SCHOONER : une encapsulation orientée objet de supports d'exécution pour applications réparties

Nathalie Furmento

Université de Nice - Sophia Antipolis
École doctorale - Sciences pour l'Ingénieur

Thèse

présentée pour obtenir le titre de

Docteur en Sciences
Spécialité : Informatique

par

Nathalie Furmento

Schooner : Une Encapsulation Orientée Objet de Supports d'Exécution pour Applications Réparties

soutenue le 10 Mai 1999 devant la commission composée de :

Président	Jean-Paul Rigault	Professeur à l'UNSA
Rapporteurs	Jean-Marc Geib	Professeur à l'Université de Lille (LIFL)
	Jean-François Méhaut	Maître de Conférences à l'Université de Lille (LIFL)
	Bernard Vauquelin	Professeur à l'Université de Bordeaux I (LaBRI)
Examinateurs	Anne-Marie Pinna	Maître de Conférences à l'UNSA
Directeurs	Françoise Baude-Dreysse	Maître de Conférences à l'UNSA (I3S-INRIA)
	Jean-Claude Bermond	Directeur de Recherche du CNRS (I3S-INRIA)

Remerciements

Je tiens par ces quelques lignes à exprimer ma gratitude à toutes les personnes sans qui cette thèse n'aurait sans doute jamais abouti.

- Françoise Baude, pour avoir accepté de diriger cette thèse. Tout au long des 5 années où nous avons travaillé ensemble, sa disponibilité constante, son aide et sa confiance m'ont grandement aidé à mener à bien mes travaux ;
- Jean-Claude Bermond, pour avoir accepté de co-diriger cette thèse, pour son accueil dans les projets PACOM et SLOOP, ainsi que pour les excellentes conditions de travail dont j'ai pu bénéficier durant cette thèse ;
- Jean-Paul Rigault, pour avoir accepté de présider mon jury de thèse malgré un emploi du temps surchargé ;
- Jean-François Méhaut, Jean-Marc Geib et Bernard Vauquelin, pour avoir accepté de rapporter sur cette thèse et pour leurs nombreuses remarques qui m'ont permis de compléter et d'éclaircir ce manuscrit ;
- Anne-Marie Pinna, pour avoir eu l'amabilité de participer à mon jury ;
- Günther Siegel et David Sagnol, pour avoir accepté de servir de cobayes à l'utilisation de SCHOONER. Nos collaborations se sont toujours passées dans les meilleures conditions et ont été très enrichissantes ;
- Raymond Namyst, pour son aide et son temps lors de notre collaboration ayant mené au portage de la librairie SCHOONER au dessus de PM^2 ;
- Olivier Dalle, pour avoir toujours pris le temps de répondre à mes nombreuses questions à propos du fonctionnement des couches réseau et ce jusqu'à la dernière minute ;
- Philippe Barette, Fabrice Belloncle, Alexandre Clavaud, Sylvain Gamel et Cédric Tonin, pour leurs collaborations fructueuses ;
- Philippe Mussi, pour son soutien constant, sa disponibilité et ses nombreux conseils ;
- Denis Caromel, Afonso Ferreira, Bruno Gaujal et Michel Syska, ainsi que tous les autres membres des projets PACOM et MISTRAL, pour leur gentillesse et leur disponibilité ;
- Ephie Deriche, Zohra Kalafi et Patricia Lachaume, pour avoir toujours une solution aux divers problèmes administratifs ainsi que pour leur gentillesse ;
- Tous mes collègues et amis du projet SLOOP : Bruno, David C., Emmanuel, Eric, Jean-Noël, Jérome, Julien, Laurent, Nausica, Olivier D., Peter, Stéphane, Tania et Yves, pour

la bonne humeur permanente qu'ils font ou qu'ils ont fait régner dans les couloirs du projet ;

- Besma, Didier, Fanny, Laurent et Nelly, ainsi que Cécile, Christelle, Domi, Franck, Hervé, Karim, Kathy, Laurence, Lol, Malek, Michel, Nadine, Philippe, Sabine, Sidi et Sylvie, pour les innombrables bons moments passés en leur compagnie ;
- Mes parents, mes soeurs, Ralf, ainsi que toute ma famille, pour leur aide et leur affection constante ;

Table des matières

Chapitre 1

Introduction

1.1 Motivations

De plus en plus de concepteurs d'applications nécessitant un volume important de calcul et de données se tournent vers les techniques du parallélisme, en les adaptant au type de matériel en vogue actuellement, schématisé par un ensemble de nœuds connectés par un réseau. Ainsi de nombreux environnements permettant l'exécution d'applications réparties adaptées à ce type de matériel, ont vu le jour. Toutefois la plupart d'entre eux rendent trop dépendante l'application du support fourni en ne permettant pas d'isoler correctement le code lié à la gestion du support d'exécution du code propre à l'application. Il est ainsi difficile de concevoir des applications portables et tout changement du support d'exécution entraîne des modifications plus ou moins lourdes dans l'application.

La gestion de la multi-activité est aussi un point non-consensuel parmi les supports existants. De ce fait, l'application va se trouver fortement liée à une politique d'ordonnancement et ne pourra en changer ou s'en abstraire qu'en adaptant son propre code.

Il apparaît donc qu'il soit nécessaire de fournir un environnement fournissant un ensemble minimal de fonctionnalités pour la gestion du support d'exécution d'applications réparties. Cet environnement, en étant minimal, garantit en partie la portabilité des applications qui sont manipulées de manière abstraite et structurée.

On a également pu constater ces dernières années une utilisation de plus en plus répandue de la programmation à objets du fait principalement de ses indéniables qualités en génie logiciel. La programmation à objets peut donc permettre de résoudre les problèmes évoqués ci-dessus.

Le problème posé est donc de concevoir un environnement impliquant de par son utilisation

une portabilité complète des applications réparties vis-à-vis du support d'exécution fourni. Ce support se doit d'être indépendant du média de communication utilisé, de donner une vision structurée et abstraite du parc de machines et des nœuds de l'application, tout en tirant bénéfice des propriétés offertes par de la programmation orientée objet.

1.2 Orientations

Le travail que nous présentons dans ce mémoire porte sur l'étude et la conception d'un environnement orienté objet pour la programmation d'applications réparties[1]. Cet environnement est principalement constitué d'un support d'exécution pour ces applications, de telle sorte à fournir à l'utilisateur une interface de programmation facilement adaptable au type d'application désirée.

Le support d'exécution offre ainsi une interface vers les concepts de base pour supporter tout type d'application concurrente répartie, en donnant libre choix à l'utilisateur pour des besoins plus spécifiques. Il sera ainsi possible de programmer une politique d'ordonnancement dédiée à une classe d'applications ou de modifier les routines d'aplatissement et de reconstruction des données en vue d'obtenir un recouvrement entre le calcul et les communications. Il est également possible de changer facilement la couche bas-niveau de communication ou à l'opposé le modèle de calcul fourni à l'utilisateur final.

Ces recherches se sont déroulées sous la direction de Françoise BAUDE-DREYSSE, tout d'abord dans le thème PaCoM du laboratoire Informatique, Signaux et Systèmes (I3S, CNRS-UNSA), puis dans le projet SLOOP (projet commun CNRS-INRIA-UNSA). L'objectif central de ce projet est le développement de méthodes et outils permettant l'utilisation efficace de machines multiprocesseurs pour la simulation à événements discrets. Ces méthodes et outils sont construits au dessus de langages à objets répartis, eux-mêmes requérant l'utilisation d'une plate-forme de communication efficace. Notre travail de thèse s'intègre donc parfaitement dans les axes de recherche du projet.

1.3 Organisation du mémoire

Ce mémoire est organisé en 3 grandes parties :

- la première partie est consacrée à l'état de l'art et s'articule autour de trois grands points. Nous nous attachons d'abord à décrire le contexte de notre travail et à définir les besoins qui en découlent : *les supports d'exécution pour applications réparties*. Une

[1] Le mot "distribué" est souvent employé dans la littérature dans le même sens que "réparti", comme traduction du mot anglais "distributed".

deuxième étape consiste alors en un tour d'horizon des solutions existantes dans ce domaine, ce qui nous amène pour finir à dégager de cette étude un modèle novateur de support d'exécution pour applications réparties.

- la deuxième partie va décrire en détail le modèle que nous proposons sous la forme de la bibliothèque de classes SCHOONER. Dans un premier chapitre, le modèle de base est décrit via son interface de programmation et d'utilisation. Un modèle étendu à la multi-activité est ensuite présenté. Après une rapide présentation de l'implémentation et des performances de la bibliothèque, l'environnement fourni en complément de SCHOONER est présenté. Il est constitué d'un ensemble d'outils d'aide à la programmation ainsi que de redéfinitions du comportement de base en vue d'améliorer les performances pour certains types d'applications.
- la dernière partie montre des exemples d'utilisation de cet environnement et permet ainsi de valider les choix effectués. Le premier chapitre décrit une version répartie d'un simulateur à événements discrets, PROSIT, ainsi qu'un client serveur hiérarchique. Le deuxième chapitre décrit une implémentation du modèle C++// au dessus de la bibliothèque SCHOONER étendue à la multi-activité. Cette application nous permettra également de proposer une technique de recouvrement des communications par du calcul.

Nous terminons ce mémoire en récapitulant les contributions principales de cette thèse et en présentant les perspectives de nos travaux.

Première partie

Cadre du travail et définition des objectifs

Chapitre 2

Contexte du travail et expression des besoins : Supports d'exécution pour applications réparties

---

✔ *Nous exposons dans ce chapitre le cadre général de notre travail, en décrivant les types d'applications qui nous intéressent et quelles doivent être les spécifications des supports permettant l'exécution de ces applications.*

2.1 Contexte général

Cette section tente de définir le cadre général de notre recherche. Celui-ci peut être exprimé en répondant aux questions suivantes : pourquoi les applications visées sont parallèles et réparties, de quels supports dispose-t-on pour leur mise en œuvre ?

Pourquoi calculer en parallèle. Malgré l'incessante augmentation des performances des stations de travail, résoudre plus rapidement certains problèmes nécessite de faire appel aux techniques du parallélisme, en particulier à celle de l'algorithmique parallèle. L'idée générale consiste à découper l'application en vue de répartir calcul et données sur plusieurs calculateurs en faisant en sorte que ces calculs s'exécutent en parallèle.

Les sources de parallélisme. Il est bien connu qu'il existe deux grandes formes de parallélisme : celui appelé couramment *de tâche* ou encore *de contrôle, de situation* et celui appelé *de données* ou encore *de résolution*. Au sein d'une même application, ces 2 sources peuvent même être conjointement exploitées.

Le parallélisme de données est généralement massif car il correspond à un traitement — souvent assez simple — à réaliser sur une quantité massive de données de même nature. Les applications exploitant un tel parallélisme sont qualifiées de SPMD (Single Program, Multiple Data). Dans le cas où les structures de données sont irrégulières, ce parallélisme de données est de plus qualifié d'*irrégulier* [Stratagème 97]. Notamment dans ce cas, la décomposition de l'application en vue d'exploiter le parallélisme génère un ensemble de tâches concurrentes à grain plus ou moins fin, éventuellement changeant dynamiquement, et plus ou moins synchronisées selon le degré d'irrégularité. Les données sont soit réparties sur l'ensemble des tâches, soit mises dans un espace commun accessible depuis n'importe quelle tâche. Mais sur un support d'exécution réparti, l'espace n'est que virtuellement commun et de façon sous-jacente, les données sont réparties parmi les tâches qui doivent se les échanger si besoin.

Extraire le parallélisme de contrôle quant à lui engendre une décomposition de l'application en tâches bien différentes, souvent faiblement synchronisées, et dont le grain est gros, voire de plus, variable. Les données peuvent être propres à une tâche, ou bien devant transiter de tâche en tâche afin d'y être traitées. Le modèle d'application correspondant est le modèle MPMD (Multiple Program, Multiple Data).

En résumé, quelles que soient les sources de parallélisme exploitées, une application en vue de son exécution sur un support physique réparti, peut être modélisée comme un graphe de tâches concurrentes, nécessitant d'interagir pour s'échanger des données.

Les supports matériels. Après l'engouement d'il y a maintenant plus d'une décennie pour les machines parallèles massives, depuis quelques années l'usage courant est de leur préférer un réseau de stations de travail. En effet, sauf dans des domaines d'application très spécifiques (par exemple, la prévision météorologique), rares sont les domaines où suffisamment d'applications présentent un potentiel de parallélisme suffisant pour justifier un investissement spécifique en matériel.

Ainsi, le support le plus courant aujourd'hui pour les applications parallèles est un ensemble de stations de travail reliées par un réseau de communication. Nous ferons donc l'hypothèse dans toute la suite de ce mémoire, que **le support physique ne présente pas de mémoire physique commune, ni de référentiel de temps commun**. Si par hasard nous sommes en présence d'une station de travail disposant de plus d'un processeur, ceux-ci communiquant par l'intermédiaire d'une mémoire physique commune, nous n'en tiendrons pas explicitement compte : même si le protocole de communication entre entités localisées sur des processeurs différents fait lui usage de la mémoire commune, nous considérons que ces entités communiquent

par échange de messages et non pas par partage de mémoire.

Utiliser un réseau de stations de travail comme une machine parallèle à mémoire répartie suppose d'avoir accès à chacune des stations : possibilité d'y lancer des calculs, d'y stocker des données, ... Bien que l'on puisse concevoir de calculer de façon répartie sur une immense machine parallèle constituée de plusieurs milliers de stations de travail reliées par l'Internet, il faut en général disposer d'un compte utilisateur sur chacune d'elles.

Ainsi les grappes de stations de travail sont le plus souvent constituées autour d'un réseau local. Ceci n'a pas que des inconvénients à l'heure actuelle, car ces réseaux sont couramment des réseaux à haut débit et de plus utilisent un faible nombre de passerelles — comparé à celui qu'il peut être nécessaire de traverser pour atteindre une machine quelconque au travers de l'Internet. Dans le cas où l'on choisirait d'utiliser l'Internet, la lenteur des communications peut alors justifier certaines optimisations, telle celle consistant à recouvrir une partie des communications par du calcul. C'est notamment une telle technique que nous avons étudiée et mise en œuvre dans un contexte de langage orienté objet. Mais il n'est pas encore temps d'en parler ...

Les supports logiciels. Les matériels décrits plus haut sont équipés de systèmes d'exploitation multi-utilisateurs, multiprocessus, tels par exemple ceux appartenant à la famille UNIX [Stevens 90]. L'existence, au sein même de cette famille, de diverses variantes implique une hétérogénéité du support logiciel et par là même le besoin d'utiliser ou de concevoir des supports d'exécution pour les applications qui soient portables au dessus de n'importe quel système d'exploitation. Notons que la présence — pour ainsi dire universelle — au sein du noyau de la pile de protocoles TCP/IP et d'une interface de programmation sockets sont des éléments positifs en ce sens. Concernant la présence de processus légers s'exécutant en mode privilégié ou tout simplement de bibliothèque de processus légers s'exécutant en mode utilisateur, l'universalité est moins évidente bien qu'un consensus se dégage autour de la norme POSIX 1003.c[1] [IEEE 96].

Comment caractériser les applications devant s'exécuter sur des réseaux de stations de travail. Un point commun aux applications s'exécutant sur un réseau de stations de travail est le caractère réparti. Si le souci majeur est d'exploiter du parallélisme, ces calculs répartis doivent être judicieusement placés sur les processeurs cibles et des techniques de répartition de charge sont donc nécessaires. Si le souci est d'arriver à effectuer un calcul portant sur une telle quantité de données qu'il serait impossible de les faire toutes tenir dans la mémoire d'une seule station de travail, la raison du caractère réparti est la répartition des données puis de ce fait des calculs. Finalement, une raison guidant la conception d'un calcul réparti, autre que la recherche de performance ou la répartition d'une masse énorme de

[1] Dans la suite de ce mémoire, nous emploierons également le terme *thread* pour désigner un *processus léger*.

données, peut être d'ordre "géographique" : par nature certaines stations de travail peuvent effectuer certaines parties spécialisées du calcul (par exemple, graphiques), alors que d'autres non. Parce que nos travaux sont en mesure d'intéresser n'importe laquelle des catégories citées, nous abandonnerons désormais le terme "applications parallèles et réparties", au profit de celui plus fédérateur de "applications réparties".

2.2 Problématique abordée : supports d'exécution pour applications réparties

A présent que le décor général de notre travail est planté, nous allons nous intéresser aux moyens d'expression des applications visées et surtout aux supports d'exécution envisageables.

Outils d'expression d'applications réparties. Les applications visées étant modélisées comme un ensemble de tâches pour ce qui est du traitement, et d'un ensemble de données à répartir sur ou à côté de ces tâches, il s'avère naturel de regrouper traitements et données au sein d'entités, pouvant être vues comme des capsules. Un paradigme particulièrement adéquat dans ce cas est le paradigme orienté objet. Ces entités qui sont amenées à interagir sont de plus réparties, mais il est reconnu que l'usage d'objets est aussi bien adapté pour la programmation répartie ([Briot 96], [Caromel 91]). Malgré ce cadre unifiant, la forme des interactions entre objets fait moins consensus. Cette interaction peut être indirecte via une mémoire répartie virtuellement partagée, orientée objet [Bal 90] ou non [Assenmacher 93a] ; sinon elle est plus directe et peut prendre la forme d'échange explicite de données (métaphore "envoi de messages" telle celle mise en œuvre dans les langages d'acteurs ou aussi [Gransart 95]) ou la forme d'appel de méthode distante. Dans cette dernière hypothèse, les données échangées le sont de façon transparente car constituant en fait les paramètres de l'appel de méthode. Dans une première approche, nous considérerons simplement que les applications visées sont décrites par des objets interagissant.

Caractéristiques générales des supports d'exécution. Afin que l'application s'exécute, chaque objet issu de la description de l'application doit se trouver supporté par une entité d'exécution. Cette entité d'exécution peut se matérialiser selon les cas par un processus lourd, par un processus léger de niveau système, par un processus léger de niveau utilisateur, ... Par ailleurs, il faut préciser sur quel calculateur du support physique les créer, en respectant d'éventuelles contraintes de placement. Lorsque les entités sont physiquement réparties, alors quelle que soit la façon d'exprimer leur interaction, celle-ci se réalise effectivement par de la transmission de messages, transmission prise en charge par le support d'exécution. Les applications considérées étant réparties, donc amenées à communiquer, il faut comprendre

le terme "support d'exécution" comme "support de calcul **et** de communication et d'entrées-sorties plus généralement".

Ainsi, un support d'exécution pour applications réparties a avant tout pour rôle de gérer tous ces éléments pour le compte de l'application (démarrage et terminaison d'entités, prise en charge de la communication, ...). On peut souhaiter en plus que cette gestion soit efficace, transparente pour l'application et permette de pouvoir utiliser n'importe quel support physique et n'importe quel système d'exploitation.

Critiques. Nous considérons les applications réparties d'un point de vue assez bas niveau, puisque c'est le support de leur exécution qui constitue l'objet de notre étude. Même si les programmeurs préféreraient décrire les applications réparties en se situant à un niveau plus élevé, en faisant ensuite appel à des traducteurs, cela n'est ni systématiquement possible, ni souhaitable lorsque la recherche d'efficacité est un critère prépondérant. Quoi qu'il en soit, nous avons pu remarquer qu'au sein d'applications utilisant un niveau aussi bas d'expressivité, **il y a souvent un manque de clarté entre le code lié à l'application proprement dite et celui relatif à la gestion de son support d'exécution**. D'autre part, la façon dont on peut gérer le support d'exécution est souvent rigide, difficilement modifiable et quelquefois fortement dépendante du support lui-même.

2.3 Objectifs et démarche

Définition de notre thèse. Nous prétendons (et allons l'argumenter tout au long de ce mémoire) qu'il peut être souhaitable d'adopter une approche orientée objet pour la gestion du support d'exécution d'applications réparties. Le résultat sera de permettre une isolation propre du code de l'application et de celui lié à la gestion de son support. Par là même, il deviendra nettement plus simple d'apporter des changements plus ou moins profonds à cette gestion sans incidence sur le code de l'application proprement dit.

Non homogénéité des supports d'exécution. Comme nous allons le détailler au cours du prochain chapitre, les supports d'exécution pour applications réparties malgré des caractéristiques générales semblables ne sont pas identiques. Outre évidemment des différences d'ordre syntaxique, les supports d'exécution présentent des différences d'ordre sémantique : en effet chacun fournit son modèle qui dépend de la nature des entités d'exécution et des moyens mis en œuvre pour leur interaction. Comme ce modèle ne peut être ignoré par le code de gestion du support, mais malheureusement parfois aussi par l'applicatif, les supports d'exécution sont rarement interchangeables.

Comment réaliser notre objectif ... Notre objectif principal est d'arriver à isoler le code de gestion du support d'exécution grâce à une encapsulation orientée objet. Bien sûr, l'encapsulation permet de gommer les différences syntaxiques que les différents supports d'exécution peuvent présenter. En outre, l'applicatif n'aura plus à se fonder sur le modèle fourni par le support d'exécution choisi, mais plutôt sur le modèle que lui présente l'encapsulation réalisée. Par ce biais, libre à nous de proposer à l'application un modèle éventuellement différent de celui du support d'exécution sur lequel elle repose effectivement. De plus, associé à ce modèle nous allons pouvoir fournir toute une batterie d'outils d'aide au développement et de mise au point du support d'exécution des applications réparties.

... et comment obtenir un portage plus large des applications. Ainsi l'approche par encapsulation que nous adoptons ouvre la voie d'un portage complet du niveau applicatif. Non seulement la tendance actuelle est que tout support d'exécution soit portable sur la plus grande gamme d'architectures (architecture au sens de la paire constituée du matériel et du système d'exploitation) mais notre démarche va permettre d'aller plus loin en proposant une encapsulation qui permette au niveau applicatif d'être portable quel que soit le support d'exécution sous-jacent.

Guide de lecture de l'état de l'art et de la conception de l'encapsulation. L'objectif du chapitre suivant est de réaliser l'état de l'art des supports d'exécution pour applications réparties, en ayant pour fil conducteur d'effectuer l'inventaire des différents modèles qu'ils proposent. Ainsi, cet inventaire nous permettra d'étayer la conception du modèle de support d'exécution que nous choisirons de mettre en œuvre dans l'encapsulation et de proposer au niveau applicatif. La conception de ce modèle se fonde essentiellement sur une synthèse des éléments communs à tous les supports d'exécution pour applications réparties[2]. Le modèle complet proposé à l'applicatif résulte alors d'une programmation qui n'a à se baser que sur cet ensemble commun et aussi minimal, de concepts. C'est donc grâce à cette démarche que le modèle est portable et par là même les applications.

Schéma de synthèse situant notre cadre de travail et nos objectifs. Grâce à l'encapsulation, les problèmes de portage sont bien isolés du niveau applicatif. De plus, l'encapsulation est le lieu idéal pour étendre, modifier, voire améliorer le comportement du support d'exécution (voir Figure 2.1).

[2] sans prétendre à l'exhaustivité bien sûr !

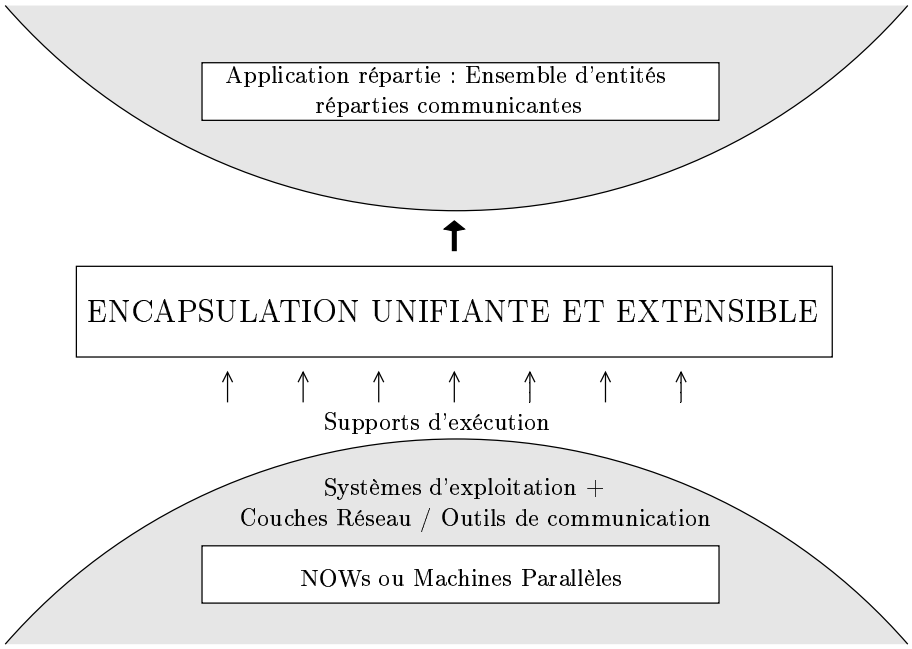

FIGURE 2.1 – *Schéma de synthèse situant notre cadre de travail.*

Chapitre 3

Supports d'exécution pour applications réparties : Etat de l'art

---

✔ *Nous étudions dans ce chapitre des exemples pertinents de supports d'exécution pour la programmation d'applications réparties. De cette étude, nous allons pouvoir dégager quels sont les éléments minimaux et nécessaires à l'exécution d'une application répartie, ce qui nous servira de base à la conception du modèle de support d'exécution présenté dans le chapitre suivant.*

3.1 Introduction

"Pour la petite histoire" ... Pour bien comprendre quel est le fil conducteur qui dirige cet état de l'art, nous suggérons que le lecteur se mette dans l'état d'esprit qui était le nôtre au début de nos travaux, qui coïncidaient avec l'élaboration scientifique du projet SLOOP. Un objectif central était d'obtenir un prototype d'un *framework* pour la simulation à événements discrets (PROSIT) qui puisse s'exécuter tout aussi bien en centralisé qu'en réparti. Les mécanismes permettant cette transparence vis-à-vis du mode d'exécution pourraient être obtenus en développant PROSIT au dessus d'une extension parallèle et répartie de C++ (C++//), dont la conception en cours reposait sur les travaux antérieurs autour d'Eiffel//. Mais rien n'empêcherait que PROSIT soit bâti en fournissant lui-même les mécanismes permettant son exécution répartie (notamment les mécanismes de réification d'appels de méthodes), c'est-à-dire sans avoir réellement besoin d'être écrit en C++//.

Schématiquement, on nous demandait d'axer notre recherche vers la conception d'un support d'exécution réparti, qui puisse satisfaire ces deux applications cibles, de façon disjointe aussi bien que conjointe ! Bien que les besoins de ces deux applications en termes de mécanismes et modèle sous-jacent pour une exécution répartie ne soient pas tout à fait identiques (nous les décrirons très brièvement ci-dessous et plus en détails lors de la présentation de chacune d'elles), aucune des deux n'était liée à un support d'exécution en particulier. Au contraire, leur conception a toujours été guidée par un souci de portabilité, à savoir pouvoir aisément changer de support d'exécution (par souci de performance ou par curiosité scientifique et technique, ...).

Description succincte des besoins des applications visées, en termes de support d'exécution. Les deux grandes applications que nous devons cibler se trouve être fondées sur la notion d'objet actif. De ce fait, le développeur doit pouvoir manipuler un objet actif sans avoir à se soucier si de manière sous-jacente, il est supporté par un processus lourd ou par un processus léger.

Au dessus de ces 2 applications, seront développées des applications finales (applications de simulation, applications coopératives) qui peuvent nécessiter un grain d'exécution plus fin que le processus lourd (car utilisant un nombre massif d'entités). Dans ce cas, il faut que de façon transparente, les 2 outils que sont PROSIT et C++// puissent faire supporter l'objet actif par un processus léger.

Ces entités sont amenées à communiquer de façon asynchrone. C'est donc au support d'exécution de fournir les mécanismes pour que soient acheminées les données d'une entité à une autre, quelle que soit leur localisation. Aucune autre contrainte n'est donnée ; on pourra donc utiliser du passage de message classique ou de l'appel de procédure à distance (dans ce cas la procédure à appeler consiste en l'opération de réception de données).

Quelle que soit l'entité supportant l'exécution, on tient à ce que l'ordonnancement soit transparent, c'est-à-dire que le programmeur n'ait pas besoin de se soucier qu'un objet actif rende la main. Si par besoin de granularité fine, plusieurs entités se partagent un processus lourd, alors se pose la question de la mise en œuvre d'un ordonnancement transparent de ces entités (dans le cas où chaque entité est supportée par un processus, l'ordonnancement est transparent et fourni par le système d'exploitation). En fait,

- soit l'ordonnancement est préemptif et éventuellement également à temps partagé : dans ce cas, il n'y a pas de souci à se faire concernant le partage de la machine, mais le support d'exécution doit éventuellement pallier aux éventuels problèmes liés au fait que le programme peut être préempté à n'importe quel endroit du code ; pour ce faire, on peut protéger par un verrou, les sections de code durant lesquelles une préemption ne doit pas avoir lieu ;
- soit l'ordonnancement n'est pas préemptif, dans ce cas, il faut profiter d'appels de cer-

taines fonctions de gestion de la part de l'entité en cours d'exécution pour lui prendre la main et la passer à une entité prête (par exemple, lorsqu'elle passe en réception). De ce fait, l'ordonnancement étant contrôlé, il n'y a pas de problème d'interruption intervenant à n'importe quel endroit du code à gérer, mais il faut suffisamment de points de préemption pour que le calcul progresse équitablement sur chacun des objets actifs.

De plus, dans PROSIT du moins, un objet actif peut-être multi-actif. Dans ce cas, il doit lui-même pouvoir lancer plusieurs fils d'exécution et avoir si besoin le loisir de gérer leur ordonnancement. Il se peut donc qu'il y ait incompatibilité entre le choix du support des objets actifs de base et de celui des activités secondaires. En particulier, le comportement des activités au sein d'un objet actif de PROSIT est celui de coroutines, alors que celui de l'objet actif lui-même est celui d'un processus au sens classique. Une solution qu'a adoptée PROSIT, est de gérer au même niveau l'activité principale d'un objet actif et ces activités secondaires. Ceci implique que le choix du support d'exécution de l'objet actif et son ordonnancement sont remontés à un niveau plus haut et nous échappent.

Il faut dès lors que le support d'exécution que nous proposons soit éventuellement capable de se décharger en partie de la gestion de l'exécution de certaines entités ! Mais il faut quand même qu'elles soient connues à notre niveau, en particulier pour pouvoir gérer les communications dont elles sont le siège.

Fil conducteur de l'état de l'art. Cet état de l'art s'organise donc autour de deux points :

1. *Quels sont les supports d'exécution* élémentaires existants que nous pourrions choisir d'interfacer afin de produire notre support d'exécution orienté objet "encapsulant" (au sens du chapitre précédent) et aussi satisfaisant les besoins — pas très précis ! — de nos 2 applications privilégiées (leurs besoins pourraient se résumer de la sorte : souplesse et portabilité). Dans ces conditions, cette étude bibliographique, évidemment non exhaustive, résulte d'une sélection parmi les différentes approches (à base de processus lourds, à base de processus légers, que le mécanisme de communication offert soit du passage de message ou de l'appel de procédure à distance) des quelques supports représentatifs[1] servant pour l'exécution d'applications en réparti. Pour chacun d'eux, nous nous attachons à dégager le modèle d'exécution associé ainsi que le modèle de programmation. En effet, si l'on interface un tel outil, il faut comprendre comment on l'utilise, donc quel est son modèle de programmation et d'utilisation.

   La présentation s'articule autour du support d'exécution d'une entité. Ainsi, nous parlerons d'abord des outils où le support d'exécution d'une entité est le processus lourd. Puis nous aborderons la présentation de ceux pour lesquels il s'agit du processus léger.

2. Plusieurs travaux autour de la notion d'objet actif et réparti existent déjà. Parmi ceux que nous avons sélectionnés, il est très instructif d'étudier la façon dont l'exécution d'un objet actif est prise en charge, et selon la manière qu'il a de communiquer, comment cela

[1] à l'époque, mais également à présent

est mis en œuvre. Cette partie de l'état de l'art n'a donc pas pour rôle de présenter des environnements que nous pourrions interfacer mais d'étudier *comment des problèmes similaires aux nôtres ont été résolus*. C'est pour cette raison que nous ne nous attarderons pas sur des travaux qui ne sont en fait que des encapsulations orientées objet des outils présentés dans le premier point ; mais plutôt sur des approches consistant à étendre, par le biais de bibliothèques de classes par exemple, un langage orienté objet de sorte à pouvoir développer des applications réparties (Panda, Java RMI, CORBA seront abordés). Nous étudierons également brièvement quelques exemples où le support d'exécution est conçu "from scratch" de sorte à cibler le calcul réparti. Cette deuxième catégorie s'apparente à la construction de mécanismes pour systèmes d'exploitation répartis (SOS, ...).

Résultats attendus au terme de cet état de l'art. Au terme de ce chapitre, nous serons en mesure de dégager le dénominateur commun aux différents représentants candidats à être un support d'exécution, en vue de la construction d'un modèle unifiant que nous mettrons en œuvre dans notre encapsulation. Nous serons également plus à même de dégager quels sont les mécanismes ou concepts que l'on ne retrouve pas systématiquement dans chaque outil, mais qu'une extension de notre encapsulation pourrait permettre d'obtenir si besoin. Nous aurons également étudié quelques exemples de conception et mise en œuvre d'exécutifs pour environnements à objets actifs répartis. Cette étude nous aura également permis de toucher du doigt les problèmes qui se posent lorsque l'on veut étendre les fonctionnalités d'un outil existant — problèmes difficiles si l'outil n'est pas orienté objet — ainsi que les utilitaires annexes qu'un développeur pourrait souhaiter avoir et qu'il n'a pas forcément (par exemple, outils de mise au point d'applications, d'aide à la gestion des ressources).

3.2 Supports d'exécution à base de processus "lourds"

A l'heure actuelle, les environnements les plus utilisés pour la programmation d'applications parallèles dont l'exécution se déroule sur un support réparti, sont dits "à gros grain", car l'entité d'exécution de base est le processus (au sens classique d'UNIX ou de WINDOWS NT). Les environnements tels que PVM, MPI, Landa [Monteil 95], Linda [Carriero 89] en sont des exemples représentatifs. A part Linda qui utilise l'échange de données par partage de mémoire (la mémoire étant symbolisée par un *tuple space*), les trois autres se fondent sur l'envoi de message "classique", les variantes possibles dépendant du mode de synchronisation instauré entre les processus émetteur et récepteur impliqués dans une opération de communication.

Nous allons dans cette section nous attarder sur le cas de PVM et de MPI car ce sont des standards (même s'il ne s'agit que d'un standard "de fait" dans le cas de PVM). Nous nous attacherons à décrire en particulier les caractéristiques de leur exécutif.

3.2.1 PVM

Motivation principale. PVM [Sunderam 90, Geist 94] est un projet de recherche de grande envergure initié au début des années 1990, à partir d'une idée simple : gérer comme un tout les réseaux de stations de travail hétérogènes afin qu'ils puissent servir de support à des applications concurrentes ou parallèles. Trois principes essentiels ont guidé la conception et l'implémentation : 1°) une API (Application Programming Interface) simple[2], 2°) une gestion telle que l'hétérogénéité des machines devient transparente, 3°) la possibilité de gérer dynamiquement la configuration (machines, tâches).

Modèle de programmation. Une façon de décrire le modèle de programmation est d'évoquer les principales fonctions fournies par l'API, API utilisable directement depuis C ou Fortran. Cette API — constituée d'une soixantaine de fonctions dans la toute dernière version (3.4) [Geist 98] — permet de programmer avant tout le contrôle de la machine parallèle virtuelle et des tâches et l'échange de messages entre tâches. En général avant chaque communication, une phase de construction du message doit précéder sa transmission : un tampon doit être initialisé, le message emballé selon un format de représentation de données adéquat, qui peut nécessiter un encodage spécifique en cas d'hétérogénéité des intervenants. Symétriquement, une phase de déballage est utile côté récepteur avant le traitement proprement dit. De façon moins courante, la programmation peut concerner la notification d'événements, la prise en charge de messages par le biais de "handlers", la gestion d'une boîte aux lettres de messages commune à l'ensemble de la machine virtuelle (à la manière du *tuple space* de Linda). Il existe des primitives de communication bipoints et multipoints, ainsi que des opérations collectives (communication multipoint agrémentée d'un traitement sur chaque message). Il est intéressant de noter que jusqu'à la version 3.4 exclue, l'échange de données entre deux tâches se réalise uniquement par une opération bidirectionnelle c'est-à-dire où la tâche émettrice réalise un envoi — qui se trouve être toujours asynchrone — par le biais de la primitive `pvm_send()` et la tâche réceptrice doit exécuter une opération de réception, telle un `pvm_receive()` pour obtenir un message en vue de le traiter. Conformément à la tendance actuelle de cibler des applications de plus en plus dynamiques et réactives, l'API de la version 3.4 inclut à présent la possibilité d'installer, même dynamiquement, des fonctions de réception-traitement automatiques de messages. En conséquence, les routines en charge de réceptionner les données arrivant sur la tâche ont dues être modifiées : elles ont pour rôle supplémentaire de filtrer les messages dont la sémantique de communication est unidirectionnelle et de démarrer la fonction de traitement associée.

Support d'exécution. Pour supporter l'exécution d'une application faisant usage de la bibliothèque PVM, sur chaque machine utilisée doit s'exécuter un démon PVM (`pvmd`). Il a

[2] qualifiée de simple par ses promoteurs, mais qui requiert tout de même une assez bonne pratique

un rôle de gestionnaire : démarrage de démons ou de tâches, prise en charge des communications non directes entre tâches grâce à la programmation d'un protocole spécifique, basée sur des sockets[3] (donc portable), ... Il est possible de définir et modifier et ce même pendant l'exécution, l'ensemble des nœuds hétérogènes composant la machine parallèle virtuelle.

Chaque tâche est matérialisée par un processus lourd disposant de son propre environnement d'exécution (fichiers au sens large, variables d'environnement, ...). La primitive `pvm_spawn()` démarre un fichier exécutable sur un des nœuds de la machine parallèle virtuelle. Le programmeur peut spécifier cette localisation, ou s'en remettre à l'exécutif PVM (au `pvmd`). Ce dernier peut sélectionner n'importe quel nœud (chacun à tour de rôle), un nœud d'une architecture donnée spécifiée par le programmeur ou délègue le choix à un programme gestionnaire de ressources dans lequel a pu entre autres être codé une stratégie ad-hoc de choix de l'emplacement d'une tâche. Cette dernière possibilité qui assouplit nettement le choix du nœud venait tout juste d'être ajoutée au moment où nos travaux en étaient à la phase de conception. On peut tout de même déplorer qu'elle nécessite une connaissance assez poussée de l'implémentation de PVM. Justement, une de nos requêtes fortes concernant un support d'exécution pour applications réparties était non seulement que l'on puisse personnaliser la stratégie de placement initial des entités d'exécution, mais que cela puisse se faire de façon **aisée**.

Environnement PVM. Outre l'API et le support d'exécution, autour de PVM gravite une large gamme d'outils : développement graphique, déverminage grâce à une collecte permanente de traces générées par l'exécutif, analyse de performances, régulation de charge, bibliothèques pour des domaines d'applications précis. Par ailleurs, les interfaces de l'API permettant d'utiliser le package PVM dans d'autres langages que C ou Fortran sont légions : Ada, Java, Perl, Tcl, Tk, ... Ces "accessoires" concourent à la popularité de PVM : plus ils sont nombreux, plus les utilisateurs se pressent ; et plus PVM a d'adeptes, plus nombreux sont les outils annexes ...

