Conception et évaluation de performance d'un Bus applicatif, massivement parallèle et orienté service

Ridha Mohammed Benosman

CONSERVATOIRE NATIONAL DES ARTS ET MÉTIERS | le cnam

École Doctorale Informatique, Télécommunications et Électronique

Centre d'étude et de recherche en informatique et communications (EA 4629)

THÈSE DE DOCTORAT

présentée par : **Ridha Mohammed BENOSMAN**

soutenue le : **12 Décembre 2013**

pour obtenir le grade de : **Docteur du Conservatoire National des Arts et Métiers**

Discipline / Spécialité : **Informatique / Systèmes distribués**

Conception et Evaluation de Performance d'un Bus Applicatif, Massivement Parallèle et Orienté Services

THÈSE DIRIGÉE PAR

M. BARKAOUI Kamel — *Professeur, CNAM (Cedric)*

RAPPORTEURS

M. DJOUANI Karim — *Professeur, Université Paris-Est Créteil (LISSI)*

M. MOREAUX Patrice — *Professeur, Polytech Annecy-Chambery- Université de Savoie (LISTIC)*

EXAMINATEURS

M. FOLLIOT Bertil — *Professeur, Université Pierre et Marie Curie (LIP6)*

M. HAMMAMI Omar — *Professeur, ENSTA ParisTech (LEI)*

Mme.BOUMERDASSI Selma — *Maître de conférences CNAM (Cedric)*

Remerciements

C'est avec beaucoup de mélancolie que je vais devoir aujourd'hui quitter le rythme de vie auquel je me suis bien habitué pour mettre un point final à ce travail laborieux qui m 'a beaucoup passionné. Parfaitement bien entouré je revois devant moi toutes les personnes qui ont été très présentes à mes cotés dans les moments difficiles et que je tiens à remercier.

Tout d'abord, je remercie infiniment et tout particulièrement Pr. Kamel BARKAOUI qui a eu la grande amabilité de m'accueillir au sein de l'équipe de recherche VESPA pour m'offrir la chance de travailler sur un sujet tout aussi passionnant que prometteur d'innovations. Je lui suis très reconnaissant d'avoir toujours été à l'écoute de mes interrogations avec un suivi substantiel accompagné de beaucoup d'encouragements : autant de conditions de travail favorables à une bonne progression de ma recherche.

Je veux demander très respectueusement, au Conservatoire National des Arts et Métiers, d'accepter ici le témoignage de ma profonde gratitude pour avoir ouvert devant moi les portes de son Laboratoire de Recherche et me donner, ainsi, l'opportunité de poursuivre mes études de recherche.

je dois reconnaître que tous les remerciements, aussi grands soient-ils que je puisse adresser au Pr. Karim DJOUANI et au Pr. Patrice MOREAUX ne sauraient être à la hauteur de l'honneur qu'ils m'ont réservé pour examiner ma thèse.

C'est avec une grande marque d'estime que tiens à remercier M. Yves ALBRIEUX et M. Jean-Luc DOU d'avoir participé à l'enrichissement de ma thèse par leurs remarques constructives ainsi que pour leurs rapports encouragements avec en perspective encore et toujours la recherche scientifique.

A tous mes collègues stagiaires, doctorants et post-doctorants, je veux dire que je garderai ancrées dans mon cœur leur aimable compagnie, les recréations et l'atmosphère de détente qui nous réunissaient après les longues heures de travail.

Je remercie vivement ma famille de m'avoir toujours aidé et encouragé à suivre la voie que je me suis tracée et à concrétiser le choix pour lequel j'ai opté.

Enfin, je dis mille fois merci à tous ceux qui ont eu confiance en moi et auxquels je compte bien donner entière satisfaction ... Merci de tout cœur à tous.

le cnam

Ridha BENOSMAN

Conception et Evaluation de Performance d'un Bus Applicatif, Massivement Parallèle et Orienté Services

le cnam

Résumé :
Enterprise Service Bus (ESB) est actuellement l'approche la plus prometteuse pour l'implémentation d'une architecture orientée services (SOA : Service-Oriented Architecture) par l'intégration des différentes applications isolées dans une plateforme décentralisée.
De nombreuses solutions d'intégration à base d'ESB on été proposées, elles sont soit open-source comme : Mule, Petals, ou encore Fuse, soit propriétaires tels que : Sonic ESB, IBM WebSphere Message Broker, ou Oracle ESB. Cependant, il n'en existe aucune en mesure de traiter, à la fois des aspects : d'intégration et de traitement massivement parallèle, du moins à notre connaissance. L'intégration du parallélisme dans le traitement est un moyen de tirer profit des technologies multicœurs/multiprocesseurs qui améliorent considérablement les performances des ESBs.
Toutefois, cette intégration est une démarche complexe et soulève des problèmes à plusieurs niveaux : communication, synchronisation, partage de données, etc.
Dans cette thèse, nous présentons l'étude d'une nouvelle architecture massivement parallèle de type ESB.
Mots clés :
Bus de services d'entreprise (Enterprise Service Bus : ESB) ; Architecture orientée services ; Traitement massivement parallèle ; Multithreading ; Systèmes distribués ; Communication inter-processus (IPC).

Abstract :
Enterprise service bus (ESB) is currently the most promising approach for business application integration in distributed and heterogeneous environments. It allows to deploy a service-oriented architecture (SOA) by the integration of all the isolated applications on a decentralized platform.
Several commercial or open source ESB-based solutions have been proposed. However, to the best of our knowledge, none of these solutions has integrated the parallel processing. The integration of parallelism in the treatment allows to take advantage of the multicore/multiprocessor technologies and thus can improve greatly the ESB performance. However, this integration is difficult to achieve, and poses problems at multiple levels (communication, synchronization, etc). In this study, we present a new massively parallel ESB architecture that meets this challenge.
Keywords :
Enterprise Service Bus (ESB) ; Service-oriented architecture (SOA) ; Massively parallel processing ; Multithreading ; Distributed and delay insensitive applications ; Inter-process communication (IPC).

Table des matières

Introduction générale

Après une phase de miniaturisation qui touche aux limites des règles de la physique (chaleur, consommation électrique, ...), la puissance des processeurs ne suit plus son rythme habituel de croissance et a tendance à stagner. Pour affronter cette immobilité, les constructeurs ont choisi de multiplier, non seulement le nombre de processeurs sur la carte mère (architecture multiprocesseurs), mais aussi celui des cœurs au sein même du processeur (architecture multicœurs).

Ainsi, la seule issue actuellement en vue pour gagner en puissance de traitement est le recours au parallélisme. Ce dernier étant un concept connu depuis bien longtemps, et dont les origines remontent aux années 70 avec l'apparition de la première génération des supercalculateurs parallèles. L'informatique actuelle, quant à elle, recherche la puissance dans un traitement parallèle de plus en plus massif.

Bien que prévisible depuis plusieurs années, cette évolution électronique est devenue actuellement une véritable révolution à tel point qu'elle a radicalement changé le paysage du monde logiciel. En effet, si le parallélisme a été réservé à l'origine aux calculs scientifiques nécessitant une grande capacité de traitement, notamment dans le domaine du calcul à haute performance (High Performance Computer : HPC), son spectre s'est peu à peu étendu en touchant sur son passage, une bonne partie du secteur des logiciels « grand public ».

Si le monde du HPC est bien au point avec l'utilisation du parallélisme, celui de la gestion, en particulier dans les architectures d'intégration orientées services : ESB (Enterprise Service Bus), et EAI (Enterprise Application Integration) continue de piétiner. Il nécessite la mise en œuvre de nouvelles façons de procéder, et de nouveaux paradigmes adaptées à son contexte. Un contexte qui appelle à la coopération, la collaboration et la synergie entre des applications distribuées et hétérogènes, qui à la base n'ont pas été conçues pour fonctionner ensemble, d'où l'insuffisance des outils (plateformes, bibliothèques, ...) classiquement utilisés en HPC.

La communication, aspect crucial de toute architecture orientée services, est la première à être impactée par cette reconversion en traitement massivement parallèle. En effet, les échanges entre applications passent obligatoirement par un réseau de communication standardisé, un véritable entonnoir qui impose une mise en série du flux de données. Cette source inévitable de sérialisation se traduit par des goulots d'étranglement qui entravent la performance du système, ce qui va à l'encontre du parallélisme tant cherché. A cela, s'ajoutent des problèmes d'intégration de tâches parallèles, de synchronisation, de cohérence et de partage de données, de montée en charge, etc.

Ainsi, il en est conclu la nécessité d'une nouvelle plateforme d'intégration d'applications remédiant à ces problèmes, et sur laquelle des solutions de type BPM (sigle anglais de Business Process Management) ou BAM (Business Activity Monitoring) peuvent se greffer.

Nous signifions par le qualificatif « massivement parallèle » attribué au Bus Applicatif (BA) proposé, sa capacité à exploiter au mieux possible le parallélisme, aussi bien au niveau applicatif qu'à tous ses niveaux internes (interconnexion, communication, ...). La prise en charge du parallélisme au niveau applicatif offre une meilleure exploitation de la concurrence inhérente des applications, tandis que son introduction au niveau des couches internes du BA améliore les

performances de notre plateforme.

Le cahier de charges est donc lourd et les verrous technologiques sont nombreux. Cela résulte d'une rencontre de deux thèmes de recherche : architectures d'intégration d'applications, et traitement massivement parallèle. Une piste qui reste, jusqu'à présent, très peu explorée.

1 Objectifs de la thèse

Les objectifs de cette thèse portent principalement sur la conception, le développement ainsi que l'évaluation de performance d'une nouvelle plateforme d'intégration d'applications massivement parallèle que nous appelons *bus applicatif (BA)*. Son rôle principal consiste à assurer l'acheminement des données circulant entre les acteurs (applications, threads, ...) de toutes les configurations interconnectées vers leurs destinations. Ce BA facilite donc le fonctionnement collaboratif et offre ainsi un système d'échange performant et sécurisé entre les différents acteurs impliqués dans un processus de traitement d informations donné.

Le BA constitue l'épine dorsale servant de fondation technologique pour l'architecture ESB, et sur laquelle des extensions de type BPM ou BAM peuvent se greffer. Nous nous focalisons dans cette thèse sur la couche déploiement de l'architecture ESB. Les couches BPM et BAM quant à elles, sortent du cadre de notre travail.

Pour arriver à cette fin, la démarche à suivre consiste dans un premier temps, à faire une étude bibliographique approfondie qui, d'une part, retrace l'évolution des architectures d'intégration d'applications, d'autre part, fait le point sur les bibliothèques de programmation parallèle qui existent à l'heure actuelle. Cette approche a pour but d'étudier les caractéristiques, ensuite d'analyser le comportement de chacune de ces technologies afin d'identifier de manière précise leurs insuffisances au regard des besoins recensés dans notre cahier de charges. À partir des informations recueillies par ce tour d horizon, la conception de l'architecture logicielle de notre plateforme peut être envisagée, après quoi il s'agit de choisir l'outil technologique de base du BA. A cet effet, une étude expérimentale préliminaire est réalisée sur plusieurs outils, dont les résultats fixent la technologie la plus adaptée à notre projet.

Le dernier volet de cette thèse porte sur l'évaluation de performance de l'architecture du BA en fonction des paramètres liés à son environnement d'exécution (bande passante, nombre de processeurs, nombre de cœurs, ...) ; C'est pourquoi nous avons opté pour une approche d'évaluation à base de tests expérimentaux. Cette étude a pour but, d'une part, d'évaluer la capacité de l'infrastructure à supporter la charge, d'autre part, de prouver l'efficacité des mécanismes de communication, de synchronisation et de partage de données que nous souhaitons mettre en œuvre.

2 Organisation du manuscrit

Ce document s'articule autour de deux parties. La première est consacrée à l'état de l'art. Elle est divisée en deux chapitres : le premier retrace l'évolution des architectures d'intégration d'applications depuis l'architecture point-à-point au ESB. Le second chapitre présente une étude sur les principales technologies utilisées dans le domaine du parallélisme, notamment celles liées à la gestion des communications. Chaque chapitre est clôturé par une analyse décrivant l'insuffisance de ces technologies par rapport à nos objectifs.

La deuxieme partie nous fait penetrer dans le vif du sujet : il s agit de dissequer en long et en large la nouvelle architecture que nous proposons. Elle tourne autour de six chapitres, le premier étant destiné à la présentation de l'architecture du Bus Applicatif. Le second fait le point sur les

principaux outils utilisés en programmation concurrente. Nous présentons dans le chapitre trois un nouvel allocateur dynamique de mémoires partagées qui remédie à certaines limitations du standards IPC (Inter-Process Communication). Le chapitre quatre décrit en détails l'architecture et le mode de fonctionnement du Terminal du Bus Applicatif (TBA). Nous consacrons le chapitre cinq à la description du mécanisme de multiplexage et démultiplexage du Bus Applicatif. Le sixième chapitre présente une évaluation de performance de l'architecture du Bus Applicatif dans son intégralité.

Nous achevons ce document par la conclusion générale et le perspectives de ces travaux de thèse.

3 Publications

Cette thèse a fait l'objet de nombreuses publications. Elles sont présentées ci-dessous par catégorie.

Conférence internationale avec comité de lecture

[A] R. Benosman, K. Barkaoui, and Y. Albrieux, "Exploiting concurrency for the ESB architecture," in *International Conference on Engineering of Complex Computer Systems. ICECCS 2013*, July 2013.

[B] R. Benosman, K. Barkaoui, and Y. Albrieux, "A new dynamic ipc-memory allocator based on a paging approach," in *International Conference on High Performance Computing and Simulation (HPCS 2013)*, 2013, pp. 382–389.

[C] R. Benosman, K. Barkaoui, and Y. Albrieux, "Design and implementation of a massively parallel esb," in *International Symposium on Programming and Systems (ISPS'2013)*, April 2013, pp. 89 – 95.

[D] R. Benosman, K. Barkaoui, and Y. Albrieux, "Performance evaluation of a massively parallel esb-oriented architecture," in *IEEE International Conference on Service-Oriented Computing and Applications. SOCA 2012*, December 2012, pp. 270–273.

Brevet

[E] Y. Albrieux and R. Benosman, "Procédé d'exécution parallèle d'une pluralité de tâches informatiques," Patent, January, 2012.

Table des figures

Première partie

État de l'art

1

Intégration d'applications : de l'architecture point-à-point à l'architecture ESB

Sommaire

1.1 Introduction

Depuis de nombreuses années, les entreprises ont toujours fait évoluer leurs systèmes d'information dans le but de répondre le mieux possible, à leurs besoins, et aux besoins de leurs partenaires. Ainsi, de nombreuses solutions ont vu le jour, souvent désignées par le terme *progiciel*, elles sont passées par une série progressive de transformation pour aboutir à la mise au point de plusieurs technologies logicielles tels que : ERP (acronyme anglais de Enterprise Ressource Planning) [EP01], CRM (sigle anglais de Customer Relationship Management) [Bul03], etc.

Le système d'information d'entreprise est généralement constitué d'un ensemble d'application hétérogènes qui ne sont pas nécessairement conformes aux standards. Cette hétérogénéité est due soit à l'utilisation de différentes technologies (langages de programmation, systèmes d'exploitation, ...), soit à l'ouverture de l'entreprise sur d'autres sociétés qui possèdent leurs propres façons de représentation de l'information. Bien que hétérogènes, ces applications ne sont pas forcement indépendantes, et peuvent collaborer dans le cadre d'un processus métier bien précis de la société.

Par ailleurs, la démocratisation d'Internet a introduit un changement capital sur le périmètre d'action des entreprises, en les rapprochant d'avantages les unes avec les autres. Cela leurs a donné une certaine ouverture sur de nouveaux besoins du marché tout en restant compétitives. Cependant, cette évolution n'a pas été sans conséquence sur le système d'information de la société, qui a peu à peu penché vers une *architecture spaghetti*. Dans cette dernière, les moyens logiciels de communication et de collaboration reposent sur des technologies différentes. Dans la plupart des cas, la mise en œuvre d'une nouvelle application se fait en fonction de la dernière technologie du moment, sans se rendre compte qu'elle risque de devenir rapidement obsolète, et donc remplacée par d'autres. Cette situation d'instabilité engendre des problèmes d'incompatibilité qui freinent le fonctionnement collaboratif de ces applications.

Les entreprises sont donc contraintes de faire communiquer non seulement leurs propres applications internes (niveau intra-entreprise), mais aussi d'autres applications appartenant à des sociétés externes (niveau inter-entreprise). Ainsi, les solutions logicielles doivent être suffisamment génériques pour prendre en compte ces deux cas de figure.

Nous présentons dans ce chapitre une étude bibliographique sur le domaine d'intégration des applications, tout en illustrant au passage les principaux concepts qui lui sont appropriés. Après quelques définitions de base de la notion d'intégration, nous présentons un état de l'art sur les plateformes d'intégration d'applications depuis l'architecture point-à-point jusqu'à l'architecture ESB (sigle anglais de Enterprise Service Bus) [ANAN10], en passant par les EAI (acronyme anglais de Enterprise Application Integration) [Man01] : une étude préliminaire d'une importance capitale à travers laquelle nous mettons en évidence les principales limitations des architectures actuelles au regard de nos besoins, ainsi que leurs inadaptations au mode de traitement massivement parallèle. Cette section s'achève par une conclusion qui présente une synthèse des connaissances acquises tout au long de l'étude.

1.2 Intégration

L'intégration des applications est une problématique née avec l'industrie du logiciel, et les solutions apportées n'ont pas cessé d'évoluer tant sur le plan architectural que technologique du logiciel. En effet, durant ces deux dernières décennies, plusieurs approche d'intégration on été admises donnant ainsi naissance à divers types d'architecture. sur toute une chaîne de solutions, chacune été élaborée en fonction des limitations que présentées la précédente, dans le but précis de construire une plateforme d'intégration flexible et adaptable tout en respectant au maximum, les grands standards du marché.

Dans la littérature, de nombreuses définitions ont été données au concept d'intégration, nous nous contentons de ne citer que celles qui ont particulièrement retenue notre attention :

- Selon Manouvrier [Man01] : l'intégration regroupe l'ensemble des méthodes et outils organisant les échanges entre applications et les processus métiers en intra ou inter entreprises. Devenue un outil stratégique pour les entreprises, elle permet une réelle réactivité du système d'information face aux évolutions métiers et par conséquent d'en optimiser fortement

les coûts.

- De son côté, Linthicum [Lin99] propose une autre définition : l'intégration est une approche dite stratégique qui permet de relier plusieurs systèmes d'informations les uns aux autres aussi bien au niveau service qu'au niveau informationnel, en permettant ainsi le partage à la fois des informations et des processus.
- De notre part, nous considérons l'intégration comme étant une démarche, dont le but consiste à incorporer de manière itérative de nouveaux acteurs (application, processus, base de données, interface graphique, ...) au sein du système d'informations (SI) d'une entreprise, à travers un ensemble de moyens logiciels (plateforme logicielle, API (sigle anglais de Application Programming Interface), ...), et ce, aussi bien dans un contexte inter-organisationnel que intra-organisationnel.

Dans le respect total de l'avis de chacun, nous pouvons nous entendre pour reconnaître que l'intégration est un concept clé pour le système d'information de l'entreprise et à travers lequel elle s'affranchit des contraintes techniques, fonctionnelles et organisationnelles pour constituer un corps unique, complet et cohérent afin d'augmenter sa réactivité et sa productivité.

D'après la définition que nous venons de proposer, nous constatons que les acteurs impliqués dans un processus d'intégration sont de nature variée. Ainsi, nous identifions plusieurs formes d'intégration selon le type d'acteur considéré. Un constat approuvé par des travaux de recherche qui recensent entre autres, trois formes d'intégration à savoir : l'intégration de données [WYZQ09], l'intégration des traitements et l'intégration des applications. Un projet d'intégration mise sur l'approche d'intégration

un projet d'intégration mise sur l'approche d'intégration choisie, celle-ci ne bénéficiant pourtant d'aucune exclusivité dans la réussite de ce projet étant donné que l'application simultanée des approches et une éventualité toujours accessible, et dépend des circonstances, comme dans le cas d'une entreprise où certaines parties reposent sur l'intégration des données, alors que d'autres sur le traitement.

1.2.1 Intégration de données

L'intégration de données est une forme d'intégration dédiée aux base de données dans le but de faciliter leurs connectivités. Lors d'un processus d'intégration de données, l'utilisateur a la possibilité de décrire toute une série de transformation : migration, consolidation, combinaison et... qui sera appliquée sur des données provenant de diverses sources : fichiers plats, bases de données relationnelles, base de données hiérarchiques, emails, etc ... et ce avant leurs mises à disposition.

La mise en pratique de cette forme d'intégration est généralement réalisée à l'aide d'une catégorie particulière d'outils appelée ETL (Extract, Transform and Load) [Vas09]. Au même titre que les IDE (Integrated Development Environment) aident le programmeur dans le développement de ses applications, les ETLs facilitent l'interaction avec les base de données en vue d'automatiser leurs traitements. Les ETLs constituent aujourd'hui la solution référentielle dans le domaine d'intégration de données (exemple : Talend [Conb]).

1.2.2 Intégration de traitements

A ce niveau, le processus d'intégration consiste à incorporer une nouvelle entité logicielle au sein d'une application existante afin de lui apporter des fonctionnalités supplémentaires. Cette entité se caractérise par l'incapacité de s'exécuter dans l'isolement et sans la présence d'une unité

logicielle de traitement qui lui apporte la logistique nécessaire. L'exemple typique de cette forme d'intégration est celui des API (Application Programming Interface).

1.2.3 Intégration d'applications

L'intégration d'applications est une démarche qui cherche à interconnecter des applications distribuées et hétérogènes qui, à la base, n'ont pas été pensées pour fonctionner ensemble, et qui sont généralement développées de manière indépendante mais surtout, de façon incompatible (exemple : deux applications provenant de deux éditeurs différents).

Une telle initiative est d'un enjeu capital pour toute société qui dispose d'un nombre important d'applications qu'elle souhaite faire collaborer afin d'atteindre un but commun, son système d'information devient donc plus flexible, plus réactif, et la modification ou la mise au point d'un nouveau processus métiers afin de répondre à de nouveaux besoins nettement plus aisée, sans pour autant avoir à investir dans un développement trop couteux.

1.3 Types d'intégration

Tout d'abord, signalons qu'une approche d'intégration ne concerne en aucun cas, la résolution des problèmes de communication entre composants internes d'une même application, mais traite des échanges entre applications aussi bien d'une même entreprise que de plusieurs. Les difficultés de communication au sein d'une même application relèvent plutôt des domaines de conception et d'architecture logicielle. Ainsi, nous distinguons deux types d'intégration :

- **Intégration intra-entreprise** : comme son nom l'indique, ce type d'intégration s'applique exclusivement sur les applications déployées au sein d'une même entreprise, et il est désignée par l'abréviation A2A (sigle anglais de Application to Application intégration).
- **Intégration inter-entreprise** : elle est représentée par le sigle B2B (Business to Business en anglais) et se distingue de celle citée précédemment par un périmètre d'intégration plus élargi qui peut s'étendre sur des applications externes, appartenant à des entreprises étrangères, ce qui complique le processus d'intégration. A cela, s'ajoutent d'autres problèmes de sécurité, et de communication réseau.

1.4 Plateformes d'intégration d'applications

Après une première génération de solutions d'intégration *point-à-point* proposées à partir des années 90, d'autres solutions plus élaborées de type *EAI* (sigle anglais de Entreprise Application Intégration) ont paru quelques années plus tard et ont réussi à résoudre un bon nombre de problèmes posés par les anciennes architectures point-à-point, notamment ceux liés à l'unification des connecteurs, et au passage à l'échelle.

L'architecture EAI ne s'est pas vraiment imposée en tant que plateforme de référence dans le domaine d'intégration, pour causes, sa conception centralisé vulnérable, et son non respect des grands standards du marché. Elle a été rapidement dépassée par une architecture répartie de type ESB (acronyme anglais de Enterprise Service Bus) plus souple, plus robuste, et entièrement basée sur les normes standards [Man01].

Dans cette section, nous présentons une vue d'ensemble sur les architectures d'intégrations d'applications, pour chacune d'elles, nous exposons les principales caractéristiques à retenir.

1.4.1 Architecture point-à-point

Ce type d'architecture a paru au début des années 90, et son principe est fondé sur une démarche d'intégration point-à-point, dans laquelle, chaque couple applications est considéré comme une brique à part, possédant ses propres connecteurs et son propre protocole de communication. Chaque application dispose d'autant de connecteurs que d'applications sur la plateforme [JcM06]. La Figure 1.1 présente la topologie d'une architecture point à point.

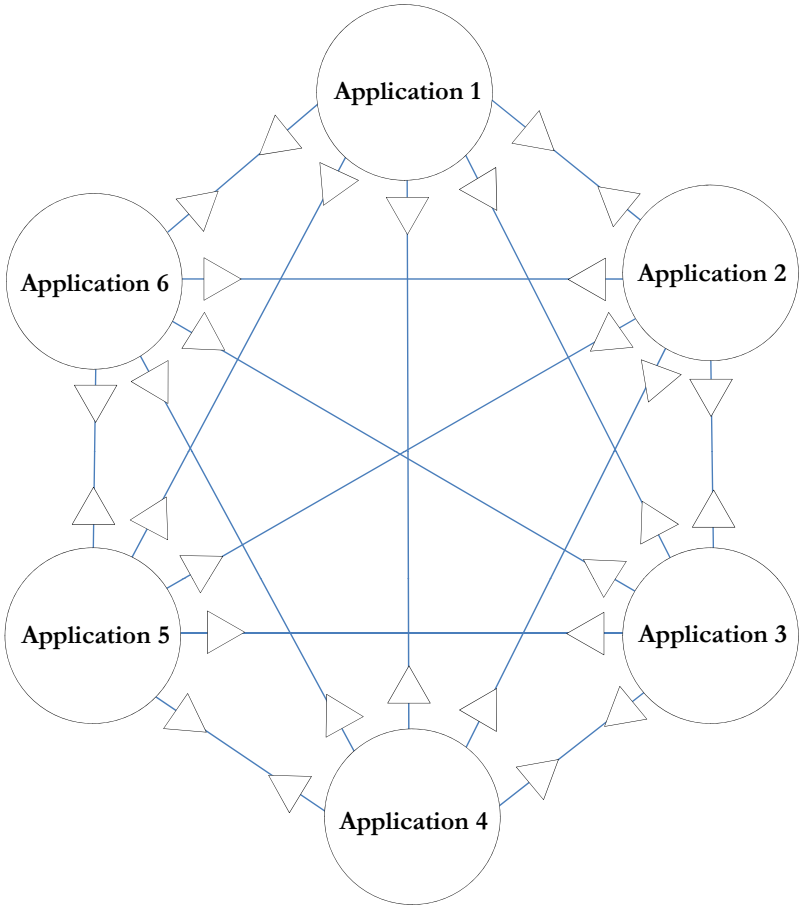

Figure 1.1 – Architecture d'intégration point-à-point.

Comme nous pouvons le constater depuis la Figure 1.1, le nombre de connecteurs augmente de manière proportionnelle au nombre d'applications à relier, un inconvenient caractéristique de cette architecture ce qui complique énormément la tâche au programmeur, qui se trouve contraint de développer tout un ensemble de connecteurs, à chaque intégration d'une nouvelle application. De plus, cette approche engendre : un coût de maintenance plus élevé, une administration plus ardue, et un risque d'erreurs plus important. En effet, avec une telle organisation, et si nous considérons qu'à l'extrême, chaque application doit communiquer avec toutes les autres applications de la plateforme, le nombre total d'interfaces est obtenu par la formule 1.1 :

$$Nb\_Interfaces = n\ (n-1)/2 \tag{1.1}$$

où n est le nombre total d'applications, et Nb_Interfaces le nombre d'interfaces nécessaires. Ainsi, pour connecter 5 applications, il faut 10 interfaces, pour en connecter 100, il en faut 4950 : un argument qui qualifie cette architecture d'*architecture spaghetti*.

Par ailleurs, cette architecture se caractérise par un couplage fort entre les applications. Par conséquent, toute modification d'une application donnée, nécessite une série de modifications au niveau des autres applications avec lesquelles elle est reliée, sous peine de causer des dysfonctionnements dans les échanges.

Modifier ou pas une application, voilà une question qui devrait être minutieusement étudiée, avant toute opération aboutissant à un quelconque changement dans le système, et dont il serait difficile de mesurer l'ampleur des adaptations requises ultérieurement. Nous comprenons donc pourquoi les responsables du SI montrent une certaine réticence envers tout type de transformation susceptible d'impacter le système.

1.4.2 Architecture EAI

L'architecture EAI (sigle anglais de Enterprise Application Integration) est née au milieu des années 90. Elle a été proposée pour répondre aux nombreuses limitations de l'architecture point-à-point, avec l'ambition de faciliter l'intégration des systèmes d'informations et en former un tout cohérent et opérationnel. L'architecture EAI se présente sous forme d'une plateforme centralisée, dans laquelle, chaque application se connecte de manière unique et indépendante à un système central, sans avoir besoin de connaitre à l'avance, la nature des autres applications qui font partie du même système. La Figure 1.2 donne une vue d'ensemble sur l'architecture EAI.

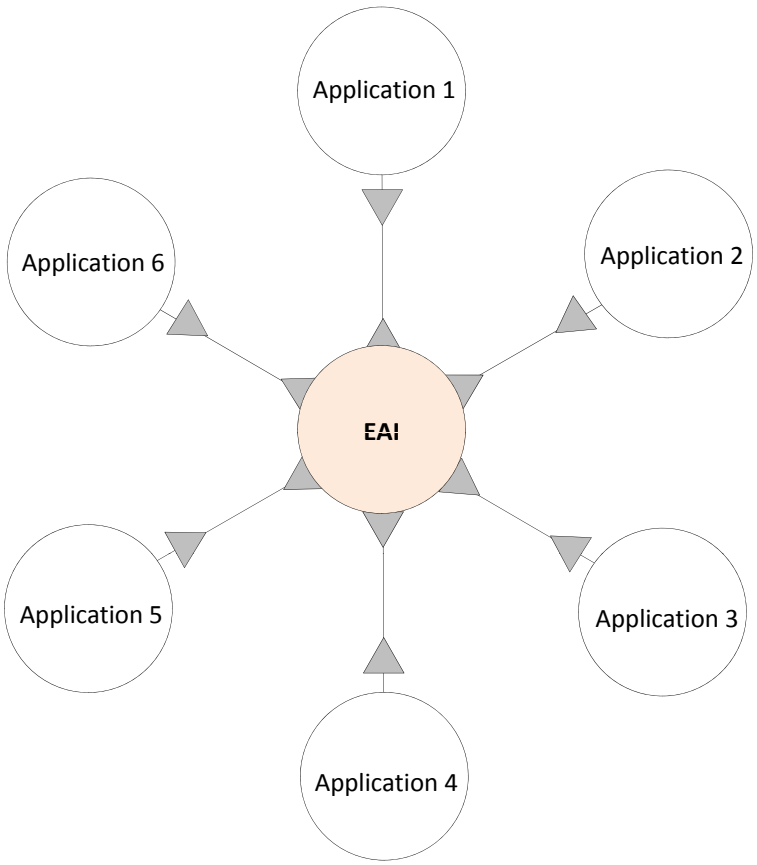

Figure 1.2 – Architecture d'intégration EAI.

En somme, les applications ne communiquent donc plus directement entre elles, mais à travers le système central qui prend en charge l'acheminement de tout le flux de communication. Des interfaces génériques standards appelées *connecteurs* assurent la connexion entre les applications et ce système central, réduisant ainsi de manière considérable : le nombre d'interfaces à gérer, le coût de développement, de maintenance et d'administration.

