée par Franck CAPPELLO, directeur de recherche INRIA FUTURS. C'est sous sa direction que mon maître de stage, Thomas HÉRAULT a effectué sa thèse dans le domaine de l'auto-stabilisation en 2002.

Le projet GRAND LARGE, créé en octobre 2003, étudie le calcul dans les systèmes à grande échelle. L'approche se situe au niveau middleware et outils de programmation bas niveau, entre les systèmes d'exploitation et les environnements de programmation de haut niveau.

L'équipe vise à :

- étudier de façon expérimentale, mais aussi formelle, les mécanismes fondamentaux de ces systèmes, c'est à dire les points durs scientifiques et les verrous technologiques,
- concevoir, réaliser, valider et tester des modules logiciels et des plateformes réelles,
- définir, évaluer et expérimenter la programmation de telles plateformes.

Les thématiques abordées concernent la sécurité, la gestion de la volatilité, l'ordonnancement à grande échelle, la circulation de données et la programmation parallèle et répartie. Plus généralement, les recherches développées dans cette action trouvent des applications dans les très grands systèmes parallèles (*clusters* équipés de plusieurs milliers de processeurs) et les grilles (grilles de PC, grilles de *clusters*). Parallèlement, il s'agit d'étudier des outils méthodologiques nouveaux permettant de rendre compte finement de phénomènes liés à la grande échelle dans un contexte d'applications réelles.

Le travail était, comme il sera détaillé plus loin, lié au projet MPICH-V s'intéressant à la gestion de la tolérance aux pannes au sein d'un ensemble de noeuds volatiles. Les principales personnes liées à ce projet sont :

- Thomas HÉRAULT, maître de conférence en informatique à Orsay,
- Aurélien BOUTEILLER, thésard sous la direction de Franck CAPPELLO,
- Pierre LEMARINIER, thésard sous la direction de Joffroy BEAUQUIER.

Chapitre 2

Cahier des charges

Afin de comprendre tous les sujets abordés dans ce cahier des charges et dans la suite de ce document, certains concepts et aspects légèrement techniques doivent être préalablement présentés dans le détail.

Ainsi, il sera repris dans cette section quelques points abordés dans l'introduction, à savoir l'architecture du projet, MPI (Message Passing Interface), MPICH, la notion de tolérance aux pannes conjointement avec MPICH-V ainsi que les *threads* et leur problématique plus précisément au sein du travail effectué.

2.1 Présentation du projet

MPICH-V a pour but d'offrir un système de *passage de message* entre noeuds d'une grille de PC ou de *cluster* qui soit tolérant aux pannes – le terme «tolérant aux fautes» serait abusif dans notre cas.

Le projet à l'avantage non négligeable d'introduire la tolérance aux pannes de manière *non intrusive*, c'est à dire ne demandant à l'utilisateur supposé aucune modification de son code afin de rendre son application distribuée tolérante aux pannes. Pour cela, il suffit que l'application utilise une implémentation générique de communication entre les machines.

MPI (Message Passing Interface), conçue en 1993-94, est une bibliothèque de fonctions, utilisable avec les langages C, C++, Ada et Fortran. Elle per-

met d'exploiter des ordinateurs distants ou multiprocesseur par passage de messages.

Il est devenu *de facto* un standard de communication pour des noeuds exécutant des programmes parallèles sur des systèmes à mémoire distribuée.

MPI a été écrit pour obtenir de bonnes performances aussi bien sur des machines massivement parallèles à mémoire partagée que sur des *clusters* d'ordinateurs hétérogènes à mémoire distribuée. Il est grandement disponible sur de très nombreux matériels et systèmes d'exploitation. Ainsi, MPI possède l'avantage par rapport au plus vieilles bibliothèques de passage de messages d'être grandement portable (car elle a été implanté sur presque toutes les architectures de mémoires) et rapide (car chaque implantation a été optimisée pour le matériel sur lequel il s'exécute).

Il existe un certain nombre d'implémentation de MPI, les principales étant :

- Open MPI : Combinant LAM/MPI, FT-MPI¹, LA-MPI² et PACXMPI³, il a pour but premier de créer la meilleure bibliothèque de MPI en associant des bibliothèques qui excellaient dans un domaine précis.
- MPICH : Extrêmement portable entre les systèmes, cette bibliothèque existe en deux versions répondant aux deux spécifications différentes de MPI.

Le choix entre l'une ou l'autre des bibliothèques est souvent purement arbitraire et esthétique. Les performances qu'offrent l'une ou l'autre implémentation ne varie que peu, et chacune aura ses programmes où elle tournera plus ou moins vite que l'autre.

Comme le nom du projet le laisse entendre, MPICH-V est basé sur MPICH. MPICH-V se comporte comme une *implémentation* de l'interface proposée par MPICH. De fait, tout logiciel utilisant MPICH pour ses communications entre noeuds n'a besoin pour être tolérant aux pannes que de configurer MPICH pour qu'il utilise l'implémentation proposée.

¹<http://icl.cs.utk.edu/ftmpi/>

³<http://www.hlrs.de/organization/pds/projects/pacx-mpi>

2.2 Principes de tolérance aux pannes

La tolérance aux pannes implique deux choses :

- Si un noeud «tombe», les autres noeuds ne doivent pas être influencés *au niveau du calcul parallèle*. Il y aura en effet sûrement une baisse de performance globale notable, mais le processus en lui même ne doit pas devenir faux.

De manière optimale, ceci implique que si un noeud redémarre, il ne doit *pas*, dans la mesure du possible, demander aux autres noeuds de redémarrer pour recommencer l'exécution.

- En cas de défaillance, le calcul doit finir par être repris.

Le choix fait au sein du projet est que seul le noeud qui est tombé redémarrera avec pour seconde contrainte que les autres noeuds ne doivent pas avoir à refaire les calculs déjà effectués.

Il s'agira donc d'avoir une mémoire, globale ou sur chaque noeud, pour sauvegarder tous les envois effectués. Quand un noeud se reconnectera, il *rejouera* son exécution, puisque ses données seront perdues, et les messages qu'il demandera aux autres noeuds seront envoyés instantanément, car déjà en mémoire.

Ainsi, lors d'une *reprise sur panne*, les messages envoyés par le noeud redémarrant seront, d'une manière ou d'une autre, ignorés car déjà reçus dans une précédente exécution. La communication «utile» sera donc dans le sens noeuds sains vers noeud redémarrant.

2.3 Principe de MPICH-V

Un schéma de communication standard entre noeuds d'un *cluster* est le suivant :

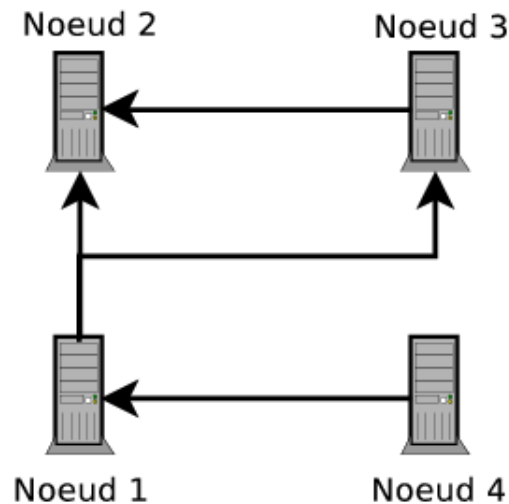


FIG. 2.1 – Communication noeuds à noeuds

Chaque noeuds est connecté à tous les noeuds avec lesquels il a besoin de communiquer. La communication est dite de pair à pair : aucune notion de *hiérarchie* entre les différents noeuds n'entre en effet en jeu.

Si l'on voulait introduire la tolérance aux pannes telle que décrite plus haut dans cette configuration, il faudrait que chaque noeud sauvegarde tous les messages qu'il *envoie*. Ainsi, une application qui deviendrait tolérante aux pannes par ce mécanisme se verrait ajouter un surcoût non négligeable en terme d'utilisation mémoire.