Critiques et conclusion. PVM est indéniablement un outil très professionnel et complet pour le développement d'applications réparties, preuve en est le riche site Internet qui lui est consacré [PVM Team 98,]. L'équipe tente d'y inclure constamment les outils permettant la mise en œuvre d'idées novatrices, comme par exemple le contrôle ou la personnalisation dynamique de l'environnement opératoire (par exemple changement de la bibliothèque de support de communication). Ceci nécessite une plus grande flexibilité de PVM qui a plutôt été conçu comme un système monolithique. Une approche orientée objet aurait permis une évolution plus en douceur.

Un point particulier méritant d'être relevé est l'absence totale de discussion autour de la

[3]Les communications directes entre tâches sont réalisées via un socket en mode connecté reliant l'émetteur au récepteur.

notion d'exécution *multithreadée*. A croire que l'unité de parallélisme étant la tâche — donc à gros grain — le seul style de programmation envisageable en son sein est un fil de contrôle séquentiel alternant entre calcul et communication, éventuellement agrémenté de messages actifs. Néanmoins des extensions de PVM intégrant des bibliothèques de processus légers seront présentées dans la prochaine section. Un point délicat que ces extensions résolvent — et que nous serions obligés d'aborder et de résoudre si nous choisissions PVM comme support d'exécution, avec l'intention d'y adjoindre nous-mêmes des processus légers — est le fait que PVM n'est pas réentrant et donc pas "thread_safe". Nous reviendrons sur ce point en temps voulu.

3.2.2 MPI

MPI [Mes 94, Malard 96] est une spécification établie par un consortium d'universitaires et d'industriels, avec comme objectif que cette spécification constitue la norme pour la programmation parallèle par passage de messages. Charge ensuite d'implémenter cette spécification sous forme d'une bibliothèque constituant une API et de fournir un support d'exécution adapté aux matériels ciblés. Il existe ainsi diverses implémentations conformes à la norme MPI, les plus connues et complètes étant MPICH (voir Section 3.2.3) et LAM [Burns 94].

Motivation principale. Historiquement les différents vendeurs de machines parallèles livraient chacun leur propre bibliothèque permettant l'échange de messages entre processus, empêchant ainsi tout portage des applications. En normalisant les différentes fonctions que se doit d'offrir une bibliothèque pour la programmation parallèle par passage de messages, la portabilité des applications devient possible ; en ne décrivant que les spécifications de ces fonctions, chaque constructeur est en mesure de réaliser l'implémentation la plus efficace possible en utilisant les fonctions de communication natives de la machine cible. Sur un réseau de stations de travail pouvant être hétérogènes, l'implémentation de la norme est obligée de faire appel à des couches réseau passe-partout — tout comme PVM— et ce au prix de performances éventuellement plus médiocres.

Modèle de programmation. Un programme MPI est constitué d'un ensemble de tâches, organisé selon une topologie donnée et statique qui correspond en fait aux interactions qu'auront les tâches. L'échange de données nécessite, comme en PVM, de nommer la tâche destinataire et plus globalement le contexte dans lequel cet échange doit se dérouler. Visiblement n'ayant pu (voulu ou su ?) réaliser une synthèse, l'ensemble des primitives de communication point-à-point et multipoints entre tâches est le plus riche connu à ce jour. Des primitives de communication unidirectionnelles sont également spécifiées depuis peu dans la version 2 de la norme. Des fonctions d'emballage et de déballage de messages encore plus générales que celles permises par PVM sont fournies.

Support d'exécution. Même si la gestion de la machine virtuelle n'est pas du tout abordée par la norme MPI, chaque implémentation est bien obligée de proposer un ensemble de primitives pour indiquer quels calculateurs utiliser, démarrer/terminer des entités supportant l'exécution des tâches, ... En effet, MPI ne suppose qu'implicitement l'existence d'un environnement dans lequel l'application s'exécute. Bien qu'une certaine forme d'interaction entre cette application et son environnement soit forcément nécessaire, elle devrait rester invisible au niveau applicatif, permettant ainsi une totale portabilité des applications.

L'implémentation LAM/MPI par exemple repose sur l'exécutif LAM par le biais duquel une gestion dynamique et modulaire de la machine virtuelle est possible. Des intégrations de PVM et MPI permettent de faire profiter les applications MPI des possibilités de gestion dynamique de la machine virtuelle qu'offre PVM. PVMPI [Fagg 96] est un exemple de proposition parmi d'autres en ce sens.

Conclusion. Comme un de nos objectifs est l'étude des supports d'exécution, l'étude de MPI seul est insuffisante et de peu d'intérêt. Nous avons néanmoins dégagé le modèle de programmation que dicte l'utilisation d'une bibliothèque de fonctions MPI. Nous allons à présent présenter MPICH qui implémente complètement la norme MPI tout en fournissant un riche environnement de développement.

3.2.3 MPICH

MPICH [Gropp 96] est une implémentation complète et performante de la norme MPI, à l'origine essentiellement conçue pour des calculateurs parallèles homogènes, mais dont la conception est telle qu'elle a naturellement convenu aux environnements hétérogènes comme des réseaux de stations de travail. La raison de cette portabilité vient d'une conception en couches présentée dans la figure 3.1.

L'extrême diversité des environnements pouvant exécuter MPICH explique une aussi grande diversité des façons de supporter l'exécution des nœuds de l'application MPI et de les démarrer. On a en général un processus par nœud MPI, mais la version au dessus de NEXUS par exemple, permet d'associer un thread à un nœud. Pour uniformiser le démarrage de l'application, une seule et unique commande est fournie, `mpirun`, qui bien que ne faisant pas partie de la norme est présente dans d'autres implémentations de MPI. Grâce à ses nombreux arguments, on peut entre autre préciser le type des machines, le nom de ces dernières provenant d'un fichier externe.

Le souci majeur de MPICH se résume bien par le "CH" de MPICH : tel un "chameleon", cet outil a été conçu de telle sorte à pouvoir parfaitement s'adapter à son environnement tout en ayant les meilleures performances possibles. L'approche par interfaces de niveau de plus en plus proche du support matériel cible, nous semble inévitable lorsque la portabilité est le

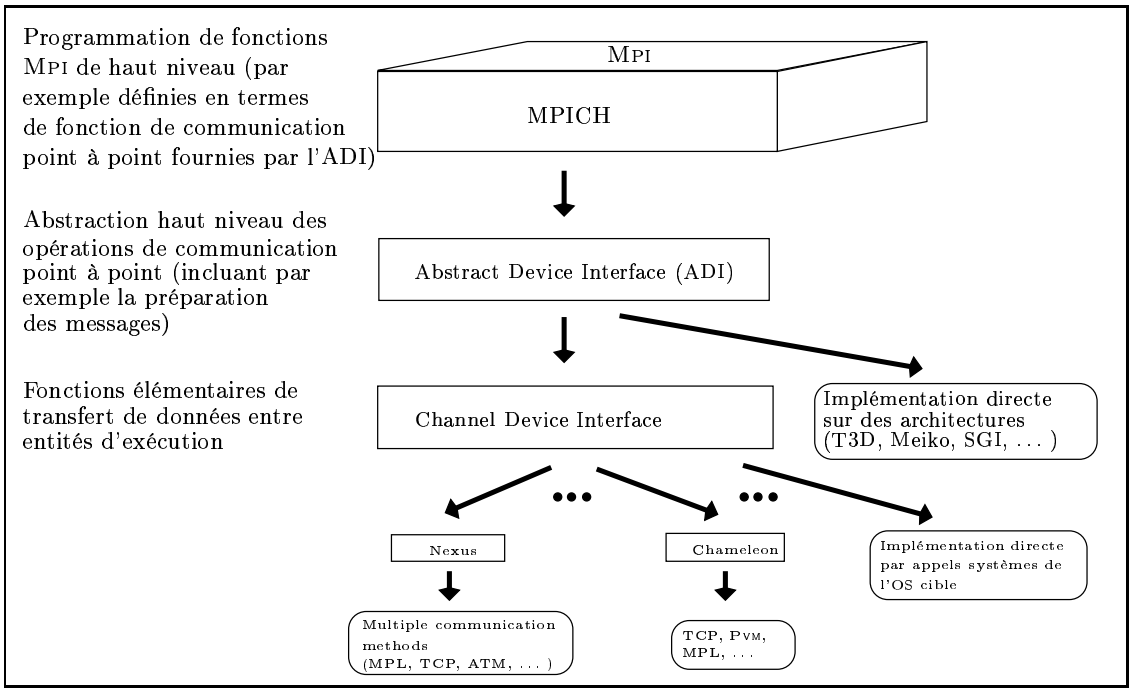

FIGURE 3.1 – *Organisation de MPICH.*

souci majeur. D'ailleurs, notre prototype SCHOONER ayant également ce souci de portabilité, il est normal, bien que notre but ne soit pas d'implémenter MPI, que nous ayons adopté une démarche semblable même si elle est orientée objet. Comme nous le détaillerons dans la section 5.6 de la partie II, notre classe système joue le même rôle que le Channel Device Interface et notre gestionnaire de communication implémente des fonctions comparables à celles de l'Abstract Device Interface.

3.3 Supports d'exécution à base de processus "légers"

Dans cette partie, nous allons aborder l'étude de supports d'exécution pour applications parallèles et réparties à grain fin. Parmi les exemples choisis, on distinguera ceux pour lesquels la communication se fait selon le mécanisme classique d'échange de message, de ceux où l'échange se fait — indirectement — par appel de procédure à distance. Dans ce deuxième cas, le calcul parallèle se schématise comme un ensemble d'exécutions de services distants — donc parallèles. Nous préciserons plus loin ce mode particulier de communication, ainsi que son utilisation dans le cas du calcul parallèle.

3.3.1 Quelques environnements utilisant des processus légers, basés sur de l'échange de message, autour de PVM

PVM ne permettant à une tâche de s'exécuter que sur un processus lourd, plusieurs travaux ont été entrepris de sorte à pallier à cette limitation en permettant à une tâche d'être prise en charge par un processus léger.

Nous allons rapidement présenter PT-PVM et TPVM. Ce sont des outils que l'on peut qualifier d'extensions de PVM car ils permettent le lancement de tâches prises en charge par des processus légers et ce à l'intérieur de tâches PVM au sens classique supportées par un processus lourd. De plus, leur modèle de programmation est tout à fait comparable à celui qu'offre PVM, même s'il peut être plus général.

3.3.1.1 PT-PVM

Modèle de programmation. En PT-PVM [Krone 95], les primitives PVM relatives à la machine parallèle virtuelle sont utilisées pour gérer les "hosts" et les "tasks" au sens PVM. Cependant, le calcul proprement dit n'est pas réalisé par les tâches PVM qui servent juste de processus hôte, mais par des tâches PT-PVM de granularité plus fine. Ainsi, la primitive `csSpawn_N()` permet de démarrer une tâche qui exécutera le code de la fonction passée en argument. Il est possible de laisser libre le choix de la tâche PVM hôte. Une fois créée, une tâche PT-PVM obtient un identifiant qui peut être utilisé pour lui adresser un message, grâce à une primitive (`csBaseSend()`) dont la sémantique est comparable à `pvm_send()`. Concernant la réception, la tâche PT-PVM dispose d'une opération spécifique (`csBaseRecv()`), car les messages sont reçus sur la tâche PVM hôte puis répartis dans des files propres aux tâches PT-PVM. Cependant sa sémantique est comparable à celle de `pvm_recv()`. La nouvelle version de PT-PVM, PT-PVM$^+$ [Krone 96] rajoute en plus la possibilité d'appeler un service à distance de façon asynchrone, sans attente du résultat. La déclaration du service se fait grâce à la primitive `csBaseSvcBind()`, puis l'invocation se fait par `csBaseSvcReq()` dont les arguments sont le nom du service et ses paramètres. Un message contenant le résultat de l'exécution du service et étiqueté par le nom du service est envoyé au client.

Support d'exécution. Outre celui fourni par l'utilisation de l'environnement PVM, et d'une bibliothèque de threads préemptive telle celle offerte par Solaris, un processus en charge de l'envoi des messages est rajouté sur chaque nœud de la machine parallèle virtuelle. Son rôle est de centraliser toutes les demandes d'envoi de messages des différentes tâches PT-PVM s'exécutant sur cette machine, demandes convoyées par un appel de service asynchrone. Un thread par tâche PVM est dédié à la réception des messages qui sont ensuite stockés ou délivrés immédiatement à la tâche PT-PVM cible. Ces ajouts sont indispensables car les tâches PT-PVM se partagent le temps machine de façon préemptive (puisque la bibliothèque de threads utilisée

est supposée l'être), or les primitives PVM ne sont pas réentrantes. Ainsi, seul le thread de réception par tâche PVM utilise des fonctions de l'API PVM, évacuant ainsi tous les problèmes liés à l'utilisation concurrente de primitives de l'API PVM.

3.3.1.2 TPVM

TPVM [Ferrari 95] est un outil dont la philosophie de programmation est identique à celle qu'offre PVM. Pour résumer, toutes les primitives PVM trouvent leur équivalent en TPVM. Il faut comme pour PT-PVM, gérer une machine parallèle virtuelle hôte, ce qui s'implémente ensuite aisément grâce à PVM et sur chaque nœud de cette machine parallèle, on peut alors démarrer des tâches TPVM de granularité plus fine. Les fonctions que peuvent exécuter les tâches TPVM doivent au préalable avoir été identifiées par une chaîne de caractères et le code local associé désigné. Une fois ce travail préparatoire effectué, il est possible de démarrer par une primitive de sémantique comparable à `pvm_spawn()`, une ou plusieurs tâches TPVM en spécifiant un identifiant et éventuellement la localisation, charge au support d'exécution de trouver un nœud PVM hôte pour ces nouvelles tâches. Une fois créée, une tâche TPVM dispose d'un identifiant `ttid` qui sert ensuite comme désignateur de source / cible d'opérations de communication. Cependant, un autre modèle de programmation répartie, dirigé lui par les données est rajouté au modèle classique par passage de messages. Pour plus de détails, on peut consulter [Namyst 97].

Implémentation. L'implémentation nécessite en plus de PVM l'utilisation d'un quelconque noyau de processus légers, dont l'ordonnancement est préemptif ou non (comme par exemple celui de ReX [Crane 93]). Seules quelques classiques primitives de gestion de processus légers sont utilisées pour l'implémentation, à savoir : création, destruction, préemption volontaire, synchronisation sur verrous nécessaire uniquement dans le cas où le noyau de processus légers est préemptif. Aucune hypothèse sur le modèle d'ordonnancement des threads ne doit être faite par le programmeur TPVM. En fait, l'ordonnancement entre tâches TPVM est entièrement géré par la bibliothèque TPVM et ce en particulier, au moment d'une opération de réception. De sorte à empêcher une tâche TPVM de se bloquer en attente d'un message non arrivé, la réception invoquée par `tpvm_recv()` s'exécute ainsi : un appel à une primitive de réception non bloquante est réalisé (`pvm_nrecv()`) et en cas de résultat négatif, la main est rendue et donnée à une autre tâche TPVM prête.

3.3.1.3 Discussion

Ces 2 extensions à PVM nous montrent les difficultés à résoudre lorsque la bibliothèque de communication utilisée n'est pas réentrante. Elles montrent également comment on peut supporter un noyau de processus légers que leur ordonnancement soit préemptif ou non. De par

leurs caractéristiques, avouons que ces outils pourraient convenir aux supports d'exécution des applications que nous ciblons. Néanmoins, au début de notre travail, ils n'en étaient qu'à des phases préliminaires. Mais surtout, nous avions comme autre ambition de prévoir un support d'exécution qui ne dépende pas fortement d'un outil existant — à la durée de vie imprévisible — et puisse tout en perdurant s'adapter à de nouvelles couches support et à d'autres modes d'interactions tel celui que nous allons étudier à présent.

3.3.2 Quelques environnements utilisant des processus légers, basés sur de l'appel de service distant

Après avoir présenté le mécanisme de l'appel de service distant, nous allons donner deux exemples de tels environnements, PM^2 et NEXUS, qui en font usage.

3.3.2.1 Mécanismes pour l'appel distant

Que le contexte soit orienté objet ou non, c'est-à-dire que l'on parle de RMI (Remote Method Invocation) ou de RPC (Remote Procedure Call, [Birrell 84]), des mécanismes permettant de réaliser l'appel de fonction entre 2 contextes d'exécution distincts, éventuellement distants (d'où le terme "remote"), sont mis en œuvre. L'objectif est de rendre le plus possible *transparent* à l'application, l'utilisation de ces mécanismes. Cette transparence est parfaite lorsque l'appelant n'a même pas à se demander si le traitement qu'il demande s'exécutera en local ou non. Cela consiste à dire qu'un appel de procédure qu'il soit local ou distant a strictement la même syntaxe qu'un appel local. Ceci n'est possible que si l'on met en œuvre avant l'exécution un préprocesseur.

Ces mécanismes pour gérer la non-localité de l'appel portent sur :
- la mise à plat des paramètres d'appel afin de permettre leur transfert entre les 2 contextes d'exécution, si besoin via la couche de communication,
- la transmission entre contextes de l'identité de la fonction ou méthode appelée et des paramètres,
- le blocage de l'entité appelante dans le cas d'un appel synchrone,
- l'obtention du résultat et sa délivrance à l'entité appelante.

Dans la suite, on pourra dénommer l'entité appelante "client" et le traitement demandé "service", cette dénomination ne s'appliquant qu'à une instance d'appel, l'appelant pouvant jouer évidemment le rôle d'un serveur c'est-à-dire répondre à des demandes de service et vice-versa.

Pour que ces mécanismes soient exécutés, en plus du service proprement dit, il faut qu'ils soients invoqués ! Ceci se fait de façon transparente grâce à l'existence d'une entité appelée "stub" (talon ou squelette), dont l'interface d'utilisation est **strictement identique** à celle

de l'appel du service par le client. Ainsi, lors de l'appel du service, il se trouve que l'exécution est détournée vers l'exécution des mécanismes gérant la non-localité, eux-mêmes se chargeant de demander l'exécution du service. Cette transparence pour le client est totale si, que l'appel d'un service soit local ou non, rien ne change, même pas la syntaxe de l'appel.

Selon les implémentations,
- le *stub* met en œuvre lui-même tous les mécanismes cités (voir [Birrell 84]),
- ou parfois, surtout en contexte orienté objet — et nous y reviendrons donc dans la prochaine section de ce chapitre —, le *stub* fabrique juste l'entité représentant la demande de service[4], puis la passe à un *proxy* (mandataire, c'est-à-dire représentant local) de l'appelé [Shapiro 86]. Ce *proxy*, qui travaille pour le compte du serveur met en œuvre tout ce qui a trait à la communication avec le serveur, éventuellement par le biais d'un réseau.

Calcul haute performance et mécanisme d'appel distant. Ces mécanismes qui s'appliquent au vaste domaine de la programmation *client/serveur* ont été spécialisés pour prendre en considération le calcul haute performance par le biais de l'exécution répartie. En présence d'un seul client, même si le client et le serveur sont localisés sur des processeurs différents, l'obtention de parallélisme n'est possible que si l'appel de service est **asynchrone**. Ainsi, le client peut poursuivre son exécution pendant que sa demande de service s'exécute ailleurs. Si l'appel de service est synchrone, donc bloquant pour le client, l'exécution de l'application ne présente du parallélisme que si plusieurs clients sont actifs, et que le support d'exécution sait passer la main à un client prêt dès qu'un client se bloque. Pour que le serveur ne soit pas un goulot d'étranglement, il est également possible que chaque exécution d'un service soit réalisée en pseudo-parallélisme vis-à-vis des autres, en utilisant plusieurs fils d'exécution au sein de l'entité supportant le serveur.

Ainsi les outils que nous avons choisis de présenter utilisant du RPC, se basent sur du RPC asynchrone d'une part, et sur du service multithread d'autre part. C'est donc pour cette dernière raison qu'ils sont présentés dans cette section "à base de processus légers".

3.3.2.2 PM^2

PM^2 [Namyst 95, Namyst 97, Namyst 98a] (Parallel Multithreaded Machine) est un environnement pour architectures réparties fournissant un modèle de calcul basé sur l'utilisation de processus légers. Il permet l'exploitation de configurations hétérogènes et est particulièrement adapté à l'exécution d'applications parallèles irrégulières. C'est l'adéquation du mécanisme de RPC au support du parallélisme massif et son intégration dans un environnement supportant

[4]La demande de service devient donc une entité de première classe, c'est-à-dire manipulable en tant que telle.

la mobilité des activités qui constituent l'originalité de PM^2.

Modèle de programmation. L'application se décrit en fait comme un ensemble de tâches séquentielles, chacune de ces tâches étant prise en charge par un processus léger. Les nœuds sur lesquels se déroulent ces processus légers, jouant indifféremment les rôles de clients ou de serveurs, sont appelés des *modules*. Les modules ont en général pour rôle d'effectuer l'initialisation — en particulier la déclaration des services exportés par le module — puis éventuellement le démarrage de quelques processus légers et finalement l'attente de la terminaison de ceux-ci sur le nœud.

Ce découpage des applications est réalisé en utilisant le mécanisme d'appel de procédure à distance léger. Il consiste en la possibilité qu'a un processus léger de déclencher l'exécution d'une fonction se trouvant sur un *module* distant, celle-ci étant prise en charge par un nouveau processus léger créé en général pour l'occasion. De plus, l'utilisateur n'a jamais à se soucier de l'ordonnancement de ces processus légers, qui est entièrement transparent car préemptif et à temps partagé.

Toutes les étapes de transport de l'appel de procédure puis de son résultat sont effectuées selon le modèle classique de RPC. Les procédures invocables à distance sont nommées de façon symbolique et exportées. De même le type de leurs paramètres d'appels est répertorié lors de l'export. L'export se matérialise par un fichier, commun en général à toute l'application, décrivant la liste des services disponibles puis leur implémentation. Il faut en plus fournir pour chacun d'eux les fonctions d'emballage et de déballage pour les arguments d'appel de service et pour l'envoi de la réponse (les 4 fonctions souches classiques). Toutes ces descriptions sont réalisées en langage C et sont peu structurées. Une approche orientée objet serait clairement un atout, d'autant plus que PM^2 s'affiche clairement comme un environnement de **développement** d'applications et non pas comme un environnement support de compilation, tel que le prétend être par exemple NEXUS.

Les différents types d'appels de procédures distants sont 1°) l'appel synchrone, c'est-à-dire avec attente bloquante pour l'appelant du résultat de la procédure ; 2°) l'appel avec attente différée qui se matérialise **explicitement** côté client, par l'utilisation d'une variable dont la sémantique est celle d'un *futur* ; 3°) l'appel asynchrone, sans aucune attente, utilisable lorsque la procédure n'envoie pas de réponse, car ne fournit pas de résultat au client. De plus, il y a deux façons de demander l'exécution de la procédure distante : avec création d'un nouveau processus léger sur le module indiqué en paramètre de l'appel ; sans création de processus léger, ce qui se met en œuvre en faisant exécuter le service immédiatement par le processus léger qui de façon transparente se charge de réceptionner les ordres d'appels depuis la couche de communication. Bien évidemment, ce dernier mode "d'exécution rapide" suppose que les services exécutés ainsi ne soient pas bloquants ...

Enfin, PM^2 introduit un mécanisme de migration de processus léger entre modules. Le

déroulement de l'opération est transparent pour le processus léger, la seule restriction étant que le code du processus léger soit dans une phase où il autorise d'être migré. Les portions de code non migrables sont donc précédées de la primitive spéciale `pm2_disable_migration()`.

Support d'exécution. PM^2 a avant tout comme objectif d'être portable. Pour ce faire, bien que les versions initiales étaient fondées sur la bibliothèque de communication PVM, la dernière se base sur une interface intermédiaire, nommée Madeleine [Bougé 98, Namyst 98b]. Elle permet dès lors une complète transparence vis-à-vis de la couche de communication réellement utilisée (TCP/IP, PVM, MPI ou toute autre). SCHOONER se fonde sur une approche comparable.

Pour résoudre la portabilité du support d'exécution des processus légers de PM^2, alors que les noyaux de processus légers ont des caractéristiques bien différentes, en particulier concernant leur ordonnancement, PM^2 repose entièrement sur la bibliothèque de processus légers, niveau utilisateur, nommée Marcel. C'est alors une implémentation portable de cette dernière qui résout la portabilité générale de PM^2.

Marcel reproduit un sous-ensemble des primitives de gestion de processus légers de la norme POSIX, qui rappelons-le ne spécifie pas la forme exacte de leur ordonnancement, mais seulement quelques grandes lignes matérialisées par des classes d'ordonnancement "plus ou moins" préemptif (*FIFO* : dès qu'une primitive de synchronisation de processus légers est invoquée, une commutation de contexte selon une politique FIFO est éventuellement initiée par le noyau de la bibliothèque ; *RoundRobin* : la différence est que la préemption se produit à intervalles de temps (quantum) réguliers et peut donc se produire à n'importe quel point du déroulement de l'exécution du processus léger, le choix se porte alors vers le suivant dans la liste des processus prêts).

L'ordonnancement des processus légers dans Marcel est basé non seulement sur des priorités, mais aussi sur un partage du temps. Toutes les portions de code non préemptibles doivent donc être protégées. Les priorités des processus légers sont programmables, mais doivent l'être **explicitement** par le programmeur. Un processus léger Marcel est toujours présent : il s'agit de celui qui a comme rôle de demander la réception **non bloquante** depuis la couche de communication. Il a bien sûr régulièrement la main.

L'implémentation du mécanisme de migration d'un processus léger Marcel qui est déclenchée par l'appel à la primitive `pm2_migrate()`, s'effectue en 3 étapes :

1. L'exécution du processus léger est gelée et son descripteur ainsi que les informations utiles de sa pile sont préparés pour envoi ;
2. L'envoi a lieu vers le module distant ;
3. Le descripteur et les parties utiles de la pile sont reçues, la pile est réallouée dans le nouvel espace d'adressage puis le processus léger est dégelé. L'exécution peut alors reprendre,

exactement à partir de la même instruction et ce de façon totalement transparente.

L'environnement PM^2 et son API associée sont disponibles sur de nombreux couples architectures/systèmes : ALPHA/OSF-1, Sparc/Solaris, Sparc/SunOs, PowerPC/Aix, PC-Linux.

3.3.2.3 NEXUS

NEXUS [Foster 94, Foster 96] est un environnement développé conjointement par le laboratoire Argonne et l'Institut de Technologie de Californie. Il fournit un support d'exécution multithreadé, permettant de développer des applications parallèles et réparties sur des environnements hétérogènes, le type privilégié d'applications étant des compilateurs de langages parallèles répartis. La bibliothèque fournit principalement un modèle de mémoire globale via des références interprocessus et un mécanisme d'événements asynchrones.

Modèle de programmation. Un programme construit au dessus de la bibliothèque NEXUS (voir Figure 3.2) est constitué d'un ensemble de *processus légers*, chacun d'entre eux s'exécutant dans un espace d'adressage appelé *contexte*, qui est alloué sur un *nœud* de l'architecture sous-jacente. Chaque processus léger exécute séquentiellement un flot d'instructions et peut lire et écrire des données partagées avec les autres processus légers s'exécutant dans le même contexte. Ces processus légers peuvent initier des *demandes de service distants* asynchrones, et donc déclencher l'exécution de procédures dans d'autres contextes, via l'utilisation de *pointeurs globaux*.

Les pointeurs globaux permettent d'obtenir un espace de nommage global, en désignant une adresse mémoire donnée dans un contexte donné. Ils ont une signification globale sur l'ensemble des contextes et peuvent donc être communiqués entre contextes.

Ces différentes entités peuvent toutes être créées et détruites dynamiquement tout au long de l'exécution de l'application.

Support d'exécution. Comme PM^2, un des objectifs de NEXUS est d'être portable. Cet objectif est atteint grâce à l'hétérogénéité fournie par l'environnement. En effet, une application NEXUS peut être construite à partir de plusieurs langages de programmation, d'exécutables, de types de processus et de protocoles réseaux différents. Les mécanismes de gestion des processus légers sont fournis par une interface constituant un sous-ensemble de l'interface POSIX *Pthread* standard [Mueller 95], interface qui est disponible sur de nombreuses architectures.

Un des points forts de NEXUS est sa facilité à le porter sur une nouvelle architecture. L'implémentation de NEXUS n'utilise qu'un ensemble minimal des fonctionnalités — de plus non spécifiques — des environnements sur lesquels il est développé, ce qui fait qu'il peut

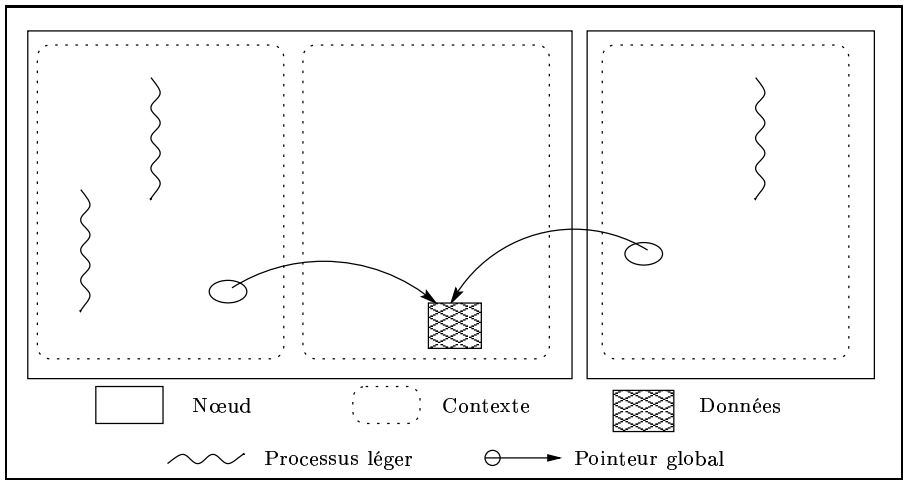

FIGURE 3.2 – *Vue d'une application construite au dessus de* NEXUS.

utiliser directement les bibliothèques et environnements disponibles sur une architecture, sans avoir besoin de développer de nouvelles interfaces.

La gestion de l'arrivée de nouveaux messages sur un *contexte* est généralement prise en compte par un processus léger spécial dédié à la scrutation. Cette technique qui est inévitable dans le cas de systèmes ne fournissant ni processus légers au niveau noyau ni primitive non bloquante de réception peut être avantageusement remplacée par d'autres politiques de scrutation :

1. *réception bloquante* par un processus léger spécifique si ces derniers sont gérés au niveau du noyau ;
2. *scrutation dispersée parmi les processus* ce qui permet d'améliorer le temps de réponse aux requêtes mais oblige au compilateur à insérer des instructions de scrutation dans le code des processus légers ;
3. *ordonnancement guidé par interruption* où l'arrivée d'une requête est notifiée par une interruption (il n'y a donc pas de scrutation dans ce dernier cas) et traitée par une fonction spécifique. Cette fonction de traitement ne peut pas contenir de code non réentrant.

3.3.2.4 Discussion

PM^2 ainsi que NEXUS fournissent des modèles de calcul et communication assez semblables, fondés sur le service distant léger invoqué de façon asynchrone. Cependant NEXUS se situe à un niveau d'abstraction légèrement plus bas car manipule la notion de pointeur global. D'ailleurs, ses auteurs ne conseillent pas aux programmeurs d'application de l'utiliser directement, contrairement au cas de PM^2. De ce fait, PM^2 gagnerait à être programmable selon le paradigme orienté objet.

3.3.3 Synthèse

Notre point de vue est de considérer les environnements utilisant l'appel de service distant comme étant au même niveau d'abstraction que ceux basés sur l'échange de message. En effet, il semble assez difficile de dire lequel des deux modes doit se trouver supporté par l'autre, la preuve étant qu'ils s'intersimulent aisément l'un l'autre : d'une part, il est naturel que le modèle à passage de messages puisse servir à décrire le mécanisme d'appel distant puisque il sert à envoyer et recevoir de façon asynchrone l'appel de service entre entités distantes ; d'autre part, bien que celà puisse paraître moins naturel le mécanisme de service distant peut s'appliquer à un service que l'on peut nommer *réception asynchrone*, l'appel correspondant à l'opération d'envoi asynchrone. Notons que pour des raisons évidentes d'efficacité, il est souhaitable que le service demandé (qui se borne à déclencher l'opération simulant la réception et la prise en compte du message) soit exécuté par un thread déjà existant. D'ailleurs, un tel mode est disponible en PM^2 (`quick_asynch_lrpc(... )`) et dans NEXUS (utilisation du type *non-threaded* lors de l'enregistrement de la fonction à exécuter en réponse à une demande de service distant (`nexus_register_handlers(... )`)). Sous ces conditions, la transmission d'information selon une méthode ou une autre a des performances similaires.

Ainsi, l'encapsulation unifiante de supports d'exécution que nous proposons se devra d'être capable de cibler n'importe lequel de ce type d'environnement pour ce qui concerne les couches "basses" de communication.

A propos de la nature des processus légers introduits par les environnements que nous avons étudiés dans cette section, constatons juste qu'il n'y a pas non plus de réel consensus quant à la nature de leur ordonnancement : préemptif à temps partagé ou pseudo préemptif avec points de préemption cachés dans les appels à la bibliothèque de gestion de processus légers. Cette absence de consensus étant, à l'heure actuelle du moins, inévitable (même la norme POSIX ne tranche pas), la meilleure stratégie consiste à fournir une encapsulation qui masque les différences. Une telle stratégie est d'ailleurs adoptée par NEXUS.

Ainsi, l'encapsulation unifiante de supports d'exécution que nous proposons se devra d'être capable, si le besoin de processus légers est exprimé par l'application, de cibler n'importe quelle bibliothèque de processus légers, quelle que soit la nature de sa politique d'ordonnancement.

3.4 Interfaces orientées objet de supports d'exécution non orientés objet

Parmi les approches orientées objet, on trouve celles qui se résument à une simple interface orientée objet d'une API, telles celles étudiées précédemment.

Il existe par ailleurs des approches qui fournissent en réalité un modèle de programmation

et d'exécution où l'entité de programmation est l'objet réparti. Pour supporter de tels objets répartis lors de l'exécution, il est courant d'utiliser un des supports d'exécution décrits précédemment ou aussi d'en développer un ad-hoc. Nous reviendrons sur ces approches dans la prochaine section de ce chapitre.

On ne peut pas dire que les interfaces d'APIs — comme celles étudiées dans les 2 précédentes sections — constituées de bibliothèques de classes utilisables en programmation orientée objet constituent des **extensions** orientées objet : il s'agit généralement d'un simple habillage de pratiquement toutes les fonctions fournies par l'API (voir par exemple la définition de l'interface C++ pour Mpi définie dans la version 2 de la norme). Il n'y a pas à proprement parler de concept d'objets actifs et répartis. Tout simplement le calcul est réalisé par une entité hébergée par un processus lourd ou léger selon le choix du support d'exécution. L'entité de calcul est codée dans un langage hôte souvent orienté objet, dont l'API est étendue par des fonctionnalités d'échange de données et de création de nouvelles entités et ce par simple interfaçage d'un des outils présentés plus haut.

Néanmoins, un des objectifs de quelques travaux comme AdaPVM [Baude 94] ou Para++ [Couland 95b] est de profiter de cet habillage et de fonctionnalités offertes par le langage hôte pour simplifier la tâche du programmeur (par exemple, grâce à la généricité, on n'a qu'une seule fonction d'emballage de messages quelque soit le type des données le constituant) ou pour classer les concepts principaux et leurs dérivés à l'aide de graphes de classes [Bangalore 94].

On note parfois une tentative d'indépendance vis-à-vis de l'API et du support d'exécution. Par exemple, Para++ qui est une bibliothèque de classes C++ préconise une programmation classique à passage de messages, dont le support peut être soit Pvm, soit MPICH. Ainsi Para++ ne peut pas être une interface complète des fonctions d'aucune de ces 2 APIs et des choix ont donc dû être réalisés. Lorsque deux objets C++ veulent échanger un message, ils disposent des opérations classiques d'entrée et de sortie sur les "streams" (<< pour émettre, >> pour recevoir), dont la sémantique est asynchrone comme en Pvm. Ainsi, toutes les nombreuses autres fonctionnalités d'envoi de messages qu'offre Mpi ne sont pas exploitables.

Conclusion. Notre ambition étant de fournir plus qu'une simple interface orientée objet de supports d'exécution classiques pour la programmation parallèle et répartie, nous ne nous attarderons pas plus sur celles existantes. Retenons néanmoins leurs apports — purement orientés objet — que nous utiliserons dans notre prototype à savoir classification des concepts que doit manipuler le programmeur, utilisation de la généricité.

3.5 Supports de programmation et d'exécution à base d'objets répartis

L'étude des outils pour la programmation et l'exécution à base d'objets répartis fait l'objet de cette section. En général, ces outils se présentent au programmeur comme des bibliothèques de classes qui traitent des points liés au parallélisme et à la répartition, que le support d'exécution adapté à la circonstance, se charge ensuite de mettre en œuvre. Il se trouve certains exemples où il est nécessaire pour avoir accès à ces extensions de modifier par adjonction de mot-clés la syntaxe du langage. De telles modifications ne sont pas souhaitables, en particulier si l'on cherche à pouvoir réutiliser le code séquentiel. L'autre approche consiste à faire hériter de classes particulières certains objets, de sorte qu'ils aient un comportement étendu. Dans tous les cas, un précompilateur se charge ensuite de générer le code additionnel (stub, proxy, utilisation de classes dédiées à la gestion de la répartition, ...).

Cette partie de l'état de l'art cherche donc à illustrer sur quelques exemples les 2 facettes permettant de faire du parallélisme et de la répartition en orienté objet : 1°) au niveau programme utilisateur, quels concepts sont fournis et comment y avoir accès (par bibliothèques et/ou par mot-clés) ; 2°) comment ces concepts sont supportés lors de l'exécution.

Nous nous attarderons bien sûr sur le second point, car notre prototype SCHOONER s'y rapporte. En effet, la bibliothèque SCHOONER constitue un support d'exécution à base d'objets répartis où le souci majeur est la portabilité par le biais d'une unification des différents supports d'exécution sous-jacents possibles. Cependant, il est impossible dans les exemples sélectionnés de parler du second point, sans avoir parlé du premier.

Rappel sur les notions de *proxy* et de *stub*. Comme nous l'avons déjà évoqué plus haut (voir Section 3.3.2.1), dans les systèmes orientés objet permettant l'invocation de méthode distante, deux entités appelées *stub* et *proxy* sont responsables de la mise en œuvre de ces mécanismes (voir Figure 3.3). Généralement, le *stub* (ou *talon*) a pour rôle de construire un objet requête (contenant à la fois l'identifiant de la méthode distante à appeler et les paramètres d'appel) et de transmettre cet objet au *proxy* qui implémente la sémantique de communication. De plus, le *stub* possède la même interface que l'objet distant et pour l'objet appelant, l'appel reste donc identique à un appel de méthode classique et tout se passe comme s'il avait une réelle référence sur l'objet distant.

3.5.1 CORBA

Motivation. Afin de répondre à une demande de normalisation des systèmes à objets répartis, l'organisation OMG (Object Management Group) a défini un modèle de mise en œuvre de tels système appelé CORBA [Geib 97] (Common Object Request Broker Architecture).

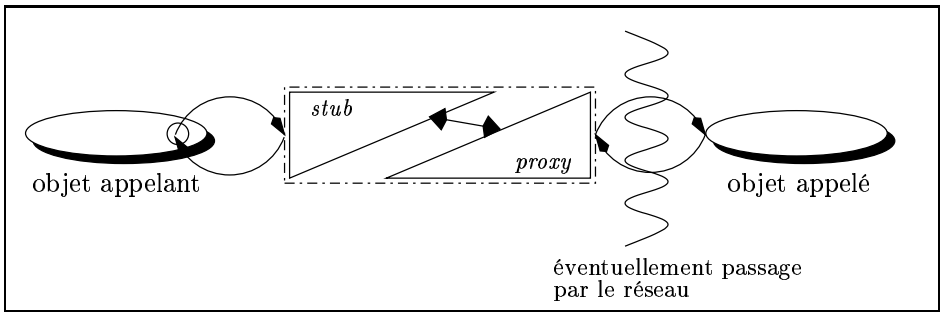

Figure 3.3 – *Architecture stub / proxy.*

Ce standard a pour but de faire communiquer des objets situés sur des plates-formes différentes grâce à l'utilisation d'un langage de définitions d'interface, l'IDL (Interface Definition Language).