L'architecture EAI a énormément fait progresser le marché des technologies d'intégration en apportant une certaine robustesse et une simplicité notamment à travers l'unification des connecteurs, ce qui n'est pas le cas dans l'architecture point-à-point. Cependant, et comme dans toute architecture centralisée, le maillon faible de l'architecture EAI est le SPOF (Single Point of Failure). En effet, tout dysfonctionnement au niveau du système central, conduit à l'arrêt de la plateforme entière.

Les concepts d'EAI et de Middleware ne sont pas complètement disjoints. En effet, les Middlewares et les EAI sont tous les deux conçus pour faciliter l'élaboration d'applications distribuées, cependant les EAI disposent d'une capacité d'intégration d'applications plus avancée que celle proposée par les Middlewares.

L'architecture EAI est composée de quatre couches de base à savoir : la couche de transport de données, la couche connexion, la couche transformation de données, et la couche orchestration des processus métiers.

1.4.2.1 Couche transport de données

La couche transport assure le routage des données échangées entre les différentes applications en s'appuyant sur un MOM (Middleware Orienté Messages) qui offre un couplage faible entre applications, les rendant ainsi moins dépendantes les unes des autres. L'ajout ou le retrait d'une application n'a donc pas d'impact sur le fonctionnement global de l'architecture.

1.4.2.2 Couche connexion

Pour qu'une application puisse s'intégrer dans l'architecture, elle doit se connecter à l'EAI à travers des interfaces dédiées (connecteurs). Une fois l'opération réussie, l'EAI établit de manière complètement transparente la connexion entre les interfaces des applications communicantes.

1.4.2.3 Couche transformation de données

Les informations échangées entre applications sont généralement de nature hétérogène, et leurs structures ne sont pas toujours les mêmes. EAI offre à ce niveau des services de conversion de données basés sur des règles de transformation. A titre d'exemple, nous citons le moteur XSL qui prend en charge la transformation des messages XML.

1.4.2.4 Couche orchestration des processus métiers

Les couches de transformation et de connexion auxquelles s'ajoute la couche de transport de données ont pour objectif d'assurer l'intégration des applications. Cependant, pour décrire un processus métier, le programmeur doit disposer d'une couche complémentaire mettant à sa disposition les outils nécessaires pour la modélisation, le déploiement de ses processus métiers de manière complètement automatisée, en décrivant la séquence d'enchainement des tâches applicatives, d'où l'intérêt de la couche orchestration.

1.4.3 Enterprise service bus (ESB)

L'émergence des nouvelles approches orientées services telles que SOA (Service-Oriented Architecture) [Li10], et la faiblesse des architectures EAI, ont conduit à l'élaboration d'une nouvelle architecture appelée ESB (Enterprise Service Bus) plus adaptée aux besoins d'intégration [Cha04]. Elle a été proposée par le Gartner Group en 2003, avec pour principal objectif, faciliter la communication entre des applications qui à la base, ne sont pas pensées pour fonctionner ensemble. Elle adopte les grands standards du marché comme XML (sigle anglais de Extensible Markup Language) ou encore les services web [YCD+09].

L'ESB constitue l'épine dorsale du SI d'une entreprise à travers laquelle circule le flux de communications entre toutes les applications. Il se caractérise par une architecture distribuée, dans laquelle, les applications sont réparties sur l'ensemble des machines de la plateforme, chacune reliée à l'ESB par le biais d'un connecteur. Cette caractéristique est d'une importance capitale car elle n'introduit aucun point de faiblesse ou de contention (SPOF), des problèmes qui ont longtemps entravé les architectures EAI.

L'architecture ESB adopte un modèle de communication par échange de messages, dans lequel, les communications sont réalisées de manière asynchrone. Pour arriver à cette fin, elle intègre un middleware orienté message (MOM) tel que JMS (Java Message Service). Avec l'introduction du couplage faible, ce mode d'échange présente un intérêt majeur pour les applications en les rendant moins dépendantes les unes des autres.

Aujourd'hui, la technologie ESB est l'approche la plus prometteuse pour l'intégration d'applications de l'entreprise. Elle facilite l'implémentation d'une architecture SOA par intégration itérative des applications isolées dans une infrastructure complètement décentralisée. La Figure 1.3 donne un aperçu sur l'architecture ESB.

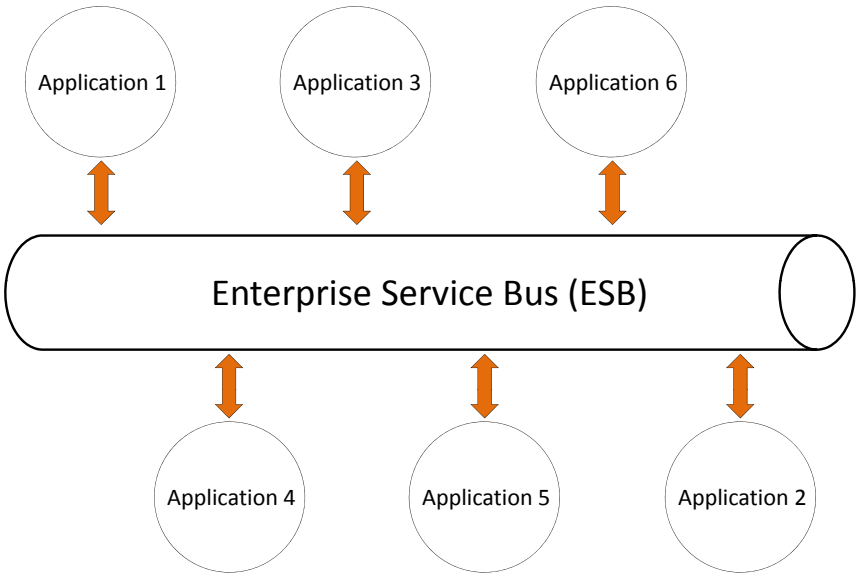

Figure 1.3 – Architecture d'intégration ESB.

1.4.3.1 Services de base

De nombreuses solutions d'intégration à base d'ESB ont été proposées, elles sont soit open-source comme : Mule [Comb], Petals [Cona], et Fuse [Coma], soit propriétaires tels que : Sonic

ESB [Sof], IBM WebSphere Message Broker [Comc], et Oracle ESB [SAKan], mais la plupart d'entre elles proposent les mêmes services de base nécessaires au fonctionnement des applications parmi lesquels nous citons :

- **Transfert de fichiers** : assure l'échange de fichiers entre les différentes applications connectées à l'ESB ;
- **Orchestration** : gestion, contrôle, et coordination automatique des services qui composent le processus métiers de l'entreprise.
- **Transmission de messages** : achemine les messages à leurs bonnes destinations. L'utilisation d'un middleware orienté messages (MOM) assure l'envoi et la réception des données, généralement en mode asynchrone et faiblement couplé (découple l'expéditeur du message de son destinataire).
- **Transformation** : convertit le format d'une donnée afin de la rendre compréhensible, donc interprétable par la ou les applications cibles, et concerne aussi bien l'aspect syntaxique que sémantique du message original [Zha10].

1.4.3.2 Java Message Service (JMS)

Java Message Service (JMS) est une API Java standard de type MOM (acronyme anglais de Message Oriented Middleware) destinée au développement d'applications distribuées [GJMCGS10]. Elle se caractérise par un mode de communication asynchrone, faiblement couplé, suivant un paradigme par échange de messages. JMS a été intégré à la plateforme J2EE (Java 2 Entreprise Edition) depuis la version 1.3.

Actuellement, le standard JMS est considéré comme étant la couche de communication de base pour la plupart des architectures ESB. Son succès revient principalement à sa portabilité, et sa robustesse. Par dessus tout, JMS intègre de nombreux mécanismes de gestion de flux qui garantissent le bon déroulement des transmissions de données tels que : contrôle d'acquittement, gestion des priorités, paramétrage des délais d'expiration, etc.

JMS est disponible en plusieurs distributions, la version la plus récente étant JMS 2.0. Le tableau 1.1 donne quelques exemples d'implémentations JMS.

Produit	Editeur	Open source	Propriétaire
SonicMQ	Sonic Software		✓
Apache ActiveMQ	Apache Software Foundation	✓	
OpenJMS	Sun Microsystems's	✓	
Oracle AQ	Oracle		✓

Tableau 1.1 – Quelques implémentation JMS.

JMS s'appuie sur l'utilisation des files d'attente en tant que structure de stockage des données, dans lesquelles les messages sont en permanence rangés jusqu'à leur consommation par le récepteur ou leur expiration, ou bien encore réordonnancés en fonction du niveau de priorité qui leur a été attribué au départ, et il arrive même que certains soient envoyés dans un ordre autre que celui dans lequel ils ont été déposés. Le standard JMS compte deux modes de communication [EFGK03] à savoir :

- **Mode point-à-point** : dans lequel, l'émetteur et le récepteur se partagent une file d'attente sur laquelle des échanges se déroulent de manière unidirectionnelle, suivant un fonctionnement est assez simple : un émetteur dépose ses messages au niveau de la file d'attente,

et le récepteur qui lui est abonné reçoit une notification l'informant de la disponibilité d'un nouveau message. Ce mode de fonctionnement est illustré dans la Figure 1.4.

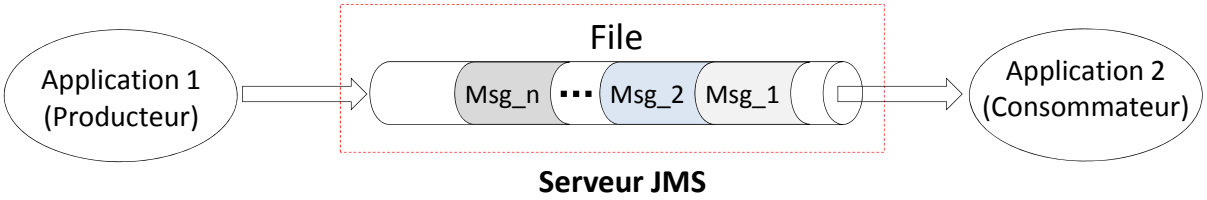

Figure 1.4 – Mode de communication point-à-point.

- **Mode publisher/suscriber** : à la différence du mode point-à-point, ce mode s'appuie sur la notion de sujet (topic) qui sert à de définir un domaine d'échange spécifique pour lequel plusieurs applications vont s'abonner simultanément. Ainsi, lorsqu'un message est adressé par un émetteur à un sujet donné, il est intercepté par toutes les applications qui lui sont préalablement abonnées. L'émetteur n'est donc pas contraint de connaître les applications avec lesquelles il va interagir, ce qui facilite l'implémentation d'une sorte de transmission broadcast sélective, c'est d'ailleurs une des particularités de ce mode. La Figure 1.5 présente le principe de fonctionnement de ce modèle :

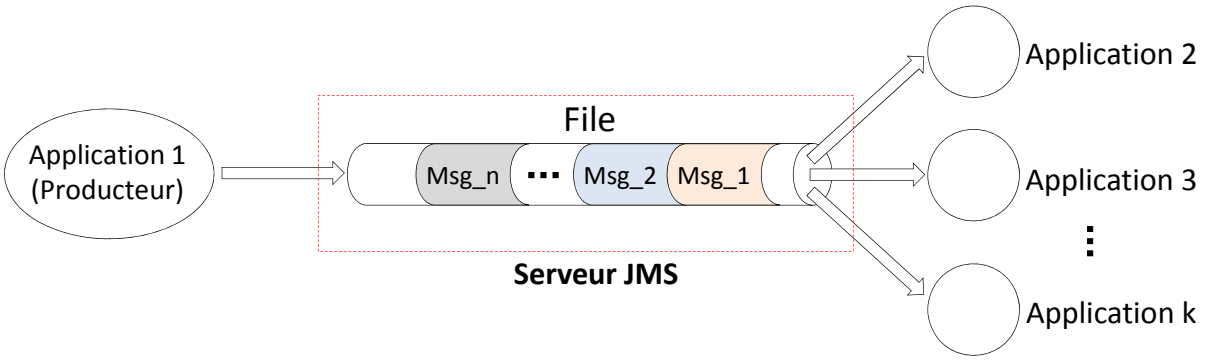

Figure 1.5 – Mode de communication publisher/suscriber.

1.5 Discussion

D'après cette étude bibliographique, nous constatons que les architectures d'intégration élaborées jusqu'à présent, qu'elles soient de type point-à-point, EAI, ou encore ESB, ne prennent pas en considération le traitement massivement parallèle, et ne proposent rien quant à sa prise en charge. Cette caractéristique s'oppose à l'évolution constante que connaissent les technologies multicoeur/multiprocesseurs, notamment ces dernières années, et pourtant, le parallélisme reste pour le moment, le seul recours prometteuse pour les applications de plus en plus en quête de puissance de traitement.

Mis à part le travail de recherche présenté dans [Lu09], par lequel les auteurs ont exploité le parallélisme pour accélérer le traitement des documents XML, il n'existe pas à notre connaissance du moins à travers note étude bibliographique aucune contribution innovante à une quelconque architecture qui traiterait, à la fois des aspects : d'intégration, de communication, et de traitement massivement parallèle.

L'étude présentée dans [Lu09] est certes intéressante, et constitue une première initiative à l'introduction du parallélisme dans le traitement, néanmoins, ses outils de développement et paradigme de programmation restent insuffisants au regard des besoins du développeur, d'où la nécessité d'une nouvelle plateforme d'intégration d'applications adaptée aux traitements massivement parallèles.

Les principales limitations des solutions ESB actuelles sont dues à leurs couches de communication, notamment concernant les points suivants :

- JMS utilise les files d'attente comme structure de données de base. Dans un environnement d'exécution massivement parallèle tel que nous l'envisageons, les entités communicantes sont majoritairement constituées de threads.
  Une éventuelle utilisation du JMS dans cet environnement laisse supposer que chaque thread aura à sa disposition, une ou plusieurs files JMS. Par conséquent, le nombre de files augmente proportionnellement au nombre de threads communicants, ce qui risque d'une part, de conduire à l'épuisement de ressources JMS en files d'attente, d'autre part, de réduire la performance globale du système puisque la gestion des files devient de plus en plus coûteuse.
- La deuxième limitation que nous reprochons au standard JMS concerne le nombre de recopies mémoires intermédiaires qui reste relativement important, et qui se traduit par un coût additionnel lors du transfert des données sur le réseau. En effet, lorsqu'une tâche hôte souhaite échanger un message avec une autre tâche distante, elle doit d'abord l'envoyer à une file d'attente avant que la tâche de destination ne le récupère. Un tel procédé nécessite donc au minimum deux opérations supplémentaires : une première chargée du dépôt de message dans la file, puis une deuxième accordant sa lecture à la tâche distante, alors qu'il serait plus judicieux d'acheminer le message directement vers l'espace d'adressage de la tâche distante, ce qui n'est , de toute évidence, pas le cas.
- Dans la norme JMS, chaque opération d'émission ou de réception nécessite au moins une opération E/S quelle que soit la taille des messages échangés, alors qu'il était plus intéressant de regrouper au maximum les messages qui ont la même destination, de manière à avoir des paquets plus compacts, et réduire ainsi le nombre d'opérations E/S consommées durant le transfert.

Pour toutes ces raisons, nous avons décidé d'implémenter notre propre mécanisme de communication réseau. Il a été élaboré de manière à remédier, entre autres, aux trois facteurs limitatifs cités ci-dessus. Une étude plus détaillée lui est consacrée dans le Chapitre 5 de la Partie II.

Pour la quasi totalité des architectures d'intégration actuelles, l'application est l'unité d'intégration de base, or, dans un contexte d'environnement massivement parallèle tel que celui que nous projetons, ces applications sont parcellisées en une pluralité de tâches, traitée chacune par un thread à part. Par conséquent, nous devons non seulement être capable d'intégrer des applications, mais aussi thread de traitement, d'où l'inadaptation des solutions actuelles. Cette forme d'intégration n'est pas sans risque d'engendrer des violations de cohérence de données manipulées lors d'un accès concurrent en lecture/écriture. Autant de problèmes auxquels nous devons faire face durant la conception et le développement de notre plateforme.

1.6 Récapitulatif

Le Tableau 1.2 présente un récapitulatif des avantages et inconvénients relatifs aux trois architectures d'intégration étudiées précédemment, à savoir : l'architecture point-à-point, l'architecture EAI, et l'architecture ESB. Cette synthèse reflète une fois de plus, le caractère bénéfique

des ESBs.

Architecture	Avantages	Inconvénients
Point-à-Point	• Légèreté des composants : les interfaces sont conçues sur mesure en fonction des besoins	• Hausse sensible des coûts de développement et de maintenance • Ne passe pas à l'échelle • Difficulté de gestion (le nombre d'interfaces augmente de manière considérable avec le nombre d'applications à intégrer) • Irrespect des standards • Redondance des données
EAI	• Réduction du nombre d'interfaces • Facilité de gestion : contrôle centralisé • Unification des connecteurs	• Centralisation de l'architecture : introduction d'un SPOF (Single Point of Failure) • Flexible réduite : technologie propriétaire
ESB	• Respect des standards • Réduction des coûts de développement et de maintenance • Flexibilité notable : rapidité d'intégration • Décentralisation de l'architecture : pas de points de faiblesse ou de contention	• Difficulté d'implémentation

Tableau 1.2 – Avantages et inconvénients des architectures d'intégration.

1.7 Conclusion

Depuis l'apparition des architectures point-à-point, deux grandes innovations ont révolutionné le monde de l'intégration, la première a abouti à la mise au point d'une plateforme EAI centralisée, tandis que la seconde a conduit à l'élaboration d'une plateforme ESB fortement distribuée, et basée essentiellement sur les standards du marché, autant d'atouts qui l'ont classée solution de référence en intégration, et à travers lesquels, elle a su d'une part, répondre aux attentes des entreprises, qui avant tout, cherchent à accroître leurs gains en réactivité et en souplesse, d'autre part, alléger un marché tellement épuisé par des projets d'intégration.

Par ailleurs, l'architecture ESB a su imposer sa fiabilité et son efficacité pour l'intégration des éléments constitutifs de la SOA. En effet, grâce aux ESBs, la technologie SOA a bien trouvé une concrétisation sur le plan pratique.

Sur la base de ce constat, nous avons la conviction que l'architecture de type bus telle que celle des ESB est la voie la plus prometteuse pour la suite de notre étude. Cependant, comme indiqué dans la section précédente, plusieurs améliorations restent à apporter à l'intégration du traitement massivement parallèle et tirer ainsi, un maximum de profit de la puissance offerte par les machines multicœurs/multiprocesseurs, d'où l'intérêt de notre plateforme. Un challenge qui s'annonce à la fois difficile et passionnant. Difficile car le parallélisme soulève bien des problèmes :

de communication, de synchronisation, et de partage et cohérence de données. Passionnant car nous il touche à la fois aux aspects d'intégration et de parallélisme.

2

État de l'art sur les interfaces de communication à haute performance

Sommaire

2.1 Introduction

La communication est un aspect primordial dans tout système réparti, elle permet à des entités déployées sur des machines distantes de communiquer à travers le réseau. La performance d'une couche de communication dépend de plusieurs facteurs liés aux caractéristiques techniques du réseau. Ces caractéristiques peuvent être :

- matérielles, exemple : support physique de transmission : fibre optique, câble Ethernet, etc.
- logicielles, exemple : Protocole de communication : TCP, UDP, etc.

Les nouvelles évolutions technologiques dans le domaine de la communication réseau, plus particulièrement dans les architectures à haute performance, permettent d'atteindre un débit de transfert très important. Cependant, l'introduction de réseaux rapides n'est pas une condition suffisante pour améliorer la latence et le débit des communications. En effet, la capacité du réseau réellement exploitée par les applications reste bien en dessous de celle offerte par le matériel, et cela revient essentiellement à l'inadaptation des interfaces de communication par rapport aux couches matérielles sous-jacentes. Cette inadaptation se traduit par des surcoûts logiciels : Recopie intermédiaire de mémoire, appels systèmes, etc... qui réduisent considérablement la performance globale du système.

La mise en œuvre de couches de communication plus adaptées aux nouvelles possibilités offertes par la couche matérielle s'impose donc pour optimiser la capacité de transmission de transmission et réduire ainsi le surcoût lié aux opérations de transfert.

Dans le cas le plus simple, les différentes entités qui composent un système distribué correspondent à des processus. Avec l'intégration du parallélisme dans le traitement, ces processus sont parcellisés *dans la mesure du possible* en un ensemble de tâches parallèles s'exécutant sur différents threads. Les besoins en termes de communication passent alors du niveau processus au niveau threads, avec le risque de poser des problèmes au niveau de la couche de communication : En effet, le nombre d'entités communicantes augmente de manière considérable, ce qui risque d'engendrer des goulots d'étranglement qui se traduisent par une baisse de performance.

Par ailleurs, la capacité du système d'exploitation en termes de ressources de communication (sockets, ports, ...) est limitée, ce qui risque d'être extrêmement contraignant dans surtout dans un environnement massivement parallèle tel que celui du bus applicatif. Nous reviendrons largement sur ces points dans la Section 4.7.1 de la Partie II.

Cette section est organisée comme suit : Dans un premier temps, nous nous intéresserons de plus près aux principes de base liés à la communication réseau. A ce titre, nous présenterons les différentes couches : Applicative, système, et matérielle qui peuvent entrer en interaction lors du transfert des données sur le réseau. Cela nous permet d'identifier le niveau sur lequel nous intervenons par la suite afin d'apporter de nouvelles solutions pour mieux gérer la couche de communication.

Dans la deuxième partie, nous proposerons un tour d'horizon sur les principales interfaces de communication utilisées dans le domaine des systèmes distribués. Pour cela, une classification à trois niveaux est proposée : interface de bas niveau , interface de niveau intermédiaire et interface de niveau applicatif. Pour chaque niveau considéré, nous donnerons quelques exemples de bibliothèques standards, tout en exposant leurs caractéristiques, leurs avantages ainsi que leurs limitations.

Des tableaux récapitulatifs seront présentés à la fin de cette partie, afin d'exposer les caractéristiques les plus pertinentes à retenir. A partir de cette étude, nous établirons notre cahier des charges pour la conception et le développement d'un nouveau mécanisme de communication adapté au modèle d'échange du Bus Applicatif.

2.2 Communication avec recopie intermédiaire de mémoire

L'accès au périphérique réseau reste le privilège exclusif du système d'exploitation. Ce dernier joue le rôle d'intermédiaire entre les applications et la carte réseau. Une application quelconque doit obligatoirement passer par des primitives systèmes pour envoyer ou recevoir des données [Bru08]. A chaque primitive d'émission ou de réception correspond une opération E/S dont le coût peut atteindre la microseconde. Il y a donc trois couches à traverser : applicative, systèmes, et matérielle durant la phase de transmission/réception des données sur le réseau [Aum02]. Pour illustrer ce principe, nous allons prendre un exemple simple, dans lequel deux entités logicielles : E1 et E2 souhaitent échanger des données à travers le réseau.

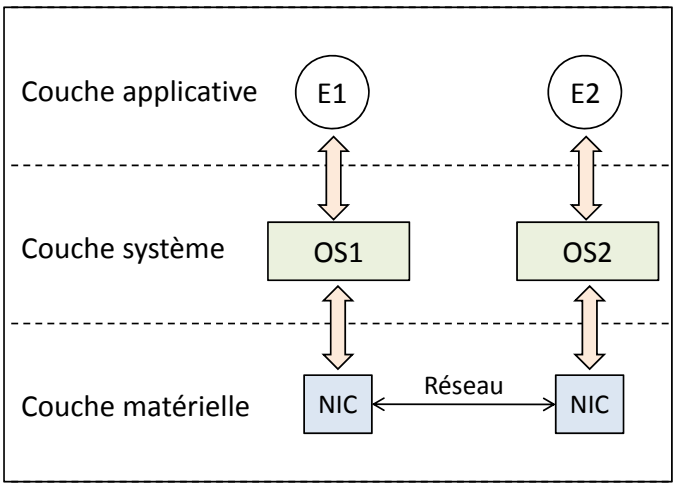

Figure 2.1 – Les différentes couches intervenant durant la transmission de données.

La Figure 2.1 illustre la superposition des trois couches : Applicative, système, et physique. Au moment de l'émission, les données à transmettre transitent depuis la couche applicative de E1 vers la couche physique en passant par la couche système. Cette dernière a pour but de contrôler les accès au périphérique sous-jacent (NIC : Network Interface Controler). Du coté de E2 (récepteur), l'opération inverse est réalisée, les données transitent cette fois-ci depuis la couche de physique vers la couche applicative, tout en passant par la couche système.

2.2.1 Émission

Dans le cas d'une émission, les données à transmettre par E1, sont tout d'abord copiées au niveau système, avant d'être transmises par la carte réseau. Cette recopie a un intérêt double ; d'une part, elle permet à l'entité E1 de réutiliser à nouveau le tampon de transmission sans risque d'interférence de données, d'autre part, elle permet de compléter les données brutes de E1 par des informations complémentaires (en-têtes) afin de reconstruire des segments adéquats au protocole utilisé. La Figure 2.2 ci-dessous montre un exemple simple du principe de la recopie intermédiaire de la mémoire.

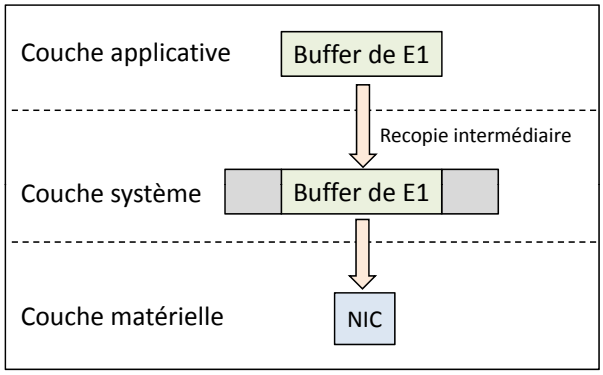

Figure 2.2 – Copie intermédiaire du tampon de transmission.

2.2.2 Réception

Dans le cas d'une réception, l'application indique à travers des fonctions spécifiques, l'endroit (tampon) dans lequel elle souhaite recevoir les données. Dès que ces données sont disponibles au niveau du périphérique, elles seront systématiquement recopiées vers un espace d'adressage propre au système d'exploitation, qui se chargera par la suite de les transmettre à l'application concernée. Au niveau de la réception, le problème qui se pose reste directement lié à l'incapacité de l'application hôte d'anticiper l'arrivée des données. L'appel de la fonction de réception peut donc parvenir soit avant, soit après, la réception effective des données.

Dans le cas idéal, les données sont reçues après l'appel de la fonction de réception, puisque le système est en mesure de fixer directement l'endroit dans lequel elles doivent être déposées. Dans le cas contraire, le système d'exploitation garde en mémoire une copie des données reçues, le temps que l'application indique le tampon dans lequel elle souhaite les recevoir. Ce mécanisme est décrit dans la Figure 2.4.

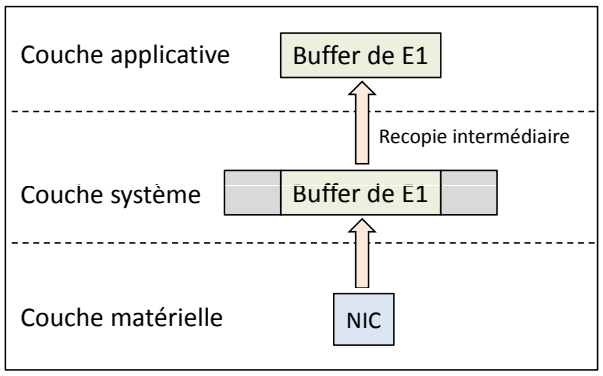

Figure 2.3 – Copie intermédiaire des données reçues.

2.3 Communication sans recopie intermédiaire de mémoire

Pour améliorer la performance du réseau, certains travaux de recherche [vEBBV95] adoptent une approche différente de celle utilisée en mode normal (avec recopie mémoire), dans laquelle les données transitent depuis la couche applicative vers la couche physique sans passer par le système

d'exploitation, ce qui a pour but d'éliminer le surcoût lié aux recopies intermédiaires de mémoire. Le principe consiste à établir des projections entre des zones mémoires appartenant à l'espace d'adressage de l'application, et les zones mémoires réservées au fonctionnement du périphérique. Il devient donc possible de dialoguer directement avec la carte réseau depuis l'espace utilisateur sans faire appel au système d'exploitation. Cette technique ressemble beaucoup à celle utilisée pour les cartes graphiques, dont le principe repose sur la manipulation de la zone mémoire réservée pour l'affichage, directement depuis l'espace utilisateur.

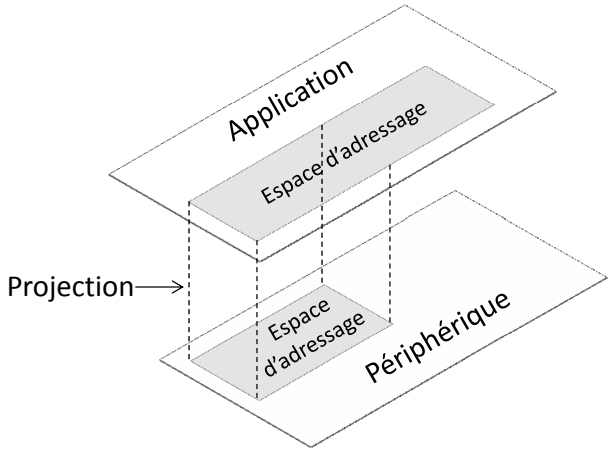

Figure 2.4 – Communication sans recopie intermédiaire de mémoire.

2.3.1 Émission

Dans le cas d'une opération de transmission, l'application doit indiquer pour l'interface réseau, la zone mémoire à transférer. Cette zone doit être protégée tout au long de l'opération de transfert afin de préserver la cohérence de données à transmettre. Ainsi, le tampon de transmission ne peut être réutilisé tant que le transfert n'est pas terminé. On utilise généralement une technique de scrutation (polling en Anglais) dans laquelle l'application interroge au fur et à mesure la carte réseau pour obtenir des informations sur l'état d'avancement du transfert.

2.3.2 Réception

Comme indiqué au début de cette section, l'opération de réception est plus difficile à réaliser que celle de l'émission. Toute la difficulté réside dans l'incapacité de l'émetteur à anticiper l'habileté d'accueil des données par le récepteur. Pour régler ce problème, les interfaces de communication utilisent généralement un mécanisme de synchronisation à base de rendez-vous qui d'une part, permet à l'émetteur d'avoir la certitude que le récepteur est prêt à recevoir les données, d'autre part, permet à la carte réseau de stocker les données directement au bon endroit.

2.4 Mode de transfert des données en mémoire

Le coût de la recopie mémoire représente le temps nécessaire pour le transfert de données entre le périphérique (carte réseau) et la mémoire ou inversement. Il existe deux modes de transfert de données : PIO (Programmed Input/Output) et DMA (Direct Memory Access) [Aum02]. Le choix

du mode de transfert est la charge de l'interface de communication qui pilote le périphérique, à l'exception de quelques interfaces comme SCSI qui laissent le choix aux usagers.

- Dans le mode PIO, Les données sont tout d'abord transférées dans une zone mémoire propre au système d'exploitation avant d'être envoyées à la carte réseau. Dans ce cas, le processeur est occupé durant tout le temps de transfert.
- Le mode DMA, permet un transfert direct de données sans avoir recours à un tampon système intermédiaire et sans faire appel au processeur. Ce dernier intervient uniquement dans les phase d'initialisation du contrôleur DMA en lui fournissant l'adresse mémoire source, l'adresse de destination et la taille de la zone à transférer.