De plus, les noeuds étant nombreux, la probabilité que deux ou plus d'entre eux subissent une panne en même temps devient grande. Cette gestion décentralisée peut donc, dans le cas où deux noeuds discutant beaucoup tombent en même temps, conduire à l'attente de production d'un des noeuds par l'autre.

L'idée dans MPICH-V est d'avoir une ou plusieurs machines, dans un premier temps réputées fiables, qui ne s'occuperont que de faire la sauvegarde des messages. Le schéma de communication est radicalement changé :

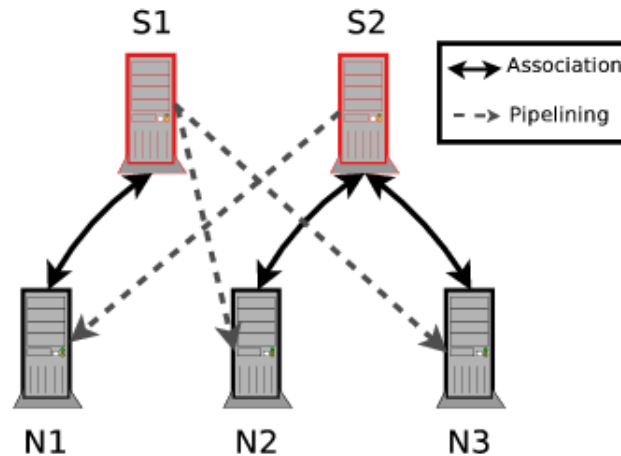


FIG. 2.2 – Communication hiérarchique - MPICH-V
Les associations font aussi du *pipelining*
(faire suivre les messages)

Chaque nœuds est associé à un serveur qui s'occupe de sauvegarder les messages à *destination* de *ses* clients. Dans Fig. 2.2, N1 est associé à S1. Si N2 veut envoyer un message à N1, il devra se connecter à S1 et lui envoyer le message en précisant que le destinataire est N1. Si N2 veut envoyer un message à N3 qui a le même serveur S2, il l'enverra directement à S2.

En cas de panne, le nœud défaillant redémarre et recommence ses calculs et ses envois de message. Chaque message est envoyé conjointement avec un indice qui permet aux serveurs de déterminer si le message a déjà été envoyé (ce mécanisme sera détaillé dans les spécifications techniques). Ainsi les serveurs peuvent ignorer ces messages, en ce sens qu'ils ne les font pas suivre aux réels destinataires, et pourront émuler la réponse, connaissant déjà la conversation.

On ajoute à la reprise sur panne la notion de *checkpoint* : de temps à autre les noeuds vont sauvegarder l'état de leur processus sur un serveur particulier. De ce fait, en cas de reprise, le noeuds relancera une *image* du processus d'un instant t et reprendra la partie du calcul qu'il était entrain d'exécuter.

Outre les avantages au niveau des pertes d'information sur le noeud défaillant qui sont sensiblement diminuées, les serveurs tirent aussi parti de la sauvegarde des processus : une fois que le processus a été sauvegardé il prévient son serveur qui peut alors supprimer de sa mémoire tous les messages qu'il destinait à une potentielle reprise.

2.4 Travail demandé

L'intitulé de mon stage tel que donné par l'INRIA FUTURS était le suivant :

Conception, implémentation et évaluation d'un protocole hiérarchique de tolérance aux fautes pour les grilles de calculs.

De cet intitulé on peut lire «MPICH-V» et évaluation. Il s'agissait de juger de l'impact de certaines options de programmation sur les performances globales des communications réseaux. Il a été vu précédemment que le rôle de tolérance aux pannes qui était associé à chaque noeud a été centralisé sur les serveurs. Il s'agit donc de minimiser les baisses de performances dues à ce choix. Pour cela, il est intéressant de doter les serveurs d'interfaces réseaux puissantes ; il faudra ensuite arriver à obtenir les performances de *gestion* réseau, c'est à dire de gestion par le système, équivalentes.

2.4.1 Multi-threading

Le *multi-threading* est une technique permettant d'exécuter de N manières parallèles différentes un processus. Le terme «parallèle» est ici utilisé alors que nous sommes sur une seule machine : en fait, il s'agit d'exécution pseudo-parallèle, en ce sens où chaque exécution à un temps pendant laquelle elle se déroule puis on passe à une autre, et d'une réelle parallélisation grâce à l'exécution sur plusieurs *processeurs* en même temps sur la machine.

Le *multi-threading* a ceci de particulier que chaque exécution partage sa mémoire avec ses pairs. Pour introduire les problématiques créées par cette spécificité, voyons ce qu'il se passe en cas d'accès *concurrents* à une même zone mémoire, c'est à dire que deux exécutions accèdent en même temps à la même variable.

Soit une variable entière partagée V codée en mémoire sur 4 octets. L'exécution 1 commence à écrire la valeur de V , elle écrit le premier octet, puis le second. L'exécution 2 a maintenant la mauvaise idée de vouloir lire V , alors qu'elle n'est pas entièrement écrite ; la valeur que lira cette exécution ne sera ni la précédente valeur de V , ni la valeur que l'exécution 1 voulait lui donner, ce sera donc une valeur *incohérente*.

Il faut donc introduire des mécanismes pour éviter ce genre d'accès. On doit, par exemple, créer des zones que seulement *un thread* à la fois peut exécuter.

2.4.2 Les *threads* au sein de MPICH-V

Le stage avait pour but de répondre à cette question :

Quelle est la meilleure politique d'utilisation des *threads* que l'on puisse utiliser dans une application telle que le «canal mémoire» ?

Autant il est séduisant de se dire que l'on va exécuter, avec peu de surcoût, N fois la même application sur une même machine à l'aide de *threads*, autant il existe bien des manières pour essayer d'atteindre ce but. Il s'agit de donner un *rôle* à chacune des exécutions particulières, de les spécialiser plus ou moins, de répliquer telle ou telle actions, . . .

Le serveur de MPICH-V, ci-après nommé le «canal mémoire» ou CM, doit répondre au fonctionnement suivant :

- Accepter les connexions de clients,
- En cas de réception de message les faire suivre,
- Sauvegarder les messages qui transitent par lui,
- Être capable de rejouer une séquence pour un noeud redémarrant.

Ce découpage fonctionnel a conduit à l'établissement de 4 conceptions de l'implémentation (on utilisera le terme «versions») du CM relatives aux *threads* :

- *mono-threadée* : N'utilisant pas de *thread*, cette version est celle «par défaut» car la plus simple à développer.
Elle utilise un mécanisme de *multiplexage des entrées* pour lire simultanément ce qu'envoient les clients. Dès qu'un message est reçu il est traité, puis on retourne dans l'attente de message.
Il n'y a donc qu'un fil d'exécution, et on évite par cette technique tous les problèmes liés aux accès concurrents. Pour autant, si le traitement d'un message en particulier est long, tous les autres clients du CM vont être en attente de lecture de leurs messages. Une solution *multi-thread* a tendance à éviter ce problème.
- *multi-threadée*, avec un *thread* par client : Dual de la version précédente, elle créera un fil d'exécution dès qu'un client se connectera et ce fil sera dédié à la réception et au traitement des messages de ce client.
On arrive très vite à créer un nombre important de *threads*, bien trop important pour les critères de rentabilité de l'utilisation des *threads*.
En effet, la gestion des accès concurrents imposera des
- *multi-threadée*, généralisation de la première version : Un nombre variable de *thread* fait du *multiplexage* de ses entrées. On a donc N fois la première version, avec N que l'on fixe au démarrage du serveur.
Il est à remarquer que si $N = 1$, on revient à la première version, et si N varie dynamiquement en fonction du nombre de client, on obtient la deuxième version.
- *multi-threadée*, utilisation d'ensemble de *threads* spécialisés :
 - Un *thread* s'occupe d'accepter les connexions donc de l'écoute sur le port du serveur,
 - N *threads* multiplexent les entrées, comme le fait la version précédente,
 - Un *thread* s'occupe de l'envoi des messages à destination des clients, donc de faire suivre *effectivement* les messages.