Modèle de programmation. La figure 3.4 présente le modèle de référence établi par l'OMG : l'Object Management Architecture (OMA) qui donne une vision globale de la construction d'applications réparties.

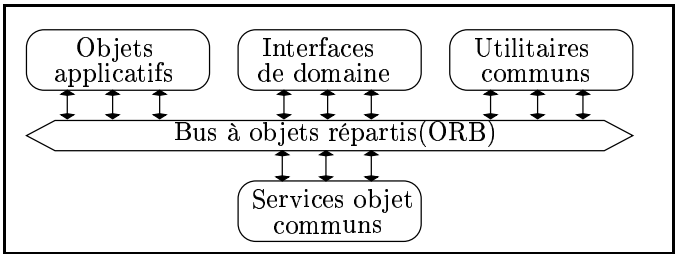

Figure 3.4 – *Vision globale de la construction d'applications réparties : OMA.*

Le bus d'objets répartis sera présenté dans le paragraphe suivant. Les différents types d'objets fournis par l'OMA permettent d'assurer l'interface entre le bus d'objets et les applications réparties et ce à différents niveaux.

- Les *services objet communs* fournissent les fonctionnalités de base nécessaires aux programmes objets répartis (comme par exemple des services pour le nommage, la persistance ou les transactions entre objets) et sont donc indépendants de tout type d'application.
- Les *utilitaires communs* ont le même rôle au niveau applicatif et standardisent l'interface utilisateur. Un exemple est l'utilitaire DDCF (Distributed Document Component Facility) qui normalise la présentation et l'échange d'objets de type document.
- Les *interfaces de domaine* sont spécifiques à un secteur d'activité comme la finance, la santé ou les télécommunications.
- Au dernier niveau, les *objets applicatifs* sont développés pour une application spécifique.

Pour cette raison, ils ne sont pas standardisés[5], mais peuvent le devenir s'il apparaît qu'ils sont utiles à plus d'une application (ils appartiennent alors à une des catégories précédentes).

Support d'exécution. La composante principale de CORBA est le bus d'objets répartis, l'ORB, Object Request Broker (voir Figure 3.5). L'ORB gère les interactions entre les différents objets. Il permet aux objets clients de s'abstraire totalement des mécanismes de communication et de localisation ou d'activation d'objets.

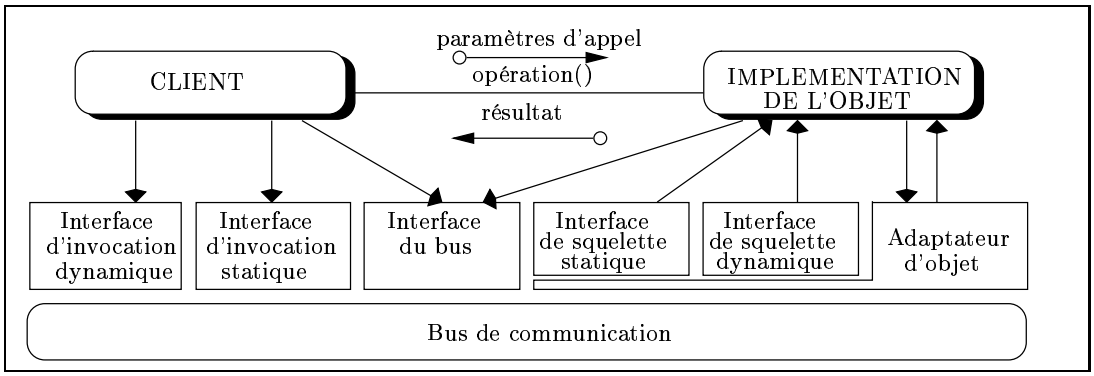

FIGURE 3.5 – CORBA *: Les composantes de l'ORB.*

L'interface de l'objet va être décrite en utilisant le langage IDL, puis compilée pour fournir deux nouvelles interfaces : 1°) l'interface d'invocation statique qui permet au client d'invoquer les méthodes de l'objet et 2°) l'interface de squelette statique dont l'implémentation de l'objet devra hériter.

Les interfaces dynamiques (invocation et squelette) permettent d'interagir directement avec les mécanismes de requêtes fournis par l'ORB sans passer par les interfaces statiques IDL.

L'interface du bus permet d'abstraire l'application de l'implémentation de l'ORB (qui peut par exemple être un ensemble de processus ou un ensemble de bibliothèques). Finalement, l'adaptateur d'objet est une interface entre l'implémentation d'objet et l'ORB qui permet entre autres l'enregistrement et l'activation des objets.

3.5.2 Java RMI

Java RMI [Microsystems 96, Microsystems 98] est un exemple de système utilisant les appels de méthode distante, conçu pour opérer au dessus du langage Java. Il est spécifiquement adapté à ce langage et peut de ce fait tirer avantage du modèle objet Java.

[5]L'OMG fournit des spécifications et non pas des applications.

Modèle de programmation. Les objets vont donc communiquer par appels de méthodes. Par héritage d'une classe spécifique, une classe peut spécifier que l'ensemble de ses méthodes publiques sont invocables de manière distante. Un identifiant global permettra aux autres objets d'obtenir une référence sur un objet de cette classe[6], l'appel de méthodes sur cet objet "distant" est ensuite analogue à tout autre appel même local.

Il est possible de spécifier des paramètres à ces fonctions distantes, la contrainte étant que ces paramètres soient des objets Java sérialisables (c'est-à-dire implémentant l'interface `java.io.Serializable`).

Ainsi au niveau strictement utilisateur, la transformation d'un programme Java en Java RMI est une opération qui est simple.

Support d'exécution et possibilités d'accès à ce support. La figure 3.6 présente l'architecture en couches du système Java RMI.

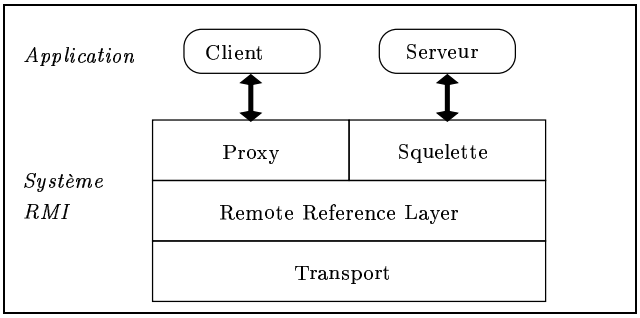

Figure 3.6 – *Architecture de Java RMI.*

Le proxy et le squelette sont identiques aux interfaces dynamiques de Corba [7] et permettent au client et au serveur de s'abstraire des mécanismes de répartition. Cependant, ces entités ne peuvent pas être redéfinies. De plus, elles sont générées automatiquement par le compilateur `rmic`.

La couche d'adressage globale (Remote Reference Layer) est responsable de la sémantique (ou protocole) de l'appel. On peut par exemple avoir une invocation point-à-point (il suffit que l'utilisateur fasse usage de la classe prédéfinie `UnicastRemoteObject`) ou invoquer la méthode sur un ensemble d'objets serveurs[8] (à condition de définir une classe appropriée).

La couche de transport est responsable de la gestion des connexions et permet de délivrer les requêtes à l'objet distant invoqué si ce dernier n'est pas dans son espace d'adressage. Celle-ci se base sur d'autres couches de transport de plus bas niveau, mais devant toutes être utilisables selon une sémantique de type socket. L'utilisateur peut même spécifier différentes natures de

[6] L'objet distant s'étant préalablement enregistré auprès du gestionnaire RMI.
[7] jusqu'à la version JDK 1.2 exclue, celle-ci ne disposant plus que de la notion de squelette
[8] On dispose donc de communication de groupes.

sockets grâce à la `RMISocketFactory`. Cependant, des protocoles de transport différents, tels ceux basés sur du *remote service request* ne peuvent en l'état actuel être interfacés. Pour qu'ils puissent l'être, il faudrait que la partie transport du système RMI propose une interface d'un niveau plus général. A titre de comparaison, SCHOONER répond à un tel besoin (mais en C++ uniquement).

D'une façon plus générale, il serait en fait souhaitable que le système supportant l'API Java RMI soit totalement ouvert, c'est-à-dire rende manipulable **toutes** les classes le composant et non seulement quelques unes. Ainsi, les utilisateurs chevronnés pourraient en modifier quelque peu le comportement, par exemple, en y introduisant du recouvrement calcul / communication comme nous avons pu le faire dans le contexte de C++// bâti au dessus de SCHOONER (voir Chapitre 9).

3.5.3 Eiffel//, C++//, Java//

Les travaux de [Caromel 91] permettent de rajouter simplement aux langages Eiffel, C++ et Java, par le biais de classes, des fonctionnalités d'objet actif, réparti, communiquant de façon transparente par le biais d'un mécanisme de RMI asynchrone, avec attente par nécessité. Nous ne détaillerons pas le modèle sous-jacent car nous y reviendrons dans le chapitre dédié à C++//.

Notons juste que le support d'exécution d'Eiffel// était conçu directement au dessus de la couche Socket. Il était donc portable. Cependant, il n'avait pas été conçu pour permettre une évolution en douceur, son rôle étant avant tout de valider les concepts proposés par le modèle. Ainsi, un objet `Process` d'Eiffel// ne peut être supporté que par un processus lourd. Toute la gestion de celui-ci, y compris les communications, est codée "en dur" dans le prototype.

Notre travail a donc eu comme retombée importante de proposer un modèle de support d'exécution pour les extensions Eiffel//, C++//, Java//. Il est clair que seul C++// en aura directement profité. Néanmoins, les idées proposées ont servi lors de la conception de Java//, même si celle-ci utilise Java RMI en tant que — et uniquement pour celà — mécanisme de base pour transporter les requêtes entre objets distants.

3.5.4 Converse

Converse [Kale 96b, Kale 96a, Kale 98] est un environnement d'exécution développé à l'Université d'Illinois à Urbana-Champaign. Il a pour principal objectif de regrouper les fonctionnalités essentielles d'un support d'exécution dans un noyau unique. Il supporte les paradigmes suivants : 1°) le modèle SPMD, 2°) le modèle basé sur l'échange de messages, 3°) la programmation orientée objet parallèle et 4°) la programmation basée sur les processus légers.

Les deux principaux atouts de Converse sont de permettre :
- un développement rapide de systèmes d'exécution pour de nouveaux paradigmes de programmation parallèle,
- la combinaison de plusieurs modules basés sur des paradigmes différents dans une même application.

L'architecture est conçue comme un ensemble de composants (voir Figure 3.7), chacun d'entre eux étant complètement accessible via une interface de programmation. L'implémentation de ces composants peut alors être modifié selon les besoins de l'application finale.

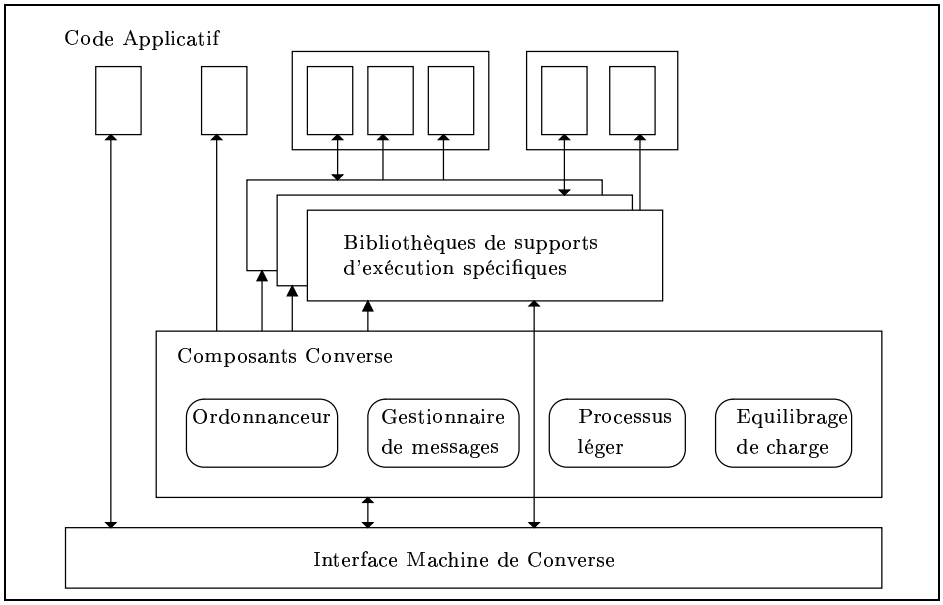

Figure 3.7 – *L'architecture en composants de converse*

La bibliothèque Schooner est conçue selon le même principe d'assemblages de modules. Par analogie à Converse, Schooner peut être vue comme un ensemble de modules plus un support d'exécution. Ainsi, Schooner ne permet pas de construire de nouveaux supports d'exécution spécifiques mais fournit elle-même un support.

Nous allons maintenant voir deux exemples de langages implémentés à l'aide de Converse : Charm++, un langage basé sur l'échange de messages et pC++, un langage data-parallèle. Les supports d'exécution de ces 2 langages ont été construit par assemblage des différents composants de Converse, ils se définissent donc comme des supports d'exécution spécifiques (voir Figure 3.7).

3.5.4.1 Charm++

Charm++ [Kalé 93] est un langage de programmation réparti consistant en une extension du langage C++. Un exécutable Charm++ peut être vu comme un ensemble d'entités actives,

réparties sur des machines, et communiquant par envoi de messages non bloquants. Charm++ distingue les objets séquentiels des objets parallèles, et le modèle d'exécution est basé sur l'envoi de message. Le partage de données entre objets parallèles est explicite et est introduit à travers des objets spécifiques.

Charm++ est en fait la version orientée objet du système Charm développé par la même équipe. Ce système permet de programmer des applications réparties indépendantes de la machine d'exécution.

Les premières versions de Charm et de Charm++ ont été développées avant Converse par la même équipe de chercheurs. Ces travaux ont ultérieurement abouti à la conception de Converse et finalement à la réimplémentation de Charm et Charm++ au dessus de Converse.

3.5.4.2 pC++

pC++ [Bodin 93a, Bodin 93b, Malony 94] est une extension parallèle objet du langage C++ développée à l'Université d'Indiana (USA). Elle fournit un modèle de programmation basé sur des structures de données distribuées, appelées *collection distribuée*. Cette collection est un ensemble structuré d'objets répartis sur les nœuds de calcul de la machine sous-jacente.

Ce modèle suppose donc un espace de mémoire partagé, simulé via le support d'exécution pour les machines à mémoire réparties. Un mécanisme de processus légers est également fourni pour supporter le parallélisme de tâches.

Le système d'exécution de pC++ est minimal, il fournit en effet seulement deux primitives de communication : `Get_Element` et `Get_Element_Part`. Ces primitives permettent aux processus légers d'accéder à n'importe quel élément en mémoire partagée. Le système doit pouvoir gérer les 3 classes d'opérations suivantes :

1. la création de nouvelles collections, ce qui implique de disposer sur chaque processeur d'un mécanisme d'identification des éléments.
2. la gestion des accès aux éléments de ces collections, cette gestion est faite via l'appel aux deux primitives de communication, `Get_Element` et `Get_Element_Part`, (implémentés soit par de la mémoire partagée soit par des communications).
3. la terminaison d'opérations collectives sur ces collections, ces opérations constituent des points de synchronisation sur l'ensemble des processus invoqués..

Ce système d'exécution appelé Tulip est donc construit à partir des composants fournis par l'environnement Converse. La compilation d'un programme pC++ consiste en fait en un traducteur de code qui convertit les constructions de pC++ en du code C agrémenté d'appels à l'exécutif Tulip.

3.5.5 Panda

Panda [Assenmacher 93a, Assenmacher 93b] est développé à l'Université de Kaiserslautern (Allemagne) et se présente comme un sur-ensemble strict de C++. Il introduit non seulement le parallélisme et la répartition mais aussi des mécanismes de persistance des objets et de ramasse-miettes.

Modèle de programmation. Un programme Panda est organisé en un ensemble d'objets communicants ; l'appel des méthodes de ces objets entraînent l'exécution d'une activité (processus léger). Tout objet non spécialement protégé peut exécuter simultanément plusieurs méthodes, ce qui génère donc du parallélisme intra-objet.

Le programmeur introduit une notion de parallélisme en héritant de la classe `UserThread` (voir Code 3.1). Après cette déclaration, l'initialisation d'un objet déclenchera en fait l'instanciation d'un objet actif (avec un processus léger associé). Ces objets auront par la suite la possibilité de migrer, explicitement via la méthode `migrate(... )`.

Code 3.1 *Déclaration et utilisation d'une classe active en Panda.*

```
class Increaser : public UserThread {
  ...
};
int main() {
  incPtr = new Increaser (10);
  ...
  incPtr→migrate(machine);
  ...
}
```

On a également la possibilité de placer n'importe quel objet dans une mémoire partagée répartie (via l'utilisation du mot-clé `DSM` lors de la création de l'objet). Une autre solution permettant d'obtenir un mécanisme global et transparent de localisation est basé sur la mobilité des processus légers. Ainsi, au lieu de migrer l'objet vers le processus léger demandant son accès, on peut choisir de déplacer le processus léger vers l'objet.

Support d'exécution. Panda est implémenté sous la forme d'un précompilateur (qui peut ajouter des synchronisations au niveau des accès concurrents aux objets actifs) et s'exécute en association avec un pico-kernel. Le système qui est conçu selon une approche orientée objet est divisé en 3 niveaux différents :

1. un pico-kernel avec une interface minimum c'est-à-dire ne fournissant que les appels "critiques" vis à vis des problèmes de protection et d'ordonnancement. Ceci permet de réduire la fréquence des appels vers le noyau, un des objectifs de Panda.

2. un support d'exécution qui gère notamment les processus légers au niveau utilisateur,
3. un niveau applicatif qui représente le code du programmeur.

Les objets à l'intérieur de chaque niveau constituent un ensemble de briques interchangeables, ce qui simplifie grandement la maintenance ou l'intégration de nouveaux ou plus spécifiques composants.

3.5.6 Gestion d'objets répartis par des bibliothèques d'abstraction

L'équipe SOR de l'INRIA a un long passé dans la conception de mécanismes orientés objet pour la répartition. Historiquement, ces mécanismes ont été définis dans le cadre de la conception d'un système d'exploitation orienté objet SOS [Shapiro 90]. Le système d'exploitation SOS fournit une bibliothèque d'appels systèmes pour la gestion d'objets élémentaires. En utilisant cette bibliothèque, le mécanisme d'objets fragmentés [Makpangou 94] a vu le jour, qui est une généralisation de l'usage de principe de mandataire [Shapiro 86] (proxy).

Les travaux ultérieurs concernant la gestion d'objets répartis font usage du concept d'Objet Fragmenté tout en étant détaché de SOS. Mais ces travaux suivent une approche qualifiée "système d'exploitation" au sens ou toute solution doit être générale, extensible, indépendante d'une classe d'applications ou de langages.

Citons BOAR [Makpangou 91] qui pour limiter la charge des programmeurs d'applications, offre un ensemble de bibliothèques de briques logicielles réalisant les politiques les plus couramment utilisées pour notamment, le contrôle de concurrence, la gestion de cohérence, la liaison, le placement des réplicats et la communication de groupe. BOAR comprend aussi une plate-forme — d'un niveau plus bas — qui offre les mécanismes de base nécessaires pour le support des objets répartis (par exemple, identification et localisation d'objets, invocation de méthodes à distance).

Citons aussi SSPC [Shapiro 92] qui étend l'usage du mécanisme de proxy en proposant celui de Chaînes de "Stubs-Scions" permettant de gérer l'adressage d'objets répartis, pouvant migrer et entre autre de faire du ramasse miettes.

3.5.7 Conclusion

Les outils présentés dans cette section permettent de gérer de manière relativement transparente des objets répartis en permettant notamment d'appeler des méthodes sur ces objets de manière identique à un appel classique. Les concepts sont proches de ceux que nous fournissons à travers la bibliothèque SCHOONER à savoir la possibilité de créer des objets sur un processus distant et par la suite de pouvoir communiquer avec ces objets. Toutefois, dans SCHOONER ces mécanismes sont directement manipulés par l'utilisateur, de ce point de vue, on pourrait

imaginer une implémentation possible de ces outils de gestion d'objets répartis au dessus de Schooner. Ceci a été d'ailleurs été fait dans le cadre de C++// comme nous le verrons dans le chapitre 9.

Chapitre 4

Bilan

✔ *Ce chapitre a pour but de faire un bilan sur l'ensemble des environnements présentés dans le chapitre précédent, afin d'en dégager les dénominateurs communs et de ce fait, justifier la conception du modèle de support d'exécution que nous mettrons en œuvre.*

4.1 Modèle proposé en fonction des besoins des applications cibles

Cherchant à concevoir un support d'exécution réparti "universel" pour les applications visées, il est normal de considérer ces applications selon le point de vue le plus proche de leur modèle à l'exécution.

Les applications que nous ciblons peuvent être schématisées par un graphe dynamique de tâches concurrentes nécessitant d'interagir pour s'échanger des données ou de manière plus simple comme un ensemble d'objets — peut importe leur granularité — interagissant. On parlera donc d'objet actif. En outre, le concepteur de l'application ne doit pas se soucier de l'ordonnancement de ces objets.

Ces applications doivent être portables et s'exécuter sur un ensemble de stations de travail — quelle que soit l'architecture de chacune d'entre elles — reliées par un réseau de communication que l'on peut considérer comme une *machine virtuelle*.

Les fondements communs des supports d'exécution de ces applications sont un besoin de pouvoir gérer un ensemble de processus lourds que l'on pourra appeler *clusters*, ces clusters

hébergeant alors les objets actifs. Ces clusters doivent être dans un premier temps créés : pour cela, l'utilisateur a besoin de manipuler individuellement les entités de sa machine virtuelle, que l'on peut appeler *computer*.

Le modèle le plus fédérateur concernant le mode d'interaction des entités du support d'exécution est le suivant : communication explicite (sans utilisation de mémoire — même virtuellement — partagée), par échange asynchrone de données, mais sans plus de choix à ce niveau, car cela pourrait alors conditionner les interactions à prendre la forme "passage de messages" ou "appel de procédure distante".

A notre niveau, il faut donc que le support d'exécution puisse transporter de façon asynchrone l'information, puis ensuite permettre de construire sur cette couche la forme d'échange réellement requise par le niveau applicatif.

Pour ce faire, les clusters auront accès à un mécanisme de communication élémentaire prenant la forme d'un mécanisme semblable à celui de *message actif*. Essentiellement, l'avantage par rapport à un mécanisme classique de passage de messages est que le récepteur n'a pas à aller s'enquérir de la présence de nouveaux messages : les messages sont en quelque sorte reçus automatiquement. De plus une méthode de traitement associée au message est automatiquement appelée lors de sa réception.

Quelle que soit la forme de l'échange de données proposée par le support sous-jacent utilisé, il est facile de s'en servir pour bâtir ce mécanisme de message actif. De plus, au dessus d'un tel mécanisme de message actif, il peut aussi être aisé de bâtir un mécanisme classique d'émission et réception tel celui utilisé dans les environnements basés sur l'échange de messages, ou bien un mécanisme fondé sur l'appel de méthode distante.

En théorie du moins, les applications que nous visons pourraient vouloir utiliser l'un ou l'autre mode. Cependant, les 2 grandes applications concrètes ciblées par ce travail se trouvent utiliser la deuxième forme (appel distant). Il a donc été difficile de ne pas faire transparaitre ces caractéristiques dans notre travail. En particulier, nous ne montrerons pas dans les détails comment on pourrait reconstruire au dessus de notre mécanisme de *message actif* un modèle classique à passage de messages.

L'encapsulation unifiante et extensible, dont la forme sera celle d'une librairie de classes C++, sera donc un support minimal — et grâce à l'approche objet — facilement extensible, permettant ainsi de supporter aussi bien des applications fonctionnant selon *l'échange de messages* ou *l'appel de méthode distante*.

Certaines applications auront également besoin de *multi-activité* afin d'obtenir des objets actifs de granularité différente. Toutefois, le modèle de gestion des processus légers notamment au niveau de l'ordonnancement n'étant pas unique (préemptif ou non par exemple), le modèle proposé ne doit pas être figé.

Ainsi, par le biais de l'encapsulation proposée, l'application pourra être en mesure, si elle le désire, d'accéder aux concepts classiques et basiques de gestion de processus légers (création, préemption, synchronisation, ...), ces concepts étant non seulement extensibles et surtout matérialisables par n'importe quel système de processus légers sous-jacent.

Un des besoins primordiaux permettant de construire des applications réparties avec un maximum de portabilité est de pouvoir bien isoler le code lié à l'application et celui relatif à la gestion de son support d'exécution. On obtient ainsi un portage complet au niveau applicatif.

Le support d'exécution "universel" que nous proposons, par le biais de l'encapsulation unifiante et extensible, peut être vu comme un ensemble de modules interchangeables avec une interface minimum regroupant les fonctionnalités de base utiles à toute application répartie. La modification ou la réécriture totale de l'implémentation d'un des modules n'aura aucune répercussion sur le code de l'application[1].

4.2 Introduction à SCHOONER

4.2.1 Objectifs de SCHOONER

L'encapsulation que nous proposons prend la forme d'une bibliothèque de classes appelée SCHOONER.

Cette encapsulation va utiliser le paradigme orienté objet pour répondre aux différents besoins précités. La portabilité vient également du fait que l'on utilise une approche par interfaces de niveau de plus en plus proche du support matériel cible. Bien que coûteuse, car engendrant de nombreux appels de fonction pour traverser les différentes interfaces, cette technique assurera la pérennité du support d'exécution qui ne doit pas dépendre trop fortement d'un outil existant et puisse facilement s'adapter à de nouvelles couches de support.

Le paradigme orienté objet permet entre autres l'utilisation de la généricité et une meilleure classification des concepts que doit manipuler le concepteur d'applications réparties.

A l'aide de cette interface de couches "basses", nous serons en mesure de construire un unique modèle sur lequel le programmeur ait à se baser. Il se trouve que ce modèle revient à *exposer* les différentes étapes et mécanismes qui rentrent en jeu pour réaliser de l'appel asynchrone de méthode entre objets distants. Mais après coup, ce n'est finalement pas un hasard, car en effet, c'est ce mode là d'interaction entre entités du niveau applicatif que notre encapsulation doit privilégier. Puisque, rappelons-le, c'est celui qui correspond aux deux grandes applications que nous devons cibler (C++// et PROSIT). Par contre, le fait que ces étapes élémentaires et ces mécanismes soient exposées et donc personnalisables s'avère

[1] tant que l'interface reste identique

très intéressant. En effet, nos deux grandes applications cibles pourraient être supportées par d'autres environnements orientés objet, comme par exemple des ORBs à la norme CORBA ou Java RMI (si écrites en Java et non en C++ !). Mais, de tels exemples ne sont pas aussi ouverts que peut l'être SCHOONER. Par exemple, en Java RMI, seul le type de sockets que l'on veut utiliser pour transporter les données est modifiable, pas le mode de transport lui-même. On pourrait très bien imaginer d'utiliser NEXUS à la place de sockets !

4.2.2 Synthèse sur les solutions existantes

Le tableau 4.1 reprend les caractéristiques principales des environnements constituant des supports d'exécution possibles pour le type visé d'applications réparties. Nous ne montrons ici que les supports d'exécution que nous considérons pouvoir être interfacés par SCHOONER (c'est-à-dire ceux présentés dans les sections 3.2 et 3.3). Les environnements présentés dans la section 3.5 offrent des services comparables à ceux de SCHOONER et nous ont permis d'étudier comment des problèmes similaires aux nôtres ont été résolus. Ils ne sont pas "interfaçables" par SCHOONER et de ce fait n'apparaissent pas dans cette section. Bien que ces solutions soient comparables à SCHOONER, nous ne les avons pas choisi pour principalement 2 raisons : 1°) nous voulions disposer d'un système ouvert afin de pouvoir y tester l'ajout de diverses techniques comme le recouvrement des communications par du calcul, la bufferisation de messages ou la génération de traces ; 2°) le modèle fourni à l'utilisateur n'était pas forcément celui requis par nos 2 applications finales (PROSIT et C++//).

Deux lectures du tableau 4.1 sont possibles. Dans une première, ce tableau indique quels supports d'exécution peuvent être choisis pour développer une application répartie donnée. Mais bien sûr, les caractéristiques du support choisi se répercutent sur la nature de l'application. Par exemple, si l'on choisit PVM, alors les applications développées sont constituées d'entités à gros grain, non multi-actives. Si l'on choisit PM^2, alors le mode d'interaction prend uniquement la forme d'appel de procédure à distance. La difficulté est que le programmeur d'application est tenté de concevoir en quelque sorte son application en fonction du support d'exécution final pressenti, nuisant souvent ainsi à l'isolation propre du code de l'applicatif de celui de la gestion de son support d'exécution. De plus, le support d'exécution envisagé ne dispose peut-être pas forcément d'une interface orientée objet, complexifiant encore plus son utilisation.

C'est pour pallier toutes ces difficultés — et leurs conséquences — que nous proposons d'une part d'encapsuler les mécanismes qui sont **communs** aux différents supports d'exécution pressentis — comme par exemple de la communication asynchrone ; d'autre part de bâtir au dessus un modèle unique et orienté objet de support d'exécution rudimentaire pour applications réparties utilisant des objets actifs : *computers*, *clusters*, *messages actifs* avec fonctions de mise à plat, reconstruction, traitement automatique, *objets communicants*, *proxy d'objets communicants*, ... Ce modèle rudimentaire consiste à *exposer* les différentes étapes

que l'on rencontre lorsque l'on veut communiquer par appel de méthode à distance au niveau applicatif. Finalement, notre dernier objectif est de rendre possible l'extension de ce modèle unique pour le personnaliser selon les besoins plus fins de l'application.

Ainsi dans une seconde lecture, les caractéristiques répertoriées dans le tableau 4.1 montrent ce que l'encapsulation que nous proposons devra elle-même définir — éventuellement par extension — lorsque le support d'exécution sous-jacent considéré sera tel ou tel autre. Par exemple, si l'on choisit d'interfacer PM^2, alors les mécanismes de processus légers que PM^2 fournit sont interfaçables directement (si bien sûr le modèle offert convient). Par contre, en choisissant PVM il faudrait que l'encapsulation les construise en son sein au prix d'un travail plus important. De la même façon pour pouvoir fournir à une application désirant communiquer par appel de méthode à distance les primitives de haut niveau qu'elle requiert, alors que le support sous-jacent ne propose pas du tout ce mode d'interaction, le rôle de l'encapsulation sera là primordial. Cette remarque s'illustre parfaitement si, par exemple, on utilise PVM de façon sous-jacente, alors que le modèle de communication du niveau applicatif est de l'appel de méthode à distance.

	Processus légers	Communication asynchrone	Appel de méthode distante
PVM	N	O	N
MPICH	N	O	N
PT-PVM	O	O	O
TPVM	O	O	O
PM^2	O	O	O
NEXUS	O	O	O

TABLEAU 4.1 – *Récapitulatif des solutions existantes*

4.3 Plan de la suite du mémoire

La **partie II** de ce mémoire va donc présenter la librairie SCHOONER que nous avons réalisée afin de servir de support d'exécution à des applications réparties. Rappelons que l'encapsulation orientée objet de supports d'exécution permet d'obtenir une portabilité complète au niveau applicatif. L'étude des supports existants nous a en effet permis de préciser quel pourrait être le modèle unificateur qui est fourni aux développeurs d'applications réparties. Les caractéristiques essentielles de ce modèle sont la présence d'entités d'exécution (*clusters* et *objets communicants*) communiquant via un mécanisme de message actif tel décrit en Section 4.1. Les spécifications de ce modèle seront décrites dans le chapitre 5.

Ce chapitre présente également quelques aspects de l'implémentation de la bibliothèque

SCHOONER et donne des résultats de performance pour les mécanismes de base.

La possibilité de redéfinir le comportement de base permettra entre autres — comme nous le verrons dans le chapitre 6 — l'introduction aisée de la multi-activité. Cette redéfinition n'entraine aucune modification du modèle et permet de diminuer la granularité des objets communicants.

Le chapitre 7 présente l'environnement fourni autour de SCHOONER. Cet environnement (voir Figure 4.1) est constitué :

- de modules redéfinissant le comportement de base comme la bufferisation (voir Section 7.2.2) ;
- d'extensions des fonctionnalités du modèle. Un exemple d'extension permettant d'obtenir un recouvrement entre calcul et communication sera présenté dans le cadre de C++// (voir Section 9.3) ;
- d'intégration d'outils existants par interfaçage qui a permis notamment de coupler SCHOONER avec un outil d'équilibrage de charge (voir Section 7.2.1).
- d'outils d'aide au développement comme le moniteur graphique (voir Section 7.1.1) qui sont en fait des applications SCHOONER spécifiques. En tant qu'applications SCHOONER, nous aurions pu également choisir de les présenter dans la partie III.

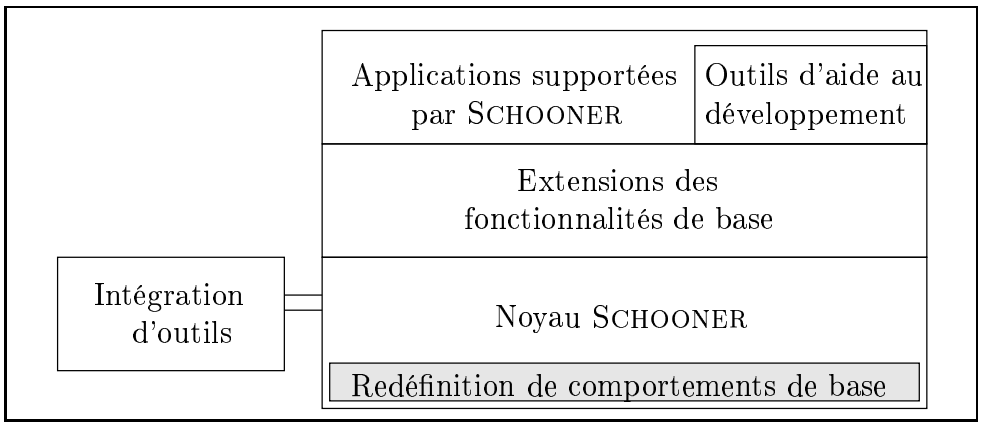

FIGURE 4.1 – SCHOONER *et son environnement.*

La **partie III** va décrire quelques applications construites au dessus de l'environnement SCHOONER. Le chapitre 8 décrit la version répartie du simulateur PROSIT ainsi qu'un exemple de client serveur hiérarchique.

Le chapitre 9 décrit une implémentation de C++// au dessus de SCHOONER. On montre ainsi comment une application grandeur réelle est supportée par SCHOONER et quels en sont alors les avantages : l'encapsulation qui permet de manipuler à sa guise les mécanismes de base fournis par le support et de fournir par programmation un support d'exécution adéquat à l'application visée (sans modifier l'application elle-même).

La figure 4.2 reprend l'ensemble des applications construites au dessus de SCHOONER.

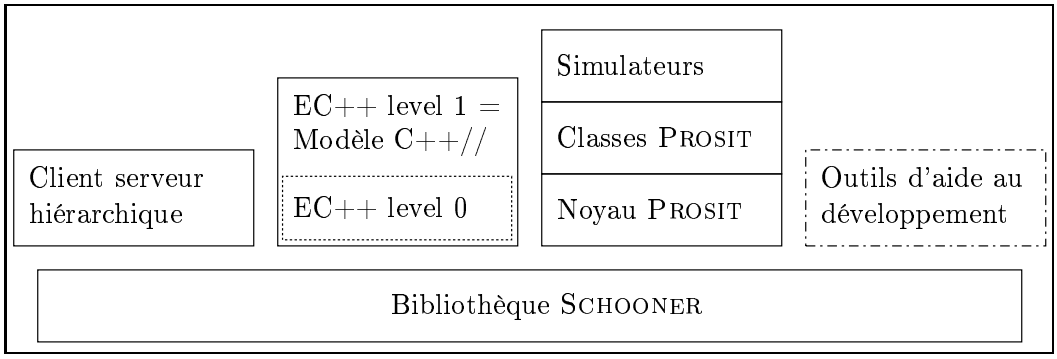

FIGURE 4.2 – *Les applications* SCHOONER.

Deuxième partie

SCHOONER : Un support d'exécution orienté objet pour applications réparties

Chapitre 5

Le modèle SCHOONER

✔ *Ce chapitre va présenter les objectifs et concepts de la bibliothèque* SCHOONER *de base, principalement en décrivant l'interface de programmation et son utilisation.*

5.1 Introduction

Le chapitre précédent a présenté les justifications du modèle de programmation que nous fournissons via la bibliothèque SCHOONER. Rappelons que cette bibliothèque doit être capable de répondre aux besoins principaux suivants :

- organiser une application répartie en un graphe dynamique de tâches de granularité différente,
- pouvoir faire interagir ces tâches via de l'échange de données,
- avoir la possibilité de pouvoir gérer des processus légers, sans toutefois avoir un modèle figé pour leur ordonnancement,
- garantir la portabilité de l'application en fournissant un support minimal.

Afin de fournir un début de solution à ces différents besoins, la bibliothèque SCHOONER de base s'organise en 3 modules :

1. une interface définissant le protocole de communication et plus généralement un ensemble minimum de mécanismes pour l'exécution d'applications concurrentes, qu'elles soient à gros grain ou à grain fin. L'interface de la bibliothèque de processus légers présentée sur la figure 5.1 ne fait pas partie du modèle de base, mais de l'extension multi-active de SCHOONER (voir Chapitre 6, page 71) ;
2. les clusters qui représentent les nœuds de calcul de l'application (entités à gros grain) ;

3. les objets communicants qui permettent d'obtenir un grain de parallélisme plus fin.

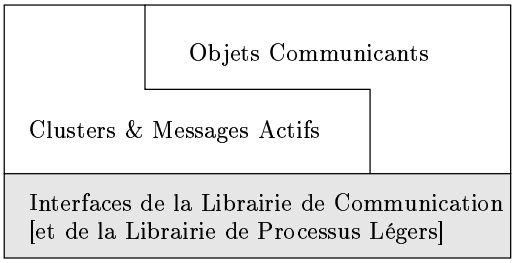

FIGURE 5.1 – *Les différents modules de* SCHOONER.

Les deuxième et troisième modules regroupent les entités de SCHOONER visibles et modifiables par l'utilisateur final. Chacune de ces entités est représentée par un objet instance d'une classe. L'utilisation de la programmation orientée objet permet une redéfinition aisée du comportement de base de ces entités — comme nous le verrons dans la suite de ce mémoire.

Ce chapitre ainsi que le suivant vont présenter le modèle de fonctionnement ainsi que le modèle d'interaction des différentes entités fournies par la bibliothèque SCHOONER. Les sections 5.3 et 5.4 présentent le deuxième module de la bibliothèque, le troisième module est détaillé dans la section 5.5.

5.2 Le module bas-niveau de SCHOONER

Le parc de machines. On suppose que l'on dispose d'un ensemble de machines dont les architectures peuvent être hétérogènes reliés par un réseau de communication (voir Figure 5.2). Ce réseau doit être capable d'effectuer des communications point-à-point entre chacune des machines présentes, et on suppose que ces communications sont FIFO[1].