Le choix du mode de transfert se fait en fonction de la taille de données à copier. il est préférable d'utiliser le mode PIO pour les messages de faible taille et le mode DMA pour les message longs, car l'opération d'initialisation du contrôleur DMA est très coûteuse en temps CPU par rapport à celle du mode PIO, mais une fois cette opération effectuée, le transfert se fait de manière directe, et sans intervention du processeur [Mor11]. La difficulté de mise en œuvre du mode DMA constitue un inconvénient majeur, puisqu'il nécessite le verrouillage des données en mémoire afin de garantir qu'elles ne soient pas « swapées » en disque pendant l'opération de transfert.

2.5 Interfaces de communication bas niveau

Les interfaces de communication de bas niveau sont généralement utilisées dans des systèmes à haute performance, dans le but d'inter-connecter les différentes entités (machines, grappes, ...) de la plateforme. Ces interfaces ont l'avantage d'offrir une faible latence et causent moins d'interférences par rapport aux interfaces Ethernet classiques car elles disposent d'un accès direct au matériel. L'échange de données avec le média physique se fait directement depuis l'espace d'adressage de l'usager sans aucun recours au système d'exploitation. Ce dernier n'intervient que dans les phases d'initialisation du périphérique [Aum02]. Parmi ces interfaces, on peut citer : Myrinet, SISCI, GM, etc. Si ce type d'interfaces offre une meilleure performance par rapport aux interfaces Ethernet standard, leur utilisation reste compliquée, ce qui ne correspond pas aux besoins des programmeurs qui ne souhaitent pas rester tributaires d'un code très bas niveau.

2.5.1 Sisci

Software infrastructure for SCI (Sisci) est une interface de communication de bas niveau réservée aux cartes réseau de type SCI (scalable coherent interface) [TGS01]. Sisci fait référence à une technologie à la fois matérielle (carte réseau, switch, ...) et logicielle (driver, interface de programmation). Ce standard a été conçu dans le but d'assurer un service d'interconnexion à haute performance entre les différents nœuds qui composent la plateforme.

L'avantage de la technologie Sisci réside dans son modèle de communication à base de mémoire partagée repartie, dans lequel, l'espace d'adressage globale est représenté par une adresse à 64bit. Le rôle principal de Sisci à ce niveau est de faire en sorte que les opérations de lecture/écriture en mémoire soient réalisées de manière implicite indépendamment de la localisation physique des segments correspondants sur le cluster. Les utilisateurs auront donc l'impression de manipuler des segments de mémoires locales, alors que ces derniers sont réellement hébergées sur d'autres noeuds distants.

2.5.2 BIP

Basic Interface for Parallelism (BIP) est une interface de communication de bas niveau destinée aux réseaux Myrinet [PT98]. Son modèle de communication repose sur un paradigme par échange de messages. Pour cela, BIP offre un certain nombre de fonctionnalités de base pour le transfert de données. Deux modes de transferts sont implémentés : bloquant et non-blocant. Les performances de BIP sont de 1Go/s de bande passante et 4.4μs de latence, pour des paquets de taille de 1096 Octet.

Parmi les aspects innovants de BIP, l'introduction de la notion de messages courts et messages longs. La même fonction BIP est utilisée pour les deux types de message aussi bien en émission qu'en réception. D'un point de vue plus technique, le transfert des messages longs nécessite le déploiement d'un mécanisme de synchronisation à base de rendez-vous entre le l'émetteur et le récepteur, ce qui permet au récepteur de copier les données reçues directement vers le tampon de destination sans avoir recours à une copie intermédiaire.

En ce qui concerne les messages de faible taille, leur transmission se fait sans utilisation de mécanisme de synchronisation. Chaque fois qu'un message est reçu, il est systématiquement stocké dans une files de messages circulaire de manière à ne pas bloquer l'émetteur en cas d'absence de demandes de réception. Le paramètre de configuration BIPSMALLSIZE permet de distinguer les messages courts des messages longs.

```
1#include "bip.h"
2int main(int argc, char* argv[])
3{
4  int iToken;
5  bip_init();
6  if(bip_mynode==0){
7    iToken=1;
8    printf("L'émission commence depuis le token 0");
9    bip_send(1,&iToken,1);
10   bip_recv(&Token,1);
11   printf("Le token est arrivé");
12 }else{
13   bip_recv(&iToken,1);
14   printf("Token%d reçu sur %d", iToken, bip_mynode);
15   bip_send((bip_mynode+1)%bip_numnodes, &iToken,1);
16 }
17 return 0;
18}
```

Figure 2.5 – Exemple de code BIP.

2.5.3 GM

GM est une interface de communication développée par la société Myricom pour la programmation des cartes réseaux Myrinet [HMMR07]. La technologie Myrinet offre une API pour les programmeurs ainsi qu'un driver à installer sur le système d'exploitation hôte. Le principe de GM repose sur le concept de port de communication qui joue un rôle double, d'une part, il permet à l'application de dialoguer avec le périphérique réseau, d'autre part, il décrit une sorte

d'adresse logique qui rend l'application hôte accessible depuis un autre poste distant. Lorsqu'un port est attribué à une application, aucune autre application hôte ne pourra l'utiliser. Le nombre maximum de ports que peut ouvrir une application est de 8 pour GM-1 et 16 pour GM-2. La Figure 2.6 donne un exemple d'organisation des ports dans GM.

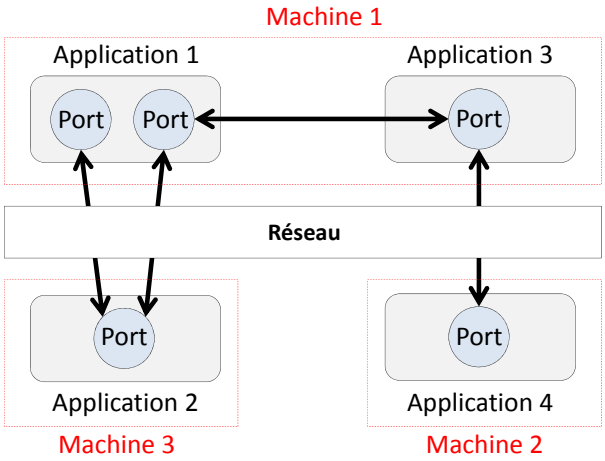

Figure 2.6 – Concept de port de communication dans GM.

Chaque port de communication est composé de trois files ; une file d'émission, une file de réception, et une file d'événements. Ces trois files sont utilisées par les applications lors des opérations de transfert de données. Les zones mémoires associées aux données manipulées peuvent être réservées de deux façons :

- En utilisant des fonctions d'allocation spécifique à la bibliothèque GM, et dans ce cas la préservation de la cohérence de données est assurée par un mécanisme de verrouillage implicite.
- En utilisant des fonctions d'allocation classiques (calloc, malloc, ...), et dans ce cas le verrouillage doit se faire de manière explicite (à la charge du programmeur).

Finalement, il est important de souligner que la l'interface GM dispose d'un mécanisme de tolérance aux pannes, dans le cas où une partie du réseau devient inaccessible.

2.5.4 Papi

Papi (PCI-DDC (PCI-Direct Deposit Component) Application Programming Interface) est une interface de communication spécialement conçue pour les composants PCI-DDC [RD01, RDF00]. Ces composants offrent aux applications la possibilité d'échanger des messages suivant deux modes de transfert ; un mode normal basé sur le concept d'écriture à distance, et dans ce cas l'émetteur doit spécifier aussi bien l'adresse mémoire source que celle de destination, et un mode de transfert réservé aux messages courts. Ce type de messages est généralement utilisé dans la phase d'initialisation d'une application afin de transmettre les adresses mémoire source/destination qui seront manipulées par la suite, raison pour laquelle la taille de ce type de messages est limitée à 8 Octets. L'aspect innovant de l'API Papi concerne habileté de l'administrateur a choisir le niveau de sécurité souhaité. A ce titre, il faut noter que la performance de l'API dépend du mode de sécurité choisi. Trois niveaux de sécurité sont implémentés :

1. NONE : Aucun mode de sécurité n'est activé. Tous les module sont chargés au niveau de l'espace utilisateur, à l'exception de quelques fonctions systèmes utilisées dans les phases d'initialisation.

2. ACCESS : Les modules sont placés au niveau du noyau du système d'exploitation. Ainsi, chaque appel de l'API au niveau de l'application se traduit par un appel système.
3. HIDE : Tout le code est placé au niveau de l'espace noyau, aucun appel système ne devient donc possible.

2.5.5 Gamma

Si la majorité des interfaces de communication citées jusqu'à présent est liée à des technologies matérielles propriétaires, l'interface Gamma (Genoa Active Message Machine) cible des technologies plus standardisées, telles que celles utilisées dans les réseaux Ethernet classiques [CC97]. Cette interface repose sur le paradigme de messages actifs, dans lequel le programmeur aura la possibilité d'associer à chaque port de communication une fonction de traitement, qui est systématiquement déclenchée lors de la réception d'un message.

Gamma a été conçue pour être déployée exclusivement sur des réseaux locaux (LAN) composés d'un ensemble de machines homogènes de type Linux. Elle dispose d'un mécanisme de retransmission dans le cas où des paquets sont perdus. Les performances de Gamma sont de l'ordre de 18.4 μs en latence et de 9.9μs en bande passante.

L'interface Gamma procure un avantage principal par sa parfaite adaptation à la technologie Ethernet standard, ce qui facilite énormément son déploiement au sein des plateformes d'exécution, sans aucun recours à des technologies matérielles particulières.

2.5.6 VIA

Virtual Interface Architecture (VIA) est issue d'une collaboration conjointe entre les sociétés : Microsoft, Compaq et Intel, et sa première version a été publiée en 1997 [vEV98]. L'objectif principal de VIA porte sur la standardisation des interfaces de communication afin de la rendre indépendante du système d'exploitation déployé, du CPU ainsi que du périphérique réseau sous-jacent [DRM+98]. Ce concept est assuré grâce à la notion d'interface virtuelle (Virtual Interface) qui se présente sous forme d'un composant logiciel permettant à l'application d'avoir un accès directe au média afin d'éviter les recopies intermédiaires de mémoire. Trois modes de transfert sont disponibles sous VIA :

– Send/Receive : Dans ce mode, l'application émettrice exécute une opération d'émission alors que l'application réceptrice exécute une opération de réception. De plus, les opérations d'émission et de réception peuvent se dérouler suivant le modèle gather/scatter dont le principe est décrit dans respectivement dans les Figures 2.7 et 2.8

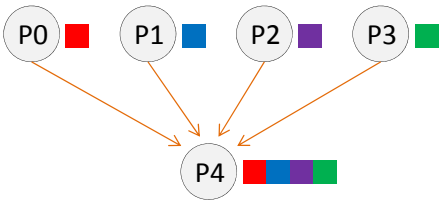

Figure 2.7 – Mode gather.

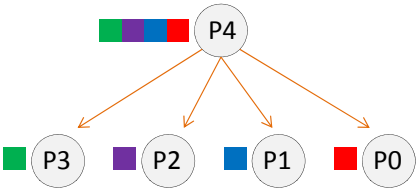

Figure 2.8 – Mode scatter.

– RDMA-Write (Remote Direct Memory Access Write) : Ce mode de transfert permet à l'application source de transférer des données directement sur un tampon de l'application de destination. Pour cela, les deux applications doivent s'entendre sur l'adresse relative au tampon de réception avant de commencer une opération RDMA-Write.

– RDMA-Read (Remote Direct Memory Access Read) : Ce mode de transfert facilite à l'application initiale le rapatriement des données hébergées sur une autre application distante. Ces données sont stockées dans une adresse sur laquelle les deux application se sont déjà entendues.

2.6 Interfaces de communication de niveau intermédiaire

Les interfaces de communication de niveau intermédiaire ont été conçues dans le but de réduire la complexité des interfaces de bas niveau. Ce type d'interfaces offre un niveau d'abstraction plus élevé qui permet d'encapsuler certains contraintes de bas niveau liées à l'architecture sous-jacente tout en assurant un bon niveau de portabilité. Le mécanisme de communication offert par ce type d'interfaces constitue la brique de base sur laquelle s'appuient d'autres exécutifs de niveau plus élevé que nous décrivons dans par la suite.

2.6.1 Madeleine

Madeleine est une bibliothèque de communication conçue spécialement pour les réseaux haute performance comme les grappes ou les grilles de calcul. Elle fournit un environnement de communication efficace et portable [Den03].

Madeleine fait partie de l'environnement de programmation PM2 [Nam01, Ant01], elle se base essentiellement sur le concept de multithreading fourni par le service de Marcel [DNR00], qui fait lui aussi partie de la plate forme PM2. Madeleine offre la communication haute performance [BAN+08] alors que Marcel assure la gestion des threads [Fre07].

En plus, Madeleine est une bibliothèque multi-protocoles dans la mesure où elle permet aux applications d'exploiter plusieurs protocoles et de nombreux réseaux physiques simultanément, et dans sa deuxième version, madeleine est capable de supporter : le protocole SISCI (pour SCI), le protocole BIP (pour Myrinet), TCP, VIA, SBP (pour Ethernet), et sa troisième version intègre un support qui permet l'interconnexion de grappes (grappes de grappes) [Den03].

L'interface de programmation Madeleine fournit un ensemble de primitives qui permet de réaliser les fonctionnalités de base, à savoir, l'envoi et la réception de messages. Chaque message est représenté sous forme d'une séquence de zones mémoires qui font partie de l'espace d'adressage de l'utilisateur. Le tableau suivant résume les fonctions de base de Madeleine [Nam01] :

Nom de la fonction	Description
mad_begin_packing	Connexion et construction du nouveau message
mad_begin_unpacking	Initialisation du canal entrant
mad_end_packing	Fin de construction du nouveau message
mad_end_unpacking	Fin de réception de données
mad_pack	Emballer le nouveau message
mad_unpack	Extraction de données

Tableau 2.1 – Quelques fonctions de base de Madeleine.

La procédure d'envoi de message se déroule en deux étapes :

– le processus émetteur doit tout d'abord déterminer le processus récepteur à l'aide de la primitive (mad_begin_packing), puis, empaqueter les données dans le message et l'envoyer au destinataire avec les deux fonctions (mad_pack et mad_end_packing).

– De son coté, le processus récepteur doit faire l'opération inverse c'est-à-dire initialiser le canal de réception (mad_begin_unpacking), extraire les données reçues (mad_unpcak) et finalement assurer la réception du message à l'aide de la fonction(mad_end_unpacking).

2.6.2 Nexus

Nexus est un exécutif pour les langages parallèles, conçu initialement pour servir de support de communication pour le langage Fortran, avant de devenir une cible pour d'autres compilateurs. Ainsi, de nombreuses technologies l'utilisent en tant que couche de communication : Compositional C++, nPerl, MPICH, etc. Nexus a été finalement utilisé comme plateforme de communication générique dans Globus [Den03].

Nexus se présente sous forme de bibliothèque utilisant la multiprogrammation. Dès le départ, l'accent a été mis sur la haute performance et la portabilité entre milieux hétérogènes, tant au niveau réseaux qu'au niveau des machines. Il constitue une couche de communication entre les langages haut niveau (MPI, CORBA, ...) et les couches basses de communication du système, à savoir : TCP et UDP [JL04].

Nexus peut supporter plusieurs protocoles de communication simultanément, ce qui favorise énormément l'interopérabilité en milieu hétérogène dans la mesure où plusieurs protocoles réseau et bibliothèques de communication peuvent être utilisés dans le même programme sans impacter l'état de fonctionnement du système. L'utilisateur ne se préoccupe pas du protocole réellement utilisé pour le transfert des données puisque Nexus peut sélectionner automatiquement le protocole adéquat en créant une interface générique, et ce, quel que soit le protocole sous-jacent [GP97].

Le mécanisme de fonctionnement de Nexus se base sur la communication point-à-point qui se présente sous forme d'un lien entre un point de départ (startpoint) et un point d'arrivée (endpoint). Plusieurs points de départ peuvent être ainsi liés à un seul point d'arrivée ce qui favorise la multiplication des informations à la sortie. Après l'établissement de la connexion unidirectionnelle entre deux points, le startpoint reste mobile pour se déplacer d'un processus (ou contexte) à un autre. L'établissement des connexions se fait d'une manière complètement dynamique, cependant, un processus ne peut pas avoir plusieurs satartpoint et endpoint à la fois. Ce mécanisme de communication est connue sous le nom « Concept de pointeur global » (ou Global Pointer : GP), par ailleurs, Nexus introduit aussi la notion d'invocation à distance(ou Remote Service Request) qui permet d'appeler des services distants. Les paramètres requis pour appeler un service sont : un pointeur global, un identificateur de service et un tampon de paramètres.

Nexus 4 est la couche de communication de base dans MPICH-G exécutif principal de Globus. Depuis la version Globus 2.0, MPICH-G2 utilise Globus I/O, une fine couche moins souple mais beaucoup plus légère. La lourdeur et la complexité d'utilisation propres à Nexus sont à l'origine de sa chute, au point qu'il a été retiré de Globus depuis la version 3.0.

2.6.3 Active Messages

Développée dans les années 1990 par David Culler à l'université de Berkeley, Active Messages est une plate forme de communication minimaliste de haute performance [vECGS98]. Les message sont calibrés et limités en taille. Elle se base essentiellement sur la notion de messages actifs, c'est-à-dire qu'un traitement bien spécifique est déclenché chaque fois qu'un message arrive. L'appel du service de réception est implicite et n'est pas à la charge des utilisateurs, le but de la plateforme Active Messages étant d'offrir une interface facile à utiliser qui exploite au maximum le matériel mis à disposition. Active Messages a été porté sur plusieurs protocoles ATM, TCP,

Myrinet avec une implémentation originale sur UDP. Aujourd'hui, Active Messages est dépassée par d'autres plateformes plus performantes, mais elle garde le mérite d'avoir exposé la notion de messages actifs qui a été d'ailleurs reprise dans d'autres interfaces de communication.

2.6.4 Illinois Fast Messages

Développée par Andrew Chien à l'université de l'Illinois de Urbana-Champaign [PKC97] au sein du group de recherche CSAG, Fast Messages s'inspire beaucoup du modèle Active Messages (principe de messages actifs) avec quelques petites nuances notamment sur le traitement relatif aux messages de faible taille. Cette adaptation a été réalisée sur la base de certaines constatations concernant la taille typique des messages pour la communication parallèle. Fast Messages a été développée sur TCP, Cray T3D, Myrinet, et a servi de support à l'implémentation de MPI-FM et une version de PVM [Den03].

2.6.5 Panda et Ibis

Développée par l'équipe d'Henri Bal à l'université d'Amsterdam, Panda est une plateforme de communication pour exécutifs parallèles qui supporte la multiprogrammation [NMH+02]. L'objectif principal de Panda [BBH+98] était d'assurer la portabilité par rapport aux réseaux (Principalement TCP et Myrinet). Quelques versions de PVM, MPI ont été portées sur Panda.

Après avoir été longtemps abandonnée, cette plateforme a été reprise par la suite et a donné naissance au projet Ibis, un environnement de communication Java. L'idée principale d'introduire le langage Java est d'apporter davantage de portabilité et de dynamisme tout en restant dans le modèle des RPC (Java RMI) sur lequel elle se base pour faire des invocations de groupe [Den03].

2.7 Interfaces de communication de haut niveau

Les interfaces de communication de haut niveau sont celles qui sont exploitées directement par les programmeurs dans le développement de leurs applications. Cependant, même si ces bibliothèques sont standardisées et assurent une certaine portabilité, elles restent sensibles aux problèmes d'interopérabilité causée par l'hétérogénéité de l'environnement de déploiement (Systèmes d'exploitation). Voici quelques interfaces de communication de haut niveau les plus connues :

2.7.1 Message Passing Interface

Message Passing Interface (MPI) est une bibliothèque destinée à la programmation des applications parallèles sur des systèmes à mémoire distribuée. Elle est basée sur un paradigme d'échange de messages entre processus. Compatible avec les langages C, C++ et Fortran, cette bibliothèque est utilisée aussi bien dans le monde scientifique que dans le monde industriel [Zid10].

MPI est née au début des années 90 et sa première version est sortie en 1993. A cette époque, le seul moyen pour faire du calcul parallèle efficace était d'acquérir une machine parallèle particulière possédant généralement sa propre bibliothèque de communication, souvent incompatible avec les versions antérieures mais jamais avec les machines des concurrents [Mar07a].

MPI permet la programmation des applications hautes performances sur les grappes ainsi que les machines parallèles. Il se base essentiellement sur l'échange de messages entre processus quelle que soit leur localisation sur le réseau [LK05], et il offre de nombreux avantages, parmi lesquels on peut citer :

- Adaptée au développement d'applications parallèles sur des systèmes distribués ou à mémoire partagées [KMS08].
- Portabilité du code.
- Le mode de communication peut être synchrone ou asynchrone, entre deux (communication point à point), ou plusieurs processus (communication collective).

Il faut noter aussi que le standard MPI existe en plusieurs versions [Mar07b]. Après la première version sortie en 1993, plusieurs implémentations se sont succédées (MPICH2 [RYA+12], Lam/MPI, OpenMPI, ...) avec un but commun qui est l'aboutissement à la meilleure implantation MPI aussi bien au niveau des performances qu'à celui des fonctionnalités proposées. Dans [Zid10] une étude approfondie du standard MPI est présentée, les auteurs ont présenté une étude comparative de plusieurs distributions de MPI. Un message est composé d'un ensemble de paquets de données qui transitent d'un processus émetteur à un ou plusieurs processus récepteurs. Outre les données (tableaux, variables scalaires, ...) à transmettre, un message contient des informations complémentaires telles que : L'identificateur de l'émetteur, le type de données à transmettre, la longueur du message, l'identificateur de(s) processus récepteur(s), etc.

Parmi les caractéristiques principales de MPI, toutes ces fonctions se basent sur la notion de communicateurs. Le communicateur par défaut créé lors de l'appel à la fonction MPI_Init, est MPI_COMM_WORLD. Il comprend alors tous les processus exécutant l'application. Le Tableau 2.2 présente quelques fonctions MPI liées à l'environnement de déploiement.

Nom de la fonction	Description
MPI_Init	Initialisation de l'environnement d'exécution
MPI_Finalize	Arrêter l'environnement d'exécution
MPI_Comm_size	Connaître le nombre de processus déployés
MPI_Comm_rank	Récupérer le rang du processus courant

Tableau 2.2 – Quelques fonctions de base de l'interface MPI.

MPI met à disposition des développeurs certains types de bases prédéfinis, mais ils peuvent toujours créer leurs propres types dérivés qui correspondent aux besoins fonctionnels de leurs applications. Le tableau 2.3 décrit les principaux types de base utilisés :

Type MPI	Type c, c++
MPI_CHAR	signed char
MPI_SHORT	signed short
MPI_INT	signed int
MPI_LONG	signed long int
MPI_FLOAT	float
MPI_DOUBLE	double
MPI_LONG_DOUBLE	long double

Tableau 2.3 – Quelques types de base de MPI.

La communication entre processus se fait par échange de messages. Pour cela, MPI offre un certain nombre de fonctions qui assurent le transfert d'une information d'un processus émetteur vers le(s) processus récepteur(s). La liste de ces fonctions est longue, on se contente de quelques unes. Leurs descriptions sont données par le Tableau 2.4

- int MPI_Send (void *buf, int count, MPI_Datatype datatype, int dest,int tag, MPI_Comm comm)
- int MPI_Recv (void *buf, int count, MPI_Datatype datatype, int source,int tag, MPI_Comm comm, MPI_tatus *status)
- int MPI_Bcast (void *buf, int count, MPI_Datatype datatype, int root, MPI_Comm comm)

Nom du champ	Description du champ
buf	Représente l'adresse du buffer d'émission pour MPI_Send et buf de réception pour MPI_Recv.
count	Représente le nombre d'éléments à envoyer ou à recevoir.
datatype	Type de données à envoyer ou à recevoir.
Source	Indice des processus source.
destination	indice des processus destinataires.
tag	Identificateur du transfert.
comm	Communicateur (groupe de processus) impliqué dans la communication.
status	Objet permettant de récupérer le compte rendu de l'opération (succès/échec).
root	Rang du processus qui va effectuer le broadcast.

Tableau 2.4 – Description des paramètres.

Le code suivant décrit les trois grandes parties d'un programme MPI qui sont : La partie initialisation, la partie traitement et la partie libération de ressources.

```
1#include <mpi.h>
2int main(int argc, char* argv[]) {
3  int rang=0, p;
4  MPI_Init(&argc,&argv); /* Initialisation de l'environnement*/
5  MPI_Comm_size(MPI_COMM_WORLD, &p); // Nombre total des processus lancés
6  MPI_Comm_rank(MPI_COMM_WORLD,&rang); // Numéro du processus courant
7  switch (rang) {
8    case 0 : /* Code du processus 0*/ break;
9    case 1 : /* Code du processus 1*/ break;
10   default : /* Autre traitement */ break;
11 }
12 MPI_Finalize(); // Fermeture propre de l'environnement MPI
13 return 0 ;
14}
```

Figure 2.9 – Architecture d'un programme MPI.

2.7.2 Bibliothèque ProActive

ProActive est un environnement de développement sur grille qui se présente sous forme d'une bibliothèque 100% Java, permettant la programmation de calculs parallèles, distribués et concurrents ; c'est aussi un middleware de grille [CL08]. Cette bibliothèque a été développée par l'INRIA de Sophia-Antipolis (la version 1 est sortie en 2001) dans le but de fournir une API complète

simplifiant la programmation d'applications distribuées sur le réseau local, sur un cluster ou sur des grilles. ProActive est construit avec des classes Java standards et n'exige donc aucun changement de la machine virtuelle. Il utilise cependant un class loader spécifique. ProActive est open source et fait partie de la communauté ObjectWeb [Car].

ProActive se base essentiellement sur la notion d'objet actif que l'on qualifie d'unité de base de distribution d'une application ProActive ; il peut être créé sur n'importe quelle machine hôte et son activité et sa localisation (locale ou distante) sont complètement transparents pour les programmeurs qui le manipulent de la même manière qu'un objet passif standard.

Un objet actif a son propre file de contrôle (voir Figure 2.10), sa propre mémoire locale, ainsi qu'un gestionnaire d'appels de méthodes qui assure le traitement des demandes. Contrairement à Java standard, le programmeur en ProActive n'a pas à manipuler explicitement les threads pour gérer les échanges entre les différents objets actifs car ce service et nativement assuré par la couche de communication de ProActive.

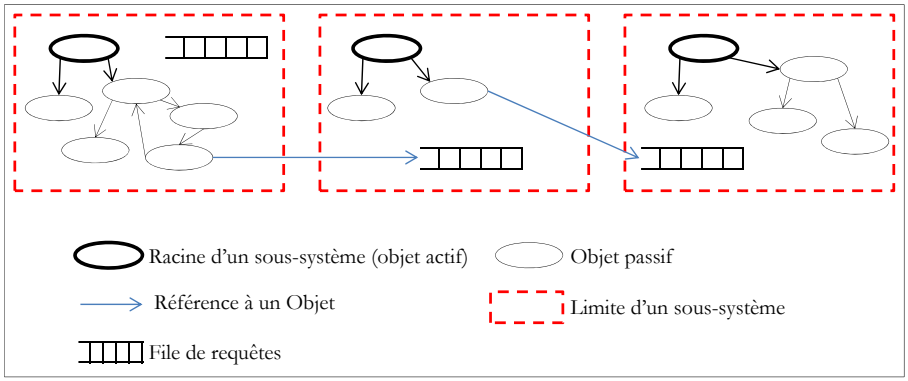

Figure 2.10 – Graphe d'objets dans ProActive.

En ProActive, une application distribuée se présente sous forme d'un ensemble d'objets actifs répartis sur les différentes machines de la plateforme, coopérant les uns avec les autres selon les contraintes decrites ci-dessous. Ces contraintes ont des conséquences importantes sur la topologie des applications et sur la procédure de communication entre les sous-systèmes.

- Chaque sous-système a un point d'entrée unique ; sa racine qui est un objet actif. Les autres objets sont des objets passifs. L'objet actif d'un sous système ne peut exécuter que ses propres méthodes (celles de l'objet actif lui-même et celles des objets passifs lui appartenant).
- Aucun mécanisme de partage d'objets passifs entre sous-systèmes n'est intégré à ProActive.
- Parmi les objets qui composent un sous-système, seul l'objet actif est accessible à distance.
- Tous les objets, actifs ou passifs, peuvent avoir une référence sur un objet actif distant.
- Si un objet O1 a une référence sur un objet passif O2, O1 et O2 appartiennent alors au même sous-système.

Les objets actifs sont déployés sur des composants logiciels appelés Nœuds qui jouent le rôle de conteneurs. Pour rendre un Nœud accessible depuis un poste distant, il faut le référencer au niveau du registre RMI. Voici un exemple d'une adresse URL d'un nœud « rmi ://IP/nom_du_noeud ».

2.7.3 CORBA

Corba est un middleware standard, d'architecture à base d'objets distribués. L'acronyme Corba signifie « Common Object Request Broquer Architecture » [BCM04], c'est une norme dé-

finie par l'OMG (Object Management Group), un véritable consortium entre universitaires et industriels dans le but d'aboutir à une normalisation des technologies orientées objet, résolvant les problèmes d'intégration d'applications [Hen08]. Le point fort de Corba réside dans son indépendance vis-à-vis du langages de programmation, du système d'exploitation déployé, et de l'architecture matérielle sous-jacente [Den03].

Le cœur de l'architecture Corba est l'ORB (Object Request Broker) qui assure la transmission ainsi que l'acheminement des requêtes qui circulent entre les différents composants distribués sur le réseau. En effet, l'utilisateur peut accéder à n'importe quel objet Corba sans se préoccuper des couches basses du système. Le bus Corba assure : l'interception les requêtes, la localisation d'objet distant, l'invocation des méthodes à distance avec les bons paramètres d'appel, ainsi que l'acheminement des valeurs de retour. Enfin, il est à noter que certains travaux de recherche ont abouti à la mise en œuvre du concept d'objets Corba parallèles qui résulte d'une combinaison du standard MPI avec celui de Corba.

2.7.4 Remote Method Invocation

Remote methode incovation (RMI), est une API JAVA offrant aux usagers la possibilité de créer ainsi que de manipuler des objets distants (c'est-à-dire des objets instanciés sur une autre machine virtuelle qui tourne sur l'une des machines connectées au réseau) de manière complètement transparente. L'utilisateur aura ainsi l'impression de manipuler des objets qui résident sur sa machine hôte [MvNV+99].

RMI est basé sur la notion de client-serveur. Le serveur offre au client la possibilité d'invoquer à distance les méthodes d'un objet qu'il instancie, ce qui nécessite la présence de deux machines virtuelles, l'une du coté du client, et l'autre du coté du serveur. Le standard RMI est fondé sur la notion d'interface qui permet de masquer l'implémentation sous-jacente lors de la création d'un objet distant. Du côté des programmeurs, ces derniers ne disposent pas d'une référence directe sur l'objet distant qui reste accessible uniquement à travers une référence intermédiaire locale, qui assure la communication avec l'objet distant de manière complètement transparente [HX09].