2.4.3 Résultats attendus

« Des courbes ! »

– Thomas HÉRAULT

Il fallait donc avec un ou des critères juger de l'efficacité d'une méthode par rapport à l'autre. Dans le domaine des communications réseaux, il suffit de s'attarder sur la bande passante et sur la latence des flux pour cela. Reste donc à identifier dans quelles conditions peut-on estimer que ces évaluations sont significatives dans le cas général.

En effet, si il est évident qu'une communication unidirectionnelle n'aura pas les mêmes performances qu'une communication en ping-pong, il faut bien voir que les rapports d'une à l'autre version du serveur pour ces deux méthodes peut être radicalement différents ; une des deux versions étant plus à l'aise que l'autre dans certains cas seulement.

Il doit être également pris en compte que l'une des versions peut-être très adaptée au traitement de deux clients tandis qu'elle aura des performances moindre qu'une autre version pour 64 clients.

Ajoutons à ça le récurrent N , variable de quelques-unes des versions, pour constater que le nombre de paramètres à faire varier pour avoir une idée précise du comportement de chaque versions est important :

- Nombre de serveurs,
- Nombre de clients,
- Agencement réseau,
- Méthode de communication,
- ...

L'agencement réseau est un critère à part entière que nous avons pu exploiter grâce aux moyens mis à notre disposition par le projet GRID'5000⁴. Il s'agit, *in fine*, de rassembler 5000 processeurs de part la France dans différents *clusters* et d'en faire une grille.

⁴<http://www.grid5000.org>

Les performances à l'intérieur d'un *cluster* sont donc optimales, en toute théorie, en terme de capacité réseau actuelle. Les *clusters* eux-mêmes sont inter-connectés grâce au réseau RENATER⁵, réseau dédié à l'éducation et à la recherche affichant des performances remarquables. 17 laboratoires différents, dont un certain nombre de l'INRIA et du CNRS⁶, participent à ce projet visant à l'établissement de 9 *clusters* de 100 à 1000 processeurs chacun en France.

⁵<http://www.renater.fr>

⁶<http://www.cnrs.fr>

Chapitre 3

Compte-rendu d'activité

Arrivé sur les lieux de travail, je reçu tout d'abord une formation courte mais suffisante à MPI et il me fut décrit le fonctionnement de MPICH-V au travers de [1]. Il était en suite question de bien modéliser les différentes parties du projet.

3.1 Spécifications techniques du protocole de MPICH-V

MPICH-V est une *surcouche* du protocole TCP/IP. Il ne sera pas ici approfondi le fonctionnement de TCP/IP, il y a juste à savoir qu'il assure qu'un message reçu a véritablement été envoyé, c'est à dire sans perte ni altération de donnée. MPICH-V *encapsule* les messages bruts de quelques informations qui seront détaillées plus loin, et ce message encapsulé est ensuite pris en charge par TCP/IP pour être acheminé aux CMs.

Reprenons un schéma de communication simple avec un seul CM :

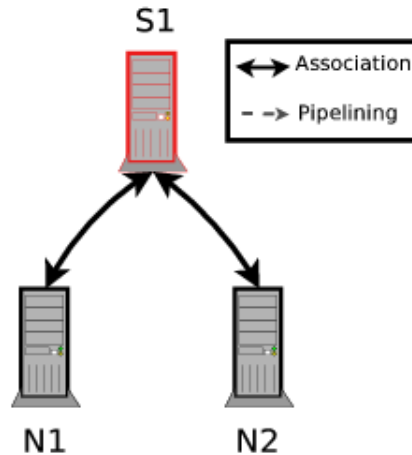


FIG. 3.1 – MPICH-V : Un seul CM, deux clients

N1 souhaite envoyer un message à N2. Le principe de MPICH-V est que N1 envoie un message au CM de N2. Afin que le CM puisse savoir à qui faire suivre le message nous avons besoin que dans l'encapsulation il y ait un champ «destinataire». Dans MPI, chaque noeud a ceci de particulier qu'il est identifié de manière unique par son *rang*, un entier.

Pour que le *forwarding*, l'action de faire suivre le message, se fasse sans perte de donnée, il faut que le CM connaisse la taille du message. Cela nous fait donc un second champ dans l'encapsulation.

Il a été évoqué plus haut que les CMs savaient si le message qu'elles recevaient avait déjà été envoyé ou pas, en cas de reprise sur pannes. Le mécanisme est le suivant :

- Chaque message est envoyé conjointement avec un entier *clock* : il est nul pour le premier message, vaut 1 pour le suivant et ainsi de suite.
- Un CM recevant un message d'*indice c* va comparer *c* avec la valeur maximale des indices des messages précédents en provenance du client qui fait l'envoi :
 - Si $c \leq n$ alors ce message a déjà été transféré,
 - Sinon, le message est à transférer.

MPI dicte le fait que les noeuds *demandent* à recevoir un message à leur CM, ils effectuent une requête selon deux paramètres :

- La source du message, ou -1 pour «n'importe quelle source»,
- Un *tag*, une étiquette à laquelle le message doit correspondre, ou -1 pour «n'importe quelle étiquette».

L'encapsulation intègre donc cette étiquette. Un message envoyé par le biais du protocole décrit sera dès lors de la forme :



FIG. 3.2 – Message envoyé via MPICH-V

3.2 Développement du client

Historiquement, ce n'est pas le client qui a été développé en premier. Pour autant, il s'agit d'un élément qui ne variera pas selon les versions des CMs : le protocole étant le même, aucune modification n'est à faire.

Dans MPICH, sur lequel se base MPICH-V, il suffit de ne développer que quelques fonctions, au sens de code C, pour faire fonctionner l'interface MPI :

probe : renvoie «vrai» si un message correspondant au «tag» en paramètre est en attente.

send : envoie un message à un destinataire avec un certain «tag».

recv : Demande la réception d'un message et attend qu'il soit complètement arrivé.

init et finalize : Initialise et détruit les variables relatives au protocole.

Tout ceci fut implémenté rapidement et seules quelques modifications furent apportées plus tard, pour permettre de tirer des statistiques sur l'utilisation mémoire, processeur, réseau et disque. Une évolution de la politique

sur la «verbosité» des binaires du projet conduisit aussi à introduire des options à la compilation qui impliquèrent des changements dans les sources du client.

3.3 Développement de la version *mono-threadée* du serveur

Ce fut le premier développement effectué. Un travail préparatif sur papier devait me permettre d'envisager plus sereinement aussi bien l'implémentation de cette version que celle de ses soeurs *multi-threadées*. Cette version fut aussi l'occasion d'utiliser une première fois des outils de développement connexes au projet :

- Subversion¹ : Outil de versionnement, permettant un partage des sources, une sauvegarde modification par modification, bref, une grande souplesse dans le développement. Il a été mis à ma disposition un dépôt SVN (SubVersioN) par Thomas HÉRAULT et l'équipe du LRI.
- Les Autotools² : Ensemble d'outils aidant à la création et la maintenance de procédures de création et d'installation des binaires issus de la compilation de fichier sources.

Ces outils m'étaient quelque peu étrangers et j'ai pu m'y familiariser grâce à l'aide que m'a apporté Thomas CLAVEIROLE, un ancien de l'ÉPITA.

- Doxygen³ : Cet utilitaire permet de tirer, d'un code commenté avec certaines balises, une documentation complète des classes et fonctions du projet.

Évidemment, personne ne repassera sur mon code ni même ne sera intéressé de savoir quel est le rôle de telle ou telle fonction ; il s'agit juste d'un principe de précaution dans un premier temps, et de forcer ses habitudes dans un second.

À en croire les entrées de ChangeLog, le développement de cette première version pris 3 semaines. Beaucoup de fichier de conception complètement générique ou relative au protocole seul purent être mis en commun dans

¹<http://subversion.tigris.org>

²<http://www.gnu.org/software/autoconf/>

³<http://www.doxygen.org>

toutes les versions du serveur, et dans une moindre mesure, avec le client.

Un des leitmotivs imposé par mon maître de stage, outre le fait qu'il faille tout coder dans le langage natif d'UNIX, le C, est qu'il fallait utiliser le moins possible des bibliothèques externes. La raison est simple, il y avait nécessité de contrôler, si ce n'est le maximum de paramètre, au moins le maximum l'utilisation qu'était faite des ressources.