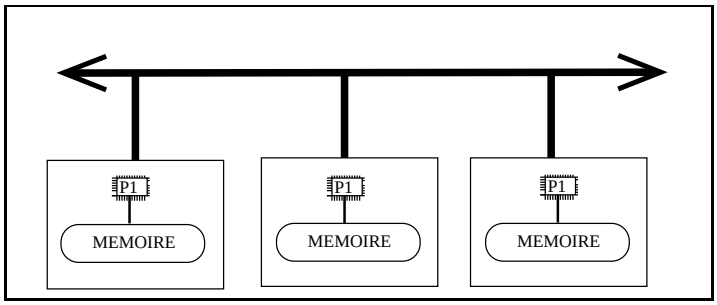

FIGURE 5.2 – *Exemple d'un parc de machines reliées par un réseau de communication.*

[1]L'ordre d'arrivée des messages sur le processus B depuis le processus A est le même que l'ordre de départ des messages du processus A vers le processus B, même si A et B s'exécutent sur deux machines distinctes.

La bibliothèque de communication. Toutes les fonctionnalités nécessaires à la gestion des processus lourds (création, communication, ...) sont regroupées dans une classe SCHOONER spécifique qui constitue ainsi l'unique point d'entrée vers la bibliothèque de communication interfacée (voir Section 5.6.2, page 66).

La bibliothèque de processus légers. De même, les fonctionnalités nécessaires à la gestion des processus légers sont regroupées dans une classe SCHOONER spécifique (voir Section 5.6.2, page 66). On dispose également d'une classe permettant d'effectuer manuellement l'ordonnancement des processus légers (à utiliser dans le cas où la bibliothèque interfacée ne serait pas préemptive).

Le deuxième module de SCHOONER permet de considérer une application comme un ensemble de nœuds de calcul répartis sur une machine parallèle virtuelle. On désigne par le terme *machine parallèle virtuelle* une architecture répartie faiblement couplée. L'ensemble de ces machines est relié par un réseau. On pourra aussi bien avoir des stations de travail que des machines multiprocesseurs sans mémoire commune. Globalement, la vision de ce parc de machines pour l'utilisateur est un ensemble de machines individuelles sans mémoire partagée reliées par un réseau de communication.

5.3 La machine virtuelle

Un *computer* (voir Code 5.1) permet de représenter une des machines cette machine virtuelle. Il est caractérisé par un nom physique (nom réseau de la machine), un nom symbolique (choisi par l'utilisateur) et par son architecture (le système d'exploitation).

Code 5.1 *Interface (partielle) de la classe* `Computer`.

```
class Computer {
protected:
  string symbolic_name;                                  // nom symbolique
  string real_name;                                      // nom réel de la machine
  Computer_Arch arch;                                    // architecture du processeur
public:
  Computer(string &, string &, Computer_Arch);           // constructeur
  static Computer *any();                                // n'importe quel computer
  static Computer *best();                               // le "meilleur" computer
  static Computer *specific(string &);                   // le computer avec le nom symbolique donné
};
```

Plusieurs objets `Computer`[2] avec des noms symboliques différents peuvent être associés à la même machine physique. L'utilisation des noms symboliques permet de définir facilement une architecture virtuelle qui peut être projetée sur différentes configurations physiques.

L'utilisation d'un fichier de configuration qui est lu au démarrage d'une application SCHOONER permet également de définir la machine virtuelle de l'application (qui est une partie ou la totalité du parc de machines effectivement disponible).

5.4 Les clusters

Cette section présente les *clusters* qui sont les nœuds de calcul de l'application.

Un *cluster* (voir Code 5.2) est une entité de calcul qui pourra être créée sur n'importe quel *computer* de la machine virtuelle. Il représente un espace d'adressage (contexte mémoire, CPU, ...) sur une machine et exécute son propre code.

Des notions similaires sont présentes dans d'autres librairies : *processus environnement* de PT-PVM, *context* en NEXUS ou bien encore *module* de PM^2. Contrairement à ces bibliothèques où l'équivalent des clusters ne représente qu'un container pour des entités à grain de parallélisme plus fin, on a la possibilité en SCHOONER d'avoir une application ne manipulant que des objets `Cluster`.

Dans la plupart des cas, un cluster sera mis en œuvre par un processus UNIX. Il est également possible en modifiant l'allocateur de la classe `Cluster` de grouper plusieurs ou l'ensemble des clusters sur un même processus UNIX. On peut ainsi contrôler facilement le nombre de processus utilisés au niveau de l'application ou au niveau de chaque *computer*.

Code 5.2 *Interface (partielle) de la classe* `Cluster`.

```
class Cluster {
protected:
  Cluster_Id *id;                                   // identifiant du cluster
public:
  bool destroy();                                   // destruction du cluster
  void *operator new (size_t , Computer *);         // allocateur
  static Cluster *any();                            // n'importe quel cluster
  static Cluster *best();                           // le "meilleur" cluster
  static Cluster *current();                        // le cluster courant
};
```

[2]La fonte `Chasse fixe` est utilisée dans ce manuscrit pour désigner les noms de classes ou d'instances de classes.

5.4.1 Création de clusters

A la création, un cluster se voit attribuer un identifiant, qui est par définition unique. La localisation du cluster est spécifiée à sa création à l'aide d'une instance de la classe `Computer`. Le code 5.3 donne des exemples de création de nouveaux objets `Cluster`. Il y a deux façons de préciser la localisation d'un nouvel objet `Cluster` :

1. le programmeur peut spécifier à l'allocateur sur quel objet de la classe `Computer` démarrer le cluster. Cela se réalise en passant comme paramètre à l'allocateur, soit une instance de la classe `Computer` (voir Code 5.3, `cl1`), soit le nom symbolique d'un *computer* préalablement créé (voir Code 5.3, `cl2`) ;
2. par défaut, le nouveau `Cluster` sera créé sur le `Computer` par défaut qui est le `Computer` sur lequel se trouve le cluster demandant la création (voir Code 5.3, `cl3`).

Ces deux méthodes permettent de couvrir l'ensemble des possibilités pour préciser la localisation d'un nouveau cluster. Il est de plus possible d'enrichir la méthode de sélection du "meilleur" *computer* de la machine virtuelle. Nous verrons dans la section 7.2.1 comment ceci peut par exemple être fait en interfaçant SCHOONER à un outil d'équilibrage et de répartition de charge.

Code 5.3 *Création de clusters.*

```
Computer *cp = new Computer("machine_1", "lorien.m_earth.fr") ;
Cluster *cl1, *cl2, *cl3 ;
cl1 = new (Computer::best()) Cluster ;          // création sur le "meilleur" computer
cl2 = new ("machine_1") Cluster ;                // création sur un computer donné
cl3 = new Cluster ;          // création sur le computer par défaut (computer courant)
```

5.4.2 Interactions entre clusters

Le mode de communication interclusters est conçu comme un mécanisme de *messages actifs* [von Eicken 92] qui permet de considérer l'échange de messages comme une opération à sens unique contrairement au modèle classique "send-receive". Dès qu'un message est reçu par un cluster, la méthode de traitement associée à ce message est automatiquement appelée.

Dans SCHOONER, un message actif est une instance d'une classe qui dérive (directement ou indirectement) de la classe `Message` (voir Code 5.4). Cette classe définit principalement les méthodes suivantes :

- `flat(... )` qui permet de linéariser (aplatir) un message (le transformer en une suite d'octets) afin de pouvoir le transmettre à un autre cluster (si besoin via le réseau) ;
- `rebuild(... )` qui permet de reconstruire un message à partir de sa forme linéarisée ;
- `process()` qui est la fonction de traitement du message. Cette fonction est appelée automatiquement quand un nouveau message de cette classe est reçu par un cluster.

Les fonctions d'aplatissement et de reconstruction doivent être définies pour toute nouvelle classe de message. Nous verrons dans le chapitre 9 un cas spécifique où nous avons pu les rendre génériques.

La classe `Message` dispose d'une opération de communication point-à-point (méthode `send(... )` qui permet d'envoyer le message à un cluster spécifique) et d'une opération de communication multipoint (méthode `mcast(... )` pour envoyer le même message à une liste de clusters). Une opération de communication globale sur un groupe de clusters — bien que non fournie dans le modèle de base — pourrait être construite au dessus de ces primitives.

Code 5.4 *Interface (partielle) de la classe* `Message`.

```
class Message {
public:
  virtual Buffer *flat(Buffer *buff)=0;                          // aplatissement
  static Message *rebuild(Buffer *buff, Message *msg=0);         // reconstruction
  virtual void process()=0;                                      // traitement

  virtual Cluster *get_sender();                                 // cluster émetteur du message
  virtual void send(Cluster *);                                  // envoi du message à un cluster
  virtual void mcast(Cluster_List *);                            // envoi du message à une liste de clusters
};
```

Pour définir une nouvelle classe de messages actifs, il suffira — comme on peut le voir dans le code 5.5 qui permet de demander à un cluster de calculer la factorielle d'un nombre n — de :

1. définir une nouvelle classe qui hérite de la classe `Message` avec les données associées ;
2. définir la méthode d'aplatissement et la méthode de reconstruction ;
3. définir la méthode de traitement automatique du message.

5.5 Les objets communicants

Le troisième module de SCHOONER offre un modèle de calcul basé sur des *objets communicants*, c'est-à-dire qu'une application construite au dessus de ce module peut être vue comme un ensemble d'objets interagissant par échange asynchrone de données. On peut avoir 1 ou plusieurs *objets communicants* placés sur un même cluster.

Code 5.5 *Définition d'une nouvelle classe de messages actifs pour le calcul de factorielle.*

```
class Message_Fact : public Message {
protected:
  int n;
  int fact(int x);                                          // calcul de fact(x)
public:
virtual Buffer *flat(Buffer *buff) {
  *buff << n;
}
static Message *rebuild(Buffer *buff, Message_Fact *msg) {
  int x; *buff >> x;
  return new Message_Fact(x);
}
virtual void process() {
  Message_Fact_Reply *msg = new Message_Fact_Reply(fact(n));
                                  // Message_Fact_Reply est une autre classe de messages actifs
  msg→send(get_sender());                                   // get_sender() héritée de Message
}
};
```

5.5.1 Création d'objets communicants

La création d'un nouvel objet communicant est faite via la création d'un objet *proxy*. Un *proxy* est le représentant local de l'objet avec qui n'importe quel autre objet va pouvoir communiquer et qui peut se trouver sur un autre *cluster* (*cluster* lui-même local ou sur un *computer* distant). C'est la création du proxy qui va automatiquement créer un objet communicant et ce sur le cluster spécifié en paramètre à la création du proxy. Chaque objet communicant a un identifiant unique qu'il suffira de transmettre à n'importe quel cluster de l'application afin que celui-ci référence l'objet communicant. Une fois l'objet communicant créé, le proxy qui le représente va permettre de lui envoyer des données.

Comme pour la création d'un nouveau cluster, il y a deux manières de spécifier la localisation du nouvel objet communicant. Il est possible de préciser au constructeur du proxy le cluster cible (voir Code 5.6, `p1`) ou sinon le noyau de SCHOONER choisira le "meilleur" cluster de l'application (voir Code 5.6, `p2`).

Code 5.6 *Création d'objets communicants.*

```
  Proxy_Comm_Object *p1, *p2;
  p1 = new Proxy_Comm_Object(cl1);                          // création sur un cluster donné
  p2 = new Proxy_Comm_Object;                               // création sur le "meilleur" cluster
```

La classe `Comm_Object` présentée dans le code 5.7 n'est pas manipulée directement, mais de façon transparente par la manipulation des objets proxy. Chaque cluster de l'application dispose d'un gestionnaire d'objets communicants (voir Section 5.6.1, page 65) qui permet entre autres de s'assurer qu'il n'existe sur un cluster qu'un seul objet proxy pour un même objet communicant. Pour des raisons de cohérence, la création d'un objet communicant sur le cluster local doit également se faire via la création d'un proxy.

Code 5.7 *Interface (partielle) de la classe `Comm_Object`.*

```
class Comm_Object {
public:
  virtual void receive(Data_Comm_Object *);
                          // fonction appelée lorsque de nouvelles données sont reçues pour l'objet
  static Comm_Object *create();
        // constructeur statique appelé automatiquement suite à la demande de création d'un proxy
  virtual void destroy();                                          // destruction de l'objet
};
```

5.5.2 Interactions entre objets communicants

L'envoi de données à un objet communicant est faite à travers l'objet proxy qui le représente. Les données à envoyer doivent être encapsulées dans une classe qui dérive (directement ou indirectement) de la classe `Data_Comm_Object` (voir Code 5.8). Pour faire l'analogie avec les messages actifs, nous désignerons, dans la suite de ce mémoire, par le terme *données actives* des objets instances d'une classe dérivant de `Data_Comm_Object`.

Code 5.8 *Interface (partielle) de la classe `Data_Comm_Object`.*

```
class Data_Comm_Object {
public:
  Comm_Object get_receiver();          // objet communicant auquel sont destinées ces données
  virtual Buffer *flat(Buffer *buff)=0;                                    // linéarisation
  static Data_Comm_Object *rebuild(Buffer *buff, Data_Comm_Object *d=0);   //
  reconstruction
  virtual void process()=0;                                                // traitement
  Proxy_Comm_Object *get_sender();
                // proxy sur l'objet communicant ayant envoyé les données (peut être non spécifié)
};
```

Lors de la réception de données pour un objet communicant, la méthode `receive(...)` (voir Code 5.7) sur cet objet communicant est automatiquement appelée. Par défaut, cette fonction appelle la méthode de traitement des données reçues (voir Code 5.8, méthode

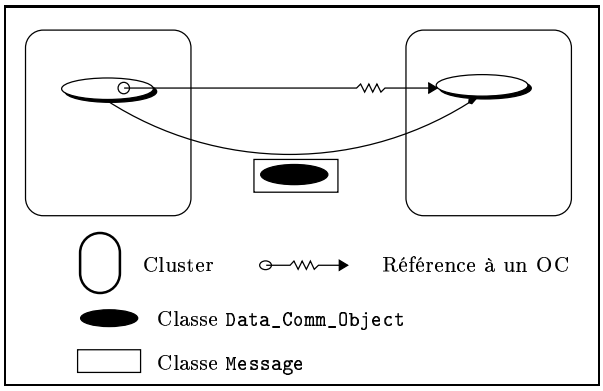

FIGURE 5.3 – *Envoi de données à un objet communicant.*

`process()`). En spécialisant cette fonction pour chaque nouvelle classe de données, il est ainsi possible d'avoir un traitement spécifique selon les données reçues.

Comme on peut le voir dans le code 5.9, la définition d'une nouvelle classe de données est comparable à celle concernant un message actif :

1. définition d'une nouvelle classe qui hérite de la classe `Data_Comm_Object` avec les données associées ;
2. définition de la méthode d'aplatissement et de la méthode de reconstruction ;
3. définition de la méthode de traitement automatique des données.

Code 5.9 *Définition d'une nouvelle classe de données actives pour le calcul de factorielle.*

```
class DCO_Fact : public Data_Comm_Object {
protected:
  int n;
  int fact(int x);                                          // calcul de fact(x)
public:
virtual Buffer *flat(Buffer *buff) {
  *buff << n;
}
static Data_Comm_Object *rebuild(Buffer *buff, DCO_Fact *data) {
  int x; *buff >> x;
  return new DCO_Fact(x);
}
virtual void process() {
  DCO_Fact_Reply *data = new DCO_Fact_Reply(fact(n));
                                  // DCO_Fact_Reply est une autre classe de données actives
  data→send(get_sender());                    // get_sender() héritée de Data_Comm_Object
}
};
```

On peut voir que le code 5.9 — qui définit une classe pour l'échange d'information entre objets communicants — est quasiment identique au code 5.5 — qui définit une classe pour l'échange d'information entre clusters. La principale différence est que dans le premier cas, il faille hériter de la classe SCHOONER `Message` et dans le deuxième cas de la classe SCHOONER `Data_Comm_Object`. Il est donc relativement facile d'adapter la granularité d'une application à l'utilisation des clusters ou des objets communicants.

5.5.3 Redéfinition du comportement des objets communicants

Une simple dérivation de la classe `Comm_Object` permet de définir un nouveau type d'objets communicants. La nouvelle classe devra au minimum définir une fonction statique de création. Dans la plupart des cas, cette fonction a juste à appeler l'opérateur de construction et retourner une référence sur le nouvel objet créé.

La spécification du comportement de la nouvelle classe se fait principalement par la redéfinition des trois méthodes suivantes :

1. l'opérateur de construction qui sera appelé sur le cluster cible via la fonction statique de création. La classe de base `Comm_Object` a un constructeur par défaut qui contient les mécanismes d'enregistrement du nouvel objet communicant, le constructeur de `Comm_Object` sera donc appelé automatiquement lors de la construction d'un objet d'une classe dérivée.

2. la méthode de réception de données (`receive(Data_Comm_Object)`, voir Code 5.7, page 62). Cette fonction est automatiquement appelée quand de nouvelles données sont réceptionnées — par le `Cluster` — pour un objet communicant, qu'il héberge. Les données reçues sont passées en paramètre à cette fonction. Par défaut, cette fonction ne fait qu'appeler la méthode de traitement associée aux données (fonction `process()` du Code 5.8). Il est par exemple possible d'ajouter un mécanisme permettant de compter le nombre de messages reçus pour un objet communicant donné ;

3. la méthode de destruction de l'objet (`destroy()`, voir Code 5.7, page 62) qui est automatiquement appelée lorsque le proxy demande la destruction de l'objet qu'il représente. Si cette méthode est redéfinie, il est essentiel que la méthode de base soit quand même appelée (mécanisme de suppression de l'enregistrement de l'objet communicant).

La possibilité de redéfinir les objets communicants sera notamment utilisée pour l'ajout de la multi-activité dans SCHOONER (voir Chapitre 6, page 71), en définissant des objets communicants multi-actifs.

5.6 Implémentation

Cette section décrit brièvement l'architecture et l'implémentation de SCHOONER. Afin de rendre SCHOONER modulable et facilement extensible, les mécanismes de base sont encapsulés au sein de classes. Ces classes sont invisibles à l'utilisateur SCHOONER "final", mais doivent être connues d'un utilisateur étendant SCHOONER pour des besoins originaux (comme nous en verrons dans le chapitre 7). Nous allons maintenant décrire ces différentes classes.

5.6.1 Les différents gestionnaires

A l'exécution, chaque nœud de calcul possède une instance de chacune des entités suivantes :

1. un gestionnaire de communication,
2. un gestionnaire de configuration,
3. un gestionnaire d'objets communicants.

Le gestionnaire de communication. Ce gestionnaire qui est matérialisé par la classe `Communication_Manager` regroupe les fonctions permettant l'échange de données entre clusters (émission vers un ou plusieurs clusters et réception).

Des méthodes pour comptabiliser le nombre de messages reçus et émis par cluster sont fournis et seront principalement utilisées pour la détection de propriétés stables (voir Section 8.1.3, page 96).

Ce gestionnaire sera notamment redéfini pour prendre en compte la bufferisation de messages (voir Section 7.2.2, page 87).

Le gestionnaire de configuration. Cette classe regroupe toutes les méthodes de gestion de l'ensemble des clusters et des computers de l'application. Elle fournit par exemple la méthode `wait_creation_clusters()` qui permet d'attendre un message de confirmation de l'ensemble des clusters créés par le cluster courant.

Le gestionnaire d'objets communicants. Ce gestionnaire centralise toutes les opérations de gestion des objets communicants (création d'identifiant, création de proxy, enregistrement d'un nouvel objet communicant). Il est matérialisé par la classe `Comm_Object_Manager` qui sera par exemple redéfinie pour implémenter le modèle des objets actifs communicants de C++// (voir Section 9.2, page 105).

La redéfinition de l'un de ces gestionnaires sera prise en compte lors de l'appel de la fonction d'initialisation du système SCHOONER (fonction qui doit être appelée en début de tout programme SCHOONER et qui prend en paramètre les objets gestionnaires à utiliser).

La figure 5.4 présente le schéma OMT montrant les liens entre ces différentes classes et la classe `Cluster`.

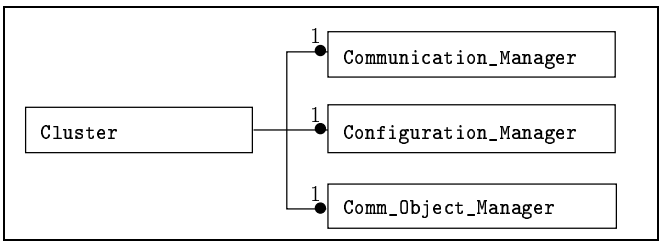

FIGURE 5.4 – *Schéma de relation OMT des gestionnaires.*

5.6.2 Les couches de support

La bibliothèque de communication interfacée. La classe `System` de SCHOONER regroupe toutes les primitives de base qui dépendent du réseau et de la bibliothèque de communication utilisés (création de processus, communication point-à-point et multipoint entre processus, ...). Seule cette classe devra être modifiée pour le portage de SCHOONER sur un autre type de machine ou pour changer la couche bas-niveau de communication.

Actuellement, la bibliothèque PVM est interfacée. Cette bibliothèque qui est reconnue comme un standard de fait pour les bibliothèques de communication possède l'avantage d'être portée sur un grand nombre de systèmes et est disponible sur plus de 30 architectures différentes.

Une implémentation au dessus de MPI est également envisageable, si ce n'est l'absence de processus dynamiques, les modèles sont les mêmes. Pour pallier à ce problème, il est également possible d'interfacer la bibliothèque Para++ [Couland 95b, Couland 95a] qui est une interface orientée objet pour les bibliothèques à échange de messages et qui encapsule les notions de processus et de communications interprocessus[3].

SCHOONER aurait pu également se baser directement sur les protocoles TCP/IP et/ou UDP, en utilisant l'API des sockets [Stevens 90].

La bibliothèque de processus légers interfacée. La classe `Thread` regroupe toutes les primitives de base nécessaires à la gestion de processus légers. Une classe `Semaphore` est également disponible afin de synchroniser les processus légers lors d'accès concurrents. Dans

[3]Une version est disponible au dessus de PVM, une autre au dessus de MPI.

le cas d'une bibliothèque de processus légers non-préemptive, la classe `Scheduler` fournit des fonctions permettant de réaliser manuellement l'ordonnancement. Nous reviendrons plus en détail sur l'implémentation des processus légers dans le chapitre 6.

5.6.3 Réception des messages

L'implémentation actuelle du modèle de base de SCHOONER ne fournit pas dans sa totalité un modèle de "messages actifs". En effet, la réception des messages au niveau des objets clusters n'est pas faite automatiquement et ceux-ci une fois initialisés doivent régulièrement penser à aller s'enquérir de la présence de messages depuis la couche de communication.

Par défaut, la mise en œuvre des messages actifs qui est réalisée au niveau du cluster se schématise par la boucle infinie suivante :

- effectuer — au niveau du module "Interfaces" de SCHOONER— la réception bloquante d'un message depuis la couche de communication ;
- ce qui une fois le message reçu déclenche automatiquement la fonction de traitement associée à ce message (destiné au cluster lui-même ou à un de ses objets communicants).

On voit immédiatement que

- pour pouvoir intercepter les messages depuis la couche de communication, le cluster n'effectue aucun calcul "utile" ou doit s'arrêter régulièrement dans son calcul ;
- tant que le traitement d'un message n'est pas terminé, il ne se passe *rien d'autre* sur le cluster ;
- et qu'ainsi il n'y a pas de concurrence à l'intérieur d'un cluster[4].

Ainsi, le mécanisme de messages actifs alors qu'il n'y a pas vraiment de calcul en cours privilégie le modèle de programmation à "appel de services distants" (à la manière de ce qu'offrent les bibliothèques d'appels de services distants telles PM^2 ou NEXUS). Cependant, en SCHOONER, les demandes de services ne peuvent pas être servies de façon concurrente sur un même nœud. Les objets communicants sont en effet des objets inertes puisqu'une fois créés, les seuls traitements[5] consistent à exécuter la fonction de réception pour chacune des nouvelle données reçues. L'intérêt du niveau objets communicants même passifs est néanmoins de permettre de structurer le serveur en serveurs de granularité plus fine (voir Section 8.2) et bien sûr de servir de base consensuelle à l'exécution d'applications réparties.

5.7 Performances

Cette section présente les mesures de performance des mécanismes de base de SCHOONER.

[4] Remarque : il y a bien évidemment de la concurrence entre clusters, constituant un potentiel de parallélisme pour l'application.

[5] outre une éventuelle fonction exécutée lors de l'initialisation de l'objet par le constructeur de la classe

Configurations logicielles et matérielles utilisées. Les tests ont été effectués sur :

- des machines PC Pentium 2/200 Mhz avec 128 mega-octets de RAM et le système Linux 2.0.29. Ces PC sont reliés grâce à un ether-switch (16 ports), en 100 Mbits/s, full duplex. Le réseau est un réseau dédié ce qui permet d'éviter les facteurs de perturbation.
- une machine Sun Ultra-5, sous Solaris 2.6 avec 192 mega-octets de RAM.

Création de clusters. Le tableau 5.1 montre le temps de création synchrone d'un cluster SCHOONER— le cluster demandant la création attend que le cluster créé lui envoie une confirmation de création avant de continuer son calcul — par rapport au temps de création synchrone d'un processus PVM. Le tableau 5.2 donne le temps de création asynchrone d'un cluster (pas d'attente de confirmation).

Le nombre d'expériences réalisé est de 100 pour chaque cas présenté. La valeur donnée est la valeur minimum de l'ensemble des expériences.

Architecture	PVM synchrone	SCHOONER synchrone	Coût SCHOONER
PC Pentium2/Linux	20891 μs	1.261 s	≈ 60
Ultra-5 Solaris	67416 μs	1.003 s	≈ 15

TABLEAU 5.1 – *Temps de création synchrone d'un cluster.*

Architecture	PVM asynchrone	SCHOONER asynchrone	Coût SCHOONER
PC Pentium2/Linux	6633 μs	8051 μs	≈ 1.2
Ultra-5 Solaris	8475 μs	10016 μs	≈ 1.2

TABLEAU 5.2 – *Temps de création asynchrone d'un cluster.*

Même si le temps SCHOONER est largement supérieur à celui de PVM, ceci n'est pas forcément pénalisant ; en effet, la plupart des applications créent les objets `Cluster` au démarrage et ce temps de création — même s'il est important — peut s'avérer être négligeable par rapport au temps total de l'exécution de l'application.

De plus, dans le cas de création asynchrone, plusieurs clusters peuvent être démarrés avant d'attendre la réception de confirmation de création pour l'ensemble d'entre eux.

Interaction entre clusters. Les figures 5.5 et 5.6 montrent le temps d'envoi d'un message entre deux clusters en fonction de la taille de celui-ci.

L'expérience est basée sur un programme effectuant un ping-pong entre deux clusters ou processus. Afin de limiter les erreurs dues à l'intrusion des instructions effectuant la mesure du temps, le message est échangé 1000 fois entre les 2 entités. Pour une taille et une architecture

données, le programme est exécuté 20 fois aussi bien pour PVM que pour SCHOONER, la figure donne la valeur minimum obtenue sur l'ensemble des exécutions.

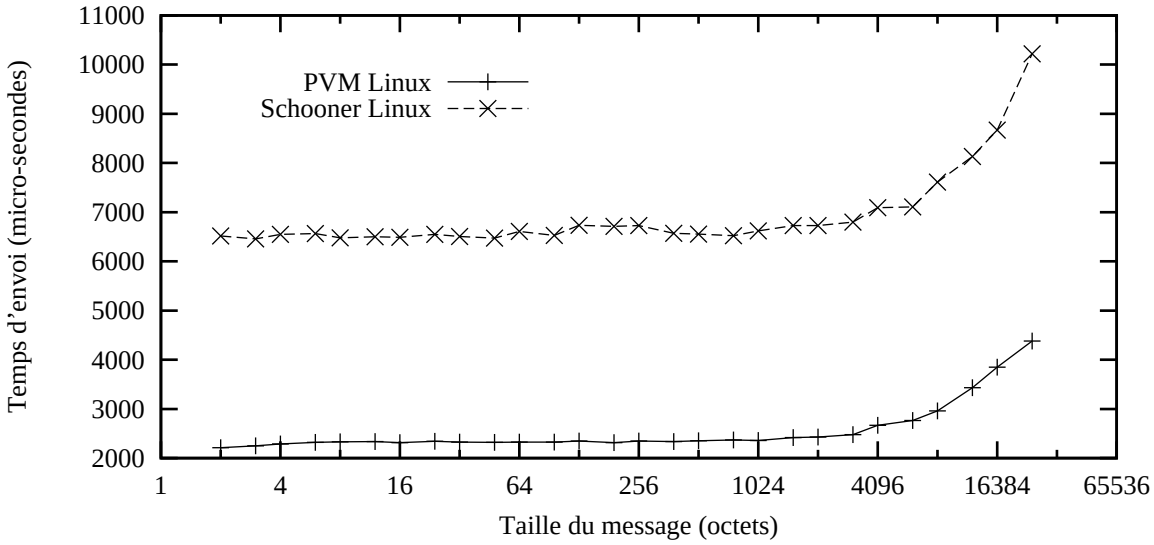

FIGURE 5.5 – *Comparaison du temps d'envoi d'un message* PVM - SCHOONER *sur une machine Linux.*

Nous pouvons voir que sur les machines Solaris, SCHOONER est environ 1.5 fois plus coûteux que PVM. Sur les machines Linux, la différence de coût est d'environ 2.5. Le surcoût de SCHOONER est donc relativement faible et est dû principalement à l'appel de fonctions virtuelles et à des allocations mémoire supplémentaires.

Les objets communicants. Les fonctions sur les objets communicants (création, envoi de données, destruction) se ramènent toutes à l'échange de messages actifs entre clusters. Nous ne présentons donc pas de mesures de performance spécifiques à la gestion des objets communicants. Toutefois, la section 6.3.1 donne des mesures de performance dans le cadre de l'utilisation d'objets communicants multi-actifs.

5.8 Conclusion

La figure 5.7 présente une vue schématique d'une application en cours d'exécution construite au dessus de la bibliothèque SCHOONER. On peut y voir toutes les entités de base de la bibliothèque.

Rappelons les points essentiels de cette bibliothèque : en plus et grâce aux avantages dus à la conception orientée objet, SCHOONER permet une manipulation aisée de la machine virtuelle

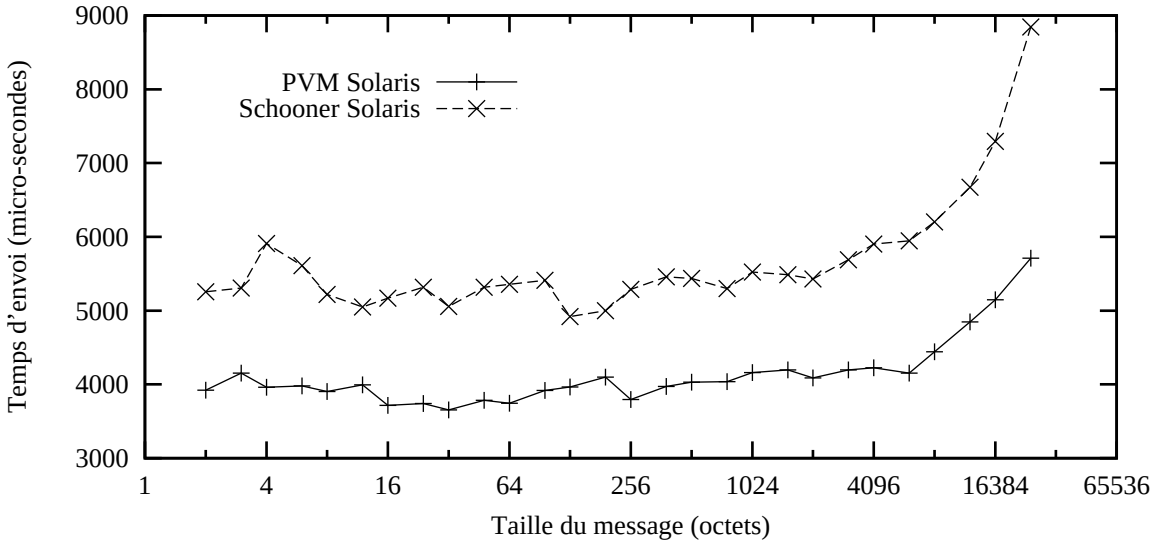

FIGURE 5.6 – *Comparaison du temps d'envoi d'un message* PVM - SCHOONER *sur une machine Solaris.*

et des entités de base de toute application répartie — les clusters et les objets communicants —, entités permettant deux niveaux de granularité différente et communiquant via un mécanisme de messages actifs.

Ceci constitue une base à tout l'environnement SCHOONER qui va être présenté dans les chapitres suivants et permet de répondre aux différents besoins évoqués dans la partie I de ce mémoire. La partie III en décrivant des applications construites au dessus de SCHOONER permettra de justifier nos choix.

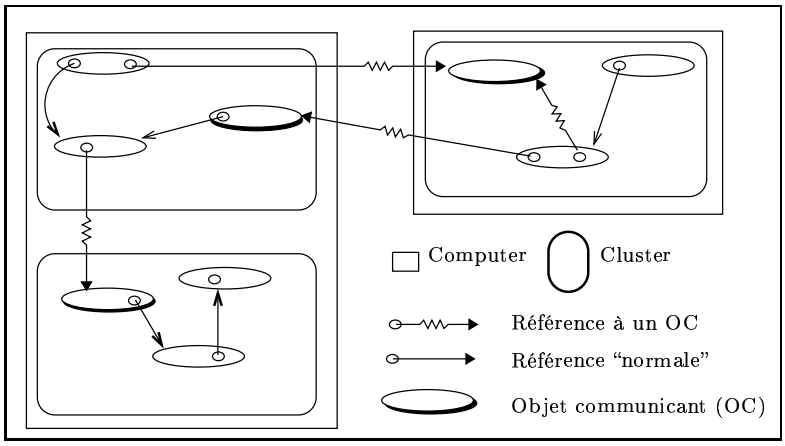

FIGURE 5.7 – *Une application construite au dessus de* SCHOONER.

Chapitre 6

La multi-activité

✔ *Ce chapitre présente l'ajout de la multi-activité dans la bibliothèque* SCHOONER. *Nous entendons ici par multi-activité la possibilité pour chaque objet communicant au sein d'un cluster d'avoir sa propre activité et donc de ne plus être un objet passif mais au contraire un objet actif.*

Nous exposons tout d'abord les justifications de cette extension et les implications éventuelles pour l'utilisateur SCHOONER, puis une mise en œuvre par interfaçage de la bibliothèque de processus légers répartis PM^2.

6.1 Principes de l'extension de SCHOONER par de la multi-activité et implications sur le modèle de programmation

La justification de l'introduction de la multi-activité en SCHOONER est d'offrir réellement un grain de parallélisme plus fin, par l'introduction d'entité d'exécution de base (un objet communicant actif) qui soit de granularité plus fine que ne l'est déjà un objet communicant de base.

De plus, à partir de la bibliothèque SCHOONER étendue à la multi-activité, il va être aisé de fournir à l'utilisateur final des paradigmes de programmation répartie autres que celui du modèle de base offert par SCHOONER, basé exclusivement sur des messages actifs (que l'on peut également voir comme des requêtes de services distants).

Grâce au fait que le comportement d'objets communicants peut être redéfini, l'introduction

de la multi-activité du point de vue de l'utilisateur final, consiste tout simplement en l'adjonction d'une méthode `Live()` dont l'exécution sera prise en charge par l'activité principale de l'objet. Cette activité principale se matérialise par un processus léger, lequel sera démarré lors de la création de l'objet et dont l'ordonnancement se devra de rester transparent. Pour ce faire, l'ordonnancement entre processus légers au sein d'un même cluster doit être pris en charge par la bibliothèque de processus légers interfacée ou le cas échéant mis en œuvre par nos soins dans la bibliothèque Schooner.

La fonction de réception d'un message (méthode `Comm_Object::receive()`, voir Code 5.7, page 62) quant à elle est toujours déclenchée de façon automatique — principe de message actif. Cette dernière consistait dans le cas d'objets communicants passifs au minimum en l'invocation de la fonction de traitement associée au type des données encapsulées dans le message (méthode `Data_Comm_Object::process()`, voir Code 5.8, page 62).

Libre au programmeur de modifier ce comportement :
- En par exemple se bornant à stocker le message reçu dans une file de messages, la routine `Live()` pouvant ultérieurement se charger de l'extraction d'un message et de l'invocation de la fonction de traitement associée au type des données qui y sont encapsulées.
- A titre d'exemple, nous montrerons dans la section 9.2.2.2, la programmation de diverses fonctions de réception de message aptes à mettre en œuvre différents modes de communication entre objets actifs du modèle C++//.
- Nous pouvons aussi obtenir un modèle "à la CSP" (Communicating Sequential Processes, [Hoare 78]) en introduisant la notion de garde fondée si besoin sur des prédicats testant la présence dans la file (mimant alors le réseau) d'un message provenant d'un canal précis.
- Nous pourrions aussi obtenir du parallélisme au sein même d'un objet communicant actif en démarrant une nouvelle activité pour traiter les données encapsulées dans chaque nouveau message reçu.

Comme on le voit, l'utilisateur a en fait la possibilité de reproduire le modèle qu'il désire.

Implications sur le modèle de programmation. Quelle que soit la manière dont sera introduite cette multi-activité dans Schooner, le mécanisme de messages actifs restant présent, il va pouvoir y avoir concurrence entre la réception de nouveaux messages et l'exécution des activités principales.

Du point de vue de l'utilisateur, seule la concurrence entre l'activité d'un objet et les opérations de réception de messages lui étant destinés *devrait* être à considérer. En effet, ces opérations peuvent impliquer des données — utilisateur — communes. Si besoin, charge donc au programmeur d'utiliser de façon adéquate des primitives de synchronisation.

Toute autre forme de synchronisation dont le besoin concerne plusieurs objets communicants actifs *devrait* être résolue dans Schooner ou par le biais de primitives réentrantes dans

les bibliothèques langages utilisées. Mais si par exemple, les activités de deux objets communicants, ou l'activité principale d'un objet et la fonction de réception d'un message, utilisent le descripteur de sortie standard, il risque d'être nécessaire — dans le cas où la bibliothèque langage n'est pas réentrante — de faire de cette utilisation une section de code critique.

6.2 Architecture logicielle et mise en œuvre par interfaçage de PM^2

Quelle que soit la bibliothèque de processus légers interfacée, il est nécessaire de prévoir une encapsulation objet de la notion d'activité (thread en anglais) et de primitives de synchronisation. Nous utilisons pour ce dernier point l'outil sémaphore car il permet de résoudre *tous* les problèmes de synchronisation.

La version étendue de SCHOONER pour la multi-activité fait usage des — nouvelles — classes suivantes : la classe `Semaphore` (voir Code 6.1) et la classe `Thread` (voir Code 6.2).

Code 6.1 *Interface (partielle) de la classe* `Semaphore`.

```
class Semaphore {
public:
  void P(int n=1);                                  // prendre n jetons
  void V(int n=1);                                  // rendre n jetons
  int value();                                      // retourne le nombre de jetons du sémaphore
  Semaphore(int init);                              // constructeur (nombre de jetons)
  ~Semaphore();                                     // destructeur
};
```

Une classe `Method` est nécessaire afin de pouvoir transmettre lors de la création d'un processus léger l'adresse de la fonction qu'il doit exécuter.

L'usage général et la norme POSIX se tournent vers des politiques d'ordonnancement préemptives des processus légers basées sur des priorités et éventuellement à partage de temps[1] c'est-à-dire dont l'ordonnancement se fait de façon transparente pour le programmeur. Ainsi ce dernier ne doit faire aucune hypothèse sur l'ordonnancement de ces processus légers. On conseille donc que toute section de code faisant usage de ressources critiques partagées entre les processus légers d'un même cluster soit protégée en parenthésant cette région par les fonctions `Thread::begin_critical_section()` et `Thread::end_critical_section()`. La création d'un objet communicant est un exemple de tel code : un identifiant pour le nouvel objet est fabriqué et doit l'être en exclusion mutuelle vis-à-vis d'autres fabrications d'identifiants.