D'un point de vue plus technique, RMI se base sur le protocole standard TCP/IP pour assurer le transfert de données à travers le réseau. Le port de service utilisé par défaut est le 1099. A partir de Java 2 version 1.3, les communications entre le serveur et les clients se font via le protocole RMI-IIOP (Internet Inter-Orb Protocol), un protocole standardisé par l'OMG et utilisé dans la technologie CORBA. La couche communication du protocole RMI repose sur les concepts de stub (implémenté par l'application cliente) et du skeleton (implémenté par l'application serveur). La couche de référence RRL (Remote Reference Layer) est intégrée au package java.rmi.Naming, et permet de référencer les objets déployés à distance (voir Figure 2.11).

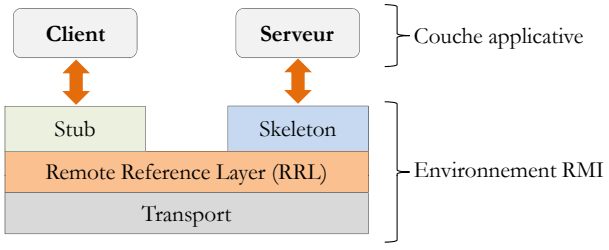

Figure 2.11 – Notion de stub et skeleton.

Tout accès à un objet depuis l'extérieur n'est rendu possible que s'il est préalablement inscrit dans un registre particulier appelé registre RMI (RMI Registry en Anglais).

2.7.5 PadicoTm

PadicoTm est une plateforme de recherche qui permet l'intégration de plusieurs middlewares (MPI, JVM, CORBA, ...) au sein du même processus [Den02]. De plus, PadicoTm gère le conflit d'accès aux ressources partagées (réseau, processeurs) afin d'assurer une cohabitation efficace et performante des différents middlewares déployés. Les programmeurs ne sont plus contraints d'utiliser un seul paradigme (Appel de fonction/procédure à distance, échange de messages) de communication dans leurs applications. Si un programmeur a besoin d'exécuter une fonction/-procédure distante (hébergée sur un processus distant) dans un programme utilisant MPI entant que interface de communication, il sera obligé de simuler un mécanisme d'appel de fonction à distance au dessus du standard MPI puisque ce dernier ne support que le paradigme d'échange de messages. PadicoTM permet de surmonter cet obstacle, en offrant aux développeurs la possibilité de combiner plusieurs paradigmes de communication à la fois. L'architecture de PadicoTM est présentée dans la Figure 2.12

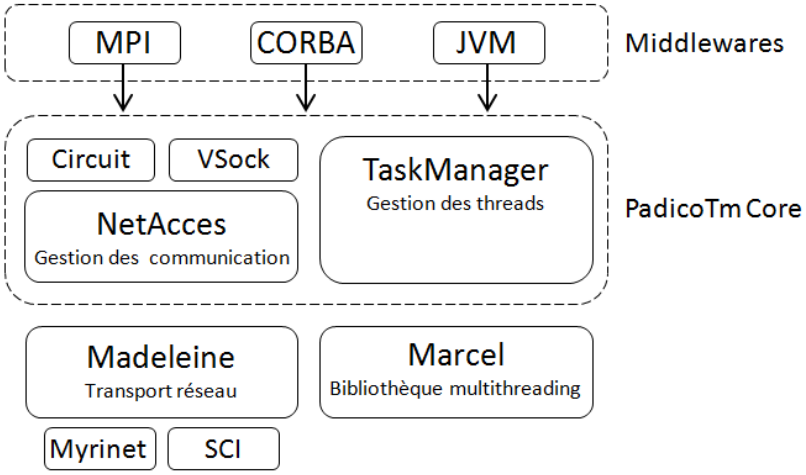

Figure 2.12 – Architecture globale de PadicoTm.

- Vsock : VSock est la version modifiée de l'interface standard Socket. C'est un module qui assure un transfert de datagrammes en mode zéro copie (sans recopie intermédiaire). De plus, Vsock sélectionne automatiquement le protocole à utiliser (NetAcces/Madeleine ou TCP/IP) en fonction des caractéristiques matérielles sous-jacentes.
- NetAcces : Ce module assure la gestion des communications sur le réseau et donne accès au service de transport de Madeleine, il met en œuvre un mécanisme de procédures de rappel, qui permet de décrire les traitements appropriés aux différents messages reçus.
- Circuit : Dans PadicoTM, la notion de groupe reflète un sous-ensemble de nœuds de la topologie. Un circuit est un canal de communication NetAcces/Madeleine restreint à un groupe.
- TaskManger : Ce module apporte une encapsulation de la bibliothèque de multithreading Marcel, et assure la gestion et la synchronisation des différents threads d'exécution.

2.7.6 PVM

Parallel virtual Machine (PVM) est l'une des plus anciennes bibliothèques de communication, et peut être considérée comme étant l'ancêtre de MPI. PVM est basée sur un paradigme

d'échange de messages entre processus, à travers une collection de fonctions de base pour l'émission/réception de données [Sun90]. L'environnement d'exécution de PVM définit une sorte de machine virtuelle constituée par un ensemble de processus, physiquement répartis sur toutes les machines de l'infrastructure. PVM est disponible en version gratuite pour les deux langages de programmation C et Fortran, et met à disposition des programmeurs un ensemble de fonctionnalités permettant : l'interconnexion entre machines, le lancement/arrêt de processus distants, l'échange de messages entre processus, etc... D'un point de vue plus technique, la couche de communication de PVM est assurée par un démon particulier (processus de service) qui doit être installé sur chaque machine de la plateforme afin de la rendre accessible à distance, et dont le rôle principal consiste à garantir l'acheminement de tous les messages qui transitent sur le réseau.

Parmi les avantages de PVM, on peut citer son habileté à supporter l'hétérogénéité des environnements d'exécution, mais aussi et surtout, sa capacité à rajouter ou supprimer dynamiquement des nœds pendant le déploiement d'une session. Cependant, son surcoût de *buffering* et son inadaptation au mode de communication entre threads [LK05] rabaissent nettement son mérite.

2.7.7 Athapascan-0a

Athapascan-0a est une bibliothèque de communication développée dans le cadre du projet APACHE. Elle repose essentiellement sur la technologie PVM et se présente sous forme d'une interface de programmation permettant la communication entre processus légers. Elle a été conçue dans le but de remédier à certaines limites du standard PVM notamment celle concernant son incapacité à gérer des communication au niveau tâches.

Le paradigme de programmation d'Athapascan repose sur l'appel de procédures à distance qui peut se faire depuis un processus léger, les communications sont effectuées de manière asynchrone, avec transmission des arguments au moment de l'appel, et synchronisation par rapport au résultat en retour, autant d'aspects innovants de ce modèle de communication dont l'intérêt principal consiste à donner au thread appelant la possibilité de continuer son exécution tant que la procédure distante n'a pas encore fini le traitement.

La synchronisation par rapport aux valeurs de retour se fait par scrutation selon les deux modes suivants :

- Bloquant : Le processus appelant est alors bloqué tant que la réponse finale n'a pas encore été reçue.
- Non-bloquant : L'indisponibilité de la réponse finale ne bloquera pas l'exécution de la tâche appelante.

La bibliothèque Athapascan est souvent critiquée pour son unique mécanisme de mise en œuvre du parallélisme qui est basé sur un paradigme d'appel de procédures à distance, qui engendre une création systémique de threads au niveau de la tâche distante. Le programmeur n'a aucune primitive de communication ou de synchronisation explicite à sa disposition, réduisant ainsi, ses possibilités en termes de communication.

2.8 Tableau récapitulatif

Nous présentons dans cette partie un résumé sur l'ensemble des bibliothèques décrites dans ce chapitre sous forme de tableaux récapitulatifs dans lesquels nous retenons leurs aspects les plus importants à savoir : Langage de programmation, objectif principal, interopérabilité et hétérogénéité, sécurité, portabilité, sûreté, mode de communication, Avantages et inconvénients.

Tableau 2.5 – Tableau récapitulatif de MPI, RMI et CORBA.

Critère \ Interface	MPI	RMI	CORBA
Niveau	Haut niveau	Haut niveau	Haut niveau
Langage de programmation	• C/C++ • FORTRAN • Python • Java • OCaml	• Java	• C/C++ • Smalltalk • Java • Ada • Cobol
Objectif	• Interface de communication pour échange de messages entre processus	• API Java pour l'invocation de méthodes à distance	• Middleware pour la programmation à base d'objets distribués
Hétérogénéité des plateformes	• Adaptée aux plateformes homogènes. Le déploiement sur une plateforme hétérogène nécessite l'installation d'une plateforme générique comme celle de Globus	• Supporte l'hétérogénéité des plateformes	• Tolère l'hétérogénéité des OS, des langages et procure un bon niveau d'interopérabilité
Sécurité et authentification	• Tunnel SSH SecretWord (mot de passe global)	• Accordée par le RMI Security Manager sur la base d'un ficher policy éditable par le policytool	• CORBA Security Service (CSIv2, ATLAS)
Portabilité	•Code portable	•Code portable	• Dépend du langage choisi
Sûreté	• Niveau de sureté faible. La reprise après défaillance est assez compliquée	• Absence de mécanisme implicite assurant la fiabilité	• Prise en charge par les services de transaction et de persistance de Corba
Mode de communication	• Synchrone ou asynchrone	• Synchrone ou asynchrone (ARMI)	• Synchrone ou asynchrone
Avantages	• Bibliothèque standard • Simplicité •Portabilité	• Standard • Simplicité • Accorde l'interopérabilité aux milieux hétérogènes (SE, réseaux, ...)	• Garantie d'un niveau élevé d'interopérabilité entre milieux hétérogènes • Portabilité du code grâce au langage IDL
inconvénients	• Absence de support explicite pour la multiprogrammation légère • Flexibilité moyenne	• Technologie purement Java • Surcoût de sérialisation des paramètres	• Lourdeur • Interopérabilité entre plusieurs bus Corba non assurée • Complexité

Tableau 2.6 – Tableau récapitulatif de PVM, Globus et Athapascan.

Interface / Critère	PVM	Globus	Athapascan
Niveau	• Haut niveau	• Haut niveau	• Haut niveau
Langage de programmation	• C/C++ • FORTRAN	• C/C++ • Java • Python	• C/C++
Objectif	• Bibliothèque de communication par échange de messages	• plateforme générique pour la programmation sur grilles	• Noyau exécutif pour la programmation parallèle
Interopérabilité et hétérogénéité	• S'oppose à l'hétérogénéité des plateformes	• Supporte l'hétérogénéité des plateformes et assure un niveau élevé d'interopérabilité	• Rejette l'hétérogénéité des plateformes
Sécurité et authentification	• Protocol RSH	• Sécurité exercée par le service GSI basé à la fois sur les certificats et les clés de cryptage	• Aucune communication
Portabilité	• Code portable	• Code portable	• Code portable
Sureté	• Détection de défaillance grâce au daemon pvmd	• Environnement de déploiement fiable assuré par les services (GRAM, GRIP, MDS, ...)	• Absence de mécanisme pour la reprise après une défaillance
Mode de communication	• Synchrone ou Asynchrone	• Synchrone ou Asynchrone	• Synchrone ou Asynchrone
Avantages	• Gestion dynamique des processus lourds • Robustesse notable garantie par son modèle à base d'échange de messages	• Machines indépendant • Garantit des services de base comme : le transfert de données(GridFTP), la gestion de ressources (GRAM), etc.	• Bibliothèque pour la multiprogrammation légère • Appel de procédure à distance depuis des threads
Inconvénients	• Modèle peu adapté au calcul fin • Ne supporte pas l'hétérogénéité des OS • Projet abandonné	• Lourdeur • Complexité	• Dépendance vis-à-vis de MPI • Mécanisme de scrutation pour suivre la progression des communications

Tableau 2.7 – Tableau récapitulatif de Active Message, Illinois Fast Messages et Nexus.

Critère \ Interface	Active Message	Illinois Fast Messages	Nexus
Niveau	• Couche basse	• Couche basse	• Couche intermédiaire
Objectif	• Bibliothèque de communication	• Bibliothèque de communication	• Exécutif qui prend en charge la communication à haute performance pour les environnements parallèles
Compatibilité	• Compatible avec tous les systèmes qui tolèrent le protocole TCP ou UDP	• Compatible ave tous les systèmes qui supportent TCP ou UDP	• Aucune communication
Hétérogénéité/ interopérabilité	• L'interopérabilité entre les systèmes hétérogènes est un point sensible puisqu'elle dépend du type de matériel	• l'interopérabilité est effective grâce à l'encapsulation par des protocoles standards comme TCP ou UDP	• Aucune communication
Avantages	• Pas de recopies intermédiaire de mémoire • Meilleure adaptation aux messages de petite taille • Accès direct à l'interface du matériel	• Permet la communication sur les WAN • Pas de recopie supplémentaire vers l'espace système	• Offre une interface uniforme qui favorise la sélection automatique du protocole adéquat • Support explicite pour la multiprogrammation légère
Inconvénients	• Inexistence de mécanisme de gestion de pannes • Format de message spécifique incompatible avec celui du standard TCP • Valable uniquement sur un LAN • Forte dépendance de l'interface du matériel	• Absence de mécanisme pour la reprise après défaillance • L'interface Socket ne peut pas être partagée entre plusieurs processus(Les appels *fork* ne sont pas supportés)	• Inadaptation aux échanges de petite taille • lourdeur • Complexité • Manque de flexibilité (les canaux de communication sont ouverts dès l'initialisation)

Tableau 2.8 – Tableau récapitulatif de Madeleine, PM2 et Padico TM.

Critère \ Interface	Madeleine	PM2	Padico TM
Niveau	• Couche intermédiaire	• Couche intermédiaire	• Couche intermédiaire
Objectif	• Bibliothèque de communication à haute performance	• Environnement générique pour la programmation parallèle	• Plate-forme de déploiement pour les grilles
Compatibilité	• Incompatibilité avec les systèmes de la famille Windows	• Inadaptation au système Windows	• Tolère l'hétérogénéité des plate-formes
Hétérogénéité et interopérabilité	• Assure un bon niveau d'interopérabilité entre réseaux hétérogènes	• Offre un bon niveau d'interopérabilité entre réseaux hétérogènes	• Procure un niveau d'interopérabilité assez élevé entre réseaux hétérogènes
Avantages	• Elle peut supporter plusieurs technologies simultanément (VIA, SBP, ...)	• Offre un support explicite à la multiprogrammation légère • Pas de copies supplémentaires des données	• Assure le déploiement de plusieurs exécutifs variés (Corba, MPI, JXTA, ...)
Inconvénients	• Problème de compatibilité avec les systèmes Windows	• Incompatibilité avec les systèmes Windows	• Lourdeur

Tableau 2.9 – Tableau récapitulatif de PROACTIVE, RPC et Panda.

Interface / Critère	PROACTIVE	RPC	Panda
Niveau	• Haut niveau	• Haut niveau	• Couche intermédiaire
Langage de programmation	• Java	• C/C++	
Objectif	• Bibliothèque Java pour la programmation parallèle intégrant le concept de groupe	• Appel de procédure à distance	• Plateforme de portabilité pour les exécutifs parallèles (Orca système)
Hétérogénéité et interopérabilité	• Supporte l'hétérogénéité des plateformes	• Tolère l'hétérogénéité des plateformes et du réseau de communication	• Aucune communication
Sécurité	• Tunnel SSH et fichier de configuration pour la politique de sécurité (fichier Policy)	• Clé restrictRemoteClients	• Aucune communication
Portabilité Fiabilité	• Code portable Absence de mécanisme implicite pour assurer la fiabilité	• Code portable peu fiable dans la mesure où il n'offre aucun moyen de reprise après défaillance	• Aucune communication
Mode de communication	Synchrone ou Asynchrone	Synchrone ou Asynchrone	• Aucune communication
Avantages	• Notion de migration • Bénéficier des avantages de la technologie Java • Notion d'objet futur • Notion de Groupe	• Offre un modèle de programmation à base d'objets distribués	Support explicite pour la multiprogrammation légère • Intègre le concept d'échange en mode multicast pour la communication efficace de groupe • Possibilité de fonctionner avec ou sans le mode zéro-copy
Inconvénients	• Dépendance vis-à-vis du protocole RMI	• Faible niveau de sécurité Dépendance vis-à-vis du langage de programmation	• Aucune communication

2.9 Synthèse

Nous avons présenté dans ce chapitre toute une gamme de bibliothèques de communication. D'une manière générale, ces bibliothèques peuvent être classées selon deux critères qui sont : le niveau d'interaction (Intermédiaire, Applicatif, ...), et le paradigme de programmation adopté (échange de messages, appel de procédure à distance, ...). Chaque interface a ses propres avantages et inconvénients. Néanmoins, la règle générale est la même : plus le niveau d'implémentation est bas, plus on gagne en performance et plus la gestion est ardue ; inversement, dès que le niveau d'interaction est élevé (MPI, PVM, ...) sauf que l'on gagne en termes de simplicité de mise en œuvre et de développement.

En principe, l'accès au périphérique réseau (carte réseau) est le privilège exclusif du système d'exploitation qui joue le rôle d'intermédiaire entre la couche applicative et le périphérique. Les opérations de base telles que l'émission ou la réception de données sur le réseau sont généralement réalisées par le biais d'un ensemble de fonctions systèmes de base (read/recv, write/send, ...).

Pour optimiser le temps de transmission sur le réseau, certaines interfaces de programmation (SISCI, BIP, ...) tentent de piloter directement les périphériques réseau sans passer par le système d'exploitation ce qui permet de minimiser le nombre de copies intermédiaires de mémoire (Depuis la mémoire système vers celle de l'application et inversement). L'accès et le contrôle de la carte réseau se feront alors directement depuis l'espace utilisateur, pour cela, certaines zones mémoires propres au périphérique doivent être référencées au niveau de la couche applicative.

Il faut signaler que dans le cas du mode d'accès direct, soit en lecture ou en écriture le système perd tout contrôle sur le périphérique et les accès ne sont plus protégés. Par conséquent, la cohérence des données en mémoire n'est plus garantie avec le risque potentiel de voir le système écraser des données utilisateurs lors du transferts des données sur le réseau.

Il faut signaler aussi que des interfaces de communication propriétaires telles que : SISCI, BIP, et GM offrent un bon niveau de performance grâce à leur débit élevé et leur faible latence, elles reposent sur des technologies matérielles et logicielles spécifiques dont l'utilisation est souvent réservée à des plateformes de communication à haute performance. Cette contrainte n'est pas toujours applicable sur les plateformes d'intégration que nous étudions, qui sont généralement composées d'un ensemble de machines reposant sur des technologies matérielles et logicielles standards (Réseau Ethernet, protocole TCP/IP, ...). La solution que nous devons proposer doit donc s'aligner sur les grands standards du marché.

Nous constatons aussi à travers ce tour d'horizon sur les interfaces de communication que le problème d'échange et de transmission de données entre thread a été rarement abordé, les seules unités communicantes considérées sont généralement des processus lourds. Pourtant, dans un contexte de programmation parallèle, les programmeurs sont souvent amenés à manipuler des unités de traitement de niveau de granularité plus fine que sont les threads, comme c'est le cas par exemple des bibliothèques MPI ou PVM, qui ont été conçues dans le but d'offrir une interface de communication entre processus lourds. L'unité processus étant la seule unité d'adressage, il est donc difficile de prendre en compte des communications au niveau threads. Pour cela, les programmeurs sont contraints de rajouter une nouvelle fonctionnalité au niveau des processus leurs permettant de dispatcher les messages sur les threads fils. Ces interfaces offrent donc peu d'éléments de réponse à notre problématique de base qui concerne l'optimisation des temps de transfert de données, dans une architecture d'intégration d'applications massivement parallèle, composée essentiellement, par un grand nombre de threads de traitement.

2.10 Discussion

Nous exposons dans cette section une analyse critique des différentes technologies présentées dans cet état de l'art, dans le but d'élucider leurs insuffisances au regard des besoins recensés dans notre cahier de charges. Deux aspects sont pris en compte par notre étude : fonctionnel et technique.

Du point de vue fonctionnel, le modèle de communication du BA nécessite un paradigme d'échange par passage de messages car il n'introduit pas de couplage fort entre applications. Cette contrainte nous fait éliminer un bon nombre de technologies qui ne répondent pas à cette exigence, à savoir :

- *CORBA* : middleware orienté objets distribués ;
- *RMI* : appel de méthodes à distance ;
- *RPC* : appel de procédures à distance ;
- *ProActive* : appel de méthodes sur des objets actifs distants ;
- *RMI* : appel de procédures à distance depuis des threads ;

Du point de vue technique, bien que performantes et basées sur un modèle de communication par échange de messages, des bibliothèques comme PAPI, VIA, ou encore Gamma, ne peuvent servir de fondations technologiques à la couche de communication du BA, et ce, pour plusieurs raisons parmi lesquelles nous citons :

- **Couverture réseau** : ces technologies sont exclusivement dédiées aux réseaux LAN (sigle anglais de Local Area Network). Par conséquent, elles ne satisfont pas aux critères de couverture réseau de l'architecture orientée services qu'est la nôtre, et dans laquelle, plusieurs applications peuvent interagir à l'échelle planétaire, c'est-à-dire à l'échelle d'un réseau WAN (acronyme anglais de Wide Area Network).
- **Standardisation du matériel** : d'un point de vue technique, ces bibliothèques ne sont disponibles que sur des technologies matérielles spécifiques généralement utilisées dans les plateformes de calcul à haute performance. Elle restent donc peu répandues dans les infrastructures grand public comme celles des entreprises, ou autres organismes, ce qui va à l'encontre de notre ambition d'aboutir à un déploiement à large échelle du BA.
- **Portabilité** : l'architecture du BA est composée d'un ensemble de machines qui ne sont pas forcement homogènes, d'où l'importance du critère de portabilité. Cette spécificité pose problème pour certaines bibliothèques comme Madeleine qui ne tolère pas l'hétérogénéité des plateformes.

Nous nous retrouvons avec seulement deux bibliothèques : PVM et MPI, deux technologies assez connues dans le domaine du HPC. Bien qu'elles aient été élaborées à deux périodes distantes (1989 pour PVM et 1993 pour MPI), elles ont plusieurs caractéristiques communes, notamment au niveau du principe de fonctionnement.

Pour ne pas rentrer dans une étude comparative infructueuse de ces deux interfaces, nous avons choisi d'aborder uniquement les caractéristiques qui les rendent déficientes au vue de nos prévisions. Ainsi, nous avons considéré les points suivants :

- **Modèle de déploiement de MPI** : Une fois la phase de déploiement entamée, plusieurs instance d'un même processus sont lancées simultanément sur le communicateur. Au cours de cette exécution, il n'existe plus aucun moyen pour intégrer de nouveaux processus. Ce mode de déploiement plutôt statique, ne correspond pas à celui de notre architecture, dans lequel, l'intégration des applications reste dynamique, tout au long de la phase d'exécution.
- **Unité d'adressage** : MPI et PVM utilisent le rang des processus comme unité d'adressage par défaut dans le communicateur. Cette spécificité constitue une source d'incompatibilité par rapport à notre plateforme, dans laquelle, les unités logicielles de traitement sont

majoritairement représentées par des threads. D'un point de vue pratique, rien n'empêche le programmeur en PVM ou MPI d'exprimer le parallélisme dans ses processus en les parcellisant en un ensemble de threads. Tant que les traitements assignés aux threads sont exécutés localement, aucun problème ne se pose, par contre, si plusieurs threads d'un même processus père communiquent simultanément avec d'autres processus/threads distants, ils auront tous la même adresse qui est le rang de leur processus père dans le communicateur. Par conséquent, les messages reçus en retour risquent de ne pas être acheminés vers leurs bonnes destinations (thread à l'origine de la demande).

- **Taille de la partie de traitement effective** : si nous nous référons à la Figure 2.9, nous constatons que l'architecture d'un programme MPI ou PVM est composée de plusieurs parties, chacune d'elles étant exécutée par un processus bien déterminé, et ce, en fonction de son rang dans le communicateur. Cette approche présente un inconvénient majeur : en effet, les processus sont entièrement chargés en mémoire alors que la partie réellement exécutée par chacun d'entre eux ne représente qu'un fragment du programme.

Toute cette argumentation justifie notre choix de mettre en œuvre notre propre mécanisme de communication entre tâches réparties. Son mode de fonctionnement est décrit en détails dans le chapitre 5 de la partie II.

Deuxième partie

Le Bus Applicatif (BA)

1

Architecture du Bus Applicatif

Sommaire

1.1 Introduction

En informatique, le terme *Bus* fait généralement référence au Bus-PCI (Peripheral Component Interconnect) qui connecte les différentes cartes d'extension (carte réseau, carte graphique, ...) sur la carte mère d'un ordinateur. Le Bus-PCI tout aussi bien que le Bus Applicatif (BA) apportent la logistique requise au bon fonctionnement aux périphériques pour le premier et aux applications impliquées dans un processus métier pour le second.

Nous consacrons cette section à la présentation de l'architecture du BA ainsi que ses principaux composants. Nous nous intéressons aussi aux aspects fonctionnels et techniques fondamentaux du BA tout en soulignant ses caractères innovants.

1.2 Définition

Dans sa définition la plus aboutie, le bus applicatif (BA) est une architecture d'intégration d'applications adaptée aux traitements massivement parallèles. Son rôle principal consiste à assu-

rer l'acheminement de toutes les données circulant entre les acteurs de toutes les configurations interconnectées vers leurs destinations. Le BA facilite donc le fonctionnement collaboratif et offre un système d'échange performant et sécurisé entre les différents acteurs impliqués dans un processus de traitement d'informations donné [AB12].

Nous désignons ici par le terme *acteur* toute entité capable d'interagir avec le BA en vue d'offrir ou de bénéficier d'un service, il peut s'agir d'une application, d'une tâche, d'une interface graphique, d'une ligne d'acquisition temps réel, d'un périphérique E/S, etc. Si nous faisons abstraction du domaine métier exercé, et nous nous intéressons purement aux aspects d'intégration, nous pouvons regrouper toutes ces instances d'acteurs sous le terme *Unité Logicielle de Traitement* (ULT), avec une nette distinction entre les unités matérielles de traitement (Processeurs, cœurs, ...) et les unités logicielles de traitement (processus, threads).

Le BA assure donc le fonctionnement collaboratif des différentes ULTs réparties sur l'ensemble des machines de la plateforme indépendamment de leurs localisations sur le réseau. Il fonctionnera sous protocole standard TCP/IP de préférence via des ports facilement admis par les pare-feu, serveurs de domaines et passerelles classiques. L'utilisation du standard TCP/IP facilitera son déploiement sur une large gamme d'infrastructures.

Le fonctionnement collaboratif des ULTs engendre des besoins en termes d'échange, de partage de données et de synchronisation. Pour répondre à ces nécessités, nous avons fait appel au standard POSIX-IPC (Inter Process Communication de la norme POSIX) qui offre tout un ensemble de dispositifs d'interaction entre tâches, notamment, son fameux mécanisme de mémoires partagées qui constitue un outil à la fois pratique et efficace pour faire communiquer des tâches/applications entre elles, il en a donc été déduit l'excellente faisabilité de la plateforme du bus sur cette base.

Les performances que nous exigeons de notre bus et dont il devra faire preuve nécessitent de notre part, l'utilisation d'un langage de programmation efficace d'où notre choix du C/C++ pour tout système admettant un compilateur adapté à ce langage.

Ce bus applicatif sera capable, selon nos objectifs de prendre en charge les liaisons du type temps réel nécessaires à certains dispositifs tels que la commande de chaînes de production, les périphériques, et dans ce cas, le BA offre une passerelle entre le monde industriel et celui de la gestion, par exemple pour alimenter en temps réel un tableau décrivant l'état d'évolution de la production ou d'un stock.

Des composants logiciels de connexion appelés *terminaux* assurent la connexion entre le BA et les ULTs et délimitent son périmètre d'accès [BBA13a]. Un Terminal de Bus Applicatif (TBA) est une sorte de case d'accueil disposant des ressources IPC nécessaires aux fonctionnement des ULTs. Quant à la couche de communication réseau, elle est assurée par un composant logiciel nommé multiplexeur du bus applicatif (MBA) dont le fonctionnement est détaillé dans le Chapitre 5 de la Partie II.

Le BA met à disposition des programmeurs une API qui leur facilite le développement d'applications parallèles distribuées. Le périmètre d'accès à l'API est externe au BA, mais son utilisation reste une contrainte technique pour les programmeurs. L'API fournit donc un niveau d'abstraction élevé qui lui permet d'encapsuler toutes les couches basses (sockets, threads, ...) mises en œuvre durant le fonctionnement collaboratif des tâches/applications.

Dans le chapitre qui suit, nous nous intéressons à la composition logicielle de l'architecture du BA tout en illustrant au passage les grandes problématiques liées à l'introduction du parallélisme dans son fonctionnement, qu'elles soient de nature technique ou fonctionnelle. Plusieurs zones d'ombre seront étudiées en détails, notamment celles concernant les interrogations suivantes :

- Comment le BA peut assurer une connectivité universelle indépendamment du domaine métier des applications ?

- Comment le BA gère-t-il l'hétérogénéité des environnements d'exécution ?
- Comment le BA gère-t-il les ressources systèmes partagées (sockets, mémoires, threads, ...) ?
- Quels sont les services de base assurés par le BA ?
- Comment les applications peuvent-elles entrer en collaboration ?
- Comment le BA intègre-t-il le parallélisme massif dans le traitement ?
- Comment le BA arrive-t-il à optimiser l'utilisation de la bande passante ?
- Comment le BA assure-t-il la synchronisation entre tâches parallèles qu'elles soient locales ou distantes ?

1.3 Applications visées par le BA

Dans le système informatique d'entreprise, plusieurs applications peuvent collaborer pour atteindre un but commun. Elles sont souvent de nature hétérogène, et traitent des domaines métiers variés tels que : finance, gestion, tertiaire (voir Figure 1.1). Le BA, comme d'ailleurs la plupart des autres solutions ESBs, n'impose aucune contrainte particulière sur la nature des applications à intégrer. Grâce à son concept de connecteur standard (Terminal de Bus Applicatif BA), le BA assure une connectivité universelle indépendamment du domaine exercé. En effet, il joue le rôle de carrefour d'échange et assure le routage et 'acheminement du flux de données échangées vers la bonne destination, tout en faisant abstraction du domaine métier des unités communicantes.

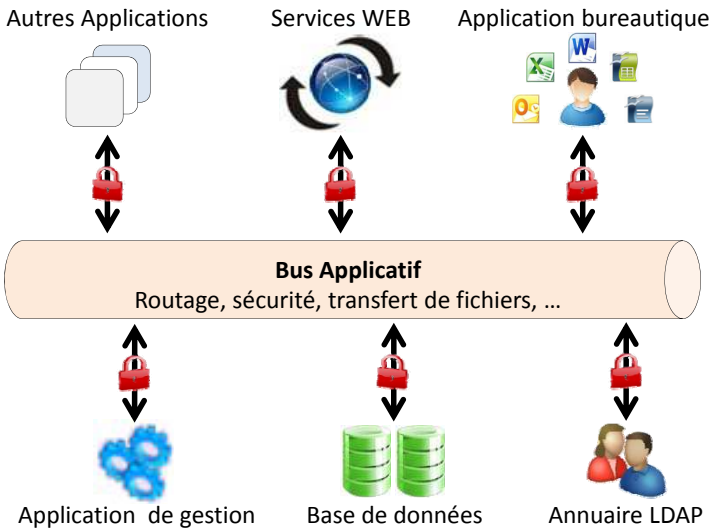

Figure 1.1 – Différents types d'applications prises en charge par le BA.