Le serveur a en mémoire pour chaque client une structure, c'est à dire un ensemble de propriété, qui évolue en fonction des messages qu'il envoie ou reçoit. Dans un réseau, l'envoi d'un message d'un point à un autre peut se faire, physiquement, en deux fois. C'est pourquoi le serveur sauve l'«état» de chaque client, c'est à dire s'il est entrain d'envoyer un message, qui peut n'être reçu par le serveur qu'à moitié, ou entrain d'envoyer une requête, ainsi de suite.

Chaque type message, à savoir démarrage du noeud, envoi de message, demande de réception, demande de présence de message et indication de début et de fin de sauvegarde du processus, est identifié de manière unique sur un octet. L'octet est la plus petite quantité de donnée que l'on puisse recevoir, ainsi, on peut déterminer dans quel état est rentré le client dès qu'il envoie un nouveau message.

D'un état, on ne peut recevoir que telle ou telle donnée : il n'y a pas de sens à ce que l'on reçoive un message après qu'il ait été indiqué qu'il allait être envoyé une requête. Cette manière contrainte de passer d'un état à un autre fait référence aux automates, on parle de programmation avec automate à état. Le serveur était donc découpé en fichier et chaque état avait le sien.

Techniquement peu compliqué, un effort a été fait sur la propreté de la conception, l'agencement et la lisibilité du code, le tout dans un soucis d'efficacité maximal de ce dernier, à grand renfort de certaines particularités sombres du langage.

3.4 Développement de la version un *thread* par client

Cette version n'avait tout d'abord pas été envisagée. En fait, tout le monde était convaincu que ce ne serait qu'une perte de temps de développer une pareille variante, tant les performances *estimées* de celle-ci étaient risibles.

Mais cette version qui ne devait pas voir le jour eut un regain d'intérêt lorsque, lors d'une présentation donnée au LRI, un docteur américain crut bon la défendre.

Du point de vue de la conception, les entrées n'ont plus à être multiplexées : en effet, chaque *thread* s'occupant d'un client, il peut se permettre de rester bloqué en attente de message seulement de sa part.

Les premiers accès concurrents à risque apparaissent donc. Pour certains, de simple *mutex* suffisent pour les résoudre. Il s'agit d'introduire des zones, ici seulement relatives à un client, dans lesquelles seul *un thread* peut entrer. Dans des cas autres, l'on doit introduire des *conditions*. Le but de ce document n'est pas de présenter en détail la théorie des *threads*, on simplifiera donc en disant que les conditions sont des signaux levés afin de faire communiquer deux ou plus *threads*.

Dans cette version, il n'apparaît pas d'accès concurrents sur simples variables globales aux *threads*, mais seulement relativement à un client donné. La résolution d'un problème de ce type a une approche tout à fait similaire, comme nous l'évoquerons plus après.

Techniquement parlant, cette première version m'a permis une première approche de la bibliothèque de *threads* Pthread⁴. Standard dans son domaine, l'implémentation de la norme POSIX relative aux *threads* par Xavier LEROY (de l'INRIA) s'est bien évidemment imposée à nous comme notre bibliothèque de travail. On peut par ailleurs noter que son intégration au sein du projet à l'aide des Autotools⁵ est extrêmement simple, et a permis une utilisation sans embûche au plus vite.

⁴<http://pauillac.inria.fr/~xleroy/linuxthreads/>

⁵<http://www.gnu.org/software/autotools/>

Du point de vue attente sur les performances, il faut avouer que nous espérons de piteux résultats. Autant cette solution est intéressante quand il y a seulement deux clients sur le CM, car nous travaillons essentiellement sur des machines à deux processeurs, autant cette solution extrême ne semblait pas convenir pour un plus grand nombre de client. Nous verrons plus loin quels sont ces résultats.

3.5 Développement de la version généralisant la première

Il s'agissait donc de créer un ensemble de *threads* reproduisant à l'identique le comportement de la première version tout en gérant les accès concurrents créés par cette généralisation à N *threads*. Cette méthode, générique vis à vis de la problématique résolue de cette manière, permettrait de trouver l'équilibre entre nombre de client et nombre de *threads* consacrés à ces derniers.

Elle fut tout d'abord implémentée par une solution hybride condition/signaux. Les signaux sont une des plus anciennes techniques dans le monde UNIX pour faire communiquer deux processus. Ils s'envoient un chiffre, 0 ou 1, et tout ce que fait le receveur est interrompu pour lui signaler la réception de ce chiffre. Cette communication inter-processus a été adaptée, dans la bibliothèque Pthread, pour les *threads*.

Cette solution était pleinement satisfaisante, du moins, je le pensais. À dire vrai, j'étais content d'avoir mélangé toutes ces techniques, implémentation durant laquelle j'ai beaucoup appris. Mais j'aurais été bien plus content si elle avait qui plus est marché.

Il existe, dans le domaine de la programmation, un terme qui est souvent utilisé avec les *threads* : «*race condition*». Il s'agit d'une erreur rare, peu reproductible, à laquelle on ne s'attendait pas du tout, et qui va être très énervante. Elle est d'autant plus irritante que sa cause peut-être d'une simplicité enfantine.

J'ai donc eu la joie d'expérimenter pour cette version de très belle *race conditions*, la plupart dues à ma faible expérience en programmation à l'aide de *threads*. Quelques grosses erreurs, celles là même qui sont trop visible pour être vues, amenèrent cette version à ne pas être complètement disponible en temps et en heure.

Nous pensions atteindre des performances plus que correct avec celle ci, pour N variant entre 2 et 4 de 2 à 64 clients. C'est en effet avec ce genre de valeur que l'on peut tirer le mieux parti de l'exécution sur deux processeurs des programmes.

Il a été évoqué la possibilité d'avoir un nombre de *thread* variant *dynamiquement* au cours de l'exécution du programme en fonction du nombre de client, mais la nature de nos tests étant plutôt stable sur ce dernier nombre, l'idée fut abandonnée. Il s'agissait de ne pas le faire varier comme la version précédente, mais K fois moins vite que les clients.

3.6 Développement de la version avec *threads* spécialisés

Il s'agit de la version la plus originale : jusqu'à présent, l'idée était de faire N fois la même chose, et de spécialiser les deux premières versions un peu plus pour qu'elles soient optimales dans leur domaine. Ici, il s'agissait de mettre une subdivision plus claires entre les taches que devaient accomplir le CM et en faire des ensembles de *threads* pour chaque.

Lors de l'explication de ce que j'aurais à faire par Thomas HÉRAULT, en début de stage, l'idée de cette version était de créer trois ensembles de *threads* :

- Un qui fera du multiplexage des entrées,
- Un autre qui s'occupera de la sauvegarde des messages,
- Un dernier qui fera les envois

La partie «sauvegarde» des messages avait une importance plus grande que l'implémentation finale : elle était censée, lorsque la mémoire prise par les messages passait un certain seuil, sauver une partie des messages non plus

en mémoire vive (RAM), mais sur le disque dur.

Cette optimisation paraissait bénéfique dans un premier temps, tant les temps d'accès à la RAM par rapport à celui du disque dur était faible. Pour autant, vouloir l'effectuer par nos propres moyens était coûteux en temps de développement pour un rendement peu sensiblement meilleur que la gestion que fait le système avec le *swap*.

Le *swap*, dans un système GNU/LINUX standard, est une partition d'un disque dur utilisée pour stocker temporairement les pages mémoires peu utilisées lorsque la RAM devient surchargée. Les méthodes pour déterminer si et quand une page doit être mise en *swap* ne font pas partie des préoccupations de ce document, mais c'est la possibilité de mieux contrôler ces paramètres qui fit originellement penser à Thomas qu'il fallait refaire cette partie.

Au final, l'implémentation différait beaucoup des premières conceptions :

- Un *thread* effectuait les lectures de nouvelles connexions, il était donc peu sollicité,
- Un ensemble de *threads* s'occupait de traiter les messages de manière multiplexée,
- Un *thread* effectué tous les envois.