[1] classes d'ordonnancement `SCHED_FIFO` et `SCHED_RR` de POSIX

Code 6.2 *Interface (partielle) de la classe* `Thread`.

```
class Thread {
protected:
  Handler_thread *th;
#ifdef NO_MT_SAFE                                   // utilisé pour le code non préemptible
  static Semaphore *sem_sched;                                          // sémaphore interne
  static int locked;                                                  // nombre d'entités bloquées
#endif NO_MT_SAFE
public:
  void wait_end();                                  // attend la fin de l'exécution du processus léger
  Thread(Comm_Object_Active *obj, Method &m);                                  // constructeur
  virtual ~ Thread()                                                            // destructeur
  static void begin_critical_section();                          // indique le début d'une région critique
  static void end_critical_section();                             // indique la fin d'une région critique
  static void set_time_slice(unsigned long tslice);
                                        // permet de fixer la tranche de temps alloué à chaque thread
  static void get_time_slice();                 // retourne la tranche de temps allouée à chaque thread
  static void delay(unsigned long millisecs);                  // stoppe la thread pendant un temps donné
};
```

Une implémentation portable de ces fonctions revient à coder l'entrée et la sortie d'un moniteur, le code critique s'exécutant alors à l'intérieur du moniteur. Les fonctions d'entrée et de sortie du moniteur peuvent se réaliser par les algorithmes décrits dans le code 6.3. Le point délicat est d'éviter l'autoblocage d'un processus léger qui tenterait d'entrer dans le moniteur alors qu'il s'y trouve déjà, on utilise pour cela le principe des sémaphores récursifs. De tels algorithmes sont portables quelle que soit la bibliothèque de processus légers interfacée, puisqu'ils ne font appel qu'aux primitives standard `P` et `V` sur un sémaphore initialisé à 1.

Les primitives `begin_critical_section()` et `end_critical_section()` peuvent être codées de manière plus spécifique en bloquant l'ordonnancement des processus légers. Ainsi le processus léger en cours d'exécution est assuré d'être seul tant qu'il ne réactive pas l'ordonnanceur. Dans le cas de la bibliothèque Marcel, cette façon de procéder peut être mis en œuvre en appelant la primitive non POSIX `pthread_set_timeslice_np(0)` qui a pour effet d'empêcher les préemptions.

Dans le cas où la bibliothèque de processus légers interfacée ne serait pas préemptive, une classe `Scheduler` permettrait de décharger le programmeur final de la gestion de l'ordonnancement des processus légers et par là même de l'ordonnancement des objets communicants actifs. En effet, il est nécessaire que le programmeur puisse faire l'hypothèse qu'un objet communicant actif ne peut pas avoir la main indéfiniment. Si tel était le cas, le programmeur devrait manuellement prévoir que l'activité d'un objet communicant rende le contrôle à

Code 6.3 *Fonctions d'entrée et de sortie du moniteur permettant de délimiter des sections de code critique.*

```
void Monitor::acquire() {
  sem→P()
  if (le processus léger appelant est le propriétaire du sémaphore récursif) {
    incrémenter le niveau de récursivité
    sem→V()                                          // sémaphore interne
  }
  else {
    sem→V()
    recursif_lock→P()
    initialiser le niveau de récursivité à 1
    indiquer que le processus léger appelant est le propriétaire du sémaphore récursif
  } // endif
}
void Monitor::release() {
  sem→P()
  vérifier que le processus léger appelant est le propriétaire du sémaphore récursif
  décrementer le niveau de récursivité
  if (le niveau de récursivité vaut 0) {
    recursif_lock→V()
  } // endif
  sem→V()
}
```

l'ordonnanceur.

Cette classe permettrait de mettre en œuvre — de façon certainement peu efficace — un mini noyau de gestion de processus légers comprenant par exemple une stratégie de réquisition basée sur le décompte du temps d'utilisation du CPU. Il est clair que cela signifierait de reprogrammer la commutation entre processus légers et reviendrait à recoder le noyau de la bibliothèque de processus légers. Nous ne pouvons donc que conseiller d'interfacer une bibliothèque de processus légers dont l'ordonnancement se fasse déjà de façon automatique grâce à une stratégie avec réquisition, telle celle à temps partagé avec priorités mise en œuvre dans Marcel.

Mise en œuvre à l'aide de PM^2. Toutes les spécifications de classes peuvent s'implémenter en interfaçant directement les fonctions adéquates de la bibliothèque de processus légers Marcel utilisée par PM^2.

PM^2 qui est une bibliothèque de processus légers *répartis* fournit en plus de Marcel des opérations de communication point-à-point entre modules (que l'on peut comparer à des clusters) et des opérations de gestion de configuration (que l'on peut comparer à certaines de celles disponibles sur les classes `Computer` et `Cluster`).

PM^2— dans sa dernière version secondée par Madeleine [Bougé 98, Namyst 98b] quant aux aspects liés à la répartition — résout ainsi les éventuels problèmes de conflit posés par l'utilisation depuis plusieurs processus légers Marcel de code non réentrant (en particulier des primitives de communication de bas-niveau). Il est donc particulièrement intéressant d'implémenter les classes Schooner `Computer`, `Cluster` et celles réalisant la gestion des communications et de la configuration en interfaçant PM^2. Pour plus de détails, se référer à [Baude 96c, Section 6].

Un autre avantage — non négligeable — à utiliser PM^2 est que les clusters Schooner sont totalement déchargés de la tâche de réception de messages depuis la couche de communication, puisqu'elle est réalisée par un processus léger Marcel spécifique au sein de PM^2. Ainsi, du point de vue de l'utilisateur final, la notion de message actif se rapproche plus de l'idée que l'on s'en fait, à savoir arrivée de message et traitement associé démarrés de façon totalement automatique (à l'inverse de ce que l'on est obligé de faire lorsque l'on utilise le modèle Schooner de base, voir exemples dans le chapitre 8).

Ce travail a été en partie réalisé lors du projet d'ingénieurs de P. Barette et de C. Tonin [Barette 96].

Cette implémentation au dessus de la bibliothèque Marcel a également nécessité de réécrire la classe `System` afin d'utiliser la version de Pvm fournie avec PM^2. En effet, les premières versions de PM^2 nécessitaient leur propre version de Pvm, ce n'est plus le cas avec la nouvelle version au dessus de Madeleine.

6.3 Définition d'une classe d'objets communicants actifs

Le petit exemple décrit ci-après (voir codes 6.4, 6.5 et 6.6) consiste en une classe d'objets communicants actifs dont le comportement — jouet — est de dormir un certain temps et si au réveil un message est disponible, le traiter, sinon se rendormir. Ce comportement sera répété exactement 3 fois, après quoi l'objet se termine.

6.3.1 Exemple de programme et performances

Du point de vue de l'utilisateur final, un programme utilisant les objets communicants multi-actifs est *très peu différent* d'un programme utilisant les objets communicants de base fournis par Schooner. En effet, bien que des processus légers aient été introduits dans le

Code 6.4 *Interface (partielle) d'une classe d'objets communicants actifs.*

```
class Comm_Object_Active_Sample : public Comm_Object {
protected:
  Thread *activity ;
  bool is_finished ;
  deque<Data_Comm_Object *> pile_dco ;
  Semaphore *sem_pile_dco ;
public:
  virtual void live() ;                                          // routine Live
  virtual void receive(Data_Comm_Object *dco) ;                  // point d'entrée
  void add_dco(Data_Comm_Object *dco) ;          // gestion d'une pile de données actives
  Data_Comm_Object *get_dco() ;                  // gestion d'une pile de données actives
  bool dco_present() ;                           // gestion d'une pile de données actives
  Comm_Object_Active() ;                                         // constructeur
  virtual void end_object() ;                                    // terminaison de l'objet
};
```

modèle, ils ne sont pas visibles globalement par l'application. L'utilisateur ne crée pas des processus légers répartis mais des objets communicants — d'une classe spécifique — dont le code sera exécuté par un processus léger.

En supposant que les bibliothèques utilisées soient réentrantes, le code ne devrait donc pas être changé que les processus légers soient utilisés ou non. La seule modification est à la création d'un nouvel objet communicant : il faudra préciser si l'on veut créer un objet communicant de la classe de base ou d'une classe dérivée (voir Code 6.7).

La figure 6.1 présente le temps d'appel d'une fonction distante permettant de calculer la factorielle de n (temps exprimé en μs) en fonction de n, donc du temps de calcul à exécuter sur le cluster distant. Le surcoût de SCHOONER par rapport à PM^2 est constant et est d'environ 24% (l'écart-type de cette valeur est de 0.89).

6.4 Conclusion

Nous avons vu dans ce chapitre comment de manière relativement simple la multi-activité a pu être introduite dans le modèle de base de SCHOONER.

Un prototype a été développé au dessus de la bibliothèque PM^2. Il faut noter que le modèle de programmation fourni par PM^2, un modèle basé sur les LRPC, n'est pas reproduit dans SCHOONER. Les LRPC sont utilisées pour implémenter les communications de SCHOONER mais ne sont pas utilisables par l'utilisateur SCHOONER.

Code 6.5 *Implémentation (partielle) de la classe* `Comm_Object_Active_Sample` *{partie 1}.*

```
Comm_Object_Active_Sample::Comm_Object_Active_Sample() : is_finished(FALSE) {
  sem_pile_dco = new Semaphore(1);
  number_of_dco = 0;
  activity = new Thread(this, live);     // création du processus léger associé à l'objet communicant
}
void Comm_Object_Active_Sample::live() {
  while (!is_finished) {
    Thread::begin_critical_section();                    // tant que l'on n'a pas reçu 3 messages
    cerr << "pas envie de traiter de message " << *this << endl;
    Thread::end_critical_section();
    Thread::delay(5);
    if (dco_present()) {
      Data_Comm_Object *dco = get_dco();
      dco→process();
      number_of_dco++;
      if (number_of_dco == 3) {
        end_object();
      } // endif
    } // endif
  } // endwhile
  Thread::begin_critical_section();
  cerr << "fin de l'objet" << *this << endl;
  Thread::end_critical_section();
  activity→wait_end();
}
```

Et nous pourrions tout à fait envisager de fournir une implémentation multi-active de SCHOONER au dessus de la bibliothèque NEXUS.

Code 6.6 *Implémentation (partielle) de la classe* `Comm_Object_Active_Sample` *{partie 2}.*

```
void Comm_Object_Active_Sample::end_object() {
  is_finished = TRUE;
}
void Comm_Object_Active_Sample::receive(Data_Comm_Object *d) {
  add_dco(d);
  Thread::begin_critical_section();
  cerr << "un message de plus reçu pour l'objet" << *this << endl;
  Thread::end_critical_section();
}
void Comm_Object_Active_Sample::add_dco(Data_Comm_Object *d) {
  sem_pile_dco→P();
  pile_dco.enq(dco);
  sem_pile_dco→V();
}
Data_Comm_Object *Comm_Object_Active_Sample::get_dco() {
  sem_pile_dco→P();
  Data_Comm_Object *dco = pile_dco.deq();
  sem_pile_dco→V();
  return dco;
}
bool Comm_Object_Active_Sample::dco_present() {
  return !pile_dco.is_empty();
}
```

Code 6.7 *Création d'objets communicants de classes différentes.*

```
  Proxy_Comm_Object *p1, *p2;
  p1 = new Proxy_Comm_Object(cl, "Comm_Object");
          // création d'un objet communicant de la classe Comm_Object (comportement par défaut)
  p2 = new Proxy_Comm_Object(cl, "Comm_Object_Active");
                          // création d'un objet communicant de la classe Comm_Object_Active
```

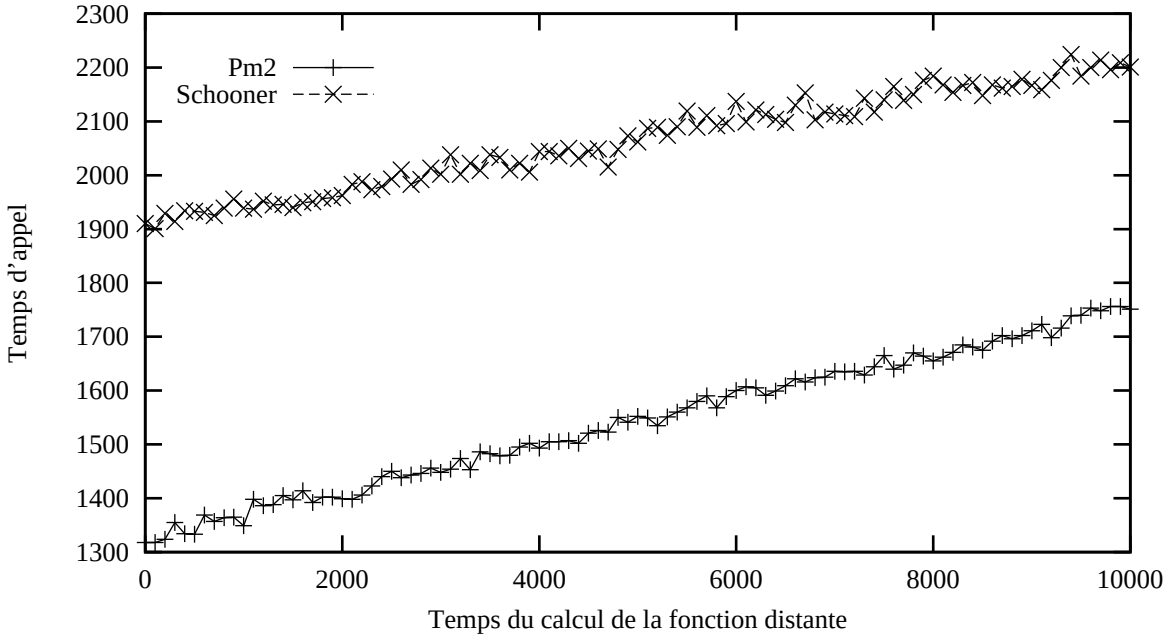

FIGURE 6.1 – *Différence* SCHOONER- PM^2 *: Temps d'appel d'une fonction distante synchrone avec résultat (μs) en fonction du temps de calcul de la fonction (valeur de n).*

Chapitre 7

Outils et bibliothèques développés

✔ *On présente dans ce chapitre les outils et bibliothèques développés en complément de* SCHOONER. *Ce sont principalement des outils d'aide à la mise au point d'applications réparties et des redéfinitions du comportement de la bibliothèque de base.*

7.1 Aide pour la mise au point de programmes SCHOONER

7.1.1 Représentation graphique

Afin de pouvoir suivre[1] l'exécution d'une application, nous avons développé un moniteur graphique permettant de récupérer diverses informations d'une application SCHOONER en cours d'exécution et de les afficher. Cet outil est codé très "facilement" à l'aide de messages actifs.

Le moniteur gère la liste des clusters de l'application que l'on désire interroger, ainsi qu'un ensemble d'actions pouvant être effectuées sur ces clusters. La sélection d'un cluster et d'une action déclenche l'envoi à ce cluster d'un message actif correspondant à l'action sélectionnée ; le moniteur se bloque alors en attente de la réponse du cluster et l'affichera une fois reçue. Il est par exemple possible d'avoir une action permettant de connaître le nombre d'objets communicants du cluster ou bien des informations sur son état courant.

[1] en anglais : monitoring

La figure 7.1 montre un exemple de fenêtre principale d'un moniteur avec la liste des clusters de l'application que l'on désire inspecter et la liste des questions (actions). La deuxième fenêtre contient la réponse du cluster sélectionné à la question qui lui avait été posée par le biais de la sélection d'une des 2 actions disponibles[2].

FIGURE 7.1 – *Exemple de moniteur graphique.*

Chaque action du moniteur est représentée par une classe spécifique de message actif. L'application à surveiller a donc juste besoin de pouvoir exécuter[3] la fonction de traitement de ces messages et son code n'a pas besoin d'être modifié. Les messages du moniteur seront reçus et traités de manière transparente parmi les autre messages internes de l'application.

Le mécanisme des messages actifs permet ainsi de manière élégante à deux applications SCHOONER distinctes de s'échanger des messages, sans perturber le code existant de l'application.

7.1.2 Mécanisme de traces

Afin de pouvoir faire de la mise au point d'applications SCHOONER, on fournit également une bibliothèque de génération de traces qui est une extension de la bibliothèque `Trace-Tool` [Gioanni 96], ainsi qu'un outil permettant la visualisation postmortem graphique des fichiers de traces produits durant l'exécution.

[2] Le bouton "Inspector exit" permet de quitter le moniteur.
[3] c'est-à-dire pouvoir trouver le code associé

Génération des traces. Lors d'une exécution répartie, chaque *cluster* va générer un fichier de traces relatives à chacun des événements dont il est le siège (communication, création de clusters, création d'objets communicants, ...). En fin d'exécution, un programme annexe est chargé de fusionner ces différents fichiers en corrigeant le déphasage entre les horloges.

Pour cela, chaque cluster reçoit en début d'exécution un message lui fournissant son décalage d'horloge avec le cluster principal de l'application. Ceci va permettre d'estampiller les différents événements de l'application selon une horloge globale.

Nous avons choisi d'avoir une visualisation des traces postmortem. Cette solution est moins intrusive que celle utilisant une visualisation "à la volée"[4]. Toutefois, cette autre solution serait également réalisable. Si l'on ne dispose pas d'un disque partagé entre les différents clusters de l'application, il sera nécessaire de rajouter un mécanisme de messages internes pour centraliser toutes les traces sur le cluster maître ou sur un autre cluster externe à l'application. De plus, avec cette solution, le décalage d'horloge doit être appliqué avant l'écriture sur le disque[5], ce qui entraîne donc un surcoût supplémentaire.

Synchronisation des horloges. L'algorithme utilisé pour calculer le décalage entre deux *clusters* est assez rudimentaire. Si un cluster C_1 sur un computer A désire connaître son décalage d'horloge avec un cluster C_2 sur un computer B, il envoie un message M à C_2 qui va répondre en lui envoyant son heure locale t_r.

Si t_1 est l'heure avant l'envoi du message sur le cluster C_1 et t_2 l'heure après la réception de la réponse de C_2 sur C_1, alors le décalage est égal à : $\frac{t_1+t_2}{2} - t_r$.

Ce calcul pose l'hypothèse que la durée de transmission du message M est strictement identique à celle de la réponse. Dans ce cas là, le temps $\frac{t_1+t_2}{2}$ sur le computer A et le temps t_r sur le computer B représentent la même durée de temps, la différence entre ces deux durées est donc bien le décalage d'horloge entre les deux machines.

Ce mécanisme bien que rudimentaire permet d'obtenir une estimation de décalage d'horloges, d'autant plus facilement que mis en œuvre par appel de la fonction `pvm_hostsync(...)` (qui implémente exactement l'algorithme présenté), mais pourrait être aisément redéfini.

Format de traces. Le format de traces utilisé est le format SDDF (Self-Defining Data Format) [Aydt 96]. Ce format de traces est très largement utilisé. Un autre format très utilisé est le format PICL [Worley 92] couplé avec l'outil de visualisation ParaGraph [Heath 91]. Un état de l'art sur les outils de visualisation pour les systèmes parallèles pourra être trouvé dans [Kraemer 93] ou dans [Vigouroux 96].

[4] effectuée au fur et à mesure du déroulement de l'application
[5] et non en phase postmortem, comme c'est fait actuellement

SDDF qui est un méta-format de traces présente l'avantage d'être portable, facilement extensible et d'avoir une représentation texte lisible[6].

La bibliothèque `Trace-Tool` utilisée pour la gestion de traces étant assez flexible, nous pourrions facilement modifier le format de traces généré (voir [Gioanni 96], [Siegel 97, Section 13.2.1]).

Visualisation. Une interface graphique permet de traduire et de visualiser les événements au format texte (générés durant l'exécution de l'application) en événements graphiques dynamiques (s'affichant en fonction de l'instant de leur création par rapport au début du programme et de leur durée). Cette interface a été développée par A. CLAVAUD [Clavaud 97].

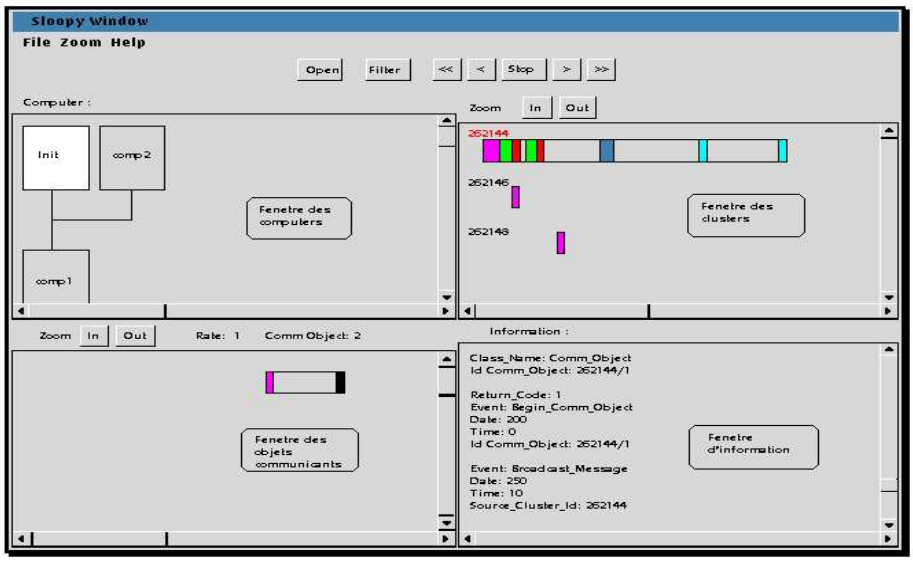

FIGURE 7.2 – *Fenêtre principale de l'interface graphique.*

Chaque événement de l'application SCHOONER correspond à deux champs différents du fichier traces : le premier indique le début de l'événement et le deuxième la fin. L'interface sera ainsi capable de représenter les différents événements relatifs à un cluster ou à un objet communicant dans un histogramme horizontal, chaque événement étant une sous-partie de cet histogramme dont la taille est fonction de la durée de l'événement[7] (voir Figure 7.2).

En plus des fonctionnalités habituelles d'un outil de visualisation, cette interface permet de filtrer les événements affichés selon une combinaison de différents critères. Les critères peuvent être par exemple un identifiant de trace, un identifiant de cluster ou d'objet communicant. Il est ainsi possible pour une même exécution de choisir de visualiser uniquement les événements

[6] Le fichier de traces peut être lu et compris sans l'aide d'un outil externe et contient la définition de ses différents événements.

[7] Chaque type d'événement différent est représenté par une couleur différente.

relatifs à l'interaction entre objets communicants ou bien uniquement les événements produits sur un computer spécifique. Le filtrage des événements pourrait également être fait en cours d'exécution, mais cette solution nécessite des modifications de code de la part de l'utilisateur (qui doit spécifier un filtre de traces dans le code même de son application) et de plus, il faudra re-éxecuter l'application pour chaque nouveau filtre.

L'exemple de filtre présenté dans la figure 7.3 permet de sélectionner uniquement les événements relatifs au computer de nom symbolique `comp2` et au cluster d'identifiant `262144` du computer de nom `Init`.

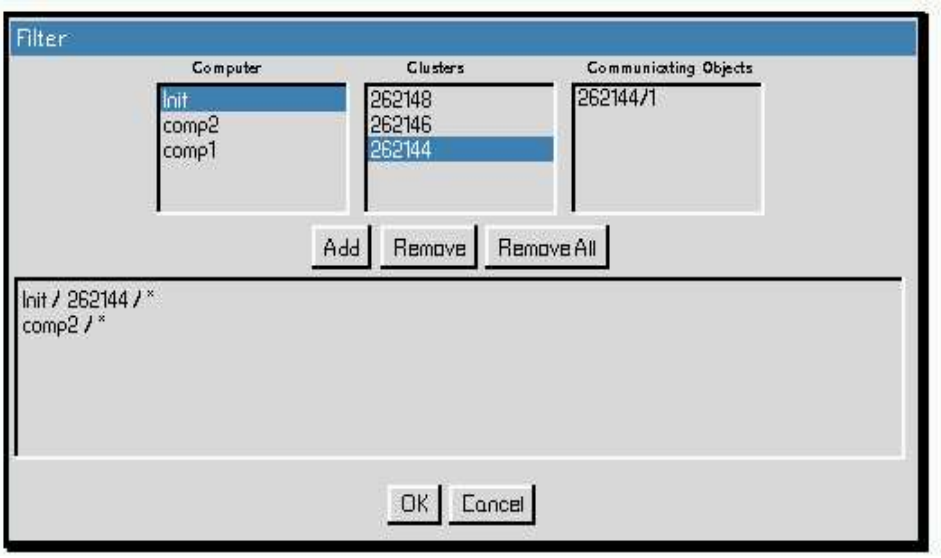

FIGURE 7.3 – *Filtre de l'interface graphique.*

Perturbations. La prise de traces dans un programme entraîne forcément des perturbations dans l'exécution de ce dernier. Dans le cas d'un programme réparti, cela peut même aller jusqu'à modifier le comportement global de l'application en provoquant par exemple des interblocages. Toutefois, nous ne nous sommes pas intéressés à ce genre de problèmes dans cette thèse, le lecteur intéressé pourra consulter [Vigouroux 96].

Nous allons présenter ici les perturbations en terme de temps d'exécution. Chaque événement SCHOONER tracé va générer deux appels de fonction, un marquant le début de l'événement et un deuxième pour la fin de l'événement et l'écriture des informations dans le fichier de traces. Ceci va générer l'appel d'un minimum de 5 méthodes C++, l'appel des méthodes récupérant les informations sur les objets tracés (par exemple, nom du computer ou identifiant du cluster), la gestion d'une table de hash et l'écriture dans le fichier de traces. Le surcoût entraîné est d'environ 5% sur le temps d'exécution total d'une application.

7.2 Redéfinition du comportement de base de SCHOONER

7.2.1 L'équilibrage de charge

Le travail présenté dans cette section provient d'une collaboration avec J-E. TANZY et O. DALLE [Tanzy 97]. Il porte sur l'intégration de mécanismes d'équilibrage de charge dans la bibliothèque SCHOONER.

Nous présentons d'abord les justifications de cette intégration, puis la mise en œuvre effectuée.

7.2.1.1 Présentation du problème et objectifs

Les applications développées au dessus de l'environnement PROSIT (voir Section 8.1, page 93) sont des applications de taille importante — aussi bien en durée, en mémoire ou en communications —, le placement des différents objets de l'application peut donc être un critère important de performances.

On s'intéressera principalement au placement des processus lourds. En effet, les simulations PROSIT créent généralement leurs clusters (processus lourds) lors du démarrage de la simulation : la qualité de leur placement initial est donc un facteur important de performance. Les objets actifs (processus légers) de l'application ayant la possibilité de migrer, le choix de leur placement initial est moins crucial. Mais il peut toutefois être important dans le cas d'environnements hétérogènes[8]. Nous ne nous intéressons donc pas ici au problème de placement des objets actifs, nous considérons en effet que la régulation de charge pour ces objets peut être fait en cours d'execution grace à des mécanismes de migration, tels ceux présents dans la bibliothèque PM^2.

Le problème va donc être de trouver un site d'exécution (quasi-)optimal pour lancer l'exécution d'un nouveau processus. Nous utilisons pour cela l'outil *DLB*, acronyme de *Dynamic Load Builder*, qui est une plate-forme de répartition de charge développée dans le projet SLOOP.

7.2.1.2 Présentation de l'outil *DLB*

On se place dans un environnement de stations de travail UNIX hétérogènes, multitâches et multi-utilisateurs. L'outil de répartition de charge peut :

– gérer simultanément l'exécution de plusieurs applications réparties ;

[8]La migration ne peut s'effectuer qu'entre deux machines du même type.

- prendre en compte les variations de charge dues à l'ensemble des applications s'exécutant sur chacune des machines ;
- prendre en compte l'hétérogénéité des machines.

L'architecture retenue est de type client/serveur : les applications réparties sont clientes d'un service indépendant d'aide à la répartition de charge mis en œuvre par *DLB*. Ce service est supposé s'exécuter en permanence sur chacune des machines. Son rôle est :

1. de collecter auprès de ses vis-à-vis les informations concernant l'état de charge des autres machines ;
2. réciproquement, de fournir ces informations lorsqu'elles lui sont demandées par une autre machine ;
3. de répondre aux requêtes et notifications des tâches qui s'exécutent localement sur sa machine ;

Les notifications pouvant être effectuées par les processus clients de *DLB* sont :
- Notification de prise en charge ;
- Notification d'état ;
- Notification de terminaison.

Une application s'étant enregistrée auprès du processus *DLB* tournant sur sa machine pourra lui soumettre les requêtes suivantes :
- Requête de placement, c'est-à-dire demander quelle est la "meilleure" machine de la configuration virtuelle considérée et ce selon différents critères (CPU, mémoire, entrées/sorties, communications réseau) en donnant des poids à chacun d'eux.
- Informations de charge sur un processeur ou un cluster.

Utilisation de *DLB* via SCHOONER. Afin de rendre accessible à l'utilisateur les fonctionnalités de l'outil *DLB*, la fonction SCHOONER de choix du meilleur *computer* de la machine virtuelle a été étendue. L'utilisateur pourra ainsi choisir de démarrer un nouveau cluster sur le computer ayant la charge CPU la moins élevée ou celui ayant le plus de mémoire disponible. Ce choix est fait en ajoutant à la méthode `Computer::best()` quatre paramètres correspondant aux poids à donner aux indicateurs de charge utilisés par l'outil *DLB* (CPU, mémoire, entrées/sorties, communications réseau).

Comme indiqué en introduction, cette interface a été testée avec succès pour le placement de clusters d'applications PROSIT.

7.2.2 La bufferisation

La plupart des applications ont souvent besoin d'échanger beaucoup de messages de petite taille. Afin de rentabiliser au maximum la capacité de la couche physique de communication, il peut être intéressant de regrouper ces messages en un seul message de plus grande taille.

Un mécanisme de bufferisation est proposé dans l'environnement SCHOONER, il permet de grouper l'envoi de plusieurs messages actifs vers un même cluster. Les paramètres de bufferisation sont :

1. le nombre de messages à bufferiser,
2. la taille maximum du buffer fixée par défaut à la place disponible pour les données dans un paquet système de la bibliothèque de communication interfacée (ex : les paquets de la version 3.3 de bibliothèque PVM font 4092 octets, 4064 étant utilisables par l'application).

Ces deux paramètres peuvent être modifiés par l'utilisateur.

Un canal de communication est associé à chacun des clusters de l'application. Le buffer courant entre une paire de clusters est envoyé lorsqu'une des deux contraintes suivantes est satisfaite :

1. le buffer contient un nombre maximum de messages,
2. la taille maximum du buffer est atteinte.

L'utilisateur a également la possibilité à tout moment de demander que soit vidé le canal de communication pour un cluster donné ou pour l'ensemble des clusters.

L'utilisateur pourra également au cours de l'application décider de désactiver le mécanisme de bufferisation et de le réactiver ultérieurement.

L'implémentation de ce mécanisme en SCHOONER s'est faite assez facilement en spécialisant l'objet servant de gestionnaire de communication (classe `Communication_Manager`). De nouvelles fonctionnalités pour la gestion de la bufferisation y ont été rajoutées :

- modifier/consulter le nombre de messages à bufferiser (primitives `set/get_nb_max_messages()`),
- modifier/consulter la taille maximum du buffer (primitives `set/get_buffer_size()`),
- indiquer si l'on veut ou non la bufferisation (primitives `set/get_bufferisation()`),
- vider un ou tous les canaux de communication (primitive `flush()`).

Pour utiliser le nouveau gestionnaire de communication, il suffit seulement de changer l'appel de la fonction d'initialisation du système SCHOONER en précisant que l'on veut utiliser ce nouveau gestionnaire (voir Section 5.6.1).

Performances. Les tests effectués permettent de mesurer le gain d'envoi en bufferisant plusieurs messages actifs. Le paramètre principal du programme est le nombre maximum de messages à bufferiser entre deux clusters. Nous mesurons le temps d'envoi de n messages, en faisant varier la taille du buffer. Les résultats sont présentés dans la figure 7.4. On peut voir que le temps d'envoi diminue quand la taille du buffer augmente : comme l'a montré [Baude 96b], le temps d'envoi d'un message d'une taille totale de T octets est en effet inférieur au temps

d'envoi de n messages d'une taille de T/n octets. L'envoi d'un message PVM ayant un coût fixe indépendant de sa taille, il est en effet préférable pour un même volume de données de les envoyer en une seule fois.

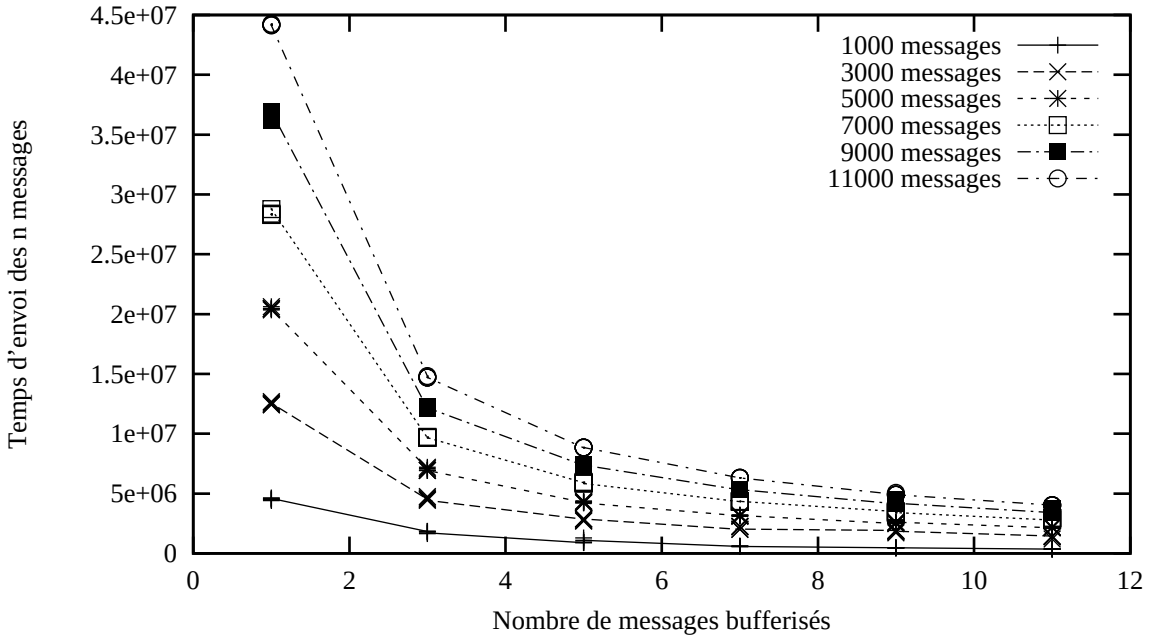

FIGURE 7.4 – *Temps d'envoi de n messages en faisant varier le nombre maximum de messages à bufferiser entre les deux clusters.*

7.3 Conclusion

Ce chapitre a présenté des outils que nous avons développé en complément des bibliothèques de base de SCHOONER afin de faciliter la programmation d'applications réparties, ou d'améliorer le comportement pour certaines classes d'application.

Nous allons maintenant voir dans la partie suivante comment l'environnement SCHOONER a été utilisé pour le développement d'applications grandeur réelle.

Troisième partie

Applications construites au dessus de SCHOONER

Chapitre 8

Exemples d'applications utilisant SCHOONER

✔ *Ce chapitre présente des applications construites au dessus de la bibliothèque* SCHOONER.

8.1 L'environnement de simulation PROSIT

PROSIT [Siegel 97] est un environnement pour la simulation à événements discrets développé au sein du projet SLOOP. Il existe un prototype de la version répartie de PROSIT, celui-ci étant construit au dessus de SCHOONER.

8.1.1 Le modèle séquentiel

Une simulation PROSIT correspond à l'exécution d'un ensemble d'objets actifs interagissant par le biais de demande de services. Un temps de simulation permet de synchroniser l'ensemble de ces objets : chaque objet possède sa propre horloge, c'est au système d'assurer le maintien de la cohérence entre les différents objets.

Les objets actifs de PROSIT gèrent un ensemble d'activités. Dans la version actuelle, ces activités sont gérées comme des coroutines, c'est-à-dire que les différentes activités de l'objet se déroulent de manière concurrente dans le temps simulé. Les interactions entre les objets actifs sont asynchrones.

Un objet actif est instance d'une classe qui hérite directement ou indirectement de la classe `Simulation_Obj`. Les objets actifs interagissent via un mécanisme de réification. A chaque demande de service correspond l'émission d'un objet requête. En cas de retour de demande bloquante, un objet réponse est émis. Cette réification est mise en œuvre via 4 points de réification : 2 au niveau de l'objet actif émetteur et 2 au niveau de l'objet actif récepteur (voir Figure 8.1).

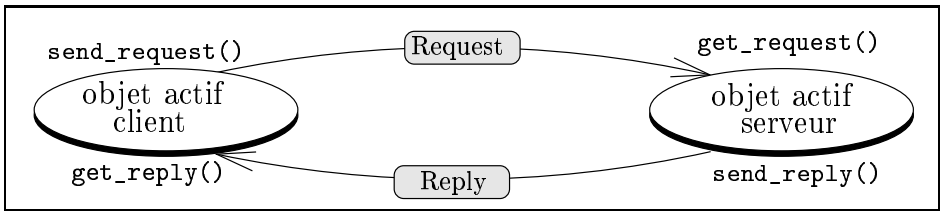

FIGURE 8.1 – *Mécanisme de réification de* PROSIT

L'ordonnancement de l'ensemble des objets actifs d'une simulation est géré par le noyau de simulation en fonction de la date de chaque objet.

8.1.2 Les besoins pour une version répartie

Un des objectifs principaux était de pouvoir passer d'une simulation séquentielle à une simulation répartie sans modification de la modélisation de la simulation et avec le minimum de modification de code. Les besoins de la version répartie sont :

1. vision structurée et abstraite du parc de machines,
2. mécanisme de placement des objets actifs et primitives de localisation,
3. migration des objets actifs,
4. synchronisation temporelle des objets actifs.

Nous allons maintenant reprendre l'ensemble de ces besoins et voir comment SCHOONER a permis de les résoudre.

Architecture distribuée. Les besoins de PROSIT à ce niveau sont les mêmes que ceux de SCHOONER. Les notions de *computer* et de *cluster* ont donc pu être directement utilisés par PROSIT. De ce fait, un cluster va constituer une entité de synchronisation locale.

Au niveau de PROSIT, le support de communication doit avoir les caractéristiques suivantes :

- réseau de communication complet,
- support de communication fiable,

– chaque message destiné à n'importe quel nœud de la simulation doit pouvoir être reçu par celui-ci (c'est-à-dire mécanisme de buffer infini).

Ces fonctionnalités sont présentes dans la bibliothèque Schooner, et sont donc directement exploitables en Prosit.

Les objets actifs. L'approche *proxy* a été utilisée pour la mise en œuvre du framework Prosit réparti : tout accès à un objet actif (local ou distant) se fera à travers un objet proxy. Le mécanisme séquentiel d'interaction est encapsulé dans ces objets *proxy* qui se chargent donc de recevoir et de transmettre les objets requêtes et réponses.

Les objets actifs Prosit obtiennent un comportement réparti en héritant de la classe Schooner `Comm_Object` et peuvent donc interagir à travers l'ensemble des nœuds de la simulation. Les proxys Prosit héritent également de cette classe pour être capable de recevoir des données et utilisent la classe Schooner `Proxy_Comm_Object` qui permet de créer un objet communicant Schooner (ou une classe dérivée) et de lui envoyer des données. La figure 8.2 montre le graphe d'héritage utilisé.