Comme nous le constatons sur la Figure 1.1, le BA, et de la même manière qu'un ESB classique, en mesure de prendre en charge des interconnexions avec des bases de données (intégration des données) ou avec d'autres technologies standards comme Lightweight Directory Access Protocol (LDAP) permettant l'interrogation et la modification des services d'annuaire.
D'un point de vue purement intégration, notre solution ne se distingue donc pas des autres solutions à base d'ESB puisque toutes les deux offrent une indépendance vis-à-vis du domaine métier exercé. En partant de ce constat, nous donnons une représentation plus générale de l'architecture présentée dans Figure 1.1 en faisant abstraction de la couche métier, cette nouvelle architecture

est présentée dans la Figure 1.2.

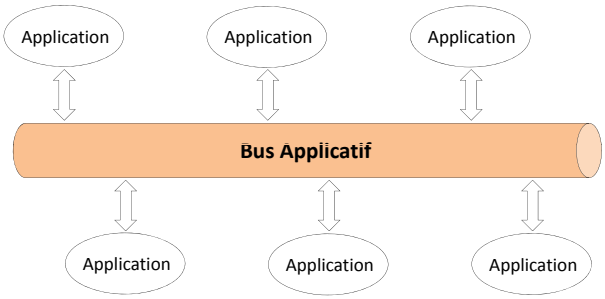

Figure 1.2 – Intégration des applications au BA.

1.4 Prise en charge du parallélisme par le BA

L'exploitation des nouvelles capacités offertes par les machines multicœurs actuelles vise le fonctionnement effectif de nombreuses tâches simultanément, l'extraction du parallélisme des applications est donc une démarche incontournable de l'évolution actuelle et très prometteuse des processeurs. L'idée principale consiste à parcelliser les applications en une pluralité de tâches indépendantes qui sont exécutées par des threads différents, et plus le nombre de ces threads augmente, plus en gagne en termes de performance à condition de disposer bien sûr, de la puissance matérielle (processeurs, cœurs) nécessaire.
Cette démarche entraine une conséquence directe sur l'entrée du BA qui n'es plus alimentée par un ensemble d'applications tel qu'il a été déjà montré dans la Figure 1.2, mais plutôt par une pluralité d'unités de traitement correspondant à des threads ou processus. Cette nouvelle architecture est présentée dans la Figure 1.3

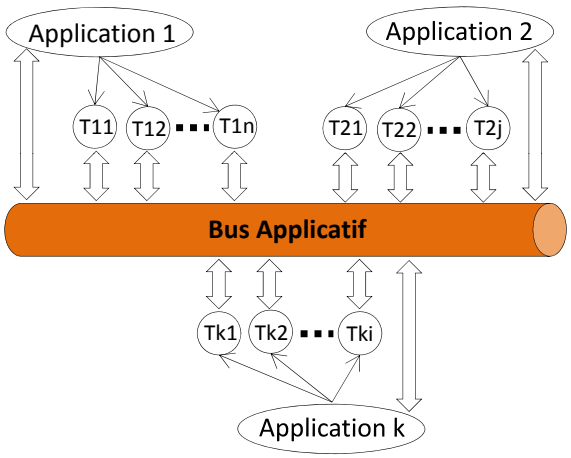

Figure 1.3 – Extraction du parallélisme des applications.

Si nous suivons l'enchaînement des idées développées jusqu'à présent, nous constatons que nous sommes partis sur la base d'une description du BA qui tient compte du domaine métier exercé (voir Figure 1.1), étant donné que cet aspect ne présente aucun impact particulier sur le processus d'intégration, nous avons désigné toutes les entités connectées au BA par le terme application

(voir Figure 1.2). Avec l'introduction du parallélisme dans le traitement, la Figure 1.2 ne s'adapte pas à la granularité des nouvelles unités de traitement considérées (threads), d'où l'intérêt de l'architecture présentée par la Figure 1.3, il ne nous reste plus qu'à introduire la notion des ULTs afin d'avoir une représentation plus générale du BA (voir Figure 1.4) qui illustre toute sa capacité à prendre en charge des unités de traitement de granularité diverse.

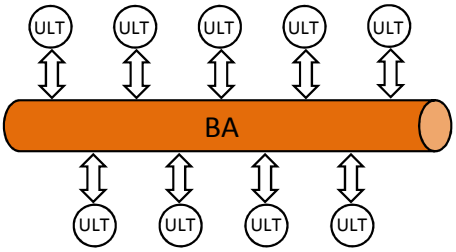

Figure 1.4 – Architecture générale du BA.

1.5 Cas d'interaction possibles au sein du BA

D'après la description donnée par la Section 1.2, les entités de traitement connectées au BA qu'elles soient locales : déployées sur la même machine hôte, ou distantes : déployées sur des machines distantes sont de deux types : processus ou threads, à partir de cette constatation, nous avons établi une cartographie des différents cas d'interaction entre thread/processus au sein du BA, qui constituera notre cahier de charges lors de la phase de conception de notre mécanisme de communication, afin qu'il puisse couvrir tous les cas recensés. La Figure1.5 illustre l'ensemble des cas identifiés.

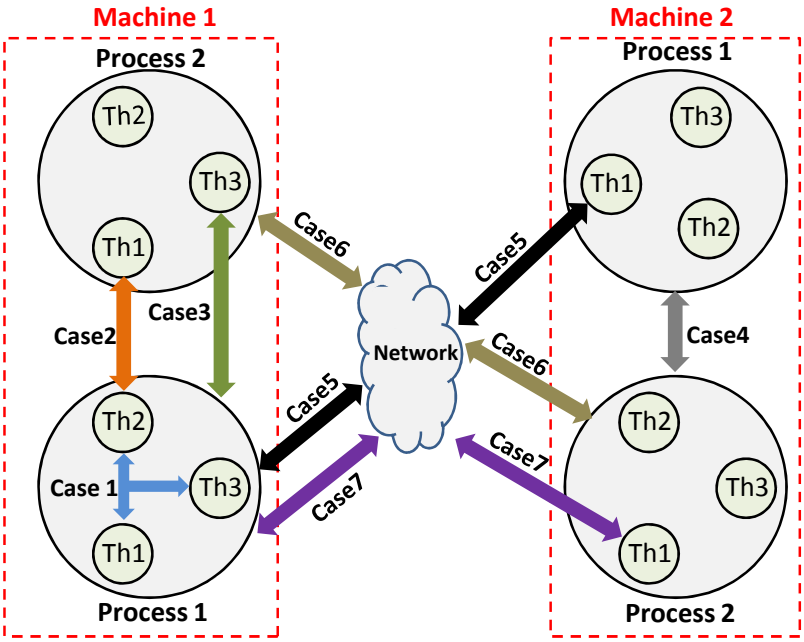

Figure 1.5 – Les différents cas d'interaction au sein du BA.

- Cas 1 : Communication entre un ensemble de threads d'un même processus.

- Cas 2 : Communication entre deux (ou plusieurs) threads de deux processus différents sur une même machine.
- Cas 3 : Communication entre un processus et un thread local.
- Cas 4 : Communication entre deux processus sur une même machine.
- Cas 5 : Communication entre deux threads distants.
- Cas 6 : Communication entre deux processus distants.
- Cas 7 : Communication entre un thread local et un processus distant.

Tous ces cas d'interaction sont pris en charge par notre mécanisme d'adressage, donnant ainsi aux programmeurs l'opportunité de réaliser des schémas de communications extrêmement complexes dans le respects des contraintes de performance et de transparence et qu'ils n'obtiendrons jamais avec une solution ESB classique. C'est donc à ce niveau que notre solution présente un intérêt majeur en termes de fluidification des communications entre les différents threads/processus répartis sur l'ensemble des machines de la plateforme.

1.6 Localisation des applications

Notre solution s'affranchit des contraintes de localisation physique des applications reparties sur toutes les machines de la constellation. Tout comme un serveur de nommage classique tel que Domain Name System (DNS), le BA dispose de son service de nommage capable de traduire l'adresse logique d'une application (adresse type BA) en une physique contenant la totalité des paramètres techniques (adresse IP, port) de la machine qui l'héberge. L'attribution et la modification des noms logiques sont à la charge de l'administrateur du BA. Un tel dispositif offre une abstraction de haut niveau qui facilite la tâche aux usagers du BA qui n'auront plus à se préoccuper des couches basses (Réseau, protocole, OS, ...) mises en jeu lors de la coopération des applications.

Avec l'introduction du parallélisme dans le traitement, nous sommes non seulement contraint de localiser des applications, mais aussi des threads : une fonctionnalité non supportée par les ESB classiques, et c'est à ce niveau que nous rapportons une valeur ajoutée avec l'introduction d'un mécanisme d'adressage plus subtil, avec lequel nous sommes en mesure de réaliser des communications au niveau threads, et ce, quelles que soient leurs localisations sur le réseau.

Continuons toujours sur cette même voie de réflexion, en supposant maintenant qu'un thread nécessite plusieurs points d'entrée/sortie sur le BA, par conséquent, nous devons non seulement localiser un thread, mais également son point d'entrée/sortie désiré. Une contrainte de plus prise en compte et résolue par notre mécanisme d'adressage, donnant ainsi au programmeur la possibilité d'accomplir des schémas de communications complexes.

1.7 Extraction automatique du parallélisme

Une bonne exploitation de notre architecture suppose donc le découpage préalable des applications en un ensemble de tâches parallèles par le programmeur avant d'être soumises au BA, il nous parait alors judicieux d'envisager un procédé de découpage automatique d'une part pour alléger la charge de travail des programmeurs, d'autre part, pour extraire le maximum de parallélisme des applications. Cette réflexion est le fruit de notre inspiration de certaines avancées de la recherche dans le domaine d'extraction automatique du parallélisme, notamment dans le domaine du calcul haute performance (HPC). Cependant, le contexte et l'environnement de déploiement du BA diffèrent de ceux que nous rencontrons en HPC, il n'est donc pas envisageable d'utiliser les mêmes techniques que celles du HPC. En effet, dans le contexte d'une architecture

SOA (Service Oriented Architecture), l'extraction automatique du parallélisme est difficile à automatiser car elle nécessite l'intervention d'un homme métier compétant jouîssant d'un savoir et savoir-faire pointus sur le domaine exercé par la société et ayant une parfaite connaissance des différentes règles et contraintes métiers qui la régissent. Pour illustrer cet aspect, nous allons prendre un exemple de processus métier simple présenté par la Figure 1.6

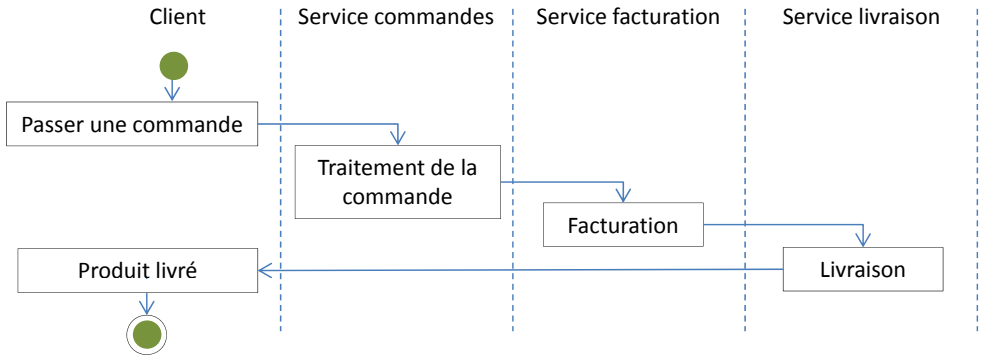

Figure 1.6 – Exemple d'un processus métier simple.

Notons que l'exécution de ce processus métier sur notre architecture ne présente aucun intérêt si les divers services sont développés suivant un code séquentiel. Supposons maintenant que le spécialiste décrive le service de facturation par toute la suite d'actions élémentaires suivantes :

- Action 1 : Collecte des informations ;
- Action 2 : Récupération du nombre de points de fidélité du client ;
- Action 3 : Connaissance du prix courant du produit ;
- Action 4 : Facturation en fonction du prix du produit, des taxes et des points de fidélité clients ;
- Action 5 : Transmission de la facture au client ;
- Action 6 : Établissement les statistiques de ventes ;
- Action 7 : Validation du paiement ;

En se basant sur cette description, ce spécialiste, et fonction d'un ensemble de règles de gestion propres à sa société peut déduire par exemple que les deux couples d'actions 2,3 et 5,6 peuvent être exécutées en concurrence. Le service de facturation peut donc être réécrit de manière à exploiter cette opportunité de parallélisme et améliorer ainsi la performance globale du système. Sur le plan pratique, la mise en œuvre de cette mutation passe par un recodage du service de facturation attribuant cette fois-ci à chaque action élémentaire un nouveau thread de traitement, nous obtenons ainsi la nouvelle composition présentée par la Figure 1.7

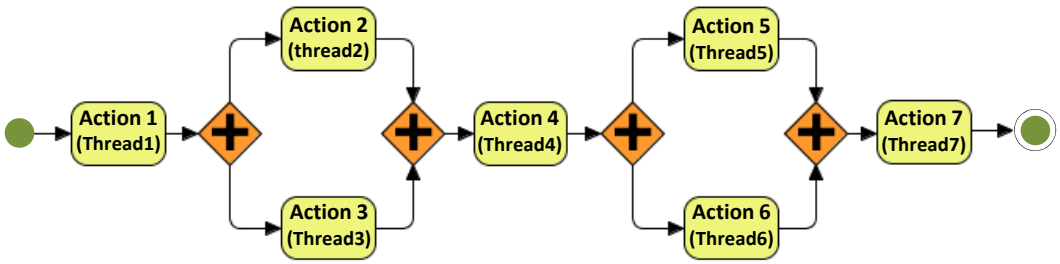

Figure 1.7 – Exploitation du parallélisme dans le traitement.

Dans cet exemple, nous nous somme intéressés uniquement au service de facturation, cependant, cette démarche d'extraction de parallélisme s'applique à tout autre service dont le traitement peut être décrit sous forme d'un ensemble d'actions élémentaires, mais en aucun cas, elle ne peut être entièrement automatisée, sans qu'il n'y ait implication d'un homme métier, raison pour laquelle 'il est extrêmement délicat de mettre en œuvre un procédé d'extraction automatique de parallélisme.

1.8 Les différents formes d'interaction au sein du BA

Nous avons identifié lors de cette étude trois formes d'interaction qui se manifestent lors d'un fonctionnement collaboratif d'un ensemble d'ULT connectées au BA, il s'agit d'interaction de partage de données, de synchronisation, et de communication qui s'appliquent à l'ensemble des cas présentés dans la Figure 1.5

1.8.1 Partage de données

Le troisième type d'interaction se présente sous forme d'un partage de données dans lequel, deux ou plusieurs ULT partagent le même espace d'adressage pour lequel le BA assure la cohérence des données qui y sont stockées lorsqu'un accès concurrent en lecture ou écriture se présente.
A partir de la description donnée par la Figure 1.5, nous distinguons trois niveaux de partage : intra-processus, inter-processus, et partage au niveau global.

- **Partage intra-processus**
  Le partage intra-processus est celui qui concerne l'ensemble des threads d'un même processus. A ce titre, rappelons que les bibliothèques de threads POSIX-PThread (pour Linux et Mac-OS) et WinThread (pour Windows) assurent nativement ce type de partage indiqué par cas1 sur la Figure 1.5. En effet, l'espace mémoire Heap [MBFM11] d'un processus est entièrement accessible à l'ensemble des threads qui le composent. La Figure 1.8 montre bien cette caractéristique.

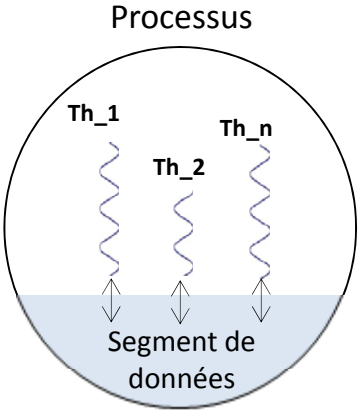

Figure 1.8 – Partage intra-processus.

- **Partage Inter-processus**
  Ce type de partage se présente lorsque deux ou plusieurs threads appartenant à des processus différents, ou lorsque deux processus distincts hébergés sur la même machine (OS)

souhaitent partager le même espace d'adressage, indiqué par cas2, cas3 et cas4 sur la Figure 1.5, et dans ce cas, le recours à un mécanisme d'échange entre processus est inévitable. A cet effet, le standard IPC-POSIX apporte des solutions pour le partage à travers de nombreux dispositifs tels que : les mémoires partagées, les files de messages et les tubes. Ce type de partage est présenté dans la Figure 1.9

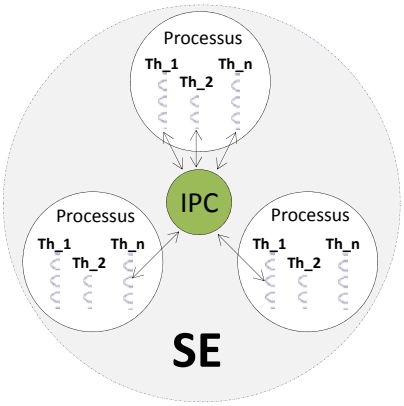

Figure 1.9 – Partage inter-processus.

- **Partage au niveau global**
  La troisième forme de partage se manifeste lorsque deux ou plusieurs ULT distantes, donc hébergées sur des machines différentes, souhaitent partager des données comme indiqué par cas5, cas6 et cas7 de la Figure 1.5, et dans ce cas, le passage par une couche de communication réseau est primordial. L'utilisation d'un protocole de transmission tel que TCP/IP répond à ce besoin. Ce type de partage est illustré par la Figure 1.10

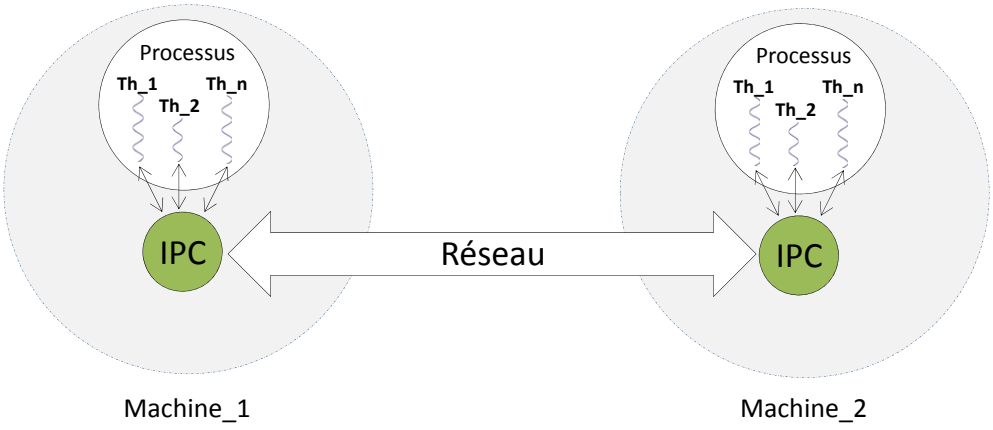

Figure 1.10 – Partage au niveau constellation.

1.8.2 Synchronisation

Parmi les difficultés dues au parallélisme nous citerons que chaque tâche va lire des données, réaliser un traitement et rendre une réponse. On comprend donc que la sortie d'une tâche est susceptible de devenir l'entrée d'une autre. Dans le cas où les deux tâches se déroulent en parallèle, il est nécessaire d'attendre d'abord que les données soient traitées par la première avant

d'être prises par la deuxième.
Les différentes ULT collaboratives sont donc reliées en amont par des contraintes de précédence d'où l'intérêt d'avoir un mécanisme de synchronisation permettant aux développeurs d'en indiquer explicitement les points. En se basant sur la description donnée par la Figure 1.5, nous constatons qu'il existe trois niveaux de synchronisation :

- **Synchronisation intra-processus**
  La synchronisation intra-processus concerne un ensemble de threads appartenant à un même processus, et dans ce cas, elle est réalisée à l'aide d'un simple mutex, et il n'est pas nécessaire de faire appel à des mécanismes lourds comme les sémaphores. Ce type de synchronisation est indiqué par cas1 et cas2 de la Figure 1.5
  La plupart des bibliothèques de programmation de threads comme PThread, et WinThread offrent des fonctions de synchronisation :
  - Point-à-point (voir Figure 1.11) : lorsque la synchronisation concerne seulement deux deux threads.
  - collectives (voir Figure 1.12) : lorsque la synchronisation concerne plusieurs threads à la fois (barrière de synchronisation).

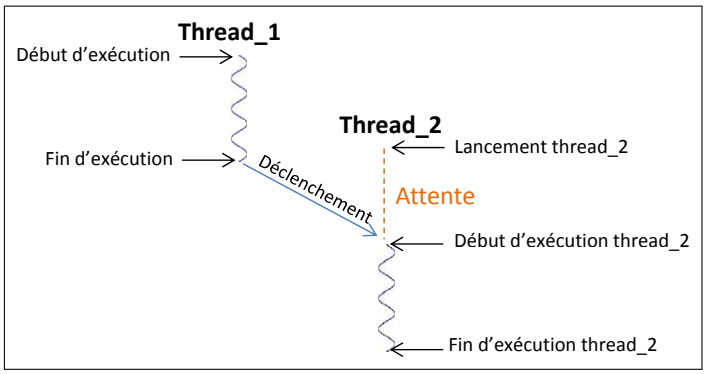

Figure 1.11 – Synchronisation point-à-point.

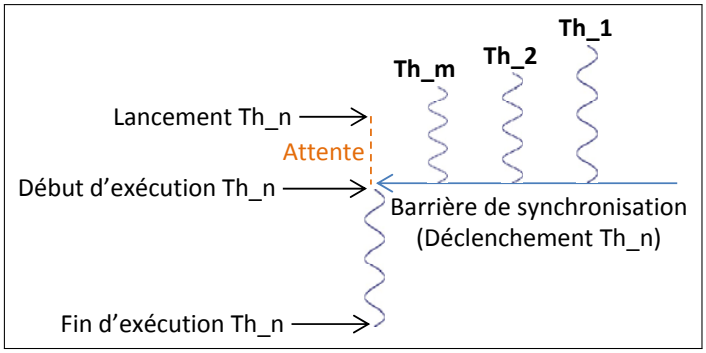

Figure 1.12 – Barrière de synchronisation.

- **Synchronisation inter-processus**
  Ce type de synchronisation se présente lorsque deux ou plusieurs ULTs ne disposant pas de lien de parenté sur une même machine hôte souhaitent se synchroniser comme indiqué par cas3, cas4 et cas5 de la Figure 1.5, dans une telle situation, l'utilisation d'un simple Mutex ne résout pas le problème puisqu'il se caractérise par une visibilité locale et ne peut pas

être utilisé par un autre ULT externe. L'utilisation du mécanisme des sémaphores répond à ce besoin.

- **Synchronisation au niveau global**
  Il s'agit de la forme de synchronisation la plus difficile à réaliser et consiste à synchroniser deux SPUs déployées sur des machines distantes, comme indiqué par cas6, cas7 et cas8 de la Figure 1.5). Dans cet autre contexte, les outils de synchronisation cités précédemment : Mutex et Sémaphores ne suffisent pas puisqu'ils ne font pas intervenir la couche réseau, d'où l'intérêt, entre autres, de notre mécanisme de synchronisation par P-instruction (parallèle instruction) qui répond parfaitement à ce besoin.

1.9 La communication

La communication est une forme d'interaction qui se distingue de celle relative au partage de données, par l'acheminement directe des données vers l'ULT ciblée sans pour autant être lues par une autre ULT externe. Elle se charge de la transmission de messages d'une ULT émettrice vers une ULT réceptrice. Nous considérons dans cette étude que l'aspect communication peut s'appliquer aussi bien pour des ULTs locales que distantes : Une une supposition qui vient désapprouver la désignation, largement utilisée, de la communication par une forme d'échange impliquant obligatoirement une couche réseau. Nous avons identifié trois types de communication :

- **Communication intra-processus**
  La communication intra-processus se présente lorsque deux threads : Th1 et Th2 appartenant à un même processus souhaitent échanger un message, qui sera conduit de Th1 et directement mis en place dans l'espace de travail de Th2 sans qu'une tierce ULT ne puisse le lire/modifier. Ce dernier sera directement déposé sur l'espace de travail de la deuxième et sans qu'il n y est possibilité de le lire/ecrire par une autre ULT tierce. Ce type de communication est représenté par les cas 1 et 2 de la Figure 1.5
- **Communication inter-processus**
  La communication inter-processus s'opère quand deux ULTs ne disposant pas de lien de parenté sur une même machine hôte souhaitent s'échanger des messages comme indiqué par cas3, cas4 et cas5 de la Figure 1.5, et dans ce cas précis, il devient nécessaire de faire appel à l'un des mécanismes de communication du standard IPC (file de messages, tube, ...) pour répondre à ce besoin.
- **communication au niveau global**
  Nous appelons ainsi, la communication entre deux ULTs distantes disposées à s'échanger des messages, tel que indiqué par cas6, cas7 et cas8 de la Figure 1.5, et dans ce cas seulement, l'introduction d'une couche réseau devient obligatoire afin d'assurer le transport des données depuis l'ULT source jusqu'à l'ULT de destination.

1.10 Composants logiciels du BA

Nous nous intéressons dans cette section à l'architecture logicielle globale du BA dans le but d'introduire les différents éléments logiciels sur lesquels elle repose. Nous nous contentons dans un premier temps d'une description brève de ces éléments juste pour illustrer la répartition de leurs rôles et fonctionnalités sur le BA, puis nous consacrons pour chacun d'eux un chapitre entier présentant une étude plus détaillée.
L'architecture du BA est repartie physiquement sur une constellation de machines. Elle comporte avantageusement les trois éléments logiciels suivants [BBA13b] :

1.10.1 Multiplexeur du Bus Applicatif (MBA)

Le Multiplexeur du Bus Applicatif (MBA) est le représentant local du BA au niveau de chaque machine, son rôle principal consiste à gérer toutes les ressources partagées sur le système qui l'héberge : mémoires partagées, threads, sémaphores, ..., dont le réseau de communication, à travers un service de multiplexage/démultiplexage en mesure de contrôler la circulation du flux de données tout en optimisant l'utilisation de la bande passante. De plus, ce composant MBA dispose d'un ramasse-miettes (garbage collector en Anglais) assurant le recyclage des ressources non utilisées suite à un mauvais usage de la part du programmeur (non libérées correctement) où à une terminaison inattendue du programme (cas d'une exception imprévue). Dans ces deux cas, le MBA détecte les ressources vouées à l'abandon et procède ainsi à leur destruction immédiate du système.
Du point de vue plus pratique, le MBA se présente sous forme d'un service (Daemon en Anglais) qui s'exécute en permanence sur le système, et sans lui, la machine devient inaccessible à distance. L'architecture du bus applicatif associe donc pour chaque machine hôte ou chaque système d'exploitation, notamment dans le cas de machines virtuelles, une instance de MBA réalisant simultanément le cryptage, le multiplexage, la transmission des données sur le réseau et la réception, le démultiplexage et le décryptage des données arrivées. La Figure 1.13 montre la répartition des MBAs sur les machines de la constellation.

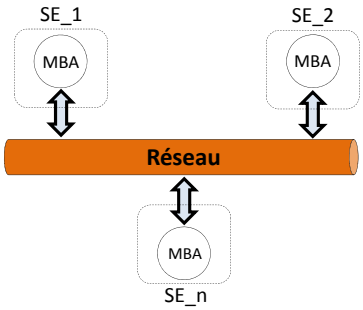

Figure 1.13 – Répartition des MBAs sur la plateforme.

1.10.2 Terminal du Bus Applicatif (TBA)

Une fois que le programmeur a défini ses ULTs au niveau applicatif, il doit les connecter au BA tout en donnant à chaque ULT, l'opportunité de disposer de son propre espace de travail où elle interagit en toute sécurité avec le BA, et sans qu'il n'y ait de risques d'interférences avec les échanges des autres ULTs. Pour pourvoir à cette nécessite, le BA est doté d'un composant logiciel nommé Terminal du Bus Applicatif (TBA) : une sorte de case d'accueil disposant des ressources nécessaires au fonctionnement des ULTs pour lesquelles il joue le rôle de connecteur et offre un point d'entrée/sortie sur le reste de la plateforme.
Le TBA donne également accès aux différents services du BA (transfert de fichiers, échange de messages, ...) et contrairement au MBA, il n'est pas une entité exécutable, il s'agit d'un composant logiciel entièrement constitué de ressources IPC nécessaires au fonctionnement des ULTs. Son cycle de vie est géré par le MBA et ce depuis sa création jusqu'à sa destitution, ainsi, le BA n'est accessible pour une ULT que via un TBA connecté au MBA hôte. La Figure 1.14 illustre la place réservée au TBA dans l'architecture du BA.

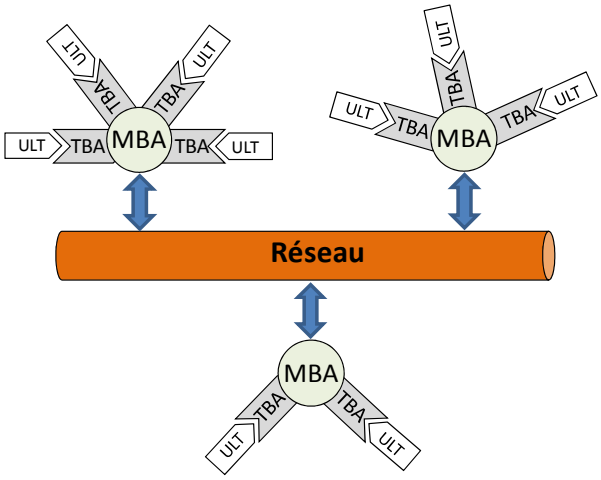

Figure 1.14 – Place du TBA dans l'architecture du BA.

1.10.3 API de Bus Applicatif

Pour rendre l'accès au TBA programmable, nous devons disposer d'une interface standard de programmation dite API (Application Programming Interface en Anglais) portable et simple à utiliser qui facilite cette tâche au programmeur qui ne porte plus aucune préoccupation pour les couches basses (sockets, mémoires, ...) mises en jeu durant l'exécution. Cette API offre également un jeu d'instructions riche qui ouvre l'accès aux multiples services du bus, et donne ainsi au programmeur la possibilité d'implémenter ses propres protocoles d'échange au niveau applicatif grâce à des fonctions génériques que nous verrons ultérieurement.
Le programmeur doit donc comprendre que détourner l'API pour accéder directement aux TBAs depuis les ULTs est une opération à hauts risques. En somme, l'API est un passage incontournable pour écarter toute forme de dysfonctionnement éventuellement causée par de mauvaises implémentations imputées au programmeur. La Figure 1.15 montre l'architecture logicielle globale du BA avec mise en relief des éléments logiciels présentés dans cette section.

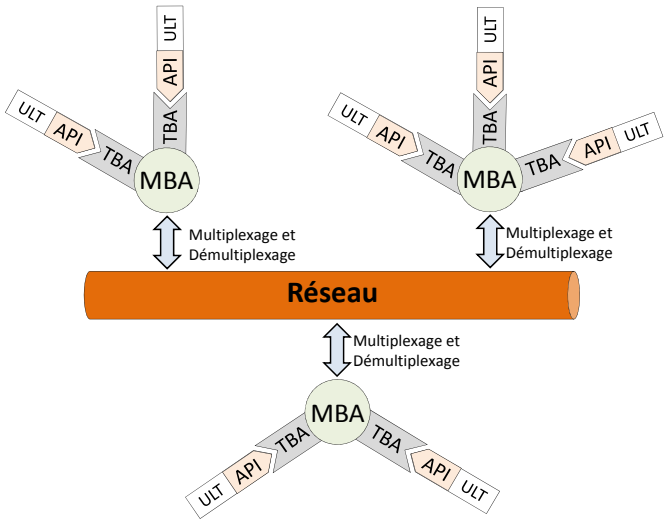

Figure 1.15 – Architecture logicielle globale du BA.