L'utilisation de *threads* spécialisés est séduisante, pour autant elle implique une contrepartie non négligeable : lors de réception d'un message alors que le client est en état neutre, il est d'abord lu l'octet discriminant, puis les données d'encapsulation si il y en a, et enfin le message dans certains cas. Étape par étape, la zone de mémoire dans laquelle est stockée le message brut se remplit. Or, avoir un *thread* qui ne s'occupe que de l'envoi amène le fait que dès qu'il y a une donnée elle est envoyée. Cela fini par impliquer un surcoût notable dans le nombre d'appel aux fonctions d'envoi de base des communications réseaux.

Mais pouvoir paralléliser les réceptions et les envois est un avantage loin d'être négligeable. Les cartes réseaux étant *full-duplex*, donc pouvant recevoir et émettre en même temps, et les machines bi-processeurs dans notre cas, il est très intéressant de pouvoir avoir ce genre d'optimisation.

3.7 Mise au point et expériences

Il fut donc entamé la série d'expériences, après la réalisation de *scripts* automatisant les exécutions sur diverses machines des clients et des CM. Mais il restait à résoudre le dilemme de l'application de test, car seul une implémentation des quelques fonctions MPICH-V avait été faite.

Notre choix se porta sur NetPIPE⁶. NetPIPE est un outil de mesure de performance indépendant du protocole de communication réseau sous-jacent. Il effectue des tests en mettant le réseau dans certaines situations pour en tirer l'essence des performances. Il utilise une méthode dite de ping-pong, les messages étant renvoyés d'un processus à l'autre, les processus étant pris par pair. Il permet d'obtenir la bande passante et la latence pour chaque taille de message.

NetPIPE offre donc la possibilité de s'exécuter en utilisant une implémentation de MPI pour protocole. Chaque exécution de NetPIPE offre une nouvelle courbe statistique. Ces courbes n'étant que données brutes non pertinentes, elles seront synthétisées dans la partie 4 (p. 31) et plus particulièrement dans le document [2].

Pour l'anecdote, il faut savoir que nos performances étaient, dans les premiers tests, jusqu'à 1000 fois moins bonnes qu'une communication sans sauvegarde de message. Cela venait du fait que TCP/IP n'envoie pas, par défaut, les messages dès que la demande en est faite par l'utilisateur, mais seulement quand :

- La zone mémoire tampon de communication TCP/IP est pleine,
- Il y a des données trop «vieilles» dans cette zone.

On peut voir que pour des petits messages, le deuxième cas était toujours celui concerné. De fait, chaque envoi de message devait attendre un certain temps avant d'être effectivement effectué. Il nous fallut nous plonger, avec Thomas HÉRAULT, dans les manuels du fonctionnement de TCP/IP pour comprendre et modifier ce comportement.

⁶<http://www.scl.ameslab.gov/netpipe>

Chapitre 4

Résultats

Cette partie du rapport s'attachera à présenter sommairement les résultats concrets de la réalisation et de l'expérimentation du projet. Je souhaiterai, par ailleurs, éviter la redite en portant l'attention du lecteur sur le fait qu'un rapport technique a été écrit dans le cadre du stage afin de traiter les expériences en elle même ([2], en annexe).

Il y sera aussi détaillé tous les acquis et les points sur lesquels j'ai pu progresser grâce à ce stage, aussi bien humainement, d'un point de vue technique et dans le monde de l'entreprise, ou plus précisément ici, celui de la recherche dans le milieu public.

4.1 Conformité au cahier des charges

Dans la globalité, le but du stage était de déterminer quelle est la meilleure politique d'utilisation des *threads* au sein d'une bibliothèque de passage de message tolérante aux fautes. En ce sens, le travail effectué a permis de répondre, avec plus ou moins de précision, à cette question.

Les résultats sont présentés et interprétés dans le détail dans [2]. Ce document est l'aboutissement du stage, puisqu'il vise la publication dans un recueil de rapport pour CoreGRID¹. CoreGRID est une initiative européenne visant au renforcement et à l'avancée en excellence scientifique et technologique dans les domaines de calcul sur grille et des communications pairs à

¹<http://www.coregrid.net>

pairs.

Ce rapport de recherche est, personnellement, le plus grand accomplissement du stage. Bien que j'y sois présenté comme auteur, mes nombreux retards à répétition, sorte de constante fort désagréable dans mon mode de vie, ne me permirent pas d'écrire en temps et en heure la partie qui m'était réservée. Je ne suis donc en rien dans la qualité rédactionnelle remarquable de ce document.

Pour autant, mon apport se trouve dans les résultats bruts présentés et dans quelques menues explications. D'un point de vue purement technique, la réalisation des versions différentes du «canal mémoire» a été faite, avec quelques parties laissées en suspens.

La première version, *mono-threadée*, de laquelle tous les morceaux de code partageables ont été pris, a été complètement développée. Elle donnait, niveau performance, le «la» auquel nous nous attendions : des résultats moyens en tout point. Encore une fois, le lecteur se référera à [2] pour des résultats plus détaillés.

La version «un *thread* par client» dépassa nos attentes en terme de performance. Complètement réalisée et déboguée, elle offrait des résultats très satisfaisant aussi bien dans un nombre de client restreint qu'important. Des explications techniques sur l'apparition de ce phénomène se trouvent, encore une fois, dans le document support des résultats, [2].

La version généralisée de cette solution *multi-threadée* avait par défaut seulement deux *threads*, valeur qui fut utilisée pour le développement et la phase de débogue. Cette valeur, hélas, était trop petite pour faire surgir une *race condition* qui fut résolue *in extremis* d'une manière peu esthétique. Une partie des expériences n'ont pas pu être faites à cause de ceci, mais les résultats obtenus sont tout de même intéressants : on observe certains cas, particulièrement dans les communications à l'intérieur d'un même *cluster*, où cette version fait des merveilles.

Enfin, la version utilisant des *threads* spécialisés souffrait de la même *race condition*, plus exacerbée car deux *threads* de multiplexage suffisaient à la provoquer. Bien évidemment, la valeur par défaut était donc de un *thread*.

Cela étant, le schéma de spécialisation put tout de même être testé dans ces conditions, et donner des résultats expérimentaux pertinents.

Ce travail s’inscrivait dans le projet MPICH-V, qui se concentre sur la conception et l’implémentation de protocole de passage de message tolérant aux pannes. En ce sens, mon travail a permis d’estimer quelle était la façon la plus efficace de développer pareil protocole, et il s’inscrit donc dans un processus de gain de performance de MPICH-V, mais aussi des bibliothèques s’attachant à atteindre le même but.

4.2 De l’apprentissage grâce au stage

Appartenant au LRDE², par lequel ce stage me fut confié, j’éprouvais une certaine attirance pour le milieu de la recherche. Public ou privée, en France ou ailleurs, mon idée n’était pas encore très précise ; d’autant que je n’avais jamais fait parti d’un laboratoire extra-ÉPITA.

Quel meilleur commencement aurais-je pu avoir que ce stage au sein du LRI, dans le cadre d’une recherche menée par l’INRIA FUTURS ? Tout de suite plongé dans le quotidien du chercheur, j’ai pu apprécier un nombre de point très divers tels que :

- Le rassemblement de connaissances quasi-encyclopédiques sur tous les domaines de l’informatique moderne, aussi bien pratiques que théoriques,
- Une ambiance sans égale où les relations hiérarchiques sont légèrement délaissées au profit d’un travail dans des conditions agréables,
- Une entraide efficace, toute une équipe autour d’un PC pour étudier un problème ou autour d’un café pour en discuter,
- Une grande souplesse dans les horaires, qui permettait d’arriver à midi pour repartir à 23h, un travail dans un milieu universitaire vivant, ...

Grâce à toute l’équipe, Thomas HÉRAULT en tête, j’ai pu acquérir et approfondir mes connaissances dans des domaines variés :

- Les systèmes de communication entre *clusters* et grilles,
- Les problématiques de tolérance aux fautes dans ce cadre,

²<http://www.lrde.epita.fr>

- Les outils utilisés pour mesurer les performances réseaux,
- Les spécificités de TCP/IP,
- La programmation à l'aide de *threads*,
- L'utilisation des Autotools,
- L'administration et la configuration de GNU/LINUX et de son noyau,
- Culture générale informatique,
- Rédaction de document en L^AT_EX, ...