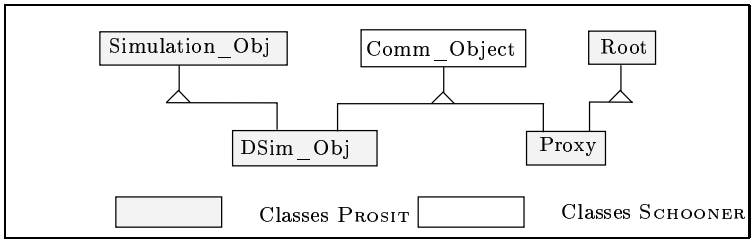

Figure 8.2 – *La hiérarchie de* Prosit *réparti pour les objets actifs et les proxys*

La localisation de nouveaux objets actifs est entièrement gérée via les méthodes fournies par la classe Schooner `Cluster`. De plus, Schooner garantissant à chaque nouvel objet communicant un identifiant global, il est aisé d'implémenter les primitives de localisation.

Concernant les 4 points de réification, il était important que leur utilisation ne diffère pas du modèle séquentiel. Pour cela, les proxys Prosit fournissent les méthodes `get_request()` et `get_reply()` qui seront appelés par les objets actifs de la même manière que dans une simulation séquentielle. C'est le rôle de l'objet proxy de transférer la requête si besoin est sur un cluster distant. Une fois la transmission effectuée, le point de réification sur l'objet actif concerné sera alors appelé afin de conserver la même sémantique de communication que dans une simulation séquentielle (voir Figure 8.3).

Les interactions entre les objets actifs se font en spécialisant la classe Schooner `Data_Comm_Object`. Chaque demande de service est associé à une classe dérivant de cette classe Schooner (et éventuellement une deuxième classe pour la réponse), ce qui permet donc d'échanger des objets requêtes et réponses entre objets actifs.

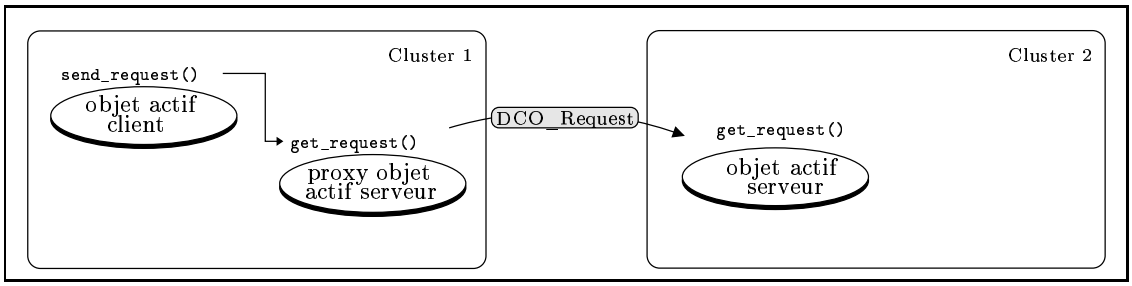

FIGURE 8.3 – *Envoi de requête dans la version répartie de* PROSIT

Migration des objets actifs. La version répartie de PROSIT a été développée au dessus de la version de base de SCHOONER (c'est-à-dire celle ne fournissant pas les processus légers). On aurait pu développer un prototype au dessus de la version avec processus légers, mais le framework PROSIT intègre déjà dans sa version séquentielle un mécanisme d'activités concurrentes.

De ce fait, les mécanismes de migration ont du être implémentés en PROSIT.

Synchronisation des objets actifs. Le mécanisme permettant d'assurer la synchronisation de l'ensemble des objets de la simulation est à deux niveaux : un premier niveau qui permet de synchroniser les objets actifs placés sur un même nœud, un deuxième niveau qui synchronisent les nœuds entre eux et donc l'ensemble des objets actifs.

Ainsi sur chaque cluster ou nœud de l'application devra être instancié un noyau chargé de l'ordonnancement des objets actifs locaux et de la synchronisation globale. Ce noyau est défini par dérivation du noyau fonctionnant en mode séquentiel. Il hérite également de la classe SCHOONER `Comm_Object` afin de pouvoir communiquer avec les autres noyaux de l'application.

8.1.3 Opérations collectives

L'environnement PROSIT ayant besoin dans sa version répartie de mécanismes de synchronisations globales, nous allons décrire dans cette section une bibliothèque permettant d'effectuer des opérations collectives sur des ensembles de clusters et entre autres un mécanisme de détection de propriétés stables.

On présente tout d'abord l'algorithme choisi pour la détection de telles propriétés puis l'implémentation faite en SCHOONER.

Bien que présenté dans le cadre de PROSIT, le mécanisme d'opérations collectives n'en est pas dépendant et pourrait être utilisé par n'importe quelle application désirant effectuer des opérations collectives sur l'ensemble de ses clusters. De plus, ce mécanisme pourrait être facilement adapté aux objets communicants.

8.1.3.1 Détection de propriétés stables

Dans [Mattern 87], Mattern donne la définition suivante :

> *A distributed computation is considered to be* globally terminated *if every process is (locally) terminated and no messages are in transit. "Locally terminated" can be understood to be a state in which a process has finished its computation and will no restart any action unless it receives a message.*

Cette définition peut être généralisée et résoudre ainsi des problèmes de détection de propriété stable tels que la détection de blocage ou la détection de terminaison. Une propriété stable est donc vérifiée sur l'ensemble des nœuds de calcul d'une application lorsque les deux conditions suivantes sont vérifiées :

1. il n'y a plus de messages en transit sur le réseau (tous les messages émis ont été effectivement reçus),
2. la propriété est vérifiée localement sur chacun des nœuds de l'application.

Pour savoir s'il n'y a plus de messages en transit, on utilise l'algorithme des 4 compteurs de Mattern [Mattern 93]. Pour cela, chaque cluster garde un compteur s_i pour le nombre de messages émis et un compteur r_i pour le nombre de messages reçus. Une première phase de l'algorithme récupère les compteurs de chaque cluster. Si S, la somme des s_i, est égale à R, la somme des r_i, la seconde phase de l'algorithme est démarrée et redemande à chaque cluster la valeur de ses deux compteurs. Soit S' la nouvelle somme des s_i et R' la nouvelle somme des r_i. Si les valeurs des quatre compteurs sont égales ($S = R = S' = R'$), cela signifie qu'il n'y a plus de messages en transit sur le réseau.

Une phase de récupération du résultat du test de la propriété sur chacun des clusters est alors démarrée. La propriété sera vérifiée globalement si et seulement si elle est vérifiée sur chacun des clusters. Afin de minimiser le nombre de messages échangés, lors de la deuxième phase de l'algorithme des 4 compteurs, le cluster va également envoyer le résultat du test de la propriété en plus de la valeur de ses 2 compteurs.

8.1.3.2 Implémentation en Schooner

La classe `Synchronization_Manager` — qui doit être utilisée par un cluster lorsque celui-ci veut déterminer un état global — permet ainsi de tester si une propriété est stable sur l'ensemble des *clusters* de l'application. Elle dispose pour cela de la méthode :

```
test_property(Message_Property *)
```

Le paramètre de la fonction est une classe de messages actifs (`Message_Property` héritant de `Message`) qui définit la propriété à tester localement sur chacun des *clusters* de l'application.

Cette fonction va donc d'abord appliquer l'algorithme des 4 compteurs de Mattern et si aucun message n'est en transit, elle pourra alors calculer la conjonction des tests de la propriété

locale sur chacun des clusters.

Pour définir une nouvelle propriété, il suffit de dériver la classe `Message_Property` et de redéfinir la fonction `local_property()`. Les figures 8.1 et 8.2 présentent un exemple d'utilisation. La fonction `local_property()` codée permet par exemple de tester que tous les clusters ont leur noyau à une date égale à 100.

Code 8.1 *Définition d'une nouvelle propriété.*

```
class Message_Sample : public Message_Property {
public:
bool local_property() {
  return kernel→get_time() == 100;
}
};
```

Code 8.2 *Test de propriété stable.*

```
  Synchronization_Manager *sync_manager = new Synchronization_Manager;
  Message_Sample *msg = new Message_Sample;
  if (sync_manager→test_property(msg)) {
    ...
  } // endif
```

Il serait possible de modifier la fonction fournie afin de la faire travailler non pas sur l'ensemble des clusters de l'application, mais sur un groupe de clusters. Un tel groupe serait défini par un identifiant accessible par l'ensemble des clusters qui pourraient à tout moment de l'application joindre ou quitter le groupe (comme c'est fait en PVM par exemple). Il serait alors nécessaire d'utiliser un autre algorithme que celui de Mattern pour vérifier qu'il n'y a plus de messages en transit sur le sous-réseau considéré.

8.1.4 Conclusion

Le développement de la version répartie de PROSIT nous a permis de mettre en évidence l'avantage lors de la conception d'application d'isoler les mécanismes de support d'exécution de l'application elle-même. Le passage d'une simulation séquentielle à une simulation répartie ne nécessite en effet pas de modification de la modélisation et se fait avec un minimum de modification de code.

Diverses extensions de SCHOONER telles que la détection de propriétés stables (voir Section 8.1.3), l'équilibrage de charge (voir Section 7.2.1) ou la bufferisation (voir Section 7.2.2) ont été expérimentées par le biais de PROSIT en tant qu'application finale.

8.2 Un client serveur hiérarchique

Nous allons voir dans cette section comment une application peut utiliser efficacement les objets communicants fournis par la bibliothèque SCHOONER. Par défaut, les objets communicants ne possèdent pas d'activité propre (voir Section 5.6.3, page 67) et ne peuvent donc pas exécuter de code, si ce n'est à la réception de données leur étant destinées. Mais bien que dans le modèle de base les seules entités actives soient les clusters, le niveau d'expression n'est cependant pas forcément celui d'un parallélisme à gros grain (comme c'est le cas en PVM, Landa ou MPI par exemple), du fait même de la possibilité de l'existence d'objets communicants au sein des clusters.

Nous allons maintenant voir dans cette section comment l'utilisation des objets communicants peut permettre de construire aisément une application possédant une structure hiérarchique.

Une application client-serveur peut être modélisée en SCHOONER de deux manières. Les serveurs peuvent être représentés soit 1°) par des clusters, soit 2°) par des objets communicants. Nous nous plaçons dans le cas d'une application nécessitant un nombre important de serveurs aussi bien au niveau du nombre de services disponibles que du nombre de serveurs effectifs.

Représenter les serveurs par des clusters. Même si le nombre de processus système est limité, il y a a la possibilité de regrouper plusieurs clusters au sein d'un même processus système. Mais cette solution a l'inconvénient d'être peu modulable. Il est possible de spécialiser les clusters qui vont représenter les différents serveurs, via la fonction de traitement d'un message actif spécifique. Selon les paramètres du message reçu, tel ou tel type de serveur sera créé. Il semble en effet naturel de vouloir représenter les serveurs à l'aide d'une classe C++ et de bénéficier ainsi des mécanismes de la programmation orientée objet tels que l'héritage. Cette idée nous amène tout naturellement à la deuxième solution qui est de représenter les serveurs par des objets communicants.

Représenter les serveurs par des objets communicants. Cette solution permet de partager très aisément des données entre un ensemble de serveurs (objets communicants), ces données pouvant être gérées soit par le cluster qui héberge les serveurs soit par un objet communicant spécifique. Cette solution a également l'avantage de pouvoir supporter très simplement un nombre important de serveurs.

Voyons maintenant de quelle manière représenter les services offerts, en d'autres termes où doit être placé le code algorithmique du service. Il est possible de :

- *représenter un service par un type particulier de données.* Cette solution nécessite de spécialiser la classe `Data_Comm_Object` (ou une classe dérivée qui sert de base à l'ensemble des demandes de service). Le code algorithmique du service est alors codé dans la méthode `process()` de la nouvelle classe des données actives.
  Cette solution — qui est très proche du modèle SCHOONER de base — ne nécessite pas de redéfinir la classe définissant les objets communicants, mais a l'inconvénient de ne pas pouvoir conserver de données entre deux exécutions distinctes du même service (si ce n'est l'éventuelle réponse du service, côté client).
- *représenter un service par un type particulier d'objet communicant.* Cette solution permet de conserver des données entre deux exécutions de service. Les données envoyées au serveur (objet instance d'une classe dérivant de la classe `Comm_Object`) représenteront les paramètres de la demande de service.
  Il est alors possible de coder la partie algorithmique du service dans la méthode `receive()` de la classe dérivant de `Comm_Object`, mais on peut également choisir de le laisser dans la classe des données actives. L'accès aux données de l'objet serveur se fera via la méthode `Data_Comm_Object::get_receiver()` (voir Code 5.8, page 62).

Dans tous les cas, l'application sera donc organisée en un ensemble de clusters hébergeant chacun plusieurs serveurs. Un objet servant à l'équilibrage de charge pourra centraliser les demandes de services afin de répartir au mieux les demandes de service sur l'ensemble des serveurs de l'application. Afin d'avoir un modèle homogène, les clients peuvent également être représentés par des classes dérivant de la classe `Comm_Object`.

Le code 8.3 présente un exemple d'application client/serveur utilisant les objets communicants de SCHOONER. On crée d'abord un objet serveur, instance de la classe `Name_Server`, puis une requête lui est envoyée pour connaître le nom — dans le domaine ARPA — de la machine ayant l'adresse 33.13.252.140.

Code 8.3 *Exemple d'application client/serveur au dessus de* SCHOONER.

```
Proxy_Comm_Object *server = new Proxy_Comm_Object(Cluster::best(), "Name_Server");
DCO_Service *request = new DCO_Service;
request→set_service(Domain::ARPA);
request→set_address("33.13.252.140");
server→send(request);
...
```

Chapitre 9

Une implémentation du modèle C++// au dessus de SCHOONER étendu

✔ *On présente dans ce chapitre le langage C++// tel qu'il est implémenté au dessus des objets communicants de* SCHOONER.

9.1 Le modèle C++//

C++// est une extension parallèle du langage C++ développée au sein du projet SLOOP. Cette extension est inspirée du travail de D. CAROMEL sur Eiffel// [Caromel 91]. Nous présentons dans la suite de cette section les caractéristiques de C++// qui auront à s'appuyer sur la bibliothèque SCHOONER. Une description plus complète du langage est disponible dans [Caromel 96].

9.1.1 Les objets actifs

Tous les objets qui sont instances d'une classe qui hérite publiquement de la classe `Process` sont des objets actifs de l'application. Les objets passifs (tous ceux qui ne sont pas actifs) appartiennent à un objet actif, ce qui donne une structure hiérarchique à une application C++//.

La figure 9.1 montre un exemple d'application construite en C++//. Les sous-systèmes (hiérarchie d'objets ayant pour racine exactement un objet actif) seront placés à l'exécution dans — ce que C++// nomme — des *contextes* via des objets `Mapping` (voir Section 9.1.3, page 104).

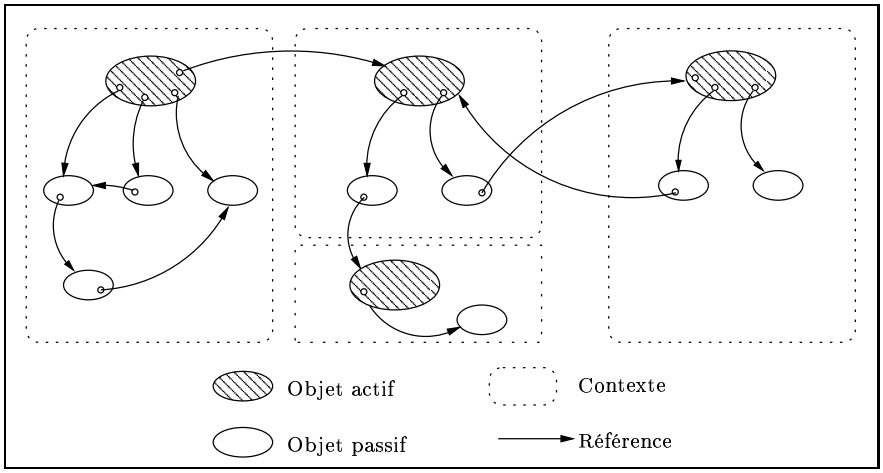

Figure 9.1 – *Exemple de hiérarchie d'objets pour une application C++//.*

9.1.2 Sémantique de la communication

Quand un objet (actif ou passif) possède une référence vers un objet actif, il a la possibilité de communiquer avec celui-ci via l'appel de l'une de ses méthodes publiques. Chaque appel de fonction sur l'objet actif résulte en l'émission d'une requête vers cet objet actif, requête qui sera déposée dans sa liste de requêtes à traiter. L'utilisateur est libre de spécifier la politique de traitement des nouvelles requêtes pour une nouvelle classe d'objets actifs, la politique de service étant par défaut FIFO[1]. Une fois une requête traitée, une éventuelle réponse est déposée dans la liste des réponses de l'appelant. L'envoi et la réception de requêtes et de réponses constituent les quatre points de réification du MOP[2] de C++// (`send_request()`, `send_reply()`, `receive_request()` et `receive_reply()`).

Il existe 3 protocoles différents qui permettent — à un objet actif ou à un objet passif — de déposer une requête (respectivement une réponse) dans la liste des requêtes (resp. réponses) d'un objet actif[3]. Chaque objet actif applique un seul et même protocole tout au long de son existence.

Ces protocoles se décrivent brièvement ainsi :

[1] Les requêtes sont servies dans l'ordre de leur arrivée.
[2] Meta-Objet-Protocol, Protocole Méta-Objet
[3] Dans la suite, nous ne parlerons plus que du cas des requêtes, celui des réponses étant identique.

1. *le protocole* ***proactif*** : la communication est non-bloquante pour l'objet émetteur et l'objet récepteur n'est pas interrompu. Ce dernier devra explicitement déclencher la routine permettant de recevoir et stocker les nouvelles requêtes dans sa file de requêtes ;
2. *le protocole* ***réactif par rendez-vous*** : 1^o) un rendez-vous est établi avec l'objet récepteur qui est interrompu ; 2^o) la requête est transmise dans la file de requêtes du récepteur ; 3^o) le rendez-vous se termine par le déblocage de l'émetteur ;
3. *le protocole* ***réactif*** : la communication est non-bloquante pour l'objet émetteur et l'objet récepteur est interrompu quand une nouvelle requête est déposée dans sa file de requêtes.

Ces protocoles se distinguent donc selon que le dépôt dans la file des requêtes d'une nouvelle requête (par exécution de la routine `receive_request()`) se fait à l'insu de l'objet actif destinataire ou non. Dans le cas où ce n'est pas à son insu, cela signifie que c'est à la charge de la routine `Live()` de l'objet — routine qui décrit le comportement de l'objet actif — de faire appel à la routine `receive_request()`.

La présence de ce mode appelé *Proactif* ([Baude 96a]) s'explique presque uniquement par d'éventuelles difficultés techniques à exécuter de façon transparente[4] le point de réification `receive_request()`.

Pour diverses raisons (dont une s'expliquant par la filiation de C++// avec Eiffel//), on dispose des 2 protocoles nommés *Réactif avec Rendez-vous* et *Réactif* où l'exécution de `receive_request()` se réalise à l'insu de l'objet actif destinataire.

Le mode *Réactif avec Rendez-vous* correspond à la sémantique standard de la communication en Eiffel// qui quoique globalement asynchrone comporte une phase synchrone. Cette dernière se décompose en une phase d'interruption de l'exécution normale de l'objet actif puis d'un rendez-vous entre l'appelant et l'appelé qui se termine lorsque l'exécution de `receive_request()` a eu lieu.

Comme le montre ROUDIER dans sa thèse [Roudier 96], le fait que l'exécution normale d'un objet actif soit interrompue pour se poursuivre vers la routine `receive_request()` est un moyen pour disposer des objets réactifs (dont le comportement réactif est tout simplement mis en œuvre par une redéfinition de `receive_request()`).

L'intérêt de la phase de rendez-vous correspondant à la transmission synchrone de la requête est de plusieurs ordres comme l'explique [Caromel 91] et de plus se prête à une modélisation aisée du point de vue d'une sémantique opérationnelle [Attali 95].

Une spécification du protocole *Réactif (sans Rendez-vous)* peut être donnée en faisant le parallèle avec celle du mode réactif avec rendez-vous. Il s'agit de ne conserver de ce dernier

[4]En effet, il peut s'avérer nécessaire de faire usage des signaux, dont une gestion parfaite — c'est-à-dire sans aucune perte — peut être difficile à réaliser.

que la spécification d'une phase d'interruption de l'exécution normale de l'objet actif pour se poursuivre par la routine `receive_request()`.

Une variante de ce protocole *Réactif*, que l'on pourrait appeler *Transparent* serait la suivante : la seule contrainte est que `receive_request()` s'effectue à l'insu de l'objet actif, mais sans forcément interrompre son exécution normale. Au contraire des 2 précédents protocoles, l'implémentation de celui-ci n'exige pas que ce soit l'activité qui supporte l'exécution normale de l'objet actif qui soit déviée vers l'exécution de la routine `receive_request()`. Ce peut donc être une nouvelle activité qui prenne en charge la réception et la mise en file de la requête. Un tel protocole autorise ainsi qu'au moment de l'exécution de `receive_request()`, l'objet actif soit multi-actif[5]. Dans ce cas, charge à l'utilisateur de gérer la concurrence entre les diverses routines qui accèdent à la file des requêtes.

9.1.3 Placement des objets actifs

[Roudier 96] propose deux mécanismes pour spécifier le placement d'un nouvel objet actif : l'*ancrage* et l'*adressage*.

1. l'*ancrage* consiste à spécifier qu'un objet actif (et donc son sous-système englobant) doit être placé sur la même machine — au sens large — qu'un autre sous-système ;
2. l'*adressage* consiste à dire explicitement dans quel espace d'adressage un sous-système doit être placé.

On peut également fixer le caractère "lourd" ou "léger" de l'objet actif, signifiant qu'un nouveau contexte doit de toute façon être créé (cas du caractère "lourd" dans lequel l'objet actif sera seul) ou qu'un contexte existant servira d'hôte (cas du caractère "léger").

Une fois fixé le caractère lourd ou léger de l'objet actif, les notions d'ancrage et d'adressage offrent ainsi 2 moyens de localisation (voir Tableau 9.1). Toutes ces possibilités sont accessibles à l'utilisateur C++// via la classe `Mapping` (voir Code 9.1).

Code 9.1 *Interface (partielle) de la classe `Mapping` de C++//.*

```
class Mapping {
public:
  virtual void on_machine(const string & m) ;          // nom de machine virtuelle
  virtual void with_process(Process *p) ;     // ancrage aux côtés de l'objet Process p
  virtual void set_light() ;                   // placement sur un contexte préexistant
  virtual void set_heavy() ;                                    // nouveau contexte
};
```

[5] L'objet actif ne gère qu'une seule activité, la multi-activité est au niveau du processus.

	Placement par adressage sur la machine logique `M`	Placement par ancrage aux côtés de l'objet actif `p`
Objet actif `light`	Sur un contexte quelconque se situant sur la machine `M`	Sur le même contexte que celui de `p` et donc sur la même machine que `p`
Objet actif `heavy`	Sur un nouveau contexte à situer sur la machine `M`	Sur un nouveau contexte à situer sur la même machine que celle où s'exécute le contexte de `p`

Tableau 9.1 – *Combinaison des critères de placement.*

9.2 De Schooner à C++//

9.2.1 Principes

Les objets actifs de C++//. Ils sont implémentés via les objets communicants de Schooner de la façon suivante :

- Le constructeur de la classe `Comm_Object_Active` (classe qui spécialise la classe Schooner `Comm_Object` et permet de définir des objets communicants actifs) démarre un processus léger qu'il associe à la routine `Live()` de la classe `Process` de l'objet actif. Les deux points de réification `receive_request()` et `receive_reply()` sont codés dans cette classe.
- La classe Schooner `Proxy_Comm_Object` est dérivée pour implémenter le concept des proxys de C++//. Une instance de cette classe servira de référence distante à un objet actif. Les deux points de réification `send_request()` et `send_reply()` sont codés dans cette classe.
- Les objets `Request` et `Reply` de C++// sont encapsulés dans des classes dérivant de la classe Schooner `Data_Comm_Object`. La classe `Request` permet de représenter n'importe quelle requête (appel de méthode sur un objet actif). En particulier, ses fonctions `flat()` et `rebuild()` sont génériques : elles savent aplatir et reconstruire n'importe quelle requête. L'aplatissement consiste à faire une copie profonde des données de l'objet, à l'exclusion des objets actifs pour lesquels une référence est passée. Ainsi, la fonction `flat()` retrouve l'identifiant global à l'application de la requête à servir, et applique la fonction générique d'aplatissement d'objets C++// sur chacun des paramètres d'appel de la requête.

  Ces méthodes génériques sont présentées dans [Gamel 96, Section 3]. Leurs algorithmes utilisent le mécanisme de typage dynamique du langage C++ (RTTI, voir [Stroustrup 97]). La stratégie est basée sur un parcours en profondeur du graphe d'objets à linéariser, le problème de détection des cycles et des références multiples (un même objet référencé plusieurs fois dans le graphe) est résolu grâce à un mécanisme d'identifiant d'objet.

Placement des objets actifs. Un objet `Mapping` sert à représenter à la fois un `Computer` qui indique la machine physique et un `Cluster` qui matérialise le contexte sur lequel on veut créer le nouvel objet actif.

Les protocoles de dépôt de requêtes. Ils sont programmés dans la méthode `send()` de la classe `Proxy_Comm_Object` et dans la méthode `receive()` d'une classe descendante de la classe `Comm_Object_Active`. Il suffit donc de dériver la classe `Comm_Object_Active` de sorte à satisfaire les différents modes de communication de requêtes. Et des objets actifs C++// n'ayant pas le même mode de communication peuvent cohabiter sur le même contexte (`Cluster`).

Chacun des protocoles est ainsi implémenté à l'aide d'une classe C++ et l'organisation de ces différentes classes en un graphe d'héritage (voir Figure 9.2) permettrait de rajouter facilement un nouveau protocole de communication.

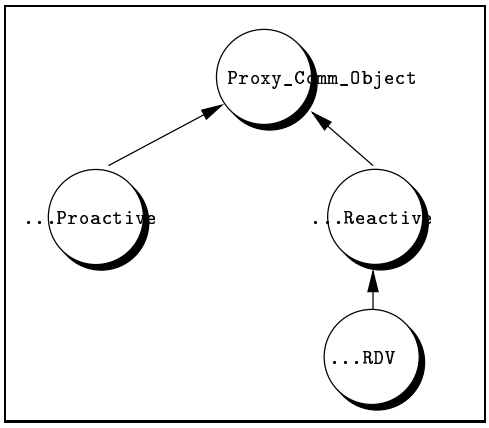

Figure 9.2 – *Hiérarchie des protocoles de dépôt de requêtes.*

9.2.2 Points concernant la mise en œuvre

Sans entrer dans les détails, il nous semble important de noter quelques points. Deux implémentations de C++// au dessus de Schooner ont été prototypées :

- une version *Objet actif lourd* (au sens du mapping C++//) ;
- une version *Objet actif léger* (au sens du mapping C++//).

9.2.2.1 Prototype "un objet actif C++// par cluster"

La version *Objet actif lourd* ne requiert que la version de base de Schooner au sens où il y a un seul objet actif C++// (c'est-à-dire instance de la classe C++// `Process`, classe mise

en œuvre dans ce prototype par dérivation de la classe Schooner `Comm_Object`) par cluster Schooner, sans utiliser du tout de processus léger.

L'unique fil d'exécution sur le cluster devant être de toute façon en charge de la réception de messages depuis la couche de communication (voir Section 8.2, page 99), il est difficile de lui faire dérouler en *même temps* la méthode `Live()` de l'objet actif (c'est-à-dire la méthode qui sert les requêtes). Ainsi, lorsqu'un message reçu sur le cluster est une requête pour l'objet actif C++//, le plus simple est que — par le biais du mécanisme de messages actifs — son traitement consiste en l'exécution de la routine `receive_request()` enchaînée immédiatement par celle du service de cette requête par l'objet actif. De plus, le cluster attend la fin de ce traitement avant de récupérer d'autres messages.

Ce prototype revient à restreindre l'implémentation du modèle C++// au cas décrit ainsi :
- service immédiat de requêtes (et donc FIFO !) ;
- file de requêtes contenant au plus un élément, avec protocole de dépôt sans interruption de l'objet actif.

Ce prototype a le mérite de la simplicité et nous a essentiellement servi de cadre de mise en œuvre et de test pour la technique de recouvrement calcul/communication présentée à la section 9.3.

9.2.2.2 Prototype "plusieurs objets actifs C++// par cluster"

La version *Objet actif léger* décrite dans [Baude 96c, Barette 96] se base sur l'extension de Schooner à la multi-activité (voir Chapitre 6, page 71) obtenue par intégration de PM^2. Décrivons en quoi cette extension se prête bien à la mise en œuvre du modèle C++//.

PM^2— plus précisément Marcel — offre des processus légers dont l'ordonnancement est totalement transparent à l'utilisateur, car à réquisition basée sur un partage de temps et des priorités. Ainsi, la routine `Live()` de chacun des objets actifs cohabitant sur le même cluster s'exécute sans intervention aucune (c'est-à-dire sans appel de code pour qu'un processus léger en préempte un autre) ni du programmeur Schooner, ni du programmeur C++//.

Tous les protocoles de dépôt de requêtes peuvent être supportés et les diverses façons de traiter une requête arrivant sur le cluster se codent dans la méthode `receive()` de la classe de l'objet actif.

En particulier, les protocoles nécessitant l'interruption de l'activité normale de l'objet actif (*réactif* avec ou sans rendez-vous) s'implémentent aisément grâce à la primitive Marcel `pthread_deviate_np(pthread_t pid, handler_func_t h, any_t arg)`. Son comportement est de détourner le processus léger d'identifiant `pid` de son calcul courant, vers l'exécution de la fonction `h(arg)`. Ainsi, la fonction `receive()` consiste en la déviation de l'activité principale de l'objet actif vers sa routine `receive_request()`.

Le mode *proactif* s'implémente en définissant un tampon qui sert de stockage intermédiaire pour les requêtes reçues sur le cluster (par le mécanisme de messages actifs) mais pour lesquelles l'objet actif n'a pas demandé le dépôt dans sa file. Ainsi la fonction `receive()` ne fait que poser la requête dans ce tampon. Charge aux objets actifs C++// ayant ce protocole de lancer l'exécution de la routine `receive_request()` depuis leur routine `Live()`, afin de récupérer les requêtes dans ce tampon et non depuis la couche de communication et de les stocker dans leur file de requêtes.

Vues les descriptions ci-dessus, nous voyons que le modèle C++// s'implémenterait tout aussi bien en utilisant directement PM^2 (ou comme cela a été expérimenté au sein de l'équipe, au dessus de Nexus [6]) puisque ces modèles fonctionnent par appel de service distant ! L'intérêt d'interfacer Schooner— outre de servir d'exemple grandeur nature — est de pouvoir rendre les couches support (`Computer`, `Cluster`) élégamment accessibles au programmeur d'applications C++//. La classe `Message` (modélisant les données que les clusters s'échangent) peut être complètement personnalisée et ainsi servir à diverses opérations de gestion ou d'observation du support et ce sans interférence avec le code de l'application proprement dite.

9.3 Recouvrement calcul et communications

9.3.1 Motivations et prérequis

L'idée de recouvrir des communications par du calcul afin de masquer le surcoût dû à la répartition des entités appelantes et appelées est attrayante mais certes pas neuve. Mais à notre connaissance, cette idée n'a jamais été exploitée dans le contexte de langages orientés objet répartis basés sur l'invocation de services distants par le biais d'appels de méthodes, tels les mécanismes de RPC en C/C++ ou l'extension RMI de Java pour ne citer qu'eux. L'optimisation du processus de copie des paramètres décrit dans [Lopes 96] est une approche différente et complémentaire.

L'idée générale est de faire en sorte que lors d'un calcul distant portant sur des données volumineuses qu'il est nécessaire de transmettre, il y ait en fait décomposition de la communication et du calcul en phases portant sur de plus petits volumes de données ; puis, il s'agit de pipeliner ces phases afin d'obtenir du recouvrement entre la phase courante de calcul distant et la transmission des données correspondant à la phase suivante du calcul distant. Cela présuppose bien évidemment de pouvoir exécuter **en même temps** du calcul distant et — au moins les étapes strictement de transmission — de la communication. Ceci peut être obtenu via une communication asynchrone.

[6] Toutefois, Nexus ne disposant pas de l'équivalent de la déviation de processus légers, seul le protocole "transparent" tel que nous l'avons défini en Section 9.1.2 est implémenté.

Si ces données doivent subir un calcul avant leur envoi, DESPREZ montre dans sa thèse [Desprez 94a] comment décomposer ce calcul local en phases et les pipeliner avec les phases de communication et de calcul distant.

Plusieurs problèmes, des plus simples aux plus complexes, se posent donc :

1. concevoir et implémenter les mécanismes élémentaires de mise en œuvre : découpage des données, transmission de paquets de données, calcul sur des données partielles, ...
2. rendre le plus transparent possible au programmeur l'utilisation des mécanismes élémentaires. Sa seule contribution pouvant être de guider le découpage des données, si le comportement par défaut n'est pas satisfaisant.
3. dans la mesure où sont précisément connus — à la compilation ou au pire, lors de l'exécution — les calculs, leur durée, les données, leur taille, les temps de transmission et donc le support d'exécution et l'architecture sous-jacents, déterminer de façon automatique (sans l'aide du programmeur) la taille optimale des paquets de données (ceci revenant à estimer la durée des différentes phases).
   Ces hypothèses font que la mise en œuvre de cette technique s'est généralement confinée au cadre de la compilation de langages à parallélisme de données pour architectures parallèles à mémoire réparties ciblées. Citons notamment HPF [Brandes 96], FortranD [Tseng 93], mais aussi LOCCS, une bibliothèque de routines de communication et de calcul [Desprez 94b].

Notre contribution est de réaliser et évaluer la mise en œuvre de cette technique dans le contexte d'un langage orienté objet étendu avec des mécanismes pour le parallélisme et la répartition, ici C++//. Cependant, nous ne résoudrons que les points 1 et 2, puisque dans toute sa généralité et en l'absence de données précises sur le calcul à réaliser et son contexte d'exécution, le point 3 ne peut être résolu dynamiquement (une stratégie est cependant développée dans [Desprez 96]).

Ce travail a fait l'objet d'une publication à la conférence HPCN'99 [Baude 99].

9.3.2 Mise en œuvre dans C++// au dessus de SCHOONER

Obtenir du recouvrement calcul-communication lors du service d'une requête C++// se fonde sur un découpage de la demande de service.

Techniques de base disponibles.

1. Le MOP sur lequel repose C++// offre un mécanisme de *futur* [Caromel 97]. Un mécanisme de continuation automatique est en cours d'implémentation [Demoustier 98] et permet que des références vers des objets de type *futur* soient transmises, sans pour

autant que la valeur soit présente. Ayant accès aux classes constituant le MOP, la technique de recouvrement mise en œuvre au niveau de Schooner, pourra donc définir des classes héritant de la classe `Future`.

2. Puisque les requêtes sont définies par des classes dérivées de la classe Schooner `Data_Comm_Object` (voir Section 5.5.2, page 62), rien n'empêche d'en dériver de nouvelles classes de sorte à surcharger les fonctions d'aplatissement et de reconstruction. Il est donc possible de redéfinir les fonctions génériques, qui rappelons-le utilisent des informations sur les classes, fournies par le MOP.
3. De plus, même si des objets apparaissant en paramètres de requêtes ne sont pas des descendants de la classe Schooner `Data_Comm_Object`, il est tout de même possible d'associer à leur classe des fonctions d'aplatissement et de reconstruction à utiliser à la place de celles standard. Ceci est faisable grâce à une fonction du MOP de C++// qui permet de tester l'existence d'une fonction précise sur une classe réifiée quelconque.

9.3.2.1 Solutions envisageables concernant le découpage — et donc la reconstruction — d'une requête

Nous allons très brièvement exposer les différentes solutions pour découper la requête de service, bien que ce soient des contraintes techniques qui nous dictent le choix final. L'idée maîtresse est l'introduction — transparente ou non — de points de césure des paramètres de la requête. A ce niveau de l'application, ces points de césure sont seulement des indications de frontières entre messages car c'est à la couche de communication de Schooner de décider s'ils délimitent vraiment des ensembles de données qui seront acheminés par autant de messages. Ainsi, par exemple, si le mécanisme de bufferisation de Schooner est actif (voir Section 7.2.2, page 87), ces points de césure ne seront pas pris en compte.

La fonction d'aplatissement est générique (voir Section 9.2.1, page 105). Il est possible de la redéfinir de sorte à ce qu'un point de césure soit inséré par exemple :

1. entre chaque donnée membre du graphe d'objets à aplatir, ce qui étant donné le caractère récursif de la fonction revient à ce que chaque donnée d'un type de base soit dissociée des autres ;
2. ou entre chaque niveau d'aplatissement du graphe d'objets dans le cas où l'algorithme de parcours de ce dernier fonctionne en largeur.

D'autre part, la possibilité de définir pour une classe donnée sa propre routine d'aplatissement pourrait permettre de personnaliser l'insertion de points de césure, par exemple en en mettant moins que ce que la stratégie numéro 1 aurait fait (par exemple voir Code 9.2).

On pourrait également imaginer de ne pas faire figurer toutes les données constituant la requête. Bien évidemment, cela réclame une routine de reconstruction adéquate, qui par exemple sait fabriquer les données manquantes et ce à coût moindre que celui de leur transmission.

Code 9.2 *Fonction `flat(... )` avec points de césure insérés manuellement.*

```
Buffer *Matrix::flat(Buffer *buff) {
  *buff << nb_line << nb_column;
  buff→cut();                                   // insertion d'un point de césure
  for (i=0; i<nb_line; i++) {
    *buff << line(i);
    buff→cut();                                 // insertion d'un point de césure
  }
  return buff;
}
```

Utilisation du principe de *futur*. Pour que la reconstruction de la requête ne soit pas bloquée en attente de l'arrivée d'un de ses éléments, mais puisse tout de même s'achever pour que l'exécution de la fonction demandée puisse démarrer, il est indispensable que l'aplatissement insère à la place d'une donnée manquante une information signalant son absence. L'algorithme de reconstruction — qu'il soit générique ou non — doit pouvoir utiliser cette information pour remplacer une donnée manquante par un objet initialement vide mais rempli ultérieurement, c'est-à-dire par un objet de même type que celui attendu, se comportant en plus selon le mécanisme de *futur* : un éventuel accès à un tel objet s'il est vide est bloquant (principe de l'attente par nécessité du mécanisme de *futur*)[7].

L'idéal serait que les données manquantes soient — *de façon transparente*[8] au programmeur C++//— instances de classes héritant de la classe `Future` définie dans le MOP de C++//. Pour ce faire, il faudrait que la routine de reconstruction dispose à l'exécution d'informations relatives à la classe dont l'objet manquant est instance. Or, si cet objet n'est ni actif, ni un *futur*, le MOP ne sait lui fournir de telles informations. Toutefois, une version en cours d'implémentation de C++// permettrait au programmeur d'indiquer quelles classes d'objets devraient tout de même "être connues" du MOP avant l'exécution (c'est-à-dire réifiées).

Solution retenue. L'état actuel de C++// force ainsi que tous les paramètres de requête qui ne seront pas transmis immédiatement — tout en permettant au service de démarrer — soient *déclarés* de type *futur*. La méthode présentée dans le code 9.3 qui est définie avec des paramètres de la classe `Matrix` pourra être appelée avec des objets de la classe `Matrix_Future`, obtenue par héritage de la classe `Matrix` et de la classe `Future` de C++//. De plus, dans ce cas, la méthode `flat(... )` de l'exemple 9.2 n'a donc plus lieu d'être définie. Le polymorphisme permet donc ici de mettre en œuvre simplement la technique de recouvrement.

En présence d'un objet de type *futur*, le mécanisme d'aplatissement se contentera lors

[7] par hypothèse, l'accès est réalisé par le biais d'un appel de méthode ; si tel n'est pas le cas, l'absence des données ne peut pas être testée ...

[8] Utilisation du mécanisme de *futur automatiques*.