1.11 Conclusion

Nous avons présenté dans ce chapitre une étude détaillée sur l'architecture globale du bus applicatif ainsi que sur les grandes caractéristiques liées à son environnement de déploiement. Avec l'introduction des différents composants logiciels appropriés à notre plateforme, nous avons identifié besoins des tâches/applications en termes de partage de données, de communication et de synchronisation. Nous avons illustré cette étude par l'analyse de notre nouveau concept d'unité logicielle de traitement (ULT) qui reflète tout le pouvoir d'itération de notre architecture, ainsi que sa capacité à prendre en charge des unités de traitements de granularité différente qu'elles soient applications ou threads avec abstraction totale du domaine métier auquel elles appartiennent : un degré de perfectionnement dans la généralisation des communications entre tâches jamais atteint jusqu'à présent par une quelconque architecture d'intégration. Nous disposons donc d'une bonne longueur d'avance, dictée par besoin d'innovation, sur les autres plateformes d'intégrations qui devront obligatoirement revenir sur leurs pas pour suivre notre itinéraire en vue d'une reconversion aux concepts de traitements massivement parallèle nouvellement mis en lumière par notre recherche.

2

Programmation concurrente : Concepts et outils fondamentaux

Sommaire

2.1 Introduction

Le développement d'applications concurrentes utilisant des mémoires partagées est une tâche complexe. Elle nécessite une certaine compétence sur le plan technique et une connaissance approfondie en matière de programmation système. Cette tâche est d'autant plus ardue lorsque la solution à élaborer est prédestinée au déploiement sur des architectures de natures hétérogènes. En effet, chaque système d'exploitation se distingue par sa propre politique de gestion de ressources, ses limites systèmes, etc ... Nous devons donc prendre en considération tous ses facteurs décisifs pendant les phases de conception et de développement d'un programme concurrent.

De nombreuses technologies ont été mises au point pour faciliter la programmation d'applications concurrentes à mémoires partagées dont nous présentons quelques exemples ici, à savoir : inter-process com munication (IPC) [VS10], OpenMP [ACDN13], TBB [ZSM11], et Cilk [TM03]. Nous proposons aussi au passage, une évaluation de performance du standard IPC afin de mieux évaluer le rendement de chacun de ses mécanismes. Cette démarche s'avère incontournable, et

suscite un intérêt majeur pour la suite de notre étude, notamment lors de la phase de conception de la plateforme du BA.

Nous avons opté pour une approche expérimentale à base de tests, dans laquelle, nous mesurons pour chaque mécanisme du standard IPC, le temps moyen relatif aux opérations fondamentales comme : la création, la destruction, la lecture, et l'écriture. Les tests ont été réalisés sur une machine Linux (Fedora 13), Pentium 4, 3 Ghz CPU, 1 Go de Ram. Pour avoir des résultats de haute précision, chaque test a subi plus de 100 essais. Toutes les mesures sont exprimées en microsecondes.

2.2 Inter-process communication (IPC)

2.2.1 Définition

Le standard IPC (inter-process communication) de la norme Posix regroupe tout un ensemble de mécanismes de communication entre processus tels que : les sémaphores, les mémoires partagées, les files de messages, les signaux et les tubes [Bla05]. Ces mécanismes sont classés selon leurs fonctionnalités : ressources d'échange de données, ressources de synchronisation, ou leurs supports de stockage : mémoire, disque. Un récapitulatif de cette classification est présenté par le Tableau 2.1.

Nom de la ressource	Fonctionnalités	Support de stockage
Mémoires partagées	Partage de données	Mémoire
Sémaphores	synchronisation	Mémoire
Tubes	Partage+synchronisation	Disque
Files de messages	Partage+synchronisation	Mémoire

Tableau 2.1 – Classification des différents mécanismes du standard IPC.

Dans cette présentation, nous n'avons prêté aucune attention particulière aux signaux dans cette présentation car ils ne sont pas disponibles sur les systèmes de la famille Windows.

2.2.2 Files de messages

La file de messages est un mécanisme de communication entre plusieurs processus hôtes suivant un paradigme par échange de messages. Le flux de messages est organisé selon une politique FIFO (First In, First Out). D'un point de vue plus pratique, les files de messages sont des listes chaînées gérées par le noyau du SE, et dont les données sont composées d'un en-tête indiquant leurs type, suivi d'un bloc contenant le corps du message.

Ce mécanisme se caractérise par sa double fonction de partage de données et de synchronisation [Gra03b]. Les programmeurs n'ont donc pas besoin d'implémenter leurs propres outils de synchronisation pour gérer les accès à la file.

L'opération de lecture (retrait de messages) se fait de manière synchrone ou asynchrone suivant les paramètres passés en arguments à la fonction de lecture. Les opérations d'écriture (dépôt de messages), quant à elles, sont toujours non-bloquantes.

Le processus d'instanciation d'une file de messages se déroule en deux étapes, dans lesquelles, l'application demande au noyau via la fonction *msgget* la mise en place d'une nouvelle file à partir d'une clé unique. Le descripteur retourné par cette fonction est utilisé par la suite dans les fonctions de dépôt/retrait de messages, comme indiqué par la Figure 2.1

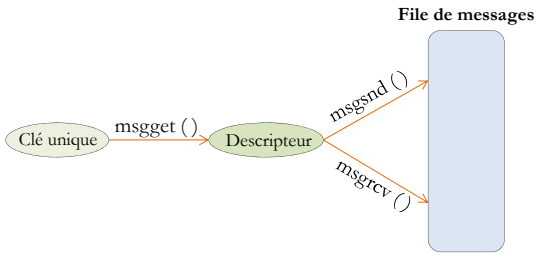

Figure 2.1 – Principe d'utilisation des files de messages.

Les temps CPU relatifs aux opérations de création et destruction d'une file de messages sont présentés dans le tableau 2.2.

Opération	Temps en microsecondes
Création	2.56μs
Destruction	1.86μs

Tableau 2.2 – Temps CPU relatif aux opérations de création et destruction d'une file de messages.

Les Figures 2.2 et 2.3 indiquent le temps CPU consommé par les opérations de dépôt et retrait de messages en fonction de leurs tailles (de 128 Octets à 1Mo). Nous faisons varier le nombre de message entre chaque itération et l'autre de manière à refléter le rapport entre le temps d'accès de la file par rapport à la taille des messages manipulés.

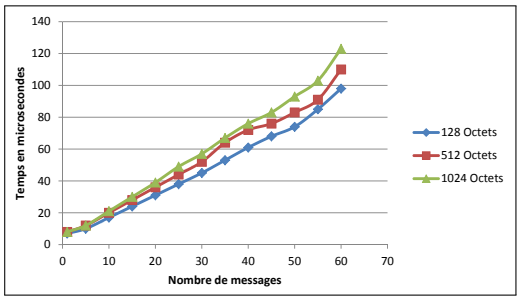

Figure 2.2 – Temps CPU consommé par l'opération de dépôt de message.

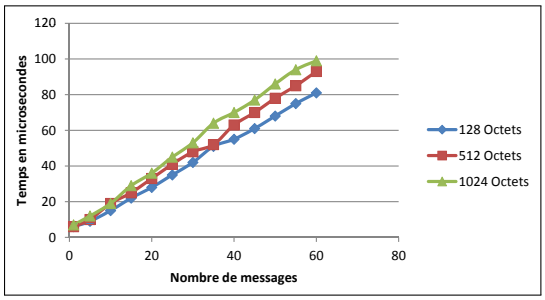

Figure 2.3 – Temps CPU consommé par l'opération de retrait de message.

2.2.3 Mémoires partagées

La mémoire partagée est un mécanisme qui donne à un ensemble de processus la possibilité de partager un même espace d'adressage. Cet espace se présente sous forme d'un tas contigu réservé au niveau de la mémoire globale du SE. Pour qu'un processus arrive à accéder à une zone mémoire partagée, il lui faut demander une adresse d'attachement via des primitives IPC spécifiques. En résumé donc, ce sont les primitives IPC qui permettent au processus de pointer sur le début du tas. les adresses d'attachement appartiennent à l'espace mémoire virtuelle du processus (adresses locales) mais pointent sur la même zone globale. Ce mapping (adresse d'attachement → adresse globale) est assuré par le SE grâce à des tables de correspondances. Ce mécanisme offre un service

avantageux d'accès direct à la mémoire, et ne nécessite pas de recopies intermédiaires, ce qui n'est pas le cas pour les files de messages par exemple, où les données sont automatiquement copiées depuis l'espace d'adressage de l'utilisateur vers celui du SE, et inversement.

Le mécanisme des mémoires partagées ne propose aucun moyen pour réaliser des indirections, il en est démunie, et la navigation dans le tas se fait donc par application d'offsets. De plus, les accès concurrents en lecture/écriture ne sont pas contrôlés par le SE, ce qui risque de causer des problèmes d'incohérence de données. Par conséquent, les programmeurs doivent déployer leurs propres moyens de synchronisation pour éviter toute forme de dysfonctionnement.

Le processus de création d'une mémoire partagée se déroule en deux étapes : la première a pour but de créer un descripteur système à partir d'une clé unique. Ce passage est réalisé à l'aide de la fonction shmget (CreateFileMapping sous Windows) du header <sys/shm.h>. La deuxième étape consiste à demander une adresse d'attachement sur le segment partagé via la fonction shmat (MapViewOfFile sous Windows). La Figure 2.4 montre le principe d'utilisation des mémoires partagées.

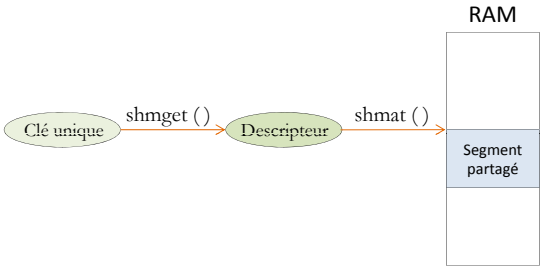

Figure 2.4 – Principe d'utilisation des mémoires partagées.

Les temps CPU relatifs aux opérations de création et de destruction d'une mémoire partagée sont présentés par le tableau 2.3. Les Figures 2.5 et 2.6 suivantes représentent respectivement les temps de retrait et de dépôt de messages en mémoire partagée.

Opération de Base	Temps CPU
Création	1.43μs
Destruction	0.93μs

Tableau 2.3 – Temps de création/destruction d'une mémoire partagée.

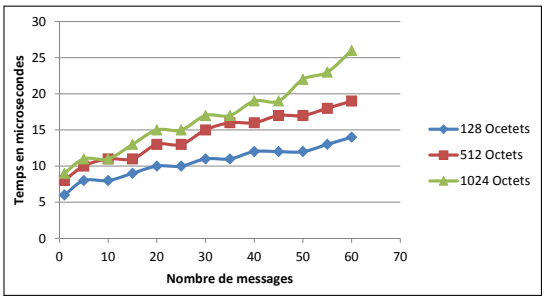

Figure 2.5 – Temps de retrait de messages en mémoire partagée.

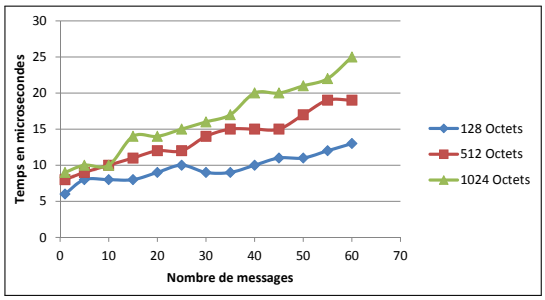

Figure 2.6 – Temps de dépôt de messages en mémoire partagée.

2.2.4 Tubes

Le tube, ou pipe, est un mécanisme de communication entre processus [LLKJ10]. Le principe de son fonctionnement consiste à utiliser des zones d'échanges au niveau disque, souvent en fichiers temporaires. il est généralement utilisé pour rediriger le flux de sortie d'un processus vers l'entrée d'un autre. Les fanatiques des lignes de commandes sous Linux par exemple, utilisent beaucoup cette technique pour combiner entre les commandes (ls | grep).

Les données circulent toujours dans le tube de façon unidirectionnelle, et une fois le sens choisi, il n'y a plus moyen de le modifier. En d'autres termes, si un processus joue le rôle de lecteur pour un tube, il le restera, sans jamais devenir rédacteur pour ce même tube et vice-versa.

Par ailleurs, les données sont gérées en flots d'octets, généralement suivant une politique FIFO : les données sont retirées selon l'ordre de leurs insertions), de plus, les lectures sont destructives, c'est-à-dire que les données lues sont systématiquement supprimées du tube.

Le tube a une capacité de stockage limitée qui dépend du tampon qui lui a été réservé. Cette limite est définie par le paramètre PIPE_BUF du fichier <limits.h>. Lorsqu'un tube est plein, les opérations d'écritures deviennent bloquantes, obligeant ainsi le processus rédacteur à temporiser jusqu'à ce qu'un nouvel espace soit disponible.

Finalement, un tube est de type *nommé* si la communication concerne deux processus indépendants, *anonyme* si elle implique deux processus possédant un lien de parenté. Il est manipulé au moyen de deux descripteurs par processus, un premier sert à la lecture alors qu'un deuxième à l'écriture. Le principe de fonctionnement des pipes est illustré par la Figure 2.7

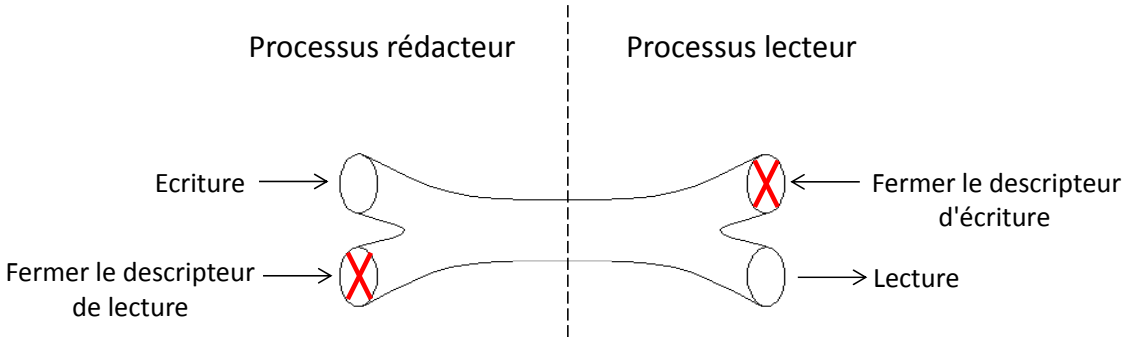

Figure 2.7 – Principe d'utilisation des pipes.

Le tableau 2.4 montre le temps CPU requis pour la création et la destruction d'un tube. Les temps de lecture et d'écriture quant à eux, sont indiqués par les tableaux 2.5 et 2.6

Opération de Base	Temps CPU
Création	4341μs
Suppression	3608μs

Tableau 2.4 – Temps nécessaire pour la création/destruction d'un tube.

Nombre de messages	128 Octets	512 Octets	1024 Octets
1	61	63	68
5	122	127	141
10	221	227	231
15	289	322	331
20	378	380	416
25	460	467	489
30	588	569	577
35	658	654	664
40	731	738	761
45	837	841	843
50	931	936	958
55	1007	1032	1044
60	1098	1126	1129

Tableau 2.5 – Temps nécessaire pour écriture dans un tube.

Nombre de messages	128 Octets	512 Octets	1024 Octets
1	61	66	78
5	115	119	133
10	218	223	229
15	291	325	336
20	382	383	419
25	465	469	482
30	592	576	584
35	667	664	675
40	733	735	775
45	844	852	862
50	936	945	960
55	1013	1041	1056
60	1112	1135	1152

Tableau 2.6 – Temps nécessaire pour lire depuis un tube.

2.2.5 Bilan

Le standard IPC regroupe tout un ensemble de mécanismes facilitant la communication entre processus hôtes. Il apporte des solutions à la fois simples et efficaces pour l'échange et la synchronisation entre tâches/applications.

Chaque mécanisme à ses propres caractéristiques et ses propres indices de performance, mais de manière générale, l'élément clef de performance d'un mécanisme quelconque réside dans la nature du support de stockage sous-jacent : RAM ou Disque.

Le standard IPC propose donc des mécanismes constituant les fondements sur lesquels repose la plateforme du BA. Sa portabilité facilite énormément son déploiement sur une large gamme de systèmes d'exploitation (Unix, Windows, Mac-OS, ...).

2.3 Interfaces de programmation pour architectures à mémoire partagée

2.3.1 OpenMP

OpenMP (Open Multi-Processing) est une un interface de programmation pour les applications parallèles à mémoires partagées [AHD11]. Elle est disponible sur la majorité des systèmes d'exploitation (Linux, Windows,...) pour les langages de programmation c, c++ et Fortran [Mor11].

L'interface de programmation OpenMP est basée sur le concept du multithreading et des directives (pragmas) pour la parallélisation des tâches. L'interprétation de ces directives par le compilateur d'OpenMP conduit à la génération automatique de code parallèle. Les séquences de code parallèles générées à l'issue de cette étape sont exécutées sur des threads différents. La gestion de ces threads (mise en place, synchronisation, destruction, ...) est implicite ce qui facilite la tâche aux programmeurs [Thi07]. Il existe deux paradigmes de programmation en OpenMP : programmation à grain fin et programmation SPMD (Single Program multiple data).

L'approche la plus simple pour générer du code parallèle à partir d'un code séquentiel de calcul est la OpenMP à grain fin qui cible les parties de calcul intégrées dans les boucles. Son principe de fonctionnement consiste à faire repartir le traitement itératif des boucles sur plusieurs threads afin d'accélérer l'exécution du programme. Voici un exemple :

```
1#include <omp.h>  // Pour integrer les fonctionnalités d'OpenMP
2#define NB_ELEMENTS 500 // Nombre d'elements d'un tableau
3 int main()
4 {
5  int i;
6  int X[NB_ELEMENTS], Y[NB_ELEMENTS], Z[NB_ELEMENTS];
7
8  #pragma omp parallel for private (i)
9  for(i=0; i<NB_ELEMENTS;i++)
10    Z[i]=X[i]*Y[i];
11
12  return 0;
13}
```

Code source 2.1 – Exemple de programme OpenMP de type grain fin

La ligne la plus intéressante dans ce code est la ligne 8. La directive *#pragma omp parallel for private (i)* a pour rôles de :

1. Initialiser l'environnement d'exécution d'OpenMP et lancer les threads de traitement ;
2. Partager les itérations sur l'ensemble des threads mis en place précédemment ;
3. Protéger la variable i (indice de la boucle) contre les accès concurrents ;

Dans le model SPMD, la distribution des données et des calculs entre threads est explicite (à la charge du programmeur). Pour des raisons de performance, cette distribution doit être faite en fonction du nombre de threads dans l'environnement d'OpenMP. L'exemple ci-dessous présente la traduction du code précédemment en mode SPMD :

```
1# include <omp.h>
2# include <stdlib.h>
3#define NB_ELEMENTS 500 // Nombre d'elements d'un tableau
4int main (){
5#pragma omp parallel
6  {
7    int i, iRang, iNbThreads, iTaille;
8
9    iNbThreads= omp_get_num_threads();
10   iRang= omp_get_thread_num();
11   iTaille= NB_ELEMENTS /iNbThreads;
12
13   int *X=(int*) calloc(iTaille+1, sizeof(int));
14   int *Y=(int*) calloc(iTaille+1, sizeof(int));
15   int *Z=(int*) calloc(iTaille+1, sizeof(int));
16
17   for(i=0; i<iTaille;i++){
18     X[i]=i\%11; Y[i]=X[i]*2; Z[i]=Y[i]/3;
19   }
20
21   for(i=0; i<iTaille; i++)
22     X[i]=Y[i]+Z[i];
23 }
24  return 0;
25 }
```

Code source 2.2 – Exemple de programme OpenMP de type grain fin

La ligne 9 est celle de la récupération du nombre de threads évoluant dans l'environnement d'OpenMP. La taille des données doit être calibrée en fonction du nombre de threads obtenue précédemment (ligne 11). Le nombre d'itérations de la boucle dépend lui aussi et encore de ce même nombre (ligne 21).

2.3.2 Threading Building Blocks (TBB)

Threading Building Blocks (TBB) est une bibliothèque c++ développée par INTEL. Elle a été conçue dans le but de faciliter la programmation d'applications parallèles sur des architectures multicœurs à mémoire partagée [INTb].

Parmi les avantages attribués à la bibliothèque TBB nous comptons sa capacité de récupérer implicitement certaines caractéristiques liées à l'environnement d'exécution, notamment le nombre de threads supportés par l'architecture matérielle sous-jacente, ce qui détermine un

meilleur calibrage du code parallèle généré en sortie par rapport à la puissance de la machine hôte.

Par ailleurs, la bibliothèque TBB intègre un ensemble d'algorithmes de base (parallel_reduce, parallel_for, parallel_scan) ainsi que des structures de contrôle (spin_mutex, mutex, queuing_mutex) qui gèrent : La création, la synchronisation et la destruction de threads.

2.3.3 Cilk

Cilk est une interface de programmation destinée au développement d'applications multi-threadées [LB10]. Elle a été développée en 1997 par le laboratoire MIT laboratory for Computer Science de INTEL [FLR98], et les langages supportés sont le C et C++ (Cilk Plus [Inta]).

Le principe de fonctionnement de Cilk est simple, il consiste à laisser au programmeur le soin d'exprimer le parallélisme dans le code par un nombre de mots clés réservés comme : spawn, sync, etc. Par la suite, l'environnement de déploiement de Cilk prend nativement en charge l'exécution des threads parallèles. Les tâches exprimées par le programmeur sont affectées à des listes gérées par le thread de l'application principale. Ce mode de gestion réduit le nombre d'opérations de synchronisation entre tâches parallèles. Voici un exemple simple d'une fonction écrite en Cilk :

```
1cilk int fibonacci (int nb)
2{
3  if (nb < 2)
4    return nb;
5  else
6  {
7    int i, j;
8    i = spawn fibonacci (nb-1);
9    j = spawn fibonacci (nb-2);
10   sync;
11   return (i+j);
12 }
13}
```

Code source 2.3 – Exemple de programme cilk.

Le code ci-dessus constitue une implémentation récursive de la suite de Fibonacci. Le mot clé spawn (ligne 8 et 9) ne fait simplement que, d'une part designer que l'appel de la fonction qui suit (fibonacci) peut être exécuté en parallèle, mais pas obligatoirement, et d'autre part indiquer au Scheduleur Clit que cet appel représente une partie indépendante du programme. La synchronisation avec la fin des traitements parallèles indiqués précédemment par le mot clé spawn se fait par le biais de l'instruction sync.

2.4 Multithreading

Le multithreading est un technique de programmation utilisée dans le développement d'applications concurrentes. Elle repose principalement sur la notion de tâche ou thread (processus léger). Un thread est une unité logicielle de traitement qui peut s'exécuter en concurrence avec d'autres.

Dans le modèle multithread, l'application principale (processus lourd) est composée d'un ensemble de threads qui se partagent des ressources système comme les fichiers, la mémoire, etc, et réalisant chacun, une séquence de code bien définie, le tout se déroule sous le contrôle du système

d'exploitation, qui se charge de diviser le temps de traitement non seulement entre les différentes applications hôtes, mais aussi entre tous les threads d'une même application. Ce modèle d'exécution propose une meilleure exploitation de la puissance des machines multiprocesseurs.

Il existe plusieurs bibliothèques de programmation de threads : GnuPth(Gnu Portable Threads), Capriccio, NPTL, pthread, etc. La plupart d'entre elles proposent le même jeu de fonctions de base pour : la création, la synchronisation et la destruction des threads.

La bibliothèque pThread est sans aucun doute la plus réputée, d'une part, pour sa conformité à la norme POSIX, d'autre part, pour son mode avancé de gestion des threads (pthread_cond_wait, pthread_mutex_lock, ...). Le coût de création d'un thread avec cette bibliothèque est de l'ordre de 38μs.

Le multithreading est un concept clé et une technologie incontournable en vue d'une meilleure exploitation de la puissance de traitement offerte par les machines multicœurs/multiprocesseurs.

2.5 Conclusion

Nous avons présenté dans ce chapitre toute une gamme de technologies destinées à la programmation d'applications concurrentes. Nous nous sommes intéressé à ce type d'outils car la mise en œuvre de notre plateforme nécessite une fondation technologique, qui lui apporte une logistique indispensable à l'échange et au partage de données, ainsi qu'à la synchronisation.

D'après la description donnée par notre cahier de charges, la première fonctionnalité demandée au BA est de faciliter la communication collaborative entre tâches (threads). Si nous nous limitons uniquement à l'aspect granularité des unités de traitement qui constitue une première contrainte, une interface comme OpenMP se présente en candidat potentiel pour servir d'assise technologique à notre plateforme, notamment pour sa capacité de générer automatiquement du parallélisme. Cependant, même si OpenMP garde le mérite de ce pouvoir, la notion de *granularité* des unités de traitement n'est pas la seule contrainte suffisante, en effet, d'autres aspects doivent aussi être pris en compte :

- **Domaine d'application** : OpenMP est parfaitement compatible avec le traitement matriciel, notamment pour la parallélisation des parties indépendantes de code dans les boucles. Cette spécificité la rend particulièrement adaptée aux domaines des calculs scientifiques et au HPC [FZRL08]. En revanche, elle ne présente aucun intérêt pour les traitements sortant de ce cadre particulier.
- **Niveau de parallélisation** : OpenMP prend en charge uniquement le parallélisme au niveau interne d'une même application, en d'autres termes, elle n'est pas en mesure de faire fonctionner deux tâches appartenant à deux applications différentes.
- **Type d'architecture** : Bien que basée sur le multithreading, OpenMP n'a pas de solutions pour le parallélisme sur des architectures distribuées car elle n'intègre aucune couche de communication réseau, c'est d'ailleurs la raison pour laquelle elle est souvent combinée avec d'autres interfaces de communication comme MPI afin de permettre la distribution des traitements. Dans le domaine du parallélisme, cette cohabitation de technologies (OpenMP + MPI) est connue sous le terme *programmation hybride*.

Ces trois critères limitatifs s'appliquent non seulement à OpenMP, mais aussi à toute interface du même acabit tel que : TBB ou Cilk. De telles interfaces ne peuvent servir donc de fondations technologiques pour notre plateforme. Leur utilisation est plutôt optionnelle, et elle n'est envisageable qu'à titre facultatif : lorsque l'exécution concerne un traitement local purement matriciel, ce qui est rarement le cas dans une architecture orientée services telle que la nôtre.

La seule technologie retenue à l'issue de cette étude est donc le standard IPC. Certes, il ne

s'agit pas d'un outil de haut niveau, il est donc difficile à manipuler, en plus, il nécessite une bonne maîtrise de la programmation système. Cependant, ce que nous ne manquons pas de noter c'est sa richesse et son efficacité qui sont d'une importance capitale et font de lui cette fois un outil décisif pour la suite de nos développements.

3

Proposition d'un nouvel allocateur dynamique de mémoires-IPC

Sommaire

3.1 Introduction

Dans l'architecture de bus applicatif que nous proposons, plusieurs processus/threads peuvent se mettre en interaction de manière simultanée. Ces processus/threads peuvent être locaux (déployés sur la même machine hôte) ou distants (déployés sur des machines distantes). Dans les deux cas, le service de communication est assuré par le multiplexeur du bus applicatif (MBA), qui se charge de toute la partie transfert de données qui au niveau local, sont systématiquement mises à disposition des processus et threads hôtes concernés chaque fois qu'elles sont reçues en provenance d'un MBA distant. Ce dispositif de partage de données entre MBA et processus/threads locaux s'appuie sur un mécanisme de mémoires partagées.

Contrairement au mode de fonctionnement des files de messages, dans lequel les données sont systématiquement copiées depuis l'espace mémoire utilisateur vers l'espace système et inversement (depuis l'espace système vers l'espace utilisateur), le mécanisme des mémoires partagées offre un moyen d'échange de données sans copie mémoire intermédiaire, grâce à un dispositif de correspondance entre les adresses d'attachement et les espaces de mémoires partagées. Ce dispositif est présenté dans la Figure 3.1

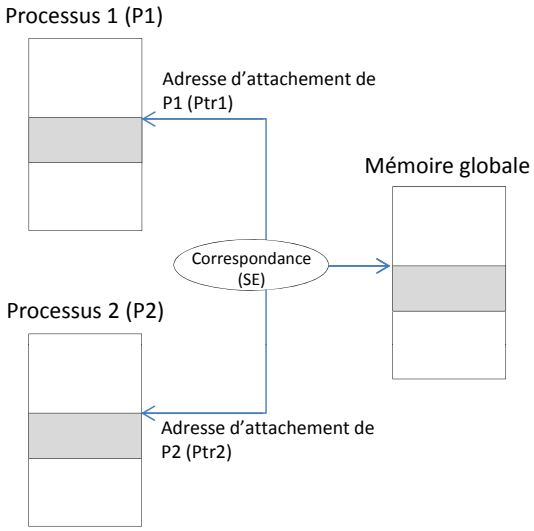

Figure 3.1 – Mécanisme de mémoires partagées.

Comme nous pouvons le constater depuis la Figure 3.1, les deux processus disposent chacun d'une adresse d'attachement (Ptr1 pour P1 et Ptr2 pour P2) qui pointent sur la même zone mémoire. Ces deux adresses peuvent être utilisées directement pour manipuler (lire et écrire) le contenu du segment partagé.

Si le standard des mémoires partagées offre un bon niveau de performance, il souffre de nombreuses limitations :

- D'une part, ce mécanisme permet uniquement le partage des données appartenant aux types de base comme les entiers (int), les réels (float), les caractères (char), et ne support pas le partages de données complexes telles que les pointeurs.
- D'autre part, les zones de mémoires partagées se présentent sous forme d'un espace contigu, ce qui risque d'engendrer des problèmes de fragmentation interne/externe, qui se posent de manière plus fréquente dans un environnement d'exécution massivement parallèle, dans lequel, le nombre de demandes d'allocation/libération de mémoires déposées par les différents processus/threads peut être très important.

Dans ce chapitre, nous présentons une nouvelle extension du standard de mémoire-IPC qui permet de remédier aux limitations citées précédemment qui a été implémentée au sein d'un nouvel allocateur dynamique de mémoires partagées, développé en langage C/C++, compilé et testé sur trois systèmes d'exploitation : Windows, Linux, et Mac-OS. En plus de la résolution de ces grandes limitations du standard des mémoires partagées, notre solution tient compte d'autres aspects annexes tels que celui de la portabilité, ce qui permet de déployer notre solution sur une large gamme d'infrastructures, de performance, ce qui minimise les temps de réponses de notre allocateur et préserve ainsi la performance globale du système, de cohérence de données partagées lors d'un accès concurrent en lecture/écriture.

3.2 Caractéristiques du mécanisme de mémoires partagées

Avant d'entamer la description de la stratégie d'allocation de mémoire partagée adoptée par notre allocateur, il serait d'abord intéressant de décrire les principales caractéristiques de ce standard qui sont à l'origine de ses limitations, et qui nous motivent à lui apporter de nouvelles extensions.