Conclusion

De nos jours, beaucoup de conjecture ou de problèmes mathématiques doivent être résolus par des calculs lourds et complexes. Par exemple, trouver une solution exacte à un problème d'optimisation combinatoire ne peut déceimment pas être effectué sur un ordinateur standard.

Le calcul distribué s'impose comme réponse à ce problème de manque de puissance. Dans ce cadre, le développement de bibliothèques performantes de passage de message est un point essentiel puisque permettant la communication efficace entre un grand nombre de machine.

Dans ce document, il a pu être entrevue les problématiques liées à la conception de telles bibliothèques dans le cadre d'un stage effectué au LRI. Il s'agissait d'étudier les possibilités d'implémentation en regard à l'utilisation de *threads* dans une bibliothèque de passage de message tolérante aux pannes. Était pris comme base le projet MPICH-V, production de l'équipe GRAND LARGE.

Diverses implémentations de MPICH-V ont été réalisées et comparées pour déterminer quelle était le meilleur choix possible quant aux *threads* ; nous avons pu constater que ce choix dépend de l'utilisation faite de la bibliothèque, et qu'aucune implémentation ne domine les autres dans tous les cas.

La production de ce stage s'inscrit dans un travail de recherche initié par l'équipe GRAND LARGE dans le domaine des communications sur grilles, et a su trouver sa place au sein du projet européen CoreGRID regroupant les pôles de recherche dans ce domaine.

Liste des acronymes

MPI..... Message Passing Interface Protocole de communication
entre noeuds conduisant un calcul ou un processus parallèle.

SVN SubVersioN Système de contrôle de version.

Glossaire

bande passante C'est la capacité d'un réseau à transmettre des informations. Elle s'exprime généralement en bits par seconde. Plus la capacité est grande, plus la quantité d'informations transmise par seconde est grande.

bibliothèques Ensemble de fonctionnalités stockées sous la forme d'un seul et même fichier. Le terme anglais est «library», souvent traduit à tort «librairie».

ChangeLog Fichier référençant tous les changements effectués sur les sources d'un projet, fichier par fichier. Chaque entrée du ChangeLog devrait correspondre à une modification précise. L'entrée la plus récente se trouve en haut du ChangeLog.

checkpoint Sauvegarde de l'image d'un processus à un instant donné. Toutes les données en mémoire du processus sont en particulier sauvés.

cluster On parle de grappe de serveurs ou de ferme de calcul (*cluster* en anglais) pour désigner des techniques consistant à regrouper plusieurs ordinateurs indépendants (appelés noeuds) pour permettre une gestion globale et dépasser les limitations d'un ordinateur en terme de disponibilité, de montée en charge et de gestion des ressources.

full-duplex Une carte, au sens matériel du terme, est dite *full-duplex* si elle peut émettre et recevoir en même temps. Ce peut-être aussi bien une carte son qu'une carte réseau, ou autre.

grille Infrastructure matérielle, ou plus rarement logicielle, permettant le partage de ressource coordonné à l'intérieur d'une organisation dynamique de machines ou d'ensemble de ressource.

latence Délai entre l'émission et la réception d'un paquet.

multi-threading Exécution d'un processus à l'aide de plusieurs *threads*, par opposition à *mono-thread*.

mutex Les algorithmes d'exclusion mutuelle sont utilisés dans la programmation avec concurrence pour permettre l'utilisation simultanée de ressources non-partageables dans des parties appelées «sections critiques».

noeuds Un noeuds est un ordinateur au sein d'un ensemble de machine mises en réseau.

ping-pong Schéma de communication réseau où un message rebondit entre deux processus. Il permet principalement la mesure de la bande passante et de la latence.

port Correspondant à la couche session du modèle OSI, la notion de port logiciel permet, sur un ordinateur donné, de distinguer différents interlocuteurs. Ces interlocuteurs sont des programmes informatiques qui, selon les cas, écoutent ou émettent des informations.

race condition Problème dans un système ou processus où la sortie dépend de manière inattendue du l'agencement temporel des évènements. Aussi appelé «*race hazard*».

swap Lorsque toute la mémoire vive (RAM) est remplie les systèmes d'exploitation, simulent de la mémoire vive sur le disque dur : on dit qu'ils *swappent* car ils échangent (*to swap* en anglais) des informations entre la mémoire vive et le disque dur. Aussi appelé «mémoire virtuelle».

TCP/IP La suite des protocoles Internet est l'ensemble des protocoles qui constituent la pile de protocoles utilisée par Internet. Elle est souvent appelée TCP/IP, d'après le nom de deux de ses protocoles : TCP (Transmission Control Protocol) et IP (Internet Protocol), qui ont été les premiers à être définis. Le document de référence sur ce sujet est le RFC 1122.

thread Les processus légers (en anglais, *thread*) sont similaires aux processus en cela qu'ils représentent tous deux l'exécution d'un ensemble d'instructions du langage machine d'un processeur. Du point de vue de l'utilisateur ces exécutions semblent se dérouler en parallèle. Toutefois là où chaque processus possède sa propre mémoire virtuelle, les processus légers appartenant au même processus père partagent une même partie de sa mémoire virtuelle.

Table des figures

1.1	Organisation du LRI	9
2.1	Communication noeuds à noeuds	14
2.2	Communication hiérarchique - MPICH-V	15
3.1	MPICH-V : Un seul CM, deux clients	22
3.2	Message envoyé via MPICH-V	23

Bibliographie

- [1] George Bosilca, Aurelien Bouteiller, Franck Cappello, Samir Djilali, Gilles Fedak, Cécile Germain, Thomas Héroult, Pierre Lemarinier, Oleg Lodygensky, Frédéric Magniette, Vincent Néri, and Anton Selikhov. Mpich-v : toward a scalable fault tolerant mpi for volatile nodes. In *SC*, pages 1–18, 2002.
- [2] Michael Cadilhac, Thomas Herault, and Pierre Lemarinier. Message Relaying Techniques for Computational Grids and their Relations to Fault Tolerant Message Passing for the Grid. *To be published in Coregrid Research Reports*, 2006.
- [3] M. Snir, S. Otto, S. Huss-Lederman, D. Walker, and J. Dongarra. *MPI : The Complete Reference*. The MIT Press, 1996.

Annexes

ChangeLog du projet

```

2005-12-23 Michael Cadilhac <michael.cadilhac@lrde.epita.fr>
Fix POLL race condition in GENE.
* src/cm_gene/loop.c: Divide poll usage of pfd's between threads.

2005-12-21 Michael Cadilhac <michael.cadilhac@lrde.epita.fr>
Fix ITPC.
* src/cm_itpc/loop.c, src/cm_itpc/act_message.c,
* src/cm_itpc/act_request.c: Fix deadlocks in ITPC.
* src/client/cm_connect_to_cm.c: Fix a warning.

2005-12-21 Michael Cadilhac <michael.cadilhac@lrde.epita.fr>
Minor fixes.
* src/cm_itpc/outgoing.h: Use a condition instead of signals.
* src/cm_itpc/incoming.c: Remove useless call to 'send-pending-message'.
* src/cm_itpc/init.c, src/cm_itpc/loop.c,
* src/cm_itpc/types.h: Remove signal handling, use a condition.
* src/cm_itpc/act_message.c: Use it.
* src/cm_itpc/outgoing.c: Use it.
* src/cm_spec/outgoing.h, src/cm_spec/init.c,
* src/cm_spec/act_message.c, src/cm_spec/loop_recv.c,
* src/cm_gene/act_message.c: Thread related fixes.

2005-12-20 Michael Cadilhac <michael.cadilhac@lrde.epita.fr>
Add a check for atomic_* functions.
* configure.ac: Prepare a check for atomic_* functions. Not installed
because the dev. computer doesn't have them.

2005-12-12 Michael Cadilhac <michael.cadilhac@lrde.epita.fr>
Use kill(2) instead of the non-portable pthread_kill_other_threads(3).
* src/cm_itpc/loop.c, src/cm_spec/loop_recv.c,
* src/cm_gene/loop.c: Update.

2005-12-12 Michael Cadilhac <michael.cadilhac@lrde.epita.fr>
Add --quit-on-noclient option to CMs and fix minor bugs.
* src/launcher.sh: kill -9 -1 when processes die.
* src/gdx-deploy.sh: New. Deployer for GDX, call launcher.sh.
* src/Makefile.am: Update accordingly.
* src/cm_itpc/cm.h, src/cm_itpc/loop.c, src/cm_itpc/main.c,
* src/cm_spec/cm.h, src/cm_spec/loop.h, src/cm_spec/loop_acc.c,
* src/cm_spec/main.c, src/cm_spec/loop_recv.c, src/cm_mthd/cm.h,
* src/cm_mthd/loop.c, src/cm_mthd/main.c, src/cm_gene/cm.h,