Code 9.3 *Fonction ayant comme paramètre des objets dérivant de* `Future`.

```
class OpMatrix : public Process {
 virtual int rang(Weight w1, Weight w2, Matrix *m1, Matrix *m2) {
   w1→plus(w2) ;
   m1→square() ;                                    // t10 (Figure 9.3)
   m1→plus(m2) ;
   return m1→result(w1) ;
}
};
```

du premier parcours du graphe d'objets d'en poser une référence. Ce sera lors d'un parcours ultérieur, que l'accès effectif à un objet de ce type sera réalisé afin de le mettre à plat — provoquant éventuellement une attente[9]. Avec cette solution, le problème de l'emplacement des points de césure est implicitement résolu : ils sont là où il y a des objets de type *futur* !

Mise en œuvre au niveau de Schooner. Le découpage d'une requête combiné à son aplatissement doit être réalisé en plusieurs étapes de parcours du graphe d'objets. La première étape se termine par l'émission d'un premier convoi associé à la requête, contenant toutes les données mises à plat qui ne sont pas de type *futur*. Puis, d'autres convois suivront pour les données de type *futur*. On peut par exemple imaginer un convoi par objet de type *futur*. Le plus élégant est que la fabrication de ces convois se réalise au niveau du point de réification `send_request()`, c'est-à-dire dans la routine `send()` d'une classe dérivée de la classe `Proxy_Comm_Object`. Une solution alternative est que les convois soient fabriqués par le gestionnaire d'objets communicants, obligeant ainsi une spécialisation des classes Schooner. Les données à convoyer s'obtiennent comme à l'ordinaire par application de la routine d'aplatissement générique — puisque elles sont d'un type héritant de `Data_Comm_Object` — ou par une routine spécifique si elle existe. Pour qu'un convoi ultérieur soit réceptionné de façon automatique côté récepteur, il suffit qu'il hérite de la classe Schooner `Data_Comm_Object`. Son traitement est celui qu'on applique classiquement à un objet de type *futur* à savoir la mise à jour — au niveau de l'objet actif en train de servir la requête — de l'objet de type *futur* correspondant avec les données convoyées (voir Paragraphe "Utilisation du principe de futur", page 111).

[9]En effet, une fonction distante peut elle-même appeler une autre fonction distante en lui passant un de ses paramètres qui au moment de l'aplatissement de la requête correspondante peut être manquant.

9.3.3 Évaluation des performances

9.3.3.1 Description de l'expérience

Programme. Comme expliqué plus haut, du fait que les *futur* automatiques ne sont pas disponibles en C++//, les fonctions que l'on peut invoquer à distance doivent avoir des paramètres déclarés de type *futur* pour tirer partie de la technique de recouvrement. De la sorte, lors d'un appel d'une fonction distante, le cluster où s'exécute l'appelant envoie un premier message avec les données qui ne sont pas de type *futur* et des références vers les objets de type *futur*. Ce n'est qu'ensuite que les objets réellement pointés par les objets de type *futur* sont envoyés.

L'expérience qui a été menée consiste à invoquer la fonction `OpMatrix::rang(... )` décrite dans le code 9.3, en faisant varier non seulement la taille des matrices, mais aussi la durée du calcul sur `m1` (fonction `m1→square()`). La méthode est donc appelée avec deux objets de la classe `Matrix_Future`[10] (voir Code 9.4).

Code 9.4 *Appel de la fonction distante en vue d'obtenir du recouvrement.*

```
OpMatrix *dom = EC_new(("host1", OpMatrix, ()));
Matrix *m1 = EC_new(Matrix_Future, (COLUMN, LINE));
Matrix *m2 = EC_new(Matrix_Future, (COLUMN, LINE));
set_value(m1); set_value(m2);                    // remplir les matrices m1 et m2
démarrer_horloge(TempsTotal);
int res = dom→rang(w1, w2, m1, m2);
arrêter_horloge(TempsTotal);
```

Plan d'expérience. Pour chaque combinaison taille de matrice — variant selon les cas par pas de 1000 ou de 10000 — et nombre de boucles constituant le calcul sur `m1`, nous avons répété le test 10 fois. Pour évaluer l'intérêt de la technique, il faut pouvoir modifier la durée de calcul (ce que nous obtenons par le biais de la variation de la taille des matrices et du nombre de boucles dans le calcul sur `m1`) et la durée d'une communication élémentaire (ce que nous obtenons par l'utilisation de 3 supports de communication dont les latences et débits sont très différents : 1°) un réseau *local* bâti sur un switch Ethernet 100 Mbits, 2°) un lien ATM entre l'IUT de Sophia-Antipolis et l'INRIA Sophia-Antipolis et 3°) un lien Ethernet — traversant l'Internet — entre un laboratoire de Lyon et l'INRIA Sophia-Antipolis). Par ailleurs, nous avons testé 2 façons d'effectuer la transmission des données par le biais de PVM. Une utilisant les démons PVM, l'autre en communication directe entre clusters. Ces 2 techniques sont également un moyen de modifier la durée des communications, mais aussi un moyen de modifier le comportement général de l'expérience, afin d'en tirer des analyses par

[10] obtenue par héritage de la classe `Matrix` et de la classe `Future` de C++//

recoupements.

9.3.3.2 Résultats et analyse

Les étapes de calcul et de communication de l'expérience sont schématisées sur la figure 9.3. Notons qu'en vue d'une analyse fine des résultats, nous avons isolé les opérations s'exécutant sur les clusters de celles s'exécutant sur les démons de communication et ce aussi bien sur le nœud émetteur (E) que sur le nœud récepteur (R).

- *Sans recouvrement* [Figure 9.3.a]. Le traitement est ici normal. Le message envoyé contient l'identifiant de la fonction à exécuter, les 2 entiers, ainsi que les deux matrices. Le calcul ne commence qu'une fois l'ensemble des données arrivé.
- *Avec recouvrement* [Figures 9.3.b*i*]. En théorie, le mécanisme de recouvrement devrait permettre d'avoir du vrai parallélisme entre des opérations sur le nœud récepteur, sur le nœud émetteur et le réseau. La figure 9.3.b1 présente le cas où chacune des deux matrices est convoyée par un seul paquet ; pour la figure 9.3.b2, le convoi de chaque matrice nécessite deux paquets.

Pour simplifier les schémas, on suppose que l'envoi d'un message se décompose en 2 étapes : la préparation et l'envoi proprement dit sur le réseau. Cette deuxième étape se décompose elle-même si besoin en autant d'étapes que de paquets. Chacune d'elles ne peut s'effectuer qu'à condition que la réception du paquet précédemment envoyé ait été acquitté (la phase de réception du dit-paquet a été effectuée). En effet même si du point de vue de l'application, la transmission est asynchrone, on peut s'apercevoir, par exemple en PVM, que les démons communiquent de façon synchrone puisqu'ils n'ont entre eux qu'un seul paquet non acquitté.

Le temps d'exécution proprement dit de la fonction distante est le même dans tous les cas. Toutefois dans les cas avec recouvrement, la durée qui s'écoule entre le démarrage de cette fonction et son achèvement est plus longue puisque y est incluse celles d'opérations liées à la transmission des deux matrices. En particulier, on s'aperçoit que dans les expériences avec recouvrement, la durée t_{10} mesurée est plus longue que pendant les expériences sans recouvrement. On voit bien sur le schéma 9.3.b2 que la durée entre le début et la fin de `m1→square()` inclut le temps de calcul lui-même (t_{10}) ET le temps de réception du premier paquet de `m2` au niveau de la couche de communication (t_{14-1}).

t_1	Préparation de l'envoi pour `mid(rang)` et les 2 entiers	t_{10}	Calcul pour la matrice `m1`
t_2	Envoi sur le réseau de `mid(rang)` et des 2 entiers	t_{11}	Préparation de l'envoi de la matrice `m2`
t_3	Transfert réseau de `mid(rang)` et des 2 entiers	$t_{12\{-i\}}$	Envoi sur le réseau de la matrice `m2`
t_4	Réception de `mid(rang)` et des 2 entiers	$t_{13\{-i\}}$	Transmission réseau de la matrice `m2`
t_5	Calcul pour `mid(rang)` et les 2 entiers	$t_{14\{-i\}}$	Réception de la matrice `m2`
t_6	Préparation de l'envoi de la matrice `m1`	t_{15}	Calcul pour la matrice `m2`
$t_{7\{-i\}}$	Envoi sur le réseau de la matrice `m1`	t_{16}	Préparation de l'envoi de `res`
$t_{8\{-i\}}$	Transmission réseau de la matrice `m1`	t_{17}	Transmission réseau de `res`
$t_{9\{-i\}}$	Réception de la matrice `m1`	t_{18}	Réception de `res`

FIGURE 9.3 – *Etapes schématiques montrant le potentiel de recouvrement calcul / communication.*

Dans le cas où les matrices sont acheminées par plus d'un paquet, il est intéressant de noter que le calcul sur la matrice `m1` peut être interrompu par l'arrivée d'un paquet de la matrice `m2`. Selon la manière dont ces 2 tâches vont être entrelacées, l'envoi du paquet suivant de `m2` pourra être plus ou moins retardé par rapport aux expériences sans recouvrement. Quand cet envoi tarde, celà augmente donc par conséquent la durée d'envoi totale de la matrice `m2`. C'est pour cette raison que certaines expériences avec recouvrement n'apporteront aucune amélioration, voire une détérioration du temps d'exécution de l'appel `res = dom→rang( ... )`.

Notons bien que la motivation de ce travail n'est pas de recouvrir les phases de calcul et de communication liées aux transmissions des paramètres de l'appel de méthode (comme par exemple un recouvrement entre t_{11} et t_8 sur le schéma 9.3.b1) mais en priorité les phases d'exécution de cette méthode avec la transmission de ses paramètres d'appel (ce qui revient à recouvrir avant tout la durée t_{10}). De ce fait, nos expériences portent sur des volumes de calcul important (t_{10}) qui ainsi rendent les phases liées aux transmissions parfois négligeables.

Potentiel de recouvrement. C'est le gain en temps concernant l'appel complet de la fonction distante — y compris la réception de la réponse du point de vue de l'appelant — que l'on cherche à maximiser. Dans la suite, ces temps d'appel seront notés $TempsTotal_{appel\_sans\_future}$ et $TempsTotal_{appel\_avec\_future}$.

Vu notre objectif prioritaire expliqué précédemment, on évalue uniquement le recouvrement du calcul sur `m1` par la transmission de `m2`, recouvrement qui est conjonction de nombreux paramètres :

1. les plates-formes et leur état (machines, réseau et leur seuil de charge, ...) ;
2. la durée du calcul sur `m1` (t_{10}, Figure 9.3) : plus ce calcul est court, moins le gain lié à son recouvrement est significatif ;
3. la durée de la transmission de `m2` (t_{13}, Figure 9.3) : plus elle est courte, moins le gain lié au fait qu'elle recouvre une certaine quantité de temps de calcul sur `m1` est visible.

Sans oublier, que la mise en œuvre de la technique de recouvrement induit un surcoût en temps (que nous évaluerons plus loin).

Ainsi, pour mieux apprécier une économie (positive, voire négative !) en temps total d'appel de la fonction distante, nous définissons la valeur suivante :

$$gain = \frac{TempsTotal_{appel\_sans\_future} - TempsTotal_{appel\_avec\_future}}{Temps\_Calcul\_sur\_m1_{appel\_sans\_future}}$$

Il est important d'avoir à l'esprit que ce gain *ne peut excéder* la durée de recouvrement de calcul sur `m1` par la transmission de `m2`. Ainsi le *gain maximum* est théoriquement de 1, en effet ce maximum est atteint lorsque toute la durée de calcul sur `m1` a pu être économisée.

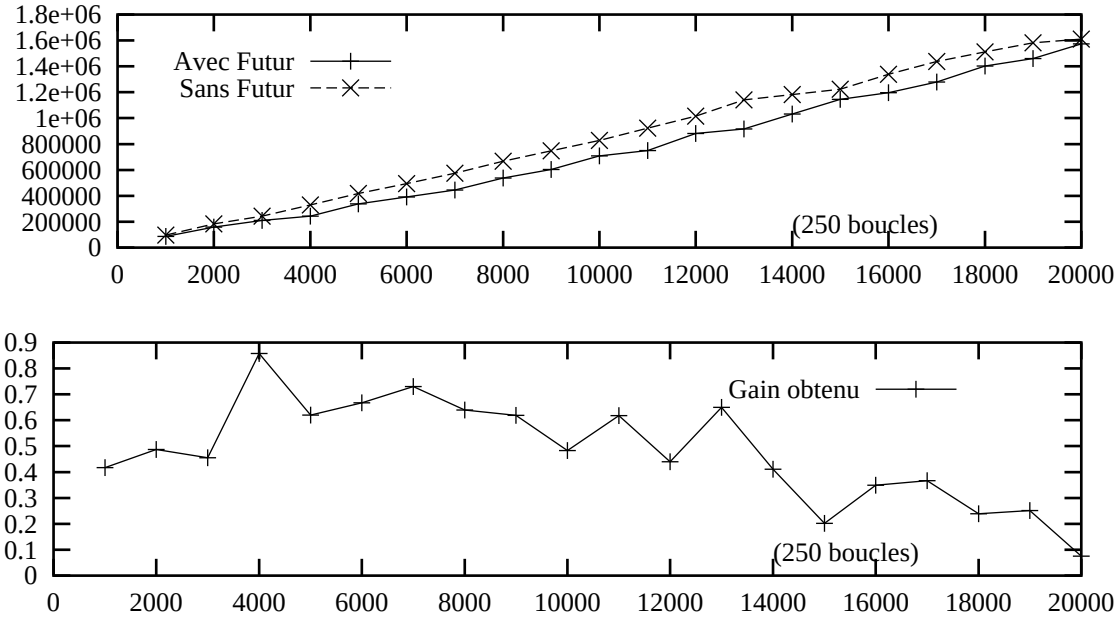

FIGURE 9.4 – *Temps d'exécution de l'appel de la fonction distante (*`res = dom→rang(... )`*, μs) en fonction de la taille des matrices (nombre d'entiers) et gain obtenu. Communication directe entre les clusters. Test effectué entre l'IUT et l'INRIA.*

Quelques résultats. Pour discuter de l'intérêt de la technique de recouvrement, nous devons omettre les résultats d'expérience pour lesquels le gain est supérieur à 1 (comme sur la deuxième courbe de la figure 9.5 par exemple). Ces points correspondent au cas où les expériences n'utilisant pas le mécanisme de *futur* ont rencontré une charge réseau toujours plus importante que celles utilisant le mécanisme de *futur*. De ce fait, les expériences sans utiliser le mécanisme de *futur* ont été anormalement "mauvaises" alors que celles l'utilisant ont été normales.

Cependant, la technique n'étant pas gratuite (nous évaluerons son coût plus loin), nous ne pensons pas que le gain maximum soit atteignable et nous observons en effet un gain inférieur à 1 dans la pratique. Il est même souvent négatif (dès lors que $TempsTotal_{appel\_avec\_future} > TempsTotal_{appel\_sans\_future}$). Si tel est le cas, 2 phénomènes additifs ont pu se produire :

1. un recouvrement même minime a eu lieu mais le surcoût dû au mécanisme lui-même l'a masqué ;
2. surtout pour un réseau partagé, la charge rencontrée par les expériences utilisant le mécanisme de *futur* leur a été toujours très défavorable (voir par exemple les points correspondant à des pics négatifs sur les courbes de la figure 9.5).

Dans toutes les expériences qui ont été menées, un recouvrement a cependant toujours eu lieu. En effet, les mesures de durée entre le démarrage — sur l'objet appelé — de la fonction qui

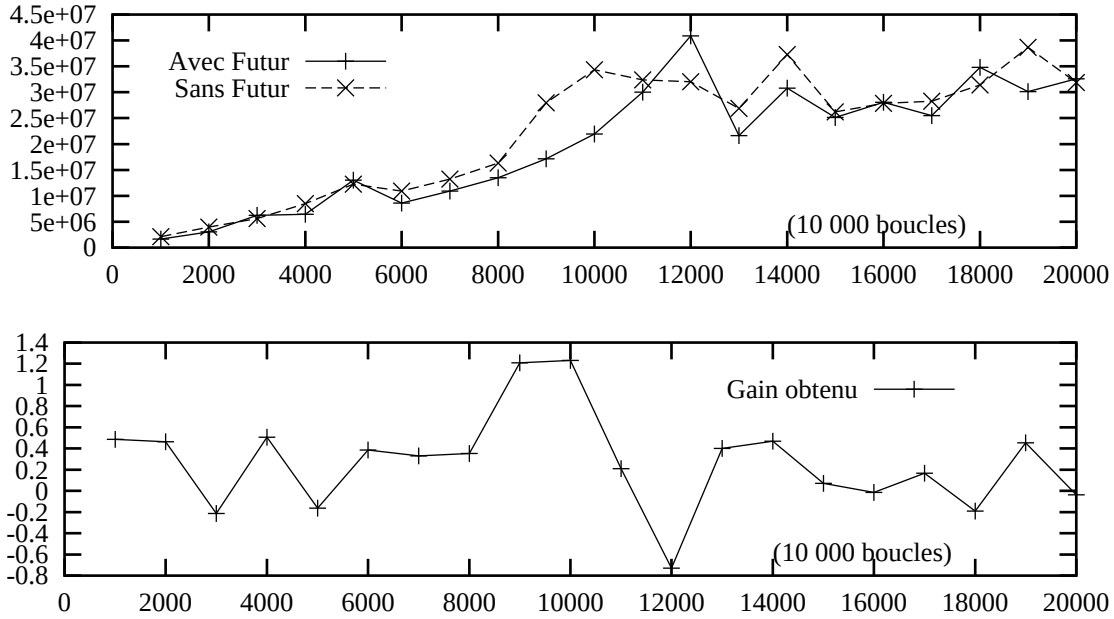

Figure 9.5 – *Temps d'exécution de l'appel de la fonction distante (*`res = dom→rang( ... )`*, µs) en fonction de la taille des matrices (nombre d'entiers) et gain obtenu. Communication directe entre les clusters. Test effectué entre Lyon et l'INRIA Sophia-Antipolis.*

utilise seulement la première matrice (`m1→square()`) et son achèvement lorsque l'on utilise le mécanisme de *futur*, sont supérieures au cas sans *futur*. De plus, ces durées croissent avec la taille de `m2` mais ne sont pas influencées par la taille du calcul sur `m1`, par rapport au cas sans *futur*. En d'autres termes : bien que le calcul distant ait démarré parce que `m1` est disponible, il est perturbé par la réception de la matrice `m2` ; l'émission et la transmission — au moins en partie — de `m2` se produisent donc alors que le calcul sur `m1` est en cours (voir Figure 9.6). Si on se reporte aux schémas de la figure 9.3, cela signifie que la durée entre le début et la fin du segment étiquetté t_{10} sur le schéma b2 est plus longue — même si ce n'est souvent que légèrement — que celle sur le schéma a.

Evaluation du coût du mécanisme provoquant du recouvrement calcul/communication. Le coût de la technique qui permet d'obtenir du recouvrement calcul/communication se décline en :

- un *surcoût fixe* indépendant de la taille des données et de la complexité du calcul. Il consiste principalement en l'appel de fonctions supplémentaires (aussi bien au niveau de Schooner qu'au niveau des processus et fonctions système en charge de la communication) pour fabriquer, réceptionner et traiter les messages supplémentaires. Mais ce nombre de messages est constant (de 2 puisque 2 objets *futur* dans l'appel de la fonction testée ; en général le nombre de paramètres d'une fonction est faible donc assimilable à

une constante).

- un *surcoût variable* selon la taille des données à convoyer. Au niveau de SCHOONER, aucun travail supplémentaire (allocation de mémoire par exemple) pouvant dépendre de la taille de chacune des matrices n'est demandé. Ce surcoût lorsqu'on expérimente la technique avec *futur*, est provoqué par une augmentation du temps de transmission de `m2`, du fait que sur le CPU où s'exécute le cluster destination s'effectue dans le même temps la tâche de réception de `m2` et du calcul sur `m1`.
  Pour tenter d'expliquer cette augmentation, étudions le comportement de PVM (puisque c'est en interfaçant cet outil de communication que les expériences ont été menées).
  - *lorsque les démons sont utilisés pour la communication* : tant qu'un paquet n'est pas acquitté, nous savons que le démon émetteur ne transmet pas le paquet suivant (la fenêtre de transmission est de 1, par défaut). Cet acquittement peut mettre plus de temps à arriver du fait du mécanisme avec *futur* puisque le démon destination peut perdre le CPU — et donc ne peut faire partir l'acquittement — au profit du cluster sur lequel s'effectuent les calculs sur `m1`. Plus l'acquittement tarde, plus le transfert des paquets suivants de `m2` tardent, provoquant ainsi une augmentation du temps total de transmission de `m2`. Ce phénomène sur un paquet peut se reproduire pour chacun et explique que le surcoût puisse dépendre de la taille de `m2` (donc du nombre des paquets utiles à sa transmission).
  - *lorsque les communications sont directes entre clusters*, la fenêtre de transmission du protocole TCP/IP [Stevens 92] peut se fermer empêchant alors le cluster émetteur de poursuivre l'émission de `m2`. C'est une conjonction de plusieurs paramètres qui peut provoquer cette fermeture de fenêtre dans le cas utilisant le mécanisme de *futur*, s'expliquant ainsi : le cluster parce qu'il est en train de calculer, ne lit pas les paquets sur son socket le reliant au cluster émetteur, ce socket finissant par être plein.
    Grâce à l'outil `tcpdump`, nous avons pu par exemple observer une fermeture de la fenêtre pendant un temps non négligeable, pour l'expérience avec mécanisme de *futur* où la taille des matrices est de 5000 entiers et la taille du calcul sur `m1` de 1000 boucles, sur une configuration de 2 Pentiums sous Linux, reliés par Myrinet (voir Annexe B). De ce fait, l'émission stoppe et le recouvrement aussi, donc dans ce cas, le gain ne pourra pas être optimal.

Malgré l'obtention systématique d'un recouvrement entre le calcul de `m1` et la transmission de `m2`, le gain n'est pas toujours bien favorable, surtout lorsque la vitesse de communication est importante (ceci est dû aux performances brutes du lien, mais également à l'absence de partage comme c'est le cas pour le réseau local utilisé qui est un switch et non un bus). Dans ce cas, les performances de communication sont tellement bonnes que le moindre facteur — lié à l'utilisation de la technique elle-même — qui ralentit l'émission de `m2` devient sensible si l'on compare 2 expériences menées exactement dans les mêmes conditions, l'une utilisant le mécanisme de *futur*, l'autre non (voir Figure 9.7).

9.3.4 Conclusion

Un recouvrement systématique est indéniablement obtenu puisque lorsque le calcul distant a démarré, la transmission des paramètres de type *futur* s'effectue encore. Mais du fait que le cluster distant est occupé à calculer, la durée de transmission de ces paramètres peut augmenter par rapport à une version sans recouvrement. De plus, dans le cas d'une communication directe entre deux clusters, le recouvrement n'est pas optimal (car limité par la taille du socket sur le cluster récepteur). L'avantage à utiliser cette technique peut donc s'estomper voire disparaître totalement, voire se transformer en inconvénient ...

Il est donc important d'avoir connaissance des phénomènes que nous avons tenté de mettre en lumière pour savoir sélectionner les situations où l'utilisation de cette technique s'avère gagnante. Tentons d'en faire la synthèse :

1. Le temps de calcul sur les paramètres déjà reçus doit être comparable ou supérieur à la durée de transmission des paramètres suivant (en gardant à l'esprit que le gain ne peut excéder le minimum de ces 2 valeurs).
2. La durée de transmission doit être suffisamment importante pour rendre négligeable une potentielle augmentation de ce même temps. Cette augmentation est due au long délai de réaction des couches systèmes d'exploitation et réseau sur le site récepteur.

Ainsi même si le réseau n'est pas aussi lent que pourrait l'être Internet, le gain peut être bon comme le montre la figure 9.4. En présence de communications à grande distance, le gain est souvent plus variable du fait de variations sensibles des durées de communication. Cependant, il peut s'avérer positif et commencer à être significatif si durant l'exécution d'une application, ce mécanisme est plusieurs fois utilisé.

Une toute autre série de mesures — que nous n'avons pas eu le loisir de mener à bien — serait de mettre les 2 clusters en état d'occupation dans les 2 catégories d'expériences, avec ou sans l'utilisation du mécanisme de *futur*. Ceci permettrait pensons-nous de ne pas subir l'augmentation du temps de transmission des paramètres de type *futur* dans les expériences utilisant le mécanisme de *futur* par rapport aux autres. Ainsi nous pensons que la technique de recouvrement produirait un gain net quant aux durées d'exécution.

L'implémentation de la technique présentée dans cette section dans le cadre d'un système à objets répartis nécessite principalement de pouvoir être capable de modifier le support d'exécution de ce système. Si tel est le cas, seules les fonctionnalités de mise à plat et de reconstruction pour l'appel de méthode distantes ont à être modifiées : l'objet représentant l'appel distant doit être fragmenté en plusieurs sous-messages indépendamment convoyés. Ceci nécessite l'utilisation d'un mécanisme de type *futur*.

Au niveau du support d'exécution, un mécanisme de message automatique est requis afin de recevoir et traiter de manière transparente les différents fragments constituant le message global.

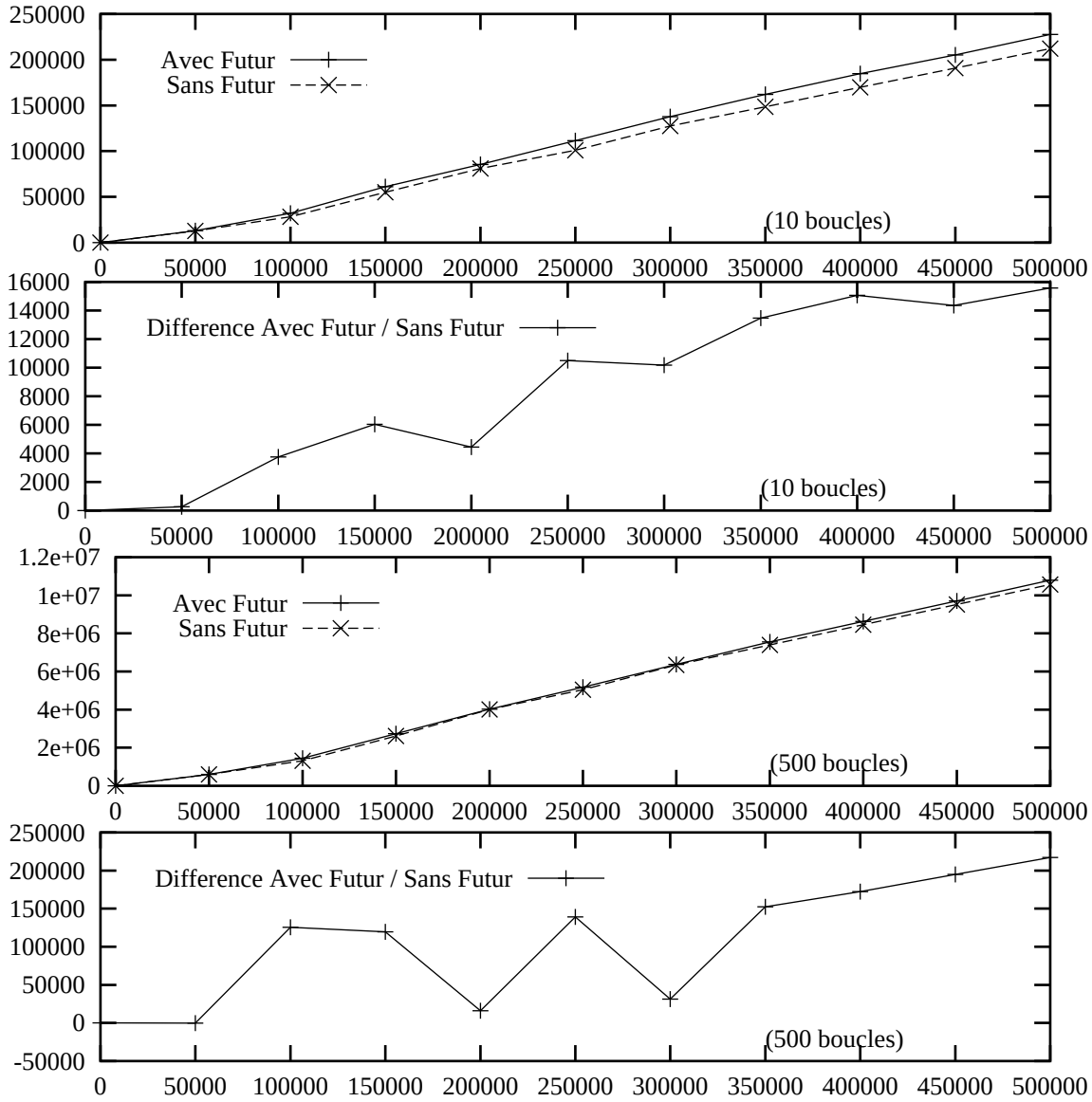

FIGURE 9.6 – *Temps d'exécution du calcul distant sur* $m1$ *(*`m1→square()`*,* t_{10} *sur Figure 9.3),* μs*) en fonction de la taille de la matrice (nombre d'entiers). Test effectué sur le réseau local. Communication utilisant le démon* PVM.

On observe une différence en durée d'exécution de la fonction distante que ce soit avec ou sans futur entre le cas "10 boucles" et le cas "500 boucles". Ceci prouve bien que pendant que le calcul sur `m1` *se déroule, des réceptions de paquets de* `m2` *au niveau du démon se produisent.*

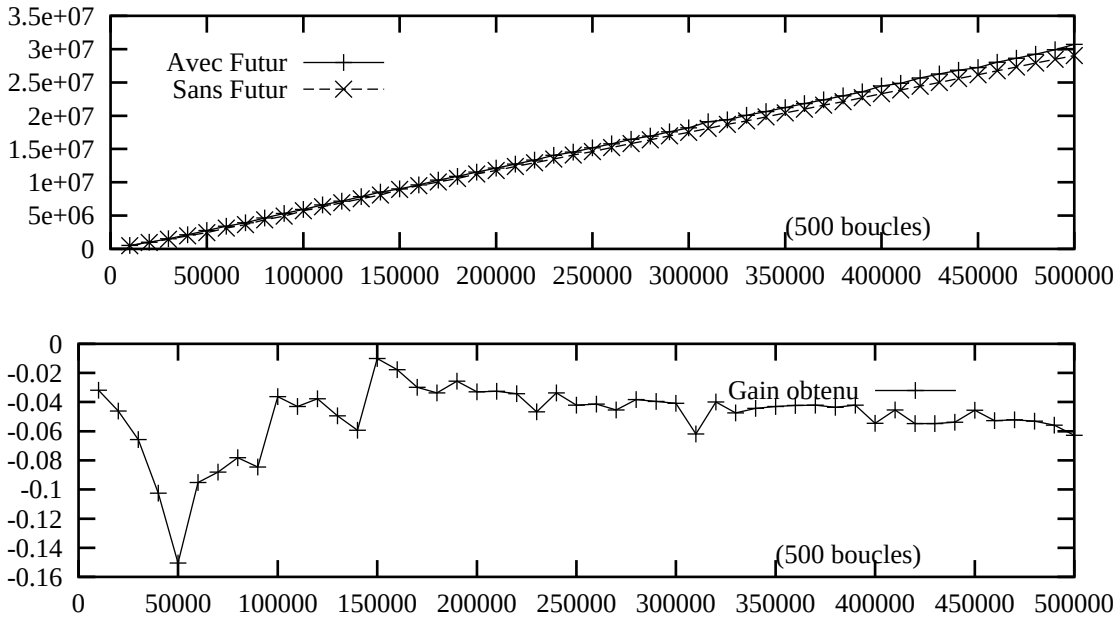

FIGURE 9.7 – *Temps d'exécution de l'appel de la fonction distante (`res = dom→rang( ... )`, μs) en fonction de la taille des matrices (nombre d'entiers) et gain obtenu. Communication directe entre les clusters. Test effectué sur le réseau local.*

Comme le réseau est rapide, il remplit très vite le socket destination, comme celui-ci n'est pas vidé, l'émission stoppe. Le recouvrement se limite donc à la transmission de données de taille égale à la taille du socket TCP (16 Ko par défaut).

Chapitre 10

Conclusion

10.1 Contributions principales

Les travaux présentés dans cette thèse portent sur la conception d'un modèle unificateur constituant un support d'exécution pour la programmation d'applications réparties, la réalisation d'une plate-forme portable de ce modèle, ainsi que d'un ensemble d'outils d'aide à la programmation. Le modèle et les librairies et outils ainsi fournis constituent un environnement appelé SCHOONER.

Cet environnement a pu être validé par le développement de deux applications "grandeurs natures" : la version répartie d'un simulateur à événements discrets, PROSIT et une extension parallèle du langage C++, C++//.

10.1.1 Le modèle de programmation et ses extensions

Le modèle SCHOONER est basé sur les entités suivantes :

1. les *computers* qui représentent les composantes de la machine virtuelle sur laquelle doit s'exécuter l'application,
2. les *clusters* ou nœuds de calcul qui sont créés sur les *computers* et représentent de manière abstraite un contexte de calcul (mémoire, CPU, etc),
3. les *objets communicants* hébergés par les *clusters* qui permettent de concevoir des applications avec différents grains de granularité.

Ce modèle permet au programmeur d'avoir une vision structurée et abstraite de son application (voir Figure 5.7, page 70). De par sa conception orientée objet, il est facilement redéfinissable et extensible, comme le montrent les exemples suivants.

Extension multi-active. Cette extension permet aux objets communicants du modèle de base de gérer leur propre activité, on obtient ainsi des objets actifs pouvant s'exécuter en concurrence au sein d'un cluster. Si ce n'est les problèmes de réentrance, la gestion de la multi-activité n'entraine aucune modification du modèle de base.

Bufferisation. Afin de minimiser les temps de communication des applications s'échangeant de nombreux messages de petite taille, nous proposons un mécanisme permettant d'envoyer en une seule fois un groupe de messages. Ce service est entièrement paramétrable par l'utilisateur et se base sur un système de canal de communication établi entre deux clusters de l'application.

Equilibrage de charge. Ce travail qui a fait l'objet d'une collaboration avec d'autres membres du projet SLOOP [Tanzy 97, Dalle 99] porte sur l'interfaçage de SCHOONER avec *DLB* une plate-forme de répartition de charge. Il est ainsi possible de placer un nouveau cluster sur le "meilleur" computer disponible. Le critère de choix se base sur différents indicateurs de charge (CPU, mémoire, entrées/sorties, communications réseau) auxquels peuvent être affectés des poids.

10.1.2 Les outils d'aide à la programmation

Afin de faciliter la mise au point d'applications SCHOONER, nous fournissons un mécanismes de traces ainsi qu'un moniteur graphique.

Mécanisme de traces. Ce système est composé d'une bibliothèque de génération de traces et d'un outil permettant la visualisation postmortem des fichiers de traces produits durant l'exécution d'une application. La génération des traces se fait par simple appel d'une fonction de la bibliothèque SCHOONER. Une fois collectés et fusionnés[1], il sera possible d'obtenir une représentation graphique des différents événements survenus au cours de l'application (création d'objets, envoi de messages, etc). L'outil de visualisation offre de plus la possibilité de filtrer les événements selon leur type.

Moniteur graphique. Un outil de suivi d'exécution a été développé, il permet de manière graphique d'interroger une application SCHOONER en cours d'exécution sur l'état de ses différents composants. Cet outil est générique dans le sens où toute application SCHOONER peut avoir son propre moniteur (la liste des questions est entièrement paramétrable).

[1] Au cours de l'application, chaque cluster génère son propre fichier de traces, il sera nécessaire par la suite de corriger le déphasage entre les horloges.

10.1.3 Les applications conçues au dessus de SCHOONER

Les choix effectués lors de la conception et du développement de l'environnement SCHOONER ont pu être validés par le développement de deux applications, PROSIT et C++//.

PROSIT. La conception de la version répartie de PROSIT a coïncidé avec la conception du modèle de SCHOONER. Cette collaboration a permis de tester et de faire évoluer de manière directe les choix effectués. PROSIT a également constitué une plate-forme de test pour différentes extensions de SCHOONER comme l'équilibrage de charge ou la définition d'opérations collectives qui a notamment permis de mettre en place un mécanisme de détection de propriétés stables.

C++//. L'utilisation de SCHOONER en tant que support d'exécution de C++// a notamment permis de proposer une technique de recouvrement du calcul par des communication. Cette technique est basée sur l'idée de découper le message représentant un appel d'une méthode distante en plusieurs petits messages afin de pouvoir commencer à calculer sur le cluster distant alors que toutes les données n'ont pas encore été acheminées. Cette technique bien que déjà existante dans le cadre de la compilation de langage à parallélisme de données n'avait pas encore été réalisée et évaluée dans le contexte d'un système à objets répartis.

10.2 Perspectives

Les objectifs annoncés dans la première partie de ce mémoire ont été atteints. L'utilisation du paradigme objet nous a permis de fournir un ensemble de bibliothèques facilement redéfinissables. Ce qui a notamment permis de tester et d'intégrer des mécanismes de bufferisation et de recouvrement calcul et communication. Bien entendu, tout ce travail est indépendant du langage objet cible, et pourrait évidemment être transposé dans le cadre du langage Java.

L'approche que nous avons identifiée dans le travail présenté ici nous semble tout à fait pertinente dans un contexte plus étendu où les objets communicants désireraient en plus utiliser de la communication multipoint. Dans ce cadre là, le choix des supports d'exécution proposant ce type de primitives est lui aussi vaste (voir par exemple, Isis [Birman 85], son successeur Horus [van Renesse 96] ou Transis [Dolev 96], ...), présentant aussi la difficulté de se situer à un niveau "processus" et non à un niveau "objet" [Guerraoui 98].

Notre solution consistant à isoler les supports d'exécution sous-jacents du niveau applicatif trouverait alors pleinement son utilité. A titre d'exemple extrême, rien ne nous empêcherait de construire — entièrement dans notre encapsulation — des primitives de communication mul-

tipoint conformes aux besoins des niveaux applicatifs, sans même que le support d'exécution sous-jacent ne propose ce type de primitive ...

En ce qui concerne les travaux relatifs au recouvrement des communications par des calculs, il pourra aussi être intéressant de compléter l'évaluation des performances en testant notre technique sur différents supports d'exécution. Cela nous permettrait également de mieux cibler les classes d'application pour lesquelles notre technique est pertinente.

Annexes

Annexe A

Liste des références sur SCHOONER

A.1 Conférences

Françoise Baude, Denis Caromel, Nathalie Furmento et David Sagnol. Overlapping Communication with Computation in the Framework of Remote Method Invocation. Dans Peter Sloot, Marian Bubak, Alfons Hoekstra et Bob Hertzberger, éditeurs, *Proceedings of the 7th International Conference - High Performance Computing Networking'99 (HPCN Europe 1999)*, volume 1593 de *Lecture Notes in Computer Science*, pages 744-753, Amsterdam (Pays-Bas), Avril 1999.

Nathalie Furmento et Françoise Baude. SCHOONER : An Object-Oriented Run-time Support for Distributed Applications. Dans K. Yetongnon and S. Hariri, éditeur, *Proceedings of Parallel and Distributed Computing Systems (PDCS'96)*, volume 1, pages 31–36, Dijon (France), Septembre 1996. International Society for Computers and their Applications (ISCA). ISBN : 1-880843-17-X.

Françoise Baude, Nathalie Furmento, Denis Caromel, Raymond Namyst, Jean-Marc Geib et Jean-François Méhaut. C++// on top of PM^2 via SCHOONER. Dans *Proceedings of Stratagem'96*, pages 41–55, Sophia Antipolis (France), Juillet 1996. ISBN-2-7261-0982-9.