Nous constatons depuis la description présentée dans la Section 2 que le principe générale de fonctionnement des mémoires partagées reste le même quel que soit le système d'exploitation considéré en l'occurrence Windows, Mac-OS et Linux. La création d'une mémoire-IPC nécessite toujours deux étapes : Une première étape dans laquelle on passe d'une clé IPC à un descripteur système, et une deuxième dans laquelle on passe d'un descripteur système à une adresse d'attachement. Cependant, les fonctions IPC de base utilisées dans ces deux étapes diffèrent de noms, de paramètres, et valeurs de retour d'un système d'exploitation à l'autre. De plus, chaque OS adopte sa propre politique de gestion pour assurer la récupération de ressources IPC qui n'ont pas été libérées proprement (ramasse-miettes). Pour toutes ces raisons, s'adapter aux différentes contraintes imposées par son environnement de déploiement, s'avère une propriété importante pour notre allocateur.

Pour chaque caractéristique considérée (durée de vie, portabilité, structuration de données, ...), nous illustrons la stratégie de gestion adoptée par chaque système d'exploitation (Windows, Linux, Mac-OS) et nous décrivons par la suite comment elle a été prise en considération dans notre allocateur dans le but d'aboutir à une solution qui serait unifiée, portable et performante.

3.2.1 Durée de vie d'un segment de mémoire partagée

La durée de vie d'un segment de mémoire partagée varie selon le système d'exploitation considéré :

- Pour les systèmes de type Linux ou Mac-OS, les segments de mémoires partagée ont une durée de vie permanente qui ne dépend pas de celle du processus créateur, même si ce dernier termine son exécution, ils restent toujours alloués au niveau du SE. Normalement la libération de ces segments doit être faite explicitement au niveau du programme (en clair à la charge du programmeur) via des fonctions IPC de base spécifiques, mais le problème se pose quand certains segments sont créés et restent alloués au niveau du SE sans jamais être libérés et ce pour deux raisons : Par un simple oubli de libération au niveau du code de la part de l'utilisateur, ou bien le programme a généré une exception qui a mis fin à son exécution de manière imprévue, et dans ces deux cas, les segments resteront toujours alloués au niveau du système sans jamais pouvoir les utiliser à nouveau.
- Windows adopte une autre stratégie de gestion car il dispose d'un ramasse-miettes dont le rôle principale consiste à détecter puis libérer les segments de mémoire partagée abandonnés. Ainsi, la durée de vie d'un segment de mémoire partagée ne dépend plus de celle du processus créateur, et sa destruction se fait de manière implicite lorsqu'il n'est plus utilisé (aucun processus ne maintient une adresse d'attachement sur le segment en question).

D'après cette description nous tirons la conclusion suivante : La stratégie adoptée par Windows étant plus raisonnable et judicieuse que celle utilisée dans Linux et Mac-OS, nous avons donc choisi de nous aligner sur une stratégie qui serait similaire à la première en apportant à notre allocateur une nouvelle fonctionnalité qui lui assure une gestion semblable à celle offerte par Windows, pour les environnements Linux et Mac-OS. Cette fonctionnalité repose sur l'utilisation de la structure shmid_ds qui donne accès aux informations relatives aux processus qui référencent un segment donné (possèdent une adresse d'attachement) et notamment, l'attribut shm_nattch qui représente le nombre de processus qui référencent un segment [Gra03a], et dont la valeur 0 déclenchera sa destruction.

3.2.2 Type de données à partager

Les segments de mémoires partagées permettent le partage d'un simple tampon en mémoire, et les données partagées doivent obligatoirement appartenir à des types de base tels que les entiers (int) les réels (float), etc, et le partage de données de type pointeur (adresses) n'étant pas supporté par les trois systèmes d'exploitation étudiés, ce qui risque d'être très contraignant pour les programmeurs.

Pour montrer la limitation du standard IPC à ce niveau, nous allons prendre un exemple simple dans lequel le système est formé par deux processus ; P1 et P2. Pour des raisons de simplification, nous supposons que le segment de mémoire partagée S est déjà créé (par P1 ou P2), et que les deux processus disposent d'une adresse d'attachement sur ce segment (GPtr1 pour P1 et GPtr2 pour P2).

Si le processus P1 souhaite partager avec P2 une structure de données composée de deux éléments : un entier et un pointeur (voir Figure 3.2). Il commence par créer une instance de la strucutre My_Struct, initialise la variable i par la valeur 22, puis reserve un espace mémoire pour l'attribut pBuf en utilisant la fonction d'allocation calloc. L'adresse retournée par cette fonction est : 0x00900068. Finalement, P1 utilise son pointeur d'attachement (GPtr1) pour copier le contenu de la structure Stest1 au niveau du segment de mémoire partagée S. Nous supposons dans ce cas que les deux variables i et pBuf sont codées sur 4 Octets.

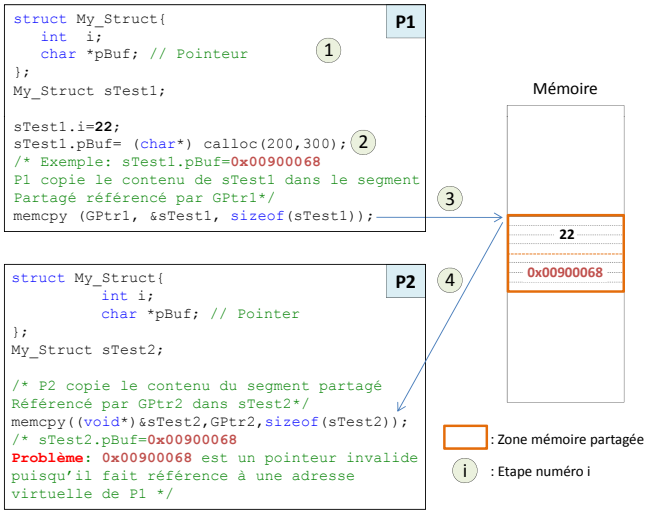

Figure 3.2 – Problème de type de données à partager.

Le processus P2 utilise son adresse d'attachement (GPtr2) pour copier le contenu du segment de mémoire partagée S dans la structure psTest2. L'utilisation du champ i de cette structure ne pose aucun problème, et son contenu peut être directement exploité au niveau du programme puisqu'il appartient aux types de base. Cependant, le contenu du champ pBuf ne peut pas être utilisé au niveau de P2 car il fait référence à une adresse virtuelle (0x00900068) qui appartient au heap de P1. La manipulation de cette adresse au niveau de P2 présente un risque potentiel pour le déclenchement d'une violation d'accès mémoire. A titre exceptionnel, cette adresse peut coïncider avec une autre qui a été déjà allouée par P2 à l'aide d'une fonction d'allocation classique (calloc, malloc, ...). Dans ce cas, on se retrouve avec deux adresses identiques, qui font référence, la première à une adresse appartenant au segment S, et la deuxième à une adresse virtuelle appartenant au heap de P2, et il est impossible de distinguer l'une de l'autre au moment de l'exécution.

Nous constatons, d'après cette description, qu'étant donné que le partage de pointeurs pose des difficultés particulières, son traitement doit être donc réalisé différemment de celui des données appartenant aux types de base. Pour répondre à cette problématique, notre allocateur met à disposition des programmeurs un ensemble de fonctions qui leurs permettent de stocker le contenu d'un pointeur local en mémoire partagée et inversement (mettre le contenu d'une mémoire partagée en mémoire locale).

3.2.3 Structuration des données

Un segment de mémoire partagée se présente sous forme d'un tas composé d'un espace contigu d'octets, à ce stade, les données partagées peuvent être décrites au niveau du processus rédacteur suivant des structures appropriées qui ne sont pas les mêmes que celle détenues par les autres processus lecteurs qui par conséquent, ne sont pas en mesure d'interpréter le contenu des données stockées en mémoire partagée. Ce cas de figure se présente, par exemple, lorsqu'un processus Pi subit une mise à jour qui modifie la structuration de ses données. Cette caractéristique constitue une autre limitation du standard IPC que nous souhaitons résoudre dans notre allocateur. Ce problème n'a pas été rencontré dans l'exemple présenté dans la Section 3.2.2 puisque les deux processus P1 et P2 utilisent la même structure de données.

Pour mieux illustrer cette limitation, nous allons prendre un exemple différent de celui donné précédemment en supposant cette fois-ci que P1 et P2 opèrent sur deux structures de données différentes : My_Struct1 pour P1 et My_Struct2 pour P2. Le scénario d'exécution de P1 et P2 est présenté dans la Figure 3.3.

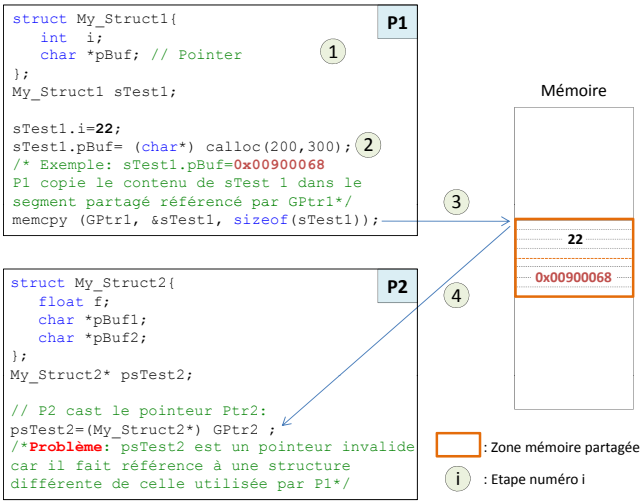

Figure 3.3 – Problème de structuration de données.

Dans cet exemple, P1 créé une instance de la structure My_Struct1 nommée sTest1, initialise ses variables ; i et pBuf, puis la stocke dans le segment de mémoire partagée S en utilisant son adresse d'attachement Gptr1.
De son côté, le processus P2 utilise son adresse d'attachement Gptr2 pour récupérer le contenu de la structure stockée précédemment par P1.
Notons que les deux processus utilisent deux structures de données différentes : My_Struct1 pour P1 et My_Struct2 for P2 pour manipuler le contenu du segment de mémoire partagé. Comme nous pouvons le constater depuis la Figure 3.3, l'interprétation de S par P2 est incorrecte puisqu'elle fait référence à des champs inexistants (pBuf2) dans la structure originale (My_Struct1). Une telle interprétation risque d'engendrer des violations d'accès mémoire lors de l'exécution.

Pour résoudre ce problème, notre allocateur donne aux processus rédacteurs la possibilité de décrire les différents éléments qui seront partagés en mémoire. Ainsi, notre API associe un champ descriptif à chaque élément à partager qui l'identifie aux autres processus lecteurs et qui se présente sous forme d'une chaîne de caractères qui peut être utilisée au moment du cast par ces processus afin d'obtenir une bonne interprétation des données.

3.2.4 Portabilité

Depuis la description présentée dans la Section 2.2.3, nous constatons que le principe de fonctionnement des mémoires partagées est le même pour les trois systèmes considérés. L'obtention d'une nouvelle adresse d'attachement se fait toujours en deux étapes : Clé IPC —> descripteur IPC, descripteur IPC —> Adresse d'attachement. Cependant, les fonctions systèmes utilisées dans ces deux étapes diffèrent d'un OS à l'autre au niveau paramètres, valeurs de retour, etc [KMK08]. Trouver un moyen d'unifier toutes ces fonctions est devenue une nécessité afin de leur offrir un jeu commun quel que soit le système considéré.

La solution implémentée dans notre allocateur repose sur le principe de la compilation conditionnelle, qui permet de compiler de façon sélective des blocs de code spécifiques tout en excluant

d'autres. Ainsi, une fonction F peut être codée en plusieurs blocs dont l'implémentation est relative à un OS déterminé. Sur le plan pratique, cette technique est réalisée à l'aide de pragma *ifdef* pour séparer entre les différents blocs d'une fonction.

1. #ifdef **TARGET_OS_MAC** for Mac-OS
2. #ifdef **__linux__** for Linux
3. #ifdef **_WIN32 or _WIN64** for Windows

3.2.5 Fragmentation de mémoire

Les problèmes de fragmentation de mémoire son souvent engendrés par l'appel répétitif des fonctions d'allocation/libération étant donné que l'espace mémoire associé à un segment partagée est formé par une zone contigüe d'octets. Les fragments désignent les parties de mémoire non exploitées par les applications pour être trop petites ou largement supérieures aux tailles demandées. Le taux de fragmentation est souvent considéré comme critère d'évaluation de performance d'une politique d'allocation. La Figure 3.4 illustre les différents types de fragmentation.

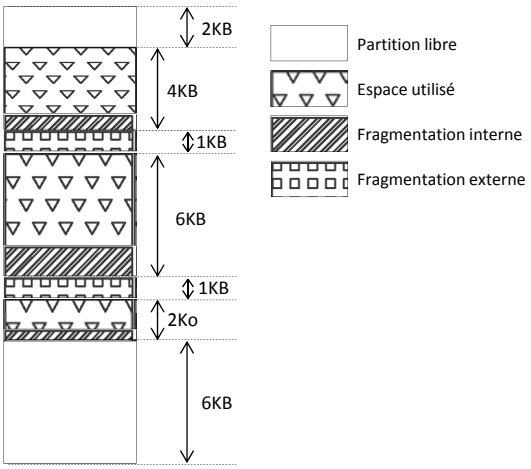

Figure 3.4 – Fragmentation interne/externe.

3.2.5.1 Fragmentation externe

La fragmentation externe fait référence aux partitions libres non exploitées par le système d'allocation, problème qui se produit lorsque des partitions de taille relativement faible sont libérées par l'application qui les maintient, alors qu'elles correspondent rarement aux nouvelles demandes d'allocation ce qui rend leurs réutilisations difficile.

Pour résoudre le problème de la fragmentation externe, les partitions libres de faible taille peuvent être regroupées en de nouvelles partitions de taille plus importante, offrant ainsi de meilleures opportunités pour d'éventuelles réutilisation.

3.2.5.2 Fragmentation interne

La fragmentation interne correspond aux zones libres non exploitées à l'intérieur des partitions. Ces « trous » de mémoire apparaissent lorsque la taille de la mémoire demandée est inférieure à celle de la partition attribuée. Pour remédier à ce problème, les partitions de grande

taille peuvent être découpées en deux pour pouvoir bénéficier d'une nouvelle partition qui couvre l'espace non exploité restant et qui peut être utilisé lors de prochaines demandes d'allocation, de manière à avoir une nouvelle partition qui couvre l'espace non exploiter (restant). Cette nouvelle partition peut être utilisée lors des prochaines demandes d'allocation.

3.2.6 Compactage de mémoire

Le compactage de la mémoire est une technique couramment utilisée pour lutter contre le problème de a fragmentation externe, elle consiste à déplacer des blocs mémoires d'une zone A (zone initiale) vers une autre zone B (zone finale), l'intérêt étant de regrouper les blocs non utilisés en un seul bloc libre, de taille plus grande, facile à exploiter. Le principe de fonctionnement du compactage de la mémoire est illustré par la Figure 3.5

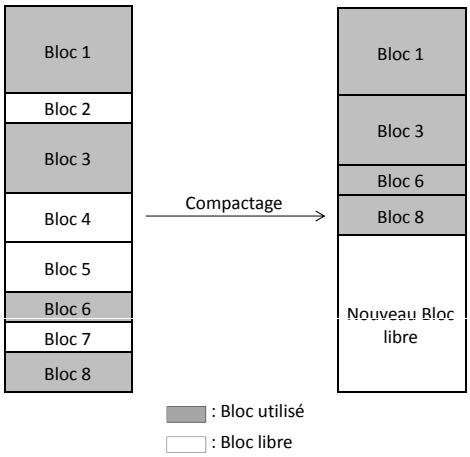

Figure 3.5 – Compactage de la mémoire.

Si la technique de compactage apporte une solution au problème de fragmentation, elle présente cependant des inconvénients majeurs car elle monopolise à la fois le CPU, La RAM et le bus système durant les opérations de transfert de données dont le temps est d'autant plus important si les pages mémoires correspondant aux blocs ne sont pas chargées en mémoires alternatives (mémoires cachees). Le système d'exploitation doit donc, les rapatrier depuis leur support de stockage de base (souvent disque dur) vers la mémoire centrale. La Figure 3.6 montre la consommation du temps CPU en fonction de la taille de mémoire compactée. Les tests ont étaient réalisés sur une machine Linux (Fedora 14), CPU 2.53 GHZ, RAM 256Mo :

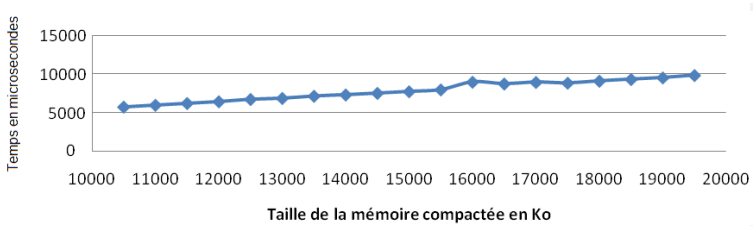

Figure 3.6 – Temps CPU consommé en fonction de la taille mémoire compactée.

3.3 Politiques d'allocation dynamique de mémoire

Le rôle d'un allocateur dynamique de mémoire (sigle anglais de DMA : Dynamic Memory Allocator) consiste à gérer, le plus efficacement possible, l'espace mémoire Heap d'un processus. Le qualificatif *dynamique*attribué au DMA exprime sa capacité d'allocation de mémoire durant l'exécution d'un programme.

Tout DMA repose sur un algorithme qui décrit sa propre politique de gestion d'espace d'adressage. De nombreuses politiques ont été proposées, et en terminologie chacune d'elles prend une désignation particulière. Elles sont classées suivant les 5 catégories indiquées ci-dessous, le reste est généralement une combinaison ou une variante de ces classes de base (pour plus de détails, se référer au panorama de Wilson [WJNB95a]).

1. Politique d'allocation contigüe : First-Fit, Next-Fit, Best-Fit, etc.
2. Politique d'allocation par subdivision : Binary Buddy and Double Buddy [YS10].
3. Politique d'allocation par listes indexées : Quick-Fits and Fast-Fits.
4. Politique d'allocation à champs de bits.
5. Politique d'allocation non-contigüe : Pagination.

Nous présentons dans ce chapitre un rapide tour d'horizon sur les principales politiques d'allocation dynamique de mémoire. Cette étude a pour objectif de nous fixer sur le choix de la stratégie la plus adaptée à notre besoin.

3.3.1 Politique d'allocation contigüe

3.3.1.1 First Fit (Première zone libre)

La politique d'allocation First Fit repose sur l'utilisation d'une liste de partitions libres. A chaque demande de réservation, l'algorithme parcours la liste depuis le début et s'arrête dès qu'il trouve une partition de taille assez grande pour contenir l'espace demandé. Si la taille de la partition choisie est trop grande par rapport à celle demandée, elle sera divisée en deux partitions afin de réduire la taille de l'espace non utilisé. Si cet algorithme est à la fois rapide (Temps de recherche faible) et facile à mettre en œuvre, il cause, d'une part, beaucoup de fragmentation externe puisque les partitions en fin de liste sont difficilement accessibles et d'autre part celles du début de liste sont découpées en une multitude de petits fragments difficilement exploitables, augmentant ainsi le temps de recherche [Her09].

3.3.1.2 Next Fit (Zone libre suivante)

La politique Next Fit est une implémentation légèrement différente de First Fit. Son algorithme ressemble beaucoup à celui décrit précédemment dans la mesure il tente de trouver la première partition suffisamment grande par rapport à demande, sauf que la recherche commence depuis la position de la dernière partition choisie. Généralement, cette technique utilise liste circulaire pour stocker la liste des partitions libres, et un pointeur de positionnement pour référencer la dernière partition choisie.
Cette politique d'allocation permet de remédier au problème d'accumulation des petits segments en début de liste, ainsi que de réduire le temps de recherche des partitions à allouer. Cependant, elle favorise la dispersion des fragments sur la mémoire, ce qui rend donc leur regroupement plus difficile [KSZ05].

3.3.1.3 Best-Fit (Meilleur ajustement)

La politique Best-Fit elle aussi, est basée sur une liste de partitions libres. Pour allouer une nouvelle partition, l'algorithme parcours l'intégralité de la liste et choisi la plus petite partition suffisamment grande pour contenir l'espace demandé. Si la taille de la partition demandée est trop petite par rapport à celle de la partition choisie, cette dernière sera divisée en deux partitions. Cet algorithme a tendance à préserver les grandes partitions pour d'éventuelles exploitations, avec le risque de fractionnement de la mémoire en plusieurs petites zones difficilement exploitables.

3.3.1.4 Worst-Fit (plus grand résidu)

L'algorithme Worst-Fit consiste à prendre la plus grande partition libre afin d'avoir le plus grand espace mémoire libre. Cet espace sera ensuite utilisé pour créer de nouvelles partitions. plusieurs travaux de recherche montrent que cette technique ne donne pas de bons résultats.

3.3.2 Politique d'allocation par subdivision (Buddy Systems)

La politique Buddy Systems consiste à subdiviser systématiquement l'espace mémoire libre en deux partitions égales qui auront généralement une taille de 2^n . Buddy Systems se distingue des autres politiques (First-Fit, Best-Fit,...) par l'utilisation de structures arborescentes pour gérer les différentes partitions, ce qui permet d'accélérer les temps de recherche, ainsi que de faciliter le regroupement des partitions non utilisées. Nous présentons deux variantes de la politique Buddy Systems [YS10].

3.3.2.1 Binary Buddy

L'espace mémoire total est représenté par un arbre binaire récursif et à chaque niveau, il est divisé en deux parties égales. Quand une demande d'allocation se présente sa taille est arrondie à la puissance de deux directement supérieure. L'arbre binaire est découpé récursivement jusqu'à ce qu'une partition de taille égale à la valeur arrondie soit rencontrée. Pour créer une partions de taille 2^x on subdivise un bloc de taille 2^{x+1}, s'il n y a plus de blocs de taille de taille 2^{x+1} on découpe un bloc de taille 2^{x+2} et ainsi de suite. Dans le cas où deux blocs adjacents se libèrent, ils seront regroupés en un seul bloc de taille deux fois plus grande.

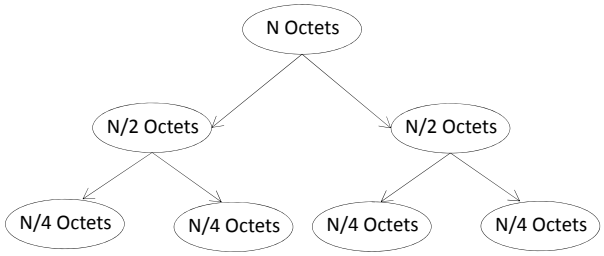

Figure 3.7 – Organisation des blocs mémoires relatifs aux partitions.

L'algorithme Buddy-Systems apporte quelques améliorations, contrairement aux algorithmes précédents (Firs Fit, Worst Fit,...). Cependant, l'arrondissement de la taille des partitions à une puissance de 2 n'est pas sans conséquence, en effet, il engendre une certaine fragmentation interne. La Figure 3.8 décrit le déroulement de la séquence d'allocation mémoire.

Taille totale	1Mo					
Demande A=111k	128k	128k	256k	512k		
Demande B=230k	128k	128k	256k	512k		
Demande C=52K	128k	64k	64k	256k	512k	
Demande D=256k	128k	64k	64k	256k	256k	256k
Libération B	128k	64k	64k	256k	256k	256k
Libération A	128k	64k	64k	256k	256k	256k
Demande E=80k	128k	64k	64k	256k	256k	256k
Libération C	128k	128k		256k	256k	256k
Libération E	512k				256k	256k
Libération D	1Mo					

Figure 3.8 – Exemple de fonctionnement de l'algorithme Buddy Systems.

3.3.2.2 Double Buddy

La politique Double-Buddy, comme Binary-Buddy utilise le même principe d'arrondissement de la taille des demandes : Deux arbres binaires sont utilisés pour réduire le taux de fragmentation interne engendrée par la politique Binary-Buddy, et chacun est structuré en fonction d'une valeur de base différente, par exemple, le premier utilise des puissances de 2 (2, 4, 8,...) et le deuxième utilise des puissances de 2 sur une base égale à 3 (3, 9, 27, ...). Cette stratégie donne une meilleure distribution des blocs mémoires sur le tas et permet de mieux les échelonner, réduisant ainsi, le taux de fragmentation interne. Ce système présente, néanmoins, un inconvénient : Le regroupement des partitions appartenant à deux bases différentes (2 et 3) n'est plus possible, ce qui augmente le taux de fragmentation externe.

3.3.3 Politiques d'allocation par listes indexées

Les politiques d'allocations présentées dans la Section 3.3.1 reposent sur l'utilisation d'une liste linéaire pour référencer les partitions libres. Une des améliorations envisagées, consiste à utiliser des structures de données plus évoluées, permettant l'indexation des partitions libres en fonction des paramètres utilisés par l'algorithme d'allocation (Tailles des partitions, Adresses début des partitions,...). Nous appelons ce type de politiques « politiques d'allocation indexées ».

3.3.3.1 Quick Fits

Quick Fit peut être considérée comme une version améliorée de l'algorithme Best Fit. Les partitions sont organisées dans un arbre binaire indexé (trié) en fonction de leurs tailles. Il n'est donc plus nécessaire de parcourir la totalité de la liste, pour trouver la partition qui correspond au mieux à la demande. Le temps de recherche augmente de manière logarithmique avec le nombre de partitions libres indexées dans l'arbre. L'algorithme Quick Fits ne prend pas en considération les adresses du début de partitions. Par conséquent, le regroupement des partitions adjacentes libres n'est plus possible.

3.3.3.2 Fast-Fits

La politique Fast-Fit est une version améliorée de Quick-Fits. Elle utilise un arbre cartésien pour indexer les différentes partitions libres. L'arbre cartésien permet l'indexation des éléments en fonction de deux paramètres. Ses éléments sont alors, totalement triés en fonction du premier paramètre, et partiellement triés selon le second. Dans l'algorithme Fast-Fits, le premier paramètre d'indexation (paramètre principale) est représenté par la taille des partitions libres,

le second (deuxième paramètre d'indexation) par leurs adresses, ce qui rend possible le regroupement des partitions adjacentes libres puisqu'elles sont ordonnées en fonction de leurs adresses.

3.3.4 Politiques d'allocation à champs de bits

Le principe fondamental de cette politique, repose sur l'utilisation d'un vecteur de bits, pour représenter les différentes zones mémoires libres. Chaque bit indique la position et l'état d'un bloc mémoire : 0 si le bloc est libre, 1 s'il est occupé. La taille des blocs est généralement fixée à 8 Octets. La Figure 3.9 donne un exemple de répartition de blocs ainsi que le vecteur de bits correspondant.

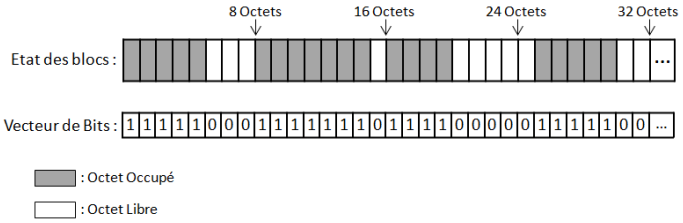

Figure 3.9 – Exemple de correspondance entre : Vecteur de Bits et blocs mémoires.

Cette politique offre deux avantages. D'une part, elle est simple à mettre en œuvre, d'autre part, elle permet d'optimiser la taille des structures utilisées pour le référencement des partitions libres. Cependant, cet algorithme souffre d'un problèmes majeur dans la mesure où il doit parcourir l'intégralité du vecteur de bits dans le but de trouver dans le but de trouver N blocs libres successifs (N zéro successifs) lorsqu'une demande d'allocation de taille N Octets se présente, ce qui impacte considérablement le temps de recherche.

3.3.5 Paramètres influençant les performances d'un allocateur

Nous avons présenté dans les sections précédentes une vue d'ensemble sur les différentes politiques d'allocation de mémoire. Dans la partie suivante, nous nous intéressons à certains paramètres qui ont une influence sur les performances d'un allocateur et qui sont au nombre de quatre :

3.3.5.1 Organisation des partitions dans la liste

La performance d'un algorithme d'allocation est sensiblement liée à la politique d'organisation des partitions libres dans une liste. Trois stratégies peuvent être citées, classées par ordre croissant d'efficacité du point de vue fragmentation, selon plusieurs études, il a été prouvé qu'au plus bas de l'échelle : l'organisation en LIFO, celle en FIFO donnant de meilleurs résultats que la première et enfin celle qui se base sur un tri par adresse étant la plus efficace de toutes, en plus elle facilite le regroupement des partitions libres adjacentes.

3.3.5.2 Découpage des partitions libres

Les tailles des demandes d'allocation correspondent rarement aux tailles exactes des partitions attribuées. L'algorithme d'allocation doit donc fixer un seuil à partir duquel, la partition sera découpée. Un seuil très bas, cause moins de fragmentation interne, mais conduit dans certain cas à l'apparition de multiples partitions minuscules, difficilement exploitables. De plus, le découpage

de partitions nécessite un coût de traitement supplémentaire, qui dégrade la performance de la politique d'allocation. Inversement, un seuil très élevé, engendre un taux plus important de fragmentation interne.

3.3.5.3 Conservation des partitions libres

L'idée consiste à favoriser l'utilisation des anciennes partitions, de manière à conserver le maximum d'espace libre contigüe en mémoire, ce qui facilite la réduction du taux de fragmentation. Les politiques Firs Fits et Best Fits intègrent nativement cette stratégie alors que Next Fit ne la prend pas en compte.

3.3.5.4 Regroupement des partitions libres adjacentes

Le regroupement des partitions libres adjacentes est un moyen pour lutter contre la fragmentation ex-terne. Cette opération ne peut être appliquée que si les partitions sont triées en fonction de leurs adresses. L'algorithme d'allocation doit définir une stratégie précise de regroupement. Plusieurs solutions sont possibles : Le regroupement peut se faire soit au moment de la libération d'une partition, quand il ne reste plus une de libre suffisamment grande pour contenir l'espace demandé, soit lorsque la taille de la mémoire utilisée atteint un certain seuil.

3.3.6 Politique d'allocation non-Contigüe

Les algorithmes que nous avons vus jusqu'à présent considèrent les partitions comme un espace contigu, ce qui augmente le risque d'apparition de fragments internes ou externes. Pour remédier à ce problème, l'espace d'adressage doit être considéré comme étant un espace non-contigu, c'est le principe de fonctionnement de la pagination [DM99] dont nous allons essayer de comprendre le mode opératoire sans trop rentrer dans les détails de la description du mécanisme de pagination utilisé dans les systèmes d'exploitation.
Le principe de la pagination consiste à structurer l'espace d'adressage en un ensemble d'unités de tailles égales appelées pages, pas nécessairement successives, qui composeront l'espace mémoire attribué à une partition qui n'est donc plus représentée par un pointeur de départ et une taille. Ces tables son généralement référencées dans une table appelée "table des pages" dans laquelle sont enregistrées toutes les correspondances "numéro de table −> numéro de partition". La technique de la pagination est souvent combinée avec d'autres notions comme :

- La segmentation : Moyen de regroupement des tables de pages
- Le swaping : Utilisation d'espace disque pour chargement et déchargement des pages.

Ces notions constituent les briques de base du concept de la mémoire virtuelle qui permet l'exécution des programmes dont la taille dépasse celle de la mémoire physique réelle et facilite à la pagination :

- Chargement à la demande : les pages ne sont chargées qu'après être référencées.
- Étendre l'espace d'adressage réel (physique).
- Transparence : L'utilisateur n'a plus à gérer explicitement ses partitions.