```

```

* src/cm_gene/loop.c,
* src/cm_gene/main.c: Add a --quit-on-noclient option that quit when
all clients have disconnected.

2005-12-08 Michael Cadilhac <michael.cadilhac@lrde.epita.fr>

Minor fixes and improvements.

* src/launcher.sh: Fix bugs, add a possibility to set options for
the client.
* src/cm_spec/loop_acc.c: Set the connexions as non blocking.
* src/cm_mthd/loop.c: Fix indexes problems.
* src/cm_gene/init.c: Include <pthread.h>.

2005-12-06 Michael Cadilhac <michael.cadilhac@lrde.epita.fr>

Make the message logging optional.

* configure.ac: Add a '--enable-logging' command.
* src/common/common.h: Include <config.h> inconditional.
* src/Makefile.am: Add 'launcher.sh' in extra dist files.

* src/cm_itpc/init.c, src/cm_itpc/incoming.c, src/cm_itpc/actions.h,
* src/cm_itpc/types.h, src/cm_itpc/act_message.c,
* src/cm_itpc/act_checkpoint.c, src/cm_itpc/act_request.c,
* src/cm_itpc/act_start.c, src/cm_itpc/outgoing.c, src/cm_spec/init.c,
* src/cm_spec/incoming.c, src/cm_spec/actions.h, src/cm_spec/types.h,
* src/cm_spec/act_message.c, src/cm_spec/act_checkpoint.c,
* src/cm_spec/act_request.c, src/cm_spec/act_start.c,
* src/cm_spec/outgoing.c, src/cm_mthd/init.c, src/cm_mthd/incoming.c,
* src/cm_mthd/actions.h, src/cm_mthd/loop.c, src/cm_mthd/types.h,
* src/cm_mthd/act_message.c, src/cm_mthd/act_checkpoint.c,
* src/cm_mthd/act_request.c, src/cm_mthd/act_start.c,
* src/cm_mthd/outgoing.c, src/cm_gene/init.c, src/cm_gene/incoming.c,
* src/cm_gene/actions.h, src/cm_gene/types.h, src/cm_gene/act_message.c,
* src/cm_gene/act_checkpoint.c, src/cm_gene/act_request.c,
* src/cm_gene/act_start.c,
* src/cm_gene/outgoing.c: Make the logging optional.

2005-11-30 Michael Cadilhac <michael.cadilhac@lrde.epita.fr>

Remove stdin treatment and minor fixes.

* src/cm_itpc/init.c: Use 'pthread_mutex_init' instead of 'memset'.
* src/cm_spec/init.c: Spec should have a pfd's of '2 * max_clients'.
* src/cm_gene/init.c: Use 'pthread_mutex_init' instead of 'memset'.
* src/cm_mthd/loop.c,
* src/cm_gene/loop.c: Remove stdin treatment.

2005-11-29 Michael Cadilhac <michael.cadilhac@lrde.epita.fr>

Add a lighth deployer.

* src/launcher.sh: New. Light deployer. See ./launcher.sh --help.

2005-11-29 Michael Cadilhac <michael.cadilhac@lrde.epita.fr>

Change ignore properties.

* ., config, src, src/cm_itpc, src/common,
* src/cm_spec,src/cm_mthd, src/cm_gene: Change property.

2005-11-28 Michael Cadilhac <michael.cadilhac@lrde.epita.fr>

Minor bug fixes in mutex management in SPEC.

* src/cm_spec/init.c,
* src/cm_spec/cm.h: Move 'send_list_*' to...
* src/cm_spec/outgoing.c,
* src/cm_spec/outgoing.h: ...here, change 'send_list_mutex'
to 'send_list_cond_mutex' and add a 'send_list_mutex'.
* src/cm_spec/loop_send.c: Use them.

2005-11-28 Michael Cadilhac <michael.cadilhac@lrde.epita.fr>

Minor bug fixes in SPEC.

```

```

* src/cm_spec/loop_acc.c: Remove useless newline.
* src/cm_spec/loop_send.c: Lock the send list when calling
'send_pending_message'.
* src/cm_spec/outgoing.c: Manage to have the send list locked
when used.

2005-11-28 Michael Cadilhac <michael.cadilhac@lrde.epita.fr>

Minor bug fixes on the SPEC version.

* src/cm_spec/types.h: Remove 'thread' as a field of 'process_t'.
* src/cm_spec/outgoing.h,
* src/cm_spec/outgoing.c: Add 'ask_for_sending'.
* src/cm_spec/act_message.c: Use it.
* src/cm_spec/main.c: Add '-r, --recv-threads' to the list of option
in the help.

2005-11-25 Michael Cadilhac <michael.cadilhac@lrde.epita.fr>

Add ugly print for Thomas.

* src/cm_ttpc/init.c, src/cm_spec/init.c, src/cm_mthd/init.c,
* src/cm_gene/init.c: Print the port on which the CM is bound on
the stdout.

2005-11-25 Michael Cadilhac <michael.cadilhac@lrde.epita.fr>

Tiny bug fixes.

* src/cm_ttpc/act_message.c: 'pthread_kill' should be called at each
reception.
* src/cm_spec/act_message.c: s/ifdef/ifndef/.
* src/cm_gene/init.c: Fix indentation problem.
* src/cm_gene/act_start.c: Bug fix: use 'old_proc'.

2005-11-25 Michael Cadilhac <michael.cadilhac@lrde.epita.fr>

Add a Makefile part for sanity check.

* src/Makefile.check: New. Common sanity check.

2005-11-24 Michael Cadilhac <michael.cadilhac@lrde.epita.fr>

Add the client. This directory should be put, or more likely
ln(1)'ed in/as mpich-1.2.7/mpid/ch_cm.

* src/client/cm.h, src/client/cm_types.h, src/client/cm_time.c,
* src/client/cm_init.c, src/client/cm_complete_ops.h,
* src/client/cm_connect_to_cm.c, src/client/cm_api.c,
* src/client/cm_final.c: New. CM specific files.

* src/client, src/client/mpid.h, src/client/bswap2.c,
* src/client/packets.h, src/client/ldtest.c, src/client/calltrace.h,
* src/client/comm.h, src/client/Makefile.in, src/client/chnrndv.c,
* src/client/chsend.c, src/client/adi2probe.c, src/client/chttime.c,
* src/client/channel.h, src/client/chpackflow.c, src/client/chflow.c,
* src/client/comments.txt, src/client/adi2hssend.c,
* src/client/adi2cancel.c, src/client/chpackflow.h,
* src/client/adi2req.c, src/client/mpiddevbase.h, src/client/send.c,
* src/client/cmnargs.c, src/client/Makefile, src/client/cmpriv.c,
* src/client/adi2hrecv.c, src/client/tr2.c, src/client/adi2ssend.c,
* src/client/mpid_time.h, src/client/flow.h, src/client/cookie.h,
* src/client/adi2pack.c, src/client/datatype.h, src/client/chnrndv.c,
* src/client/chconfig.h, src/client/design, src/client/req.h,
* src/client/adi2recv.c, src/client/chhetero.c, src/client/mpid_debug.h,
* src/client/chtick.c, src/client/chhetero.h, src/client/mprerr.c,
* src/client/attach.h, src/client/chshort.c, src/client/chstartdb.c,
* src/client/sbcnst2.c, src/client/chchkdev.c, src/client/shmemdebug.c,
* src/client/sbcnst2.h, src/client/objtrace.c, src/client/adi2mpack.c,
* src/client/mpiddev.h, src/client/objtrace.h, src/client/chbeager.c,
* src/client/chcancel.c, src/client/chnodename.c,
* src/client/adi2hssend.c, src/client/chdebug.c, src/client/adi2init.c,
* src/client/chlocal.c, src/client/reqalloc.h, src/client/ch_ldebug.c,
* src/client/chbeager.c, src/client/README, src/client/chinit.c,
* src/client/chdef.h, src/client/chprobe.c, src/client/adi2mpass.c,
* src/client/dev.h, src/client/queue.c, src/client/chbrndv.c,
* src/client/adi2send.c, src/client/mpimem.h, src/client/calltrace.c,