Nathalie Furmento. SCHOONER : Une librairie de communications orientée-objet pour le développement d'applications parallèles. Dans *Huitièmes Rencontres Francophones du Parallélisme (Renpar8)*, page 206, Bordeaux (France), Mai 1996.

A.2 Rapports de recherche

Nathalie Furmento et Françoise Baude. SCHOONER : An Object-Oriented Run-time Support for Distributed Applications. Rapport de recherche RR95-50, Laboratoire I3S, Sophia Antipolis (France), Juillet 1995.

A.3 Rapports de stage

Alexandre Clavaud. Interface graphique pour la visualisation d'exécution d'applications distribuées. Projet de 3ème année, ESSI, Sophia-Antipolis, Avril 1997.

Sylvain Gamel. Transmission efficace d'objets pour le langage C++//. Stage de DEA Informatique, ESSI – UNSA, Sophia-Antipolis, Juillet 1996.

Philippe Barette et Cédric Tonin. Implémentation d'une librairie de communications pour un langage C++ parallèle au dessus de PVM et d'une bibliothèque de threads. Projet de 3ème année, ESSI, Sophia-Antipolis, Avril 1996.

A.4 Documents électroniques

- Site SCHOONER : `http://www.inria.fr/sloop/schooner/`
- Site PROSIT : `http://www.inria.fr/sloop/prosit/`
- Site C++// : `http://www.inria.fr/sloop/c++ll/`

Annexe B

Extrait d'un fichier de traces de l'outil `tcpdump`

On peut voir ici le fichier traces produit par la commande `tcpdump` entre 2 Pentiums sous Linux (machines de nom `pinta14` et `pinta15`), reliés par Myrinet. Jusqu'à la ligne 39, la fenêtre de réception sur la machine `pinta15` a une taille de 16352 octets, cette fenêtre passe ensuite à une taille de 11680 octets (ligne 48), puis diminue encore aux lignes 60, 61 et 64, avant de reprendre une taille de 16060 octets à la ligne 66.

Cette diminution est due au fait que le socket situé sur la machine `pinta15` n'a pas pu délivrer à l'application (au cluster) les données reçues et n'a donc pas la capacité de recevoir de nouvelles données.

L'utilisation de cet outil nous a permis de mieux comprendre ce qui se passait dans le cadre de la mise en œuvre de notre technique de recouvrement des communications par du calcul en C++// (voir Section 9.3). Pour l'expérience avec mécanisme de *futur* où la taille des matrices est de 5000 entiers et la taille du calcul sur `m1` de 1000 boucles, on peut ainsi observer une fermeture de la fenêtre pendant un temps non négligeable.

```
1  17:05:47.166966 pinta14.inria.fr.1096 > pinta15.inria.fr.1050: S 1131201777:1131201777(0) win 512 <mss 1460>

   17:05:47.166966 pinta15.inria.fr.1050 > pinta14.inria.fr.1096: S 3088977806:3088977806(0) ack 1131201778 win 16352 <mss 1460>

5  17:05:47.166966 pinta14.inria.fr.1096 > pinta15.inria.fr.1050: . ack 1 win 16060 (DF)
   17:05:47.166966 pinta14.inria.fr.1096 > pinta15.inria.fr.1050: P 1:33(32) ack 1 win 16060 (DF)
   17:05:47.176966 pinta14.inria.fr.1096 > pinta15.inria.fr.1050: P 33:93(60) ack 1 win 16060 (DF)

   17:05:47.186966 pinta15.inria.fr.1050 > pinta14.inria.fr.1096: . ack 93 win 16352 (DF)
10
   17:05:47.186966 pinta14.inria.fr.1096 > pinta15.inria.fr.1050: P 93:125(32) ack 1 win 16060 (DF)
   17:05:47.186966 pinta14.inria.fr.1096 > pinta15.inria.fr.1050: P 125:213(88) ack 1 win 16060 (DF)
   17:05:47.186966 pinta14.inria.fr.1096 > pinta15.inria.fr.1050: P 213:245(32) ack 1 win 16060 (DF)

15 17:05:47.206966 pinta15.inria.fr.1050 > pinta14.inria.fr.1096: . ack 245 win 16352 (DF)
```

```
17:05:47.206966 pinta14.inria.fr.1096 > pinta15.inria.fr.1050: P 245:397(152) ack 1 win 16060 (DF)
17:05:47.206966 pinta14.inria.fr.1096 > pinta15.inria.fr.1050: P 397:429(32) ack 1 win 16060 (DF)
17:05:47.206966 pinta14.inria.fr.1096 > pinta15.inria.fr.1050: . 429:1889(1460) ack 1 win 16060
17:05:47.206966 pinta14.inria.fr.1096 > pinta15.inria.fr.1050: . 1889:3349(1460) ack 1 win 16060

17:05:47.216966 pinta15.inria.fr.1050 > pinta14.inria.fr.1096: . ack 3349 win 16352 (DF)

17:05:47.216966 pinta14.inria.fr.1096 > pinta15.inria.fr.1050: . 3349:4809(1460) ack 1 win 16060
17:05:47.216966 pinta14.inria.fr.1096 > pinta15.inria.fr.1050: . 4809:6269(1460) ack 1 win 16060
17:05:47.216966 pinta14.inria.fr.1096 > pinta15.inria.fr.1050: . 6269:7729(1460) ack 1 win 16060
17:05:47.216966 pinta14.inria.fr.1096 > pinta15.inria.fr.1050: . 7729:9189(1460) ack 1 win 16060
17:05:47.216966 pinta14.inria.fr.1096 > pinta15.inria.fr.1050: . 9189:10649(1460) ack 1 win 16060

17:05:47.226966 pinta15.inria.fr.1050 > pinta14.inria.fr.1096: . ack 10649 win 16352 (DF)

17:05:47.226966 pinta14.inria.fr.1096 > pinta15.inria.fr.1050: . 10649:12109(1460) ack 1 win 16060
17:05:47.226966 pinta14.inria.fr.1096 > pinta15.inria.fr.1050: . 12109:13569(1460) ack 1 win 16060
17:05:47.226966 pinta14.inria.fr.1096 > pinta15.inria.fr.1050: . 13569:15029(1460) ack 1 win 16060
17:05:47.226966 pinta14.inria.fr.1096 > pinta15.inria.fr.1050: . 15029:16489(1460) ack 1 win 16060
17:05:47.226966 pinta14.inria.fr.1096 > pinta15.inria.fr.1050: . 16489:17949(1460) ack 1 win 16060
17:05:47.226966 pinta14.inria.fr.1096 > pinta15.inria.fr.1050: . 17949:19409(1460) ack 1 win 16060 (DF)

17:05:47.236966 pinta15.inria.fr.1050 > pinta14.inria.fr.1096: . ack 19409 win 16352 (DF)

17:05:47.236966 pinta14.inria.fr.1096 > pinta15.inria.fr.1050: P 19409:20557(1148) ack 1 win 16060 (DF)
17:05:47.236966 pinta14.inria.fr.1096 > pinta15.inria.fr.1050: P 20557:20589(32) ack 1 win 16060 (DF)
17:05:47.236966 pinta14.inria.fr.1096 > pinta15.inria.fr.1050: . 20589:22049(1460) ack 1 win 16060
17:05:47.236966 pinta14.inria.fr.1096 > pinta15.inria.fr.1050: . 22049:23509(1460) ack 1 win 16060
17:05:47.236966 pinta14.inria.fr.1096 > pinta15.inria.fr.1050: . 23509:24969(1460) ack 1 win 16060
17:05:47.236966 pinta14.inria.fr.1096 > pinta15.inria.fr.1050: . 24969:26429(1460) ack 1 win 16060
17:05:47.236966 pinta14.inria.fr.1096 > pinta15.inria.fr.1050: . 26429:27889(1460) ack 1 win 16060

17:05:47.246966 pinta15.inria.fr.1050 > pinta14.inria.fr.1096: . ack 27889 win 11680 (DF)

17:05:47.246966 pinta14.inria.fr.1096 > pinta15.inria.fr.1050: . 27889:29349(1460) ack 1 win 16060
17:05:47.246966 pinta14.inria.fr.1096 > pinta15.inria.fr.1050: . 29349:30809(1460) ack 1 win 16060
17:05:47.246966 pinta14.inria.fr.1096 > pinta15.inria.fr.1050: . 30809:32269(1460) ack 1 win 16060
17:05:47.246966 pinta14.inria.fr.1096 > pinta15.inria.fr.1050: . 32269:33729(1460) ack 1 win 16060
17:05:47.246966 pinta14.inria.fr.1096 > pinta15.inria.fr.1050: . 33729:35189(1460) ack 1 win 16060
17:05:47.246966 pinta14.inria.fr.1096 > pinta15.inria.fr.1050: . 35189:36649(1460) ack 1 win 16060
17:05:47.256966 pinta14.inria.fr.1096 > pinta15.inria.fr.1050: . 36649:38109(1460) ack 1 win 16060 (DF)
17:05:47.256966 pinta14.inria.fr.1096 > pinta15.inria.fr.1050: . 38109:39569(1460) ack 1 win 16060 (DF)

17:05:47.256966 pinta15.inria.fr.1050 > pinta14.inria.fr.1096: . ack 38109 win 5840 (DF)
17:05:47.276966 pinta15.inria.fr.1050 > pinta14.inria.fr.1096: . ack 39569 win 4380 (DF)

17:05:47.276966 pinta14.inria.fr.1096 > pinta15.inria.fr.1050: P 39569:40717(1148) ack 1 win 16060 (DF)

17:05:47.296966 pinta15.inria.fr.1050 > pinta14.inria.fr.1096: . ack 40717 win 3232 (DF)
17:05:47.746966 pinta15.inria.fr.1050 > pinta14.inria.fr.1096: . ack 40717 win 16060 (DF)
17:05:47.766966 pinta15.inria.fr.1050 > pinta14.inria.fr.1096: P 1:33(32) ack 40717 win 16060 (DF)
17:05:47.766966 pinta15.inria.fr.1050 > pinta14.inria.fr.1096: P 33:113(80) ack 40717 win 16060 (DF)

17:05:47.766966 pinta14.inria.fr.1096 > pinta15.inria.fr.1050: P 40717:40749(32) ack 113 win 16060 (DF)
17:05:47.766966 pinta14.inria.fr.1096 > pinta15.inria.fr.1050: P 40749:40901(152) ack 113 win 16060 (DF)
17:05:47.776966 pinta14.inria.fr.1096 > pinta15.inria.fr.1050: P 40901:40933(32) ack 113 win 16060 (DF)
17:05:47.776966 pinta14.inria.fr.1096 > pinta15.inria.fr.1050: . 40933:42393(1460) ack 113 win 16060 (DF)
17:05:47.776966 pinta14.inria.fr.1096 > pinta15.inria.fr.1050: . 42393:43853(1460) ack 113 win 16060 (DF)
17:05:47.776966 pinta14.inria.fr.1096 > pinta15.inria.fr.1050: . 43853:45313(1460) ack 113 win 16060 (DF)
17:05:47.776966 pinta14.inria.fr.1096 > pinta15.inria.fr.1050: . 45313:46773(1460) ack 113 win 16060 (DF)
17:05:47.776966 pinta14.inria.fr.1096 > pinta15.inria.fr.1050: . 46773:48233(1460) ack 113 win 16060 (DF)
17:05:47.776966 pinta14.inria.fr.1096 > pinta15.inria.fr.1050: . 48233:49693(1460) ack 113 win 16060 (DF)
17:05:47.776966 pinta14.inria.fr.1096 > pinta15.inria.fr.1050: . 49693:51153(1460) ack 113 win 16060 (DF)
17:05:47.776966 pinta14.inria.fr.1096 > pinta15.inria.fr.1050: . 51153:52613(1460) ack 113 win 16060 (DF)

17:05:47.786966 pinta15.inria.fr.1050 > pinta14.inria.fr.1096: . ack 49693 win 16060 (DF)

17:05:47.786966 pinta14.inria.fr.1096 > pinta15.inria.fr.1050: . 52613:54073(1460) ack 113 win 16060
17:05:47.786966 pinta14.inria.fr.1096 > pinta15.inria.fr.1050: . 54073:55533(1460) ack 113 win 16060
17:05:47.786966 pinta14.inria.fr.1096 > pinta15.inria.fr.1050: . 55533:56993(1460) ack 113 win 16060
17:05:47.786966 pinta14.inria.fr.1096 > pinta15.inria.fr.1050: . 56993:58453(1460) ack 113 win 16060
17:05:47.786966 pinta14.inria.fr.1096 > pinta15.inria.fr.1050: . 58453:59913(1460) ack 113 win 16060
17:05:47.786966 pinta14.inria.fr.1096 > pinta15.inria.fr.1050: P 59913:61093(1180) ack 113 win 16060 (DF)
17:05:47.786966 pinta14.inria.fr.1096 > pinta15.inria.fr.1050: . 61093:62553(1460) ack 113 win 16060 (DF)
17:05:47.786966 pinta14.inria.fr.1096 > pinta15.inria.fr.1050: . 62553:64013(1460) ack 113 win 16060 (DF)
17:05:47.786966 pinta14.inria.fr.1096 > pinta15.inria.fr.1050: . 64013:65473(1460) ack 113 win 16060 (DF)
```

```
17:05:47.786966 pinta15.inria.fr.1050 > pinta14.inria.fr.1096: . ack 52613 win 16060 (DF)

17:05:47.786966 pinta14.inria.fr.1096 > pinta15.inria.fr.1050: . 65473:66933(1460) ack 113 win 16060
17:05:47.786966 pinta14.inria.fr.1096 > pinta15.inria.fr.1050: . 66933:68393(1460) ack 113 win 16060

17:05:47.796966 pinta15.inria.fr.1050 > pinta14.inria.fr.1096: . ack 64013 win 13140 (DF)
```

Bibliographie

[Assenmacher 93a] H. Assenmacher, T. Breitbach, P. Buhler, V. Hübsch et R. Schwarz. PANDA — Supporting Distributed Programming in C++. Dans *Proceedings of the 7th European Conference on Object Oriented Programming (ECOOP)*, volume 707 de *Lecture Notes in Computer Science*, pages 361–383, Germany, Juillet 1993. Springer-Verlag.

[Assenmacher 93b] H. Assenmacher, T. Breitbach, P. Buhler, V. Hübsch et R. Schwarz. The PANDA System Architecture — A Pico-Kernel Approach. Dans *Proceedings of the Fourth Workshop on Future Trends of Distributed Computing Systems (FTDCS'93)*, pages 470–476, Lisbonne, Portugal, 1993. IEEE Computer Society Press.

[Attali 95] Isabelle Attali, Denis Caromel et Sidi Ould Ehmety. *An Operational Semantics for the Eiffel// Language*. Rapport de Recherche RR-2732, INRIA Sophia Antipolis, France, Novembre 1995.

[Aydt 96] Ruth A. Aydt. *The Pablo Self-Defining Data Format*. Departement of Computer Science, University of Illinois, Urbana, Illinois 61801, USA, Septembre 1996. `http://www-pablo.cs.uiuc.edu/Projects/Pablo/documents.html`.

[Bal 90] H. E. Bal, M. F. Kaashoek et A. S. Tannenbaum. Experience with distributed programming in Orca. Dans *Proceedings of the International Conference on Computer Languages*, pages 77–89, New-Orleans, LA, USA, Mars 1990. IEEE Computer Society Press.

[Bangalore 94] P.V. Bangalore, N. E. Doss et A. Skjellum. MPI++ : Issues and Features. Dans *Proceedings of the Object Oriented Numerics Conference (OONSKI'94)*, pages 465–472, Janvier 1994. `ftp://aurora.cs.msstate.edu/pub/reports/Message-Passing/oon-ski94.ps.Z`.

[Barette 96] Philippe Barette et Cyril Tonin. Implémentation d'une librairie de communications pour un langage C++ parallèle au dessus de PVM et d'une bibliothèque de threads. Projet de 3ème année, ESSI – UNSA, Sophia Antipolis, France, Avril 1996. Encadreurs : SLOOP.

[Baude 94] Françoise Baude, Nathalie Furmento et Daniel Lafaye de Micheaux. Managing true parallelism in ADA through PVM. Dans *Proceedings of the 1st European* PVM *Users' Group Meeting (EuroPVM'94)*, Rome, Italy, Octobre 1994. Also : Research rapport I3S RR94-62. `http://www.netlib.org/pvm3/epvmug94/`.

[Baude 96a] Françoise Baude, Fabrice Belloncle, Denis Caromel, Nathalie Furmento, Philippe Mussi, Yves Roudier et Günther Siegel. Parallel Object-Oriented Programming for Parallel Simulations. *Information Sciences : An International Journal*, 93 : 35–64, Août 1996. Special Issue 1-2.

[Baude 96b] Françoise Baude et Olivier Dalle. *Analyse des performances de communication du protocole* PVM. Rapport de Recherche RR96-08, I3S, Sophia Antipolis, France, Mars 1996.

[Baude 96c] Françoise Baude, Nathalie Furmento, Denis Caromel, Raymond Namyst, Jean-Marc Geib et Jean-François Méhaut. C++// on top of PM^2 via SCHOONER. Dans *Proceedings of Stratagem'96*, pages 41–55, Sophia Antipolis, France, Juillet 1996. ISBN-2-7261-0982-9.

[Baude 99] Françoise Baude, Denis Caromel, Nathalie Furmento et David Sagnol. Overlapping Communication with Computation in Distributed Object Systems. Dans Peter Sloot, Marian Bubak, Alfons Hoekstra et Bob Hertzberger, éditeurs, *Proceedings of the 7th International Conference - High Performance Computing Networking'99 (HPCN Europe 1999)*, volume 1593 de *Lecture Notes in Computer Science*, pages 744–753, Amsterdam, The Netherlands, Avril 1999.

[Birman 85] Kenneth P. Birman, Amr El Abbadi, Wally Dietrich, Thomas A. Joseph et Thomas Raeuchle. An Overview of the ISIS Project. *IEEE Distributed Processing Technical Committee Newsletter*, Janvier 1985. Also available as technical report TR 84-642 from Indiana University.

[Birrell 84] Andrew D. Birrell et Bruce Jay Nelson. Implementing Remote Procedure Calls. *ACM Transactions on Computer Systems*, 2(1) : 39–59, Février 1984.

[Bodin 93a] F. Bodin, P. Beckman, D. Gannon, S. Naranaya et S. X. Yang. Distributed pC++ : Basic Ideas for an Object Parallel Language. *Scientific Programming*, 2(3), 1993.

[Bodin 93b] F. Bodin, P. Beckman, D. Gannon, S. Yang, S. Kesavan et A. Malony a nd B. Mohr. Implementing a Parallel C++ Runtime System for Scalable Parallel S ystems. Dans *Proceedings of the Supercomputing'93 Conference*, Portland, Oregon, Novembre 1993.

[Bougé 98] Luc Bougé, Jean-François Méhaut et Raymond Namyst. Madeleine : an efficient and portable communication interface for multithreaded environ-

ments. Dans *Proceedings of the 1998 International Conference 'Parallel Architectures and Compilation Techniques' (PACT'98)*, ENST, Paris, France, 1998. Also available as a LIP Research Report RR98-26.

[Brandes 96] T. Brandes et Frédéric Desprez. Implementing Pipelined Computation and Communication in an HPF Compiler. Dans *Proceedings of the 2nd European conference on Parallel computing (Euro-Par'96)*, volume J, pages 459–462, Août 1996.

[Briot 96] Jean-Pierre Briot et Rachid Guerraoui. Objets pour la programmation parallèle et répartie : Intêrets, évolutions et tendances. *Techniques et Sciences Informatiques (TSI)*, Hermés, Juin 1996.

[Burns 94] Gregory D. Burns, Raja B. Daoud et James R. Vaigl. LAM : An Open Cluster Environment for MPI. Dans *Proceedings of Supercomputing Symposium '94*, Toronto, Canada, Juin 1994.

[Caromel 91] Denis Caromel. *Programmation parallèle asynchrone et impérative : études et propositions.* PhD thesis, Université de Nancy I, France, Février 1991.

[Caromel 96] Denis Caromel, Fabrice Belloncle et Yves Roudier. *Parallel Programming Using C++*, chapitre The C++// System, pages 257–296. MIT Press, 1996. ISBN 0-262-73118-5.

[Caromel 97] D. Caromel, A. McEwan, J. Nolte, J. Poole, Y. Roudier, D. Sagnol, J.-M. Challier, P. Dzwig, R. Kaufman, H. Liddell, P. Mussi, D. Parkinson, M. Rigg, G. Roberts et R. Winder. *EUROPA Parallel C++.* `http://www.inria.fr/sloop/europa/`, Septembre 1997. The EUROPA Working Group on Parallel C++, Final report, HPCN Esprit Contract No 9502.

[Carriero 89] N. Carriero et D. Gelernter. Linda in context. *Communications of the ACM*, 32(4) : 444–458, Avril 1989.

[Clavaud 97] Alexandre Clavaud. Interface graphique pour la visualisation d'exécution d'applications distribuées. Projet de 3ème année, ESSI – UNSA, Sophia-Antipolis, France, Avril 1997. Encadreurs : SLOOP.

[Couland 95a] Olivier Couland et Eric Dillon. Para++ : C++ bindings for message passing libraries. Dans J. Dongarra, M. Gengler, B. Tourancheau et X. Vigouroux, éditeurs, *Proceedings of the 2nd European* PVM *Users' Group Meeting (EuroPVM'95)*, volume 5 de *Parallélisme, réseaux et répartition*, pages 167–172, Lyon, France, Septembre 1995. Hermés, Paris.

[Couland 95b] Olivier Couland et Eric Dillon. *PARA++ : C++ Bindings for Message Passing Libraries : User Guide.* Rapport technique RT-174, INRIA Lorraine, France, Juin 1995.

[Crane 93] S. Crane. *The* REX *lightweight process library.* Rapport technique, Imperial College of Science and Technology, London, England, 1993. `ftp: gummo.doc.ic.ac.uk`.

[Dalle 99] Olivier Dalle. *Techniques et outils pour les communications et la répartition dynamique de charge dans les réseaux de stations de travail.* PhD thesis, Université de Nice - Sophia Antipolis, France, Janvier 1999.

[Demoustier 98] Christian Demoustier. Asynchronisme et Continuations Automatiques dans les Langages à Objets Parallèles et Distribués. Stage de DEA Réseaux et Systèmes Distribués, ESSI – UNSA, Sophia Antipolis, France, Juin 1998. Encadreurs : SLOOP.

[Desprez 94a] Frédéric Desprez. *Procédures de Base pour le Calcul Scientifique sur Machines Parallèles à Mémoire Distribuée.* PhD thesis, Ecole Normale Supérieure de Lyon, LIP, Lyon, France, Janvier 1994.

[Desprez 94b] Frédéric Desprez et Bernard Tourancheau. LOCCS : Low Overhead Communication and Computation Subroutines. Dans *Future Generation Computer Systems*, volume 10, pages 279–284, Juin 1994.

[Desprez 96] Frédéric Desprez, Pierre Ramet et Jean Roman. Optimal Grain Size Computation for Pipelined Algorithms. Dans *Proceedings of the 2nd European conference on Parallel computing (Euro-Par'96)*, volume T, pages 165–172, Août 1996.

[Dolev 96] Danny Dolev et Dalia Malki. The Transis Approach to High Availability Cluster Communication. *Communications of the ACM*, 39(4), Avril 1996.

[Fagg 96] Graham E. Fagg et Jack J. Dongarra. PVMPI : an integration of the PVM and MPI systems. *Calculateurs parallèles*, 8(2) : 151–166, 1996.

[Ferrari 95] Adam Ferrari et V.S. Sunderam. TPVM : Distributed Concurrent Computing with Lightweight Processes. Dans *Proceedings of the Fourth IEEE International Symposium on High Performance Distributed Computing*, pages 211–218, Washington, DC, USA, Août 1995. IEEE Computer Society Press. `http://uvacs.cs.virginia.edu/~ajf2j/docs/tpvm_paper.ps`.

[Foster 94] Ian Foster, Carl Kesselman et Steven Tuecke. *Nexus : Runtime Support for Task-Parallel Programming Languages.* Rapport technique, Mathematics and Computer Science Division, Argonne National Laboratory, USA, 1994.

[Foster 96] Ian Foster, Carl Kesselman et Steven Tuecke. The Nexus Approach to Integrating Multithreading and Communication. *Journal of Parallel and Distributed Computing*, 37 : 70–82, 1996.

[Gamel 96] Sylvain Gamel. Transmission efficace d'objets pour le langage C++//. Stage de DEA Informatique, ESSI – UNSA, Sophia Antipolis, France, Juillet 1996. Encadreurs : SLOOP.

[Geib 97] Jean-Marc Geib, Christophe Gransart et Philippe Merle. *CORBA : des concepts à la pratique.* Collections InterEditions, Editions MASSON, Paris, France, 1997. ISBN : 2-225-83046-0.

[Geist 94] A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Mancheck et V. Sunderam. PVM *Parallel Virtual Machine : a user's guide and tutorial for networked parallel computing.* MIT Press, 1994.

[Geist 98] G.A. Geist, J.A. Kohl, P.M. Papadopoulos et S.L. Scott. *Beyond* PVM *3.4 : What We've Learned, What's Next and Why.* `http://www.epm.ornl.gov/harness/`, 1998.

[Gioanni 96] Frédéric Gioanni. Mécanismes de traces dans PROSIT. Projet de 3ème année, ESSI – UNSA, Sophia Antipolis, France, Avril 1996. Encadreurs : SLOOP.

[Gransart 95] Christophe Gransart. *BOX : Un Modèle et un Langage à Objets pour la Programmation Parallèle et Distribuée.* PhD thesis, Université des Sciences et Technologies de Lille, U.F.R d'I.E.E.A, Bât M3, 59665 Villeneuve d'Ascq CEDEX, France, Janvier 1995.

[Gropp 96] William Gropp, Ewing Lusk, Nathan Doss et Anthony Skjellum. A high-performance, portable implementation of the MPI message passing interface standard. *Parallel Computing*, 22(6) : 789–828, Septembre 1996.

[Guerraoui 98] Rachid Guerraoui, Pascal Felber, Benoit Garbinato et Karim Mazouni. System Support for Objects Groups. Dans *Proceedings of the ACM Conference on Object Oriented Programming Systems, Languages and Applications (OOPSLA'98)*, Vancouver, Canada, Octobre 1998.

[Heath 91] M.T. Heath et J.A. Etheridge. Visualizing the performance of parallel programs. *IEEE Software*, 8(5) : 29–39, Septembre 1991.

[Hoare 78] C.A.R Hoare. Communicating Sequential Processes. *Communications of the ACM*, 21(8) : 666–677, Août 1978.

[IEEE 96] IEEE. *Information Technology — Portable Operating System Interface (*POSIX *) — Part 1 : System Application : Program Interface (API) [C Language].* Institute of Electrical and Electronical Engineers, This edition incorporates extensions for realtime applications (1003.1b-1993, 1003.1i-1995) and threads (1003.1c-1995), 9945-1 (ISO/IEC) [IEEE/ANSI Std 1003.1] 1996.

[Kalé 93] L. V. Kalé et S. Krishnan. CHARM++ : A Portable Concurrent Object Oriented System Based on C++. Dans Andreas Paepcke, éditeur, *Proceedings of the 8th Annual Conference on Object-Oriented Programming Systems, Languages and Application (OOPSLA'93)*, volume 28, pages 91–108. ACM Press, Septembre 1993. Also : Technical Report

UIUCDCS-R-93-1796, March 1993, University of Illinois, Urbana, IL. `http://charm.cs.uiuc.edu/`.

[Kale 96a] Laxmikant V. Kale, Milind Bhandarkar, Narain Jagathesan et Sanjeev Krishnan andJoshua Yelon. Converse : An Interoperable Framework for Parallel Programming. Dans *Proceedings of the International Parallel Processing Symposium*, Honolulu, Hawaii, Avril 1996.

[Kale 96b] Laxmikant V. Kale, Joshua M. Yelon et T. Knauff. Threads for Interoperable Parallel Programming. Dans *Proceedings of the conference on Languages and Compilers for Parallel Computing*, 1996.

[Kale 98] Laxmikant V. Kale, Milind Bhandarkar, Robert Brunner et Joshua Yelon. Multiparadigm, Multilingual Interoperability : Experience with Converse. Dans *Proceedings of the 2nd Workshop on Runtime Systems for Parallel Programming (RTSPP)*, Orlando, Florida, USA, Mars 1998.

[Kraemer 93] Eileen Kraemer et John T. Stasko. The Visualization of Parallel Systems : An Overview. *Journal of Parallel and Distributed Computing*, 18(18) : 105–117, Juin 1993.

[Krone 95] Oliver Krone, Marc Aguilar et Béat Hirsbrunner. PT-PVM : Using PVM in a multi-threaded environment. Dans J. Dongarra, M. Gengler, B. Tourancheau et X. Vigouroux, éditeurs, *Proceedings of the 2nd European* PVM *Users' Group Meeting (EuroPVM'95)*, volume 5 de *Parallélisme, Réseaux et Répartition*, pages 83–88, Lyon, France, Septembre 1995. Hermés, Paris.

[Krone 96] Olivier Krone, Béat Hirsbrunner et Vaidy Sunderam. PT-PVM$^+$: A Portable platform for multithreaded coordination languages. *Calculateurs Parallèles*, 8(2) : 167–182, 1996.

[Lopes 96] Christina Videira Lopes. Adaptive Parameter Passing. Dans *Proceedings of the 2nd International Symposium on Object Technologies for Advanced Software (ISOTAS'96)*, JSSST International Symposium Series, Kanazawa, Japan, Mars 1996.

[Makpangou 91] Mesaac Makpangou, Yvon Gourhant et Marc Shapiro. BOAR : A Library of Fragmented Object Types for Distributed Applications. Dans *Proceedings of the International Workshop on Object-Orientation in Operating Systems (I-WOOOS)*, Palo Alto, CA, USA, Octobre 1991.

[Makpangou 94] Mesaac Makpangou, Yvon Gourhant, Jean-Pierre Le Narzul et Marc Shapiro. Fragmented Objects for Distributed Abstractions. Dans Casavant et M. Singhal, éditeurs, *Readings in Distributed Computing Systems*. IEEE Computer Society Press, 1994.

[Malard 96] Joël Malard. *MPI : A Message-Passing Interface Standard*. Technology Watch Report Version 1.1, Edinburgh Parallel Computing Centre, 1996. `http://www.epcc.ed.ac.uk/epcc-tec/documents/techwatch.html`.

[Malony 94] A. Malony, B. Mohr, P. Beckman, D. Gannon, S. Yang et F. Bodin. Performance Analysis of pC++ : A Portable Data-Parallel Programming System for Scalable Parallel Computers. Dans *Proceedings of the 8th International Parallel Processing Symbosium (IPP S)*, Cancún, Mexico, Avril 1994.

[Mattern 87] Friedemann Mattern. Algorithms for Distributed Termination Detection. *Distributed Computing*, 2(3) : 161–175, 1987.

[Mattern 93] Friedemann Mattern. Distributed Control Algorithms (Selected Topics). Dans *Ecole d'Informatique sur l'algorithmique répartie : Calculs répartis et causalité*, Roscoff, France, Septembre 1993. Also in *Parallel Computing on Distributed Memory Multiprocessors*, Springer Verlag, 1993, pp 167-185.

[Mes 94] Message Passing Interface Forum. *Document for a Standard Message-Passing Interface*, Février 1994.

[Microsystems 96] Sun Microsystems. *Java RMI Tutorial*, Novembre 1996. `http://java.sun.com/products/jdk/1.1/docs/guide/rmi/`.

[Microsystems 98] Sun Microsystems. *Java RMI Tutorial*, Octobre 1998. `http://java.sun.com/products/jdk/1.2/docs/guide/rmi/`.

[Monteil 95] Thierry Monteil, Jean-Marie Garcia et Pierre Guyaux. LANDA : Une machine virtuelle parallèle. *Calculateurs Parallèles — Environnements d'Exécution de Programmes Parallèles*, 7(2) : 119–137, 1995.

[Mueller 95] Frank Mueller. *Pthreads Library Interface.* Rapport technique, Florida State University, Department of Computer Science, USA, Juillet 1995.

[Namyst 95] Raymond Namyst et Jean-François Méhaut. PM^2 : Parallel Multithreaded Machine. A computing environment for distributed architectures. Dans *Proceedings of the International Conference on Parallel Computing (ParCo'95)*, Gent, Belgium, Septembre 1995.

[Namyst 97] Raymond Namyst. *PM^2 : un environnement pour une conception portable et une exécution efficace des applications parallèles irrégulières.* PhD thesis, Université des Sciences et Technologies de Lille, U.F.R d'I.E.E.A, 59655 Villeneuve d'Ascq, Lille, France, Janvier 1997.

[Namyst 98a] Raymond Namyst, Yves Denneulin, Jean-Marc Geib et Jean-François Méhaut. Utilisation des processus légers pour le calcul parallèle distribué : l'approche PM^2. *Lettre des Calculateurs Parallèles*, 1998. A paraître.

[Namyst 98b] Raymond Namyst et Jean-François Méhaut. Madeleine : Une interface de communications efficace pour les environnements multithreads. Dans *10es Rencontres Francophones du parallélisme (RenPar'10)*, Strasbourg, France, Juin 1998.

[PVM Team 98] The PVM Team. PVM *: Parallel Virtual Machine.* `http://www.epm.orln.gov/pvm/pvm_home.html`, 1998.

[Roudier 96] Yves Roudier. *Abstractions réactives pour les langages à objets paralléles : modèles et programmation.* PhD thesis, Université de Nice - Sophia Antipolis, France, Décembre 1996.

[Shapiro 86] Marc Shapiro. Structure and encapsulation in distributed systems : The proxy principle. Dans *Proceedings of the 6th International Conference on Distributed Computing Systems*, pages 198–204, Cambridge, Massachusetts, USA, Mai 1986. IEEE Computer Society.

[Shapiro 90] Marc Shapiro, Yvon Gourhant, Sabine Habert, Jean-Pierre Le Narzul, Laurence Mosseri, Michel Ruffin et Céline Valot. *Un bilan du système réparti à objets SOS.* Rapport technique RT-1242, INRIA Rocquencourt, France, Juin 1990.

[Shapiro 92] Marc Shapiro, Peter Dickman et David Plainfossé. *SSP Chains : Robust. Distributed References Supporting Acyclic Garbage Collection.* Rapport de recherche RR-1799, INRIA Rocquencourt, France, Novembre 1992.

[Siegel 97] Günther Siegel. PROSIT *: Un Environnement pour la programmation de simulations à événements discrets.* PhD thesis, Université de Nice - Sophia Antipolis, France, Septembre 1997.

[Stevens 90] W. Richard Stevens. *UNIX Network Programming.* Prentice Hall Software Series, 1990. ISBN 0-13-949876-1.

[Stevens 92] W. Richard Stevens. *Advanced Programming in the UNIX Environment.* Addison-Wesley Publishing Company, 1992.

[Stratagème 97] Stratagème. *Une méthodologie de programmation parallèle pour les problèmes non structurés.* Final rapport 97/004, PRiSM Laboratory, 45, avenue des Etats-Unis – 78035 Versailles Cedex – France, Janvier 1997.

[Stroustrup 97] Bjarne Stroustrup. *The C++ Programming Language.* Addison-Wesley, AT&T Bell Laboratories, 3rd edition, Septembre 1997.

[Sunderam 90] V. S. Sunderam. PVM : A Framework for Parallel Distributed Computing. *Concurrency - Practice and Experience*, 2(4) : 315–339, Décembre 1990.

[Tanzy 97] Jane-Elise Tanzy. Répartition dynamique de charge dans les réseaux de stations de travail. Rapport de thèse professionnelle et de DEA, UNSA-EURECOM, Sophia Antipolis, France, Août 1997. Encadreurs : SLOOP.

[Tseng 93] C.W. Tseng. *An Optimizing Fortran D Compiler for MIMD Distributed-Memory Machines.* PhD thesis, Rice University, USA, Janvier 1993.

[van Renesse 96] Robbert van Renesse, Kenneth P. Birman et Silvano Maffeis. Horus : A Flexible Group Communications System. *Coomunications of the ACM*, 39(4), Avril 1996.

[Vigouroux 96] Xavier Vigouroux. *Analyse Distribuée de Traces d'Exécution de Programmes Parallèles.* PhD thesis, Ecole Normale Supérieure de Lyon, LIP, Lyon, France, Janvier 1996.

[von Eicken 92] Thorsten von Eicken, David E. Culler, Seth C. Goldstein et Klaus E. Schauser. Active Messages : a Mechanism for Integrated Communication and Computation. Dans *Proceedings of the 19th International Symposium on Computer Architectures*, pages 256–267, Gold Coast, Australia, Mai 1992. ACM Press.

[Worley 92] Patrick H. Worley. *A New PICL Trace File Format.* Rapport technique ORNL/TM-12125, Oak Ridge National Laboratory, P.O. Box 2008, Bldg. 6012 — Oak Ridge, TN 37831-6367, USA, Septembre 1992.

Liste des figures

Liste des tableaux

Liste des exemples

Schooner : Une Encapsulation Orientée Objet de Supports d'Exécution pour Applications Réparties

Le sujet de cette thèse est la conception d'un support d'exécution orienté objet pour applications réparties. Un des principaux objectifs est de permettre de correctement isoler le code lié à la gestion du support d'exécution du code propre à l'application. D'autre part, un tel support se doit d'être portable sur le plus grand nombre de plate-formes ; pour cela son interface de programmation doit être minimale tout en restant extensible. La prise en compte de tels critères permet d'obtenir un support pour une grande variété d'applications réparties.

Nous avons donc conçu et implémenté un support d'exécution sous la forme d'une bibliothèque de classes appelée Schooner. Le modèle de programmation de la bibliothèque s'articule autour des notions de machine virtuelle et d'entités réparties communiquant par messages actifs. En plus de ce modèle de base, nous avons également développé une extension multi-active permettant l'utilisation de processus légers. Afin de compléter et valider Schooner, des outils d'aide au développement et de mise au point d'applications réparties sont également fournis.

Une des caractéristiques importantes de l'environnement fourni est d'être facilement personnalisable selon les besoins spécifiques d'une application en permettant par exemple une amélioration des performances. Dans ce contexte, nous proposons un mécanisme de bufferisation des messages entre deux entités communicantes, mécanisme entièrement paramétrable par l'utilisateur. Il est également envisageable dans le cadre de la version multi-active de modifier l'ordonnancement des entités actives.

Cet environnement a pu être validé par le développement de deux applications de taille conséquente : la version répartie d'un simulateur à événements discrets orienté objet, Prosit et une extension répartie et parallèle du langage C++, C++//.

Mots clés : Répartition – Parallélisme – Support d'exécution – Support de communication – Programmation orientée objet – Portabilité.

Schooner : An Object-Oriented Runtime Support for Distributed Applications

In this thesis, we focus on the design of an object-oriented runtime support for distributed applications. One of the main goals is to make a clear distinction between the managment of the runtime support and the application itself. On the other hand, such a runtime support has to be portable to a wide number of platforms; in this respect, the programming interface has to be minimal while still being extensible.

The design and implementation of a C++ library called Schooner is the main result of our work. Its programming model rests on the concepts of virtual computers and distributed entities communicating via active messages. We also provide a concurrent extension of this basic model which allows the use of lightweight processes. In order to validate the Schooner applications, debugging tools have also been built.

One of the main features of our environment is that it is an open system: the user can choose to modify the behaviour of the runtime components according to the specific needs of an application in order to improve the performance. For example, the policies for message buffering between any two entities is completely under the control of the application. Likewise, in the concurrent version of Schooner, the scheduling policies can be controlled from the application.

The Schooner environment has been validated through the development of 2 large applications: the distributed version of Prosit, an object-oriented environment for discrete event simulation programming, and C++//, a distributed and parallel extension of the C++ language.

Keywords: Distribution - Parallelism - Runtime support - Communication Support - Object-Oriented programming - Portability.