```

```

* src/client/chget.c, src/client/mpid_bind.h,
* src/client/chevent.c: New. MPICH-1.2.7 copied files.

2005-11-22 Michael Cadilhac <michael.cadilhac@lrde.epita.fr>

Fix compilation details for GCC 4.x.

* src/cm_itpc/Makefile.am, src/cm_itpc/outgoing.c,
* src/cm_spec/loop_acc.c, src/cm_spec/main.c,
* src/cm_spec/Makefile.am, src/cm_spec/outgoing.c,
* src/cm_mthd/Makefile.am, src/cm_mthd/outgoing.c,
* src/cm_gene/Makefile.am,
* src/cm_gene/outgoing.c: Make this compile with GCC 4.x.

* src/Makefile.am: Add the client as a tarball.

2005-11-15 Michael Cadilhac <michael.cadilhac@lrde.epita.fr>

Fix autotools detail.

* configure.ac,
* src/Makefile.am: s/cm_mthr/cm_mthd/.

2005-11-15 Michael Cadilhac <michael.cadilhac@lrde.epita.fr>

Minor fixes in the common part.

* src/common/list.h: Add some invariant test, and fix some bugs.
* src/common/list.h (Prefix##_list_delete_list): Add this function.
* src/common/contract.h: Add tabulations, and make 'Terminate'
a function.
* src/common/message_id.h: No need for ACK.
* src/common/verbose.h: VerboseVar should be set to the name
of the global variable telling if we're verbose.
* src/common/common.h: Add 'when' and 'unless'.

2005-11-15 Michael Cadilhac <michael.cadilhac@lrde.epita.fr>

Autotools minor fixes.

* configure.ac: Add Makefile for other versions of CM.
* config/Makefile.am: Add 'compile'.

2005-11-15 Michael Cadilhac <michael.cadilhac@lrde.epita.fr>

Add three other versions of the CM.

* src/cm_itpc, src/cm_itpc/send.h, src/cm_itpc/outgoing.h,
* src/cm_itpc/incoming.c, src/cm_itpc/init.c, src/cm_itpc/cm.h,
* src/cm_itpc/actions.h, src/cm_itpc/incoming.h, src/cm_itpc/init.h,
* src/cm_itpc/loop.c, src/cm_itpc/types.h,
* src/cm_itpc/act_message.c, src/cm_itpc/act_checkpoint.c,
* src/cm_itpc/loop.h, src/cm_itpc/act_request.c,
* src/cm_itpc/act_start.c, src/cm_itpc/main.c,
* src/cm_itpc/Makefile.am, src/cm_itpc/send.c,
* src/cm_itpc/outgoing.c: New. Version ITPC (One Thread Per Client).

* src/cm_gene, src/cm_gene/send.h, src/cm_gene/outgoing.h,
* src/cm_gene/incoming.c, src/cm_gene/init.c, src/cm_gene/cm.h,
* src/cm_gene/actions.h, src/cm_gene/incoming.h, src/cm_gene/init.h,
* src/cm_gene/loop.c, src/cm_gene/types.h,
* src/cm_gene/act_message.c, src/cm_gene/act_checkpoint.c,
* src/cm_gene/loop.h, src/cm_gene/act_request.c,
* src/cm_gene/act_start.c, src/cm_gene/main.c,
* src/cm_gene/Makefile.am, src/cm_gene/send.c,
* src/cm_gene/outgoing.c: New. Version GENE (GENERALISATION of the
first one).

* src/cm_spec, src/cm_spec/send.h, src/cm_spec/outgoing.h,
* src/cm_spec/incoming.c, src/cm_spec/init.c, src/cm_spec/cm.h,
* src/cm_spec/actions.h, src/cm_spec/incoming.h, src/cm_spec/init.h,
* src/cm_spec/types.h, src/cm_spec/act_message.c,
* src/cm_spec/act_checkpoint.c, src/cm_spec/loop.h,
* src/cm_spec/act_request.c, src/cm_spec/loop_acc.c,
* src/cm_spec/loop_send.c, src/cm_spec/main.c,
* src/cm_spec/act_start.c, src/cm_spec/loop_recv.c,
* src/cm_spec/Makefile.am, src/cm_spec/send.c,

```

```

* src/cm_spec/outgoing.c: New. Version SPEC (One or more SPECialised
thread for each role).

* src/cm: Move to...
* src/cm_mthd: New. ... Here.

* src/Makefile.am: Update accordingly.

2005-11-14 Michael Cadilhac <michael.cadilhac@lrde.epita.fr>

End first version.

* src/cm/types.h, src/cm/act_message.c, src/cm/act_checkpoint.c,
* src/cm/act_request.c, src/cm/act_start.c, src/cm/main.c,
* src/cm/Makefile.am, src/cm/send.c, src/cm/outgoing.c: Adjust.

2005-10-03 Michael Cadilhac <michael.cadilhac@lrde.epita.fr>

Fix 'psend' function.

* src/cm/send.c: Fix small bug in 'psend'. Don't abort when errno
is EAGAIN.

2005-10-03 Michael Cadilhac <michael.cadilhac@lrde.epita.fr>

Complete CM creation.

* configure.ac: Fix --enable-verbose, add --enable-pipeline.

* AUTHORS, NEWS, README, config,
* config/Makefile.am: New. Distro files.

* src/cm/send.h, src/cm/outgoing.h, src/cm/init.c,
* src/cm/incoming.c, src/cm/cm.h, src/cm/actions.h,
* src/cm/incoming.h, src/cm/init.h, src/cm/loop.c,
* src/cm/types.h, src/cm/act_message.c, src/cm/loop.h,
* src/cm/act_checkpoint.c, src/cm/act_request.c, src/cm/act_start.c,
* src/cm/main.c, src/cm/Makefile.am, src/cm/send.c,
* src/cm/outgoing.c: New (or not). act_* are actions files (received
messages treatments), the others are self-explained.

* src/common/list.h: New. Pseudo circular list library.
* src/common/contract.h: Adjuste.
* src/common/message_id.h: New. Message ids.
* src/common/verbose.h: Adjuste.
* src/common/dummy.c: New. Dummy C file to have $(COMPILE) fonctionnal.
* src/common/common.h: Adjuste.
* src/common/Makefile.am: Adjuste.

* Makefile.am: Add 'config' sub-directory.

2005-09-22 Michael Cadilhac <michael.cadilhac@lrde.epita.fr>

First draft for the cm.

* src: New. Source directory.
* src/cm: New. CM's sources directory.
* src/common: New. Common sources directory.

* configure.ac, bootstrap.sh, Makefile.am,
* src/common/Makefile.am,
* src/Makefile.am,
* src/cm/Makefile.am: New. Autotools things.

* src/cm/init.c,
* src/cm/init.h: New. Initialisation, binding.

* src/cm/cm.h: New. Main header file.
* src/cm/main.c: New. Option parsing and main.

* src/cm/loop.h,
* src/cm/loop.c: New. Main loop.

* src/common/contract.h: New. assert(3) replacement.
* src/common/verbose.h: New. Verbosity functions.
* src/common/common.h: New. Other common definitions.

```