3.3.7 Travaux relatifs

Les travaux de recherche présentés dans [hCZ11] décrivent le mécanisme des mémoires partagées pour les trois noyaux suivants : L4, L4FiFO, et Minix3. Les auteurs proposent aussi un nouveau mécanisme de gestion des segments partagés dont le but est de réduire le nombre de copies mémoires depuis l'espace utilisateur vers l'espace système et inversement (depuis l'espace

système vers l'espace utilisateur). A ce niveau, il faut signaler que notre allocateur mise sur l'utilisation du standard IPC système V qui optimise implicitement le nombre de recopies mémoires grâce à un mécanisme de correspondance direct entre les adresses d'attachement virtuelles et les adresses globales (voir Figure 3.1).

Nous pouvons mentionner aussi les travaux de recherche présentés dans [Ron03] qui proposent un nouvel allocateur plus maniable à l'usage des mémoires partagées, basé sur la template STL (Standard Template Library), et grâce auquel, plusieurs processus peuvent partager un ensemble d'objets sur la même machine hôte, cependant il n'est pas sans inconvénient : En effet, il ne supporte pas le partage de pointeurs et impose l'appartenance des données manipulées aux types de base. Ce qui explique d'ailleurs que les auteurs de cet allocateur ont présenté toute une gamme de tests, mais uniquement sur des types de base (Integer).

Dans [FMMA11], les auteurs présentent une étude comparative de 7 allocateurs dynamiques de mémoire, à savoir : Hoard, Ptmalloc, Miser, TCMalloc, Jemalloc, Ptmalloc, and TLSF. Les tests sont réalisés sur des applications concurrentes réelles. Cependant, ces algorithmes ne sont appliqués que sur l'espace mémoire heap. Par conséquent, nous ne pouvons donc pas faire une comparaison directe avec les résultats présentés dans cette étude.

Les travaux de recherche présentés dans [BDC10] proposent une nouvelle politique d'allocation dynamique de mémoire dont les performances sont évaluées sur des blocs mémoires de 32 Octets. Cette taille est trop faible par rapport à celle envisagée par notre allocateur. De plus, cette politique a été exclusivement appliquée sur la mémoire heap. Par conséquent, ses performances ne peuvent être garanties dans notre contexte.

Dans [RKLC08], les auteurs présentent un nouvel allocateur dynamique de mémoire intelligent destiné aux systèmes embarqués qui ont une mémoire de faible taille. Il repose sur l'utilisation conjointe d'une table de recherche avec d'un champ de bits pour optimiser les temps de gestion des blocs mémoires. Une fois de plus, les auteurs n'abordent pas le problème d'allocation de mémoire partagée et se contentent de la mémoire heap.

3.3.8 Bilan

Nous constatons depuis cette état de l'art que les approches d'allocation peuvent être classées en deux catégories : contigüe et non-contigüe. L'espace d'adressage est souvent mal exploité dans les approches contigües. En effet, les problèmes de fragmentation interne/externe sont très difficiles à contourner, et Il y a toujours un compromit entre l'efficacité de l'algorithme déployé (Temps de réponse) et le taux de fragmentation interne/externe engendré, plus le temps de réponse est court, plus le risque de fragmentation devient important. L'algorithme idéal n'existe pas à partir du moment où l'on se place dans un environnement fortement dynamique.

L'espace d'adressage est cependant bien exploité par les approches d'allocation non-contigüe comme celle de la pagination qui élimine complètement le problème de fragmentation externe puisque les pages ont une taille uniforme. Le seul cas de fragmentation interne possible est celui qui concerne la dernière page d'une partition si elle n'est pas totalement remplie.

Par ailleurs, nous remarquons aussi que la majorité des allocateurs sont destinés à la gestion d'espace mémoire heap. Chaque processus dispose de son propre espace heap qui a une visibilité interne et ne peut pas être atteint depuis un autre processus hôte, aucun de ces allocateurs ne touche au problème d'allocation de segments de mémoire partagées. Cela ne correspond pas aux besoins décrits dans notre cahier de charges, dans lequel nous souhaitons mettre en œuvre un mécanisme de partage des segments mémoires qui ont une visibilité globale dans le système et peuvent être partagés par tous les processus de la même machine hôte.

Vu tout l'avantage offert par les stratégies d'allocation non-contigüe, et l'intérêt particulier

qu'elles suscitent, nous avons opté pour une approche d'allocation par pagination pour bénéficier de tous les atouts offerts par ce modèle. La mise en œuvre d'une telle approche nécessite l'intégration d'un mécanisme de pagination au dessus de celui des mémoires partagées afin d'assurer une meilleure gestion de l'espace mémoire d'un TBA.

3.4 Contexte d'utilisation de notre allocateur

Nous avons présenté dans le chapitre 1 l'architecture logicielle de notre bus applicatif dont l'intérêt est de faciliter l'intégration d'applications métiers dans des environnements hétérogènes et distribués. Dans cette architecture, plusieurs processus peuvent entrer en interaction dans le but de réaliser un objectif commun. Pour exploiter la puissance de traitement offerte par les machines multicœur, les applications sont parcellisées dans la mesure du possible en un ensemble de tâches parallèles. Le rôle du Bus applicatif à ce niveau est d'assurer l'acheminement de tous les messages qui circulent entre tous les threads interconnectés à ce bus. Si nous faisons abstraction des échanges distants (nécessitant une couche de communication réseau) et nous considérons uniquement les échanges locaux qui se déroulent entre les différents threads de la même machine hôte, nous pouvons identifier les cas d'échange suivants :

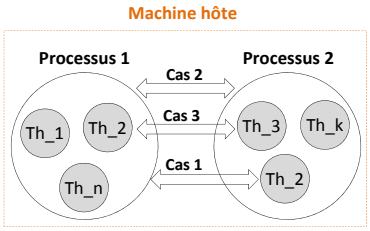

Figure 3.10 – Les différents cas de communication en mémoire partagée.

- Case 1 : Communication entre un thread et un processus.
- Case 2 : Communication entre deux processus différents.
- Case 3 : Communication entre deux threads appartenant à deux processus différents.

3.5 Présentation de notre solution

Nous proposons un mécanisme d'allocation dynamique de mémoire basé sur une approche par pagination [DM99]. Le déploiement d'une telle approche permet d'une part, de résoudre de manière efficace les problèmes de fragmentation engendrés par les allocations successives de mémoire, d'autre part, il offre une meilleure distribution des zones d'adressage allouées sur l'espace mémoire total disponible.

Notre allocateur met à disposition des programmeurs une API qui contient un ensemble de fonctionnalités de base qui facilite la manipulation des canaux du terminal, et qui a été développée et testée sur trois systèmes d'exploitation : Windows, Mac-OS et Linux. Les fonctionnalités de notre allocateur regroupent les tâches suivantes :

- Création, allocation, et ré-allocation dynamique de l'espace mémoire d'un canal d'échange.
- Transfert des données depuis l'espace utilisateur vers l'espace d'adressage des canaux d'échanges.
- Description du type de la donnée stockée au niveau d'un canal d'échange, ce qui permet aux autres processus d'avoir une bonne interprétation de la donnée partagée.

– Assurance de la cohérence des données stockées dans les canaux d'échange dans le cas d'un accès concurrent par multiple threads.
– Destruction et libération des ressources associées à un canal d'échange.

L'espace mémoire assigné à un canal d'échange n'est donc plus représenté par un zone d'octets contiguë identifiée par un pointeur de départ et un offset, mais par un ensemble de pages de tailles égales, qui ne sont pas forcement successives au niveau du segment de mémoire partagée qui les héberge, et dans ce cas, une phase de concaténation est nécessaire pour regrouper le contenu des différentes pages éparpillées, assurée par notre allocateur sans aucune charge pour les programmeurs.

La taille des pages est un critère paramétrable, fixé au moment de la création de la TDI et l'idéal serait qu'elle soit définie en fonction de la taille moyenne des données à échanger sur les canaux.

Il faut signaler que le compromis entre la taille des pages et celle de la TDI est toujours présent et que la seule solution équitable pour le résorber est de trouver un certain équilibre entre la taille attribuée aux pages et celle associée aux données à échanger car, en effet, plus les partitions sont de petite taille plus le nombre de pages à indexer augmente et par conséquent les tables d'indexation deviennent plus volumineuses et inversement.

Le mécanisme de pagination implémenté dans notre allocateur repose sur l'utilisation de deux tables de données : la table des indexes (TDI) et la table des données (TDD). La TDI est une table de description qui permet d'une part, d'indexer tous les canaux d'échange d'un terminal afin de minimiser les temps de recherche, d'autre part, de détenir tous les paramètres de configuration comme : les entêtes, taille des pages, nombre de canaux actifs, etc. La TDD quant à elle, est utilisée pour stocker le contenu des pages associées aux différents canaux d'échange d'un terminal.

La Figure 3.11 donne un exemple de répartition des pages sur la TDD à travers laquelle nous pouvons constater que l'espace mémoire d'un canal est représenté par un ensemble de pages qui ne sont pas nécessairement successives, remédiant ainsi tout problème de fragmentation externe.

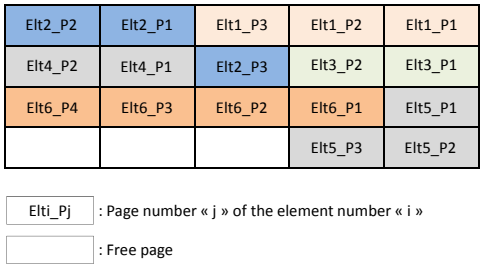

Figure 3.11 – Exemple d'affectation des pages dans la TDD.

Nous savons désormais que la principale limitation du mécanisme des mémoires partagées réside dans son incapacité à supporter le partage de pointeurs, ce qui réduit le choix de la structure de données utilisée au niveau de la TDI pour l'indexation des différentes pages de la TDD. Ainsi, l'utilisation de structures de données complexes telles que les listes chaînées n'est pas possible. Nous avons donc utilisé des listes chaînées particulières, dans lesquelles les champs de type « adresse » sont remplacés par des indices de décalage (Offset en Anglais). En partant sur cette base, nous avons proposé un premier prototype de la TDI, dans lequel, les différentes pages de la TDD sont maintenues dans une table contiguë et où chaque entrée fait référence à une page unique au sein de la TDD. La constitution du premier prototype de la TDI que nous avons mis au point est illustrée par la Figure 3.12 :

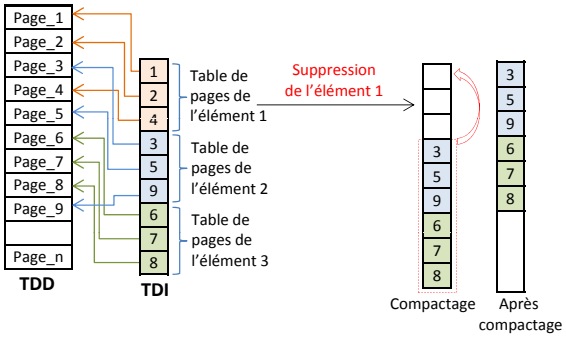

Figure 3.12 – Problème de compactage dans la TDI.

Malheureusement, nous nous sommes rapidement rendu compte que cette solution présente un inconvénient majeur : la réapparition des problèmes de fragmentation et de compactage au niveau de la TDI, après avoir été éliminés de la TDD. En effet, si nous regardons bien la Figure 3.12, nous constatons que la suppression du canal numéro 1 conduit à l'apparition d'une zone fragmentée qui doit être compactée afin de rester disponible à une une éventuelle allocation ultérieure. Suite à ce constat, nous avons proposé une nouvelle architecture pour la TDI, causant moins de fragmentation, toujours basée sur des structures de données de type liste chaînée, dans lesquelles, les champs *liens* sont remplacés par des offsets. Ainsi, tout changement apporté à la structure de la TDI, est réalisée par une simple modification des offsets de décalage, toutes les zones mémoire restant à leur place. La Figure 3.13 présente la nouvelle composition de la TDI.

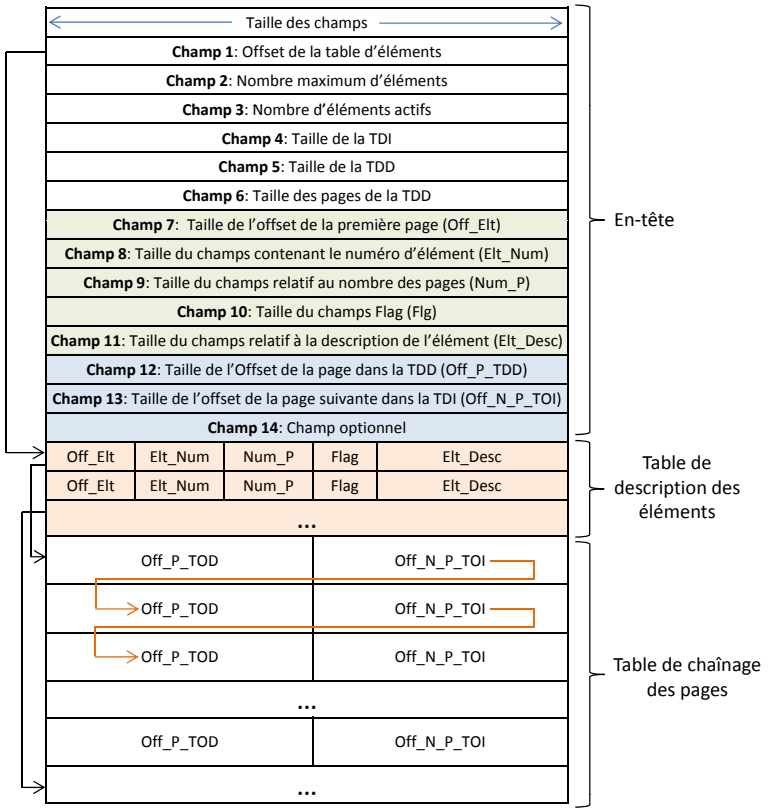

Figure 3.13 – Structure de la TDI.

La description des différents champs qui composent la TDI est présenté dans le tableau récapitulatif suivant :

Nom du champs	Description
Taille Champs	Taille en Octet de tous les autres champs de la TDI
Offset bloc canaux	Offset d'accès à la table de description des canaux utilisée pour indexer toutes les canaux dont le contenu est stocké dans la TDD
Nombre canaux	Nombre maximal de canaux qui peuvent être référencés
Nombre canaux exploités	Nombre de canaux réellement exploités par les tâches
Taille bloc données	Taille mémoire totale réservée pour la TDD
Taille page données	Taille des pages manipulées dans la TDD
Taille offset partition	Taille du champs offset dans la table de description des canaux
Taille numéro canal	Taille du champs réservé au numéro du canal, unique identifiant utilisé
Taille nombre pages	Attribut indicateur de la taille du champs qui détient la totalité des pages réservées à un canal
Taille drapeau	Taille du champs drapeau indicateur de l'état du canal, et pouvant avoir deux valeur : équivalente à 1 si le canal est occupé, zéro sinon.
Taille extension	Taille du champs extension contenant des informations complémentaires concernant le type des données stockées dans le canal.
Taille Numéro Page	Taille du champs contenant l'offset qui donne accès à la page dans la TDD
Taille Lien Page	Taille du champs indiquant l'offset vers la page suivante dans la table de chainage des pages
Offset	Champs qui donne accès à la tête de liste contenant toutes les pages qui composent un canal donné
Nb Pages	Nombre de pages qui composent le canal d'échange en question
Drapeau	Drapeau indiquant l'état (libre ou occupé) du canal d'échange
Extension	Champs descriptif permettant d'indiquer le type des données en mémoire
Offset bloc Données	Offset donnant accès à la page stockée dans la TDD
Offset Élément suivant	Offset permettant de retrouver l'élément suivant dans la liste chainée

Tableau 3.1 – Description des différents champs de la TDI.

Nous avons choisi d'adopter un mécanisme de référencement par adressage absolu dans lequel tous les champs offset sont représentés en décimal, ce qui crée moins de dépendances entre les différents champs qui donnent accès aux canaux d'échange indexés dans TDI. Ainsi, l'accès au contenu d'un canal peut être réalisé de manière indépendante des placements affectés aux autres canaux. Le processus de reconstitution, par notre allocateur, des données d'un canal existant, se fait en deux phases, et nécessite l'utilisation des deux tables décrites précédemment (TDI et TDD) :

3.5.1 Phase de parcours

Cette phase a pour objectif de récupérer les références de toutes les pages qui composent un canal donné. Pour cela, l'allocateur commence par se positionner sur le début de la TDI afin de récupérer la taille des différents champs qui la constituent. Le champs "Offset bloc canaux" est particulièrement intéressant car il donne accès à la table de description des canaux, dont

le parcours oriente vers l'entrée correspondante au canal recherché. Cette entrée contient de nombreuses informations sur ce canal, notamment l'offset qui pointe sur la tête de liste dans la table de chainage de pages elle-même détenant les références de toutes les pages qui composent le canal.

3.5.2 Phase d'agrégation

Toutes les pages trouvées dans la phase de parcours vont être rassemblées en un seul bloc de données, qui sera renvoyé comme résultat. A ce niveau, deux cas de figure peuvent se présenter :

- Les pages sont positionnées de manière successive dans la TDD, dans ce cas, aucune concaténation de pages n'est nécessaire, et la phase d'agglomération se résume simplement au renvoi de la référence de la première page.
- Les pages relatives au canal recherché sont dispersées, et dans ce cas, le processus d'agglomération est plus long, et consiste à rassembler les données de toutes les pages en un seul tas dont la constitution se fait de manière incrémentale en recopiant au fur et à mesure le contenu des pages éparpillées sur la TDD et qui sera renvoyé en tant que réponse.

3.5.3 Exemple

Dans cette section, nous présentons un exemple simple de TDI, sur laquelle nous déroulerons l'algorithme de notre allocateur afin de récupérer le contenu du canal numéro 2, tout en mettant à l'épreuve, les deux phases de traitement citées précédemment. Toutes les adresses manipulées sont indiquées par le caractères "@". L'adresse @1 (0xb74ef000) pointe directement sur le début de la TDI, et représentera l'adresse de base. La composition de la TDI est donnée par la Figure 3.14.

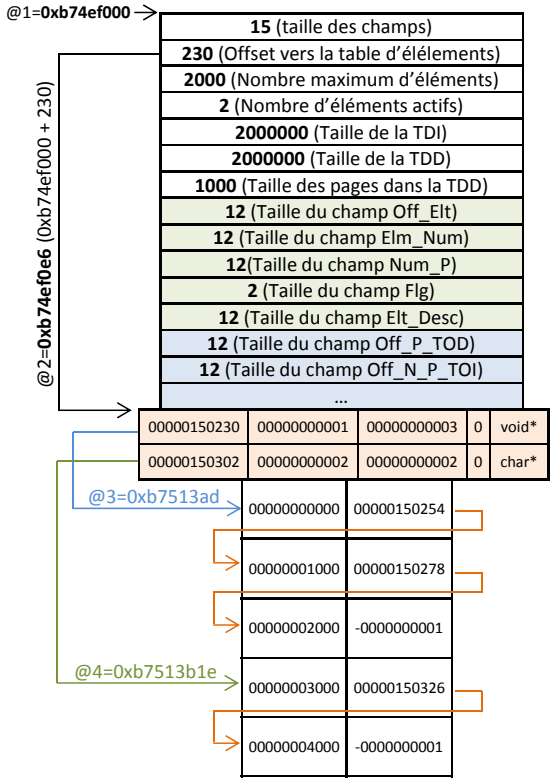

Figure 3.14 – Composition de la TDI.

Durant la phase de parcours, notre allocateur commence par extraire le premier champ de la TDI qui indique la taille de tous les champs qui le suivent qui est de l'ordre de 15 Octets, puis avec cette valeur, il se positionne sur le deuxième champ qui vaut 230 (en Décimal), et trouve accès, ainsi, à la table de description des canaux indiquée par l'adresse @2 (0xb74ef0e6) qui sera parcourue à la recherche d'une entrée relative au canal désiré avec comme critère de recherche, le champ numéro canal :

- Si aucune entrée n'a été trouvée, un code d'erreur est retourné indiquant l'absence du canal recherché.
- Si le canal recherché est trouvé, notre algorithme procède à la récupération des références des pages stockées dans la TDD avec l'application du champ Offset de l'entrée retrouvée sur la tête de liste dans la table de chaînage des pages, ce qui donne la nouvelle adresse de branchement @4 (0xb7513b1e). Au passage, l'allocateur mémorise la valeur du champ description qui fournit des indications sur le type des données à interpréter ultérieurement. Le parcours de la liste de chaînage mentionnée précédemment permet de récupérer les offset de toutes les pages relatives au canal désiré indiquées par les offsets 00000003000 et 00000004000, et dont le contenu est stocké dans la TDD. C'est à ce niveau que la phase d'agrégation entre en jeu afin de concaténer le contenu des différentes pages. A l'issue de cette étape, le résultat se présentera sous la forme d'un tas composé d'une zone mémoire contiguë dont l'interprétation est fonction du champs description extrait précédemment ce qui constitue un atout supplémentaire pour notre allocateur, en effet, il donne aux programmeurs la possibilité de partager des données de type pointeur.

3.6 Évaluation de performance

3.6.1 Introduction

L'évaluation de performance d'une stratégie d'allocation de mémoire est un sujet qui a été largement étudié dans le passé. Ces études ont abouti à la mise au point de nombreuses approches d'évaluation, parmi lesquelles, on retrouve les démarches classiques de simulation, qui se basent sur un jeu de tests soumis à un programme : le simulateur reproduit le comportement d'une stratégie d'allocation de mémoire afin d'en mesurer les performances. On peut également citer celles dont le principe repose sur la mesure effective des différentes métriques de performance d'un allocateur à travers un ensemble d'exécution réelles. Deux démarches d'évaluation ont été proposées pour ce type d'approches :

- La première, au début des années 1960, dont le principe de base repose sur l'utilisation d'un générateur synthétique de requêtes d'allocation/libération [Col61]. elle présente l'avantage d'être facile à développer, favorise le calibrage des paramètres liés à la fréquence ainsi que la taille mémoire des requêtes générées par le simulateur, dans un soucis de meilleure analogie avec les demandes des applications réelles.
- La deuxième, en 1992, par Benjamin Zorn et son équipe dont les travaux ont abouti à l'élaboration d'une nouvelle démarche de teste, dans laquelle, l'évaluation de performance de la politique d'allocation de mémoire est exclusivement basée sur des applications réelles [GZ93, GZH93, DDZ94, BZ93]. Les résultats obtenus seront, par la suite, comparés à ceux du modèle synthétique. Effectivement, avec un taux d'erreur atteignant parfois les 20%, les auteurs de cette recherche sont parvenus à prouver que les résultats obtenus à partir d'une démarche synthétique ne correspondaient pas à ceux du modèle réel : Une conclusion intéressante donnant une meilleure fiabilité au modèle basé sur les applications réelles par rapport au synthétique.

Wilson et son équipe ont continué sur cette même piste, et on évalué près de 50 politiques d'allocation de mémoire différentes sur la base de 8 applications réelles [WJNB95b, Nee96, Joh97, JW98], tout en soulignant à chaque fois, l'inadéquation entre les mesures obtenues sur le modèle réel et celles du modèle synthétique, et toute l'insuffisance de ce dernier à reproduire fidèlement des comportements analogues à ceux d'une application réelle.

La performance d'un allocateur dynamique de mémoire peut être évaluée en fonction de plusieurs critères relatifs au temps de réponse, aux taux de fragmentation interne/externe, ou compactage [BSKM11], le premier étant considéré, dans la plupart des cas, le facteur le plus important dans une démarche d'évaluation. Nous avons donc volontairement écarté les deux derniers facteurs, pour ne garder que le facteur temps de réponse pour les raisons suivantes :

- L'allocateur dynamique de mémoire que nous proposons constitue une brique de base de l'architecture du bus applicatif, par conséquent, l'évaluation de performance de l'infrastructure dans son intégralité nécessite la connaissance du coût relative à chacun de ses mécanismes (gestion des communications, multiplexage/demultiplexage, ...) dont celui destiné à l'allocation dynamique de mémoire.
- Étant donné que nous avons opté dans notre allocateur pour une approche basée sur un mécanisme de pagination, les problèmes de fragmentation et de compactage ne se posent alors plus. Il n'est donc pas nécessaire de prendre en considération des facteurs comme celui du taux de fragmentation ou de compactage.
- La mise au point du mécanisme de pagination que nous proposons est passée par plusieurs étapes laborieuses avant d'arriver à la version version finale présentée dans cette section. Tout nouvel algorithme conçu ou toute nouvelle structure de données obéit en priorité à une évaluation des performances réalisées et en fonction desquelles sera mesuré le gain (ou perte) de temps réponse acquis entre l'ancienne version et la nouvelle, d'où l'importance d'une quantification du temps CPU consommé par notre allocateur.

3.6.2 Méthodologie

Dans notre étude, nous avons suivi une méthodologie fondée sur la mesure de performance en temps de réponse de notre allocateur en nous appuyant sur des applications réelles de la plateforme YAL [Alb10], et sur lesquelles nous recueillerons les temps CPU consommés par les différentes demandes d'allocation/libération de mémoire. La mise en pratique d'une telle démarche nécessite au passage, l'utilisation d'un mécanisme de capture de temps dont le rôle consiste à recueillir les durées de réponse de l'allocateur. Ainsi, à chaque appel d'une primitive d'allocation ou libération de mémoire au niveau de l'API correspond un appel d'une fonction de mesure assurant le recueil du temps CPU consommé. La phase d'évaluation constituera donc une tâche de fond qui peut se dérouler de manière indépendante lors de la mise au point des autres parties de la plateforme.

L'activation ou la désactivation de ce mode (évaluation) se fait par le biais de certains paramètres de configuration au moment de l'initialisation de l'allocateur. Ainsi, le résultat final est un fichier log contenant tout le comportement de l'application ainsi que les différentes mesures récoltées au passage, sur lesquelles, des valeurs moyennes seront calculées par la suite afin d'avoir des résultats plus pertinents.

3.6.3 Résultats expérimentaux et comparaison

Nous avons choisi une machine de type linux (Fedora 13) pour réaliser toutes les mesures car ce type de SE, en comparaison avec un SE type Windows, se caractérise par un haut degré de

précision, notamment grâce à la fonction clock_gettime, pour avoisiner une exactitude de l'ordre de la nanoseconde. Quant aux caractéristiques matérielles de la machine de tests, il s'agit d'un modèle Pentium 4, 3 Ghz CPU, 1 Go de Ram.

Le tableau 3.2 présente un récapitulatif des valeurs obtenues. La première colonne indique la taille de mémoire à réserver alors que la second représente le temps CPU consommé.

Taille mémoire à réserver en Ko	Temps de réponse en μs
16	14,18
32	17,61
64	20,53
128	24,4
256	27,11
512	29,91
1024	34,56
2048	39,05
4096	42,33
8192	46,72
16384	49,8
32768	63,65

Tableau 3.2 – Temps de réponse de notre allocateur en fonction de la taille mémoire à réserver.

Le contexte d'utilisation de notre allocateur diffère de celui des autres allocateurs classiques. Par conséquent, la comparaison directe des temps de réponse n'est pas une démarche pertinente, et ce, pour les raisons suivantes :

- **Niveau d'accès mémoire** :
  Les stratégies classiques d'allocation dynamique de mémoire sont destinées à la gestion de l'espace d'adressage heap d'un processus, qui rappelons-le, ne peut être partagé avec d'autres processus, à la différence de notre allocateur, qui assure la gestion d'un ensemble de segments de mémoires partagées, réservés au niveau global du SE, et qui sont accessibles à tous les autres processus de la machine hôte. Nous constatons donc à ce niveau, qu'il ne s'agit pas du même type de mémoire et par conséquent, les temps d'accès dans notre allocateur sont plus longs.
- **Contraintes de structuration de données** :
  Les problèmes de structuration et de description des données ne se posent pas dans les allocateurs classiques puisque l'interprétation des données se fait au niveau interne d'un même processus. Notre allocateur assure cette caractéristique par le biais d'un ensemble de champs descriptifs, qui révèlent aux autres processus externes la structure de la donnée partagée et dont la manipulation engendre nécessairement un coût supplémentaire qui affecte les temps de réponse en les accroissant considérablement.
  Nous avons remarqué que la partie dédiée au formatage des champs descriptifs nécessitait un temps de traitement important et ce malgré l'utilisation de la fonction basique de formatage sprintf qui d'une part, écarte toute hypothèse relative à une lourdeur au niveau des algorithmes implémentés, d'autre part, ne remet aucunement en cause l'efficacité du mécanisme de pagination proposé.
  Pour avoir une idée précise sur les temps consommés par la fonction sprintf, nous avons procédé à une série de tests simples, dans laquelle nous avons mesuré le temps CPU consommé

par rapport à la taille du champ à formater.
Pour une meilleure pertinence des résultats, chaque itération est exécutée 100 fois, en prenant à chaque fois, la valeur moyenne. Ces tests on été réalisés sur une machine Linux (Fedora 14), CPU 2.53 GHZ, RAM 256Mo. Les résultats sont présentés sur la Figure 3.15.

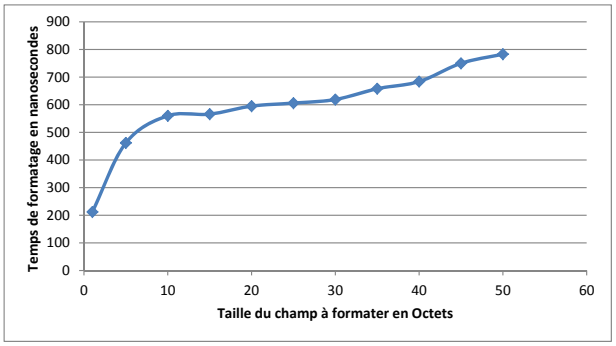

Figure 3.15 – Temps consommés par la fonction sprintf.

- **Structure de la TDI** :
  Comme nous l'avons expliqué dans la section 3.2.2, l'utilisation de pointeurs n'est pas possible. Une conséquence directe de cette caractéristique est qu'aucune structure de données complexe telle que les listes chaînées, ou arbre, ne peut être utilisée pour référencer les différentes pages indexées dans la TDI. Nous étions donc forcés de proposer une alternative dans laquelle les listes sont représentées à l'aide d'un ensemble d'offsets, dans laquelle, les opérations de mise à jour sont réalisées pas une simple modification des valeurs des offsets, toutes les zones mémoires restent à leur place. Cependant, l'inconvénient majeur de cette approche est que nous somme contraint de parcourir la TDI chaque fois qu'on souhaite lui apporter une nouvelle modification, avant d'appliquer le changement.

3.7 Conclusion

Dans cette section, nous avons présenté une nouvelle politique d'allocation dynamique de mémoires partagées, tout en exposant au passage, un état de l'art sur les politiques les plus connues. Cet allocateur a été conçu dans le but de remédier à certaines limitations du standard IPC-Posix, notamment concernant le mécanisme des segments partagés. En effet, ce dernier n'intègre aucune alternative pour lutter contre les problème de fragmentation ou de compactage, de plus, il ne permet pas le partage de données de type pointeur. L'originalité de l'algorithme que nous proposons réside dans l'utilisation d'un mécanisme de pagination au-dessus de mécanisme des mémoires partagées de base.
Les principaux objectifs de notre étude, fixés dès le départ et concernant d'abord l'élimination des problèmes de fragmentation et de compactage puis le partage de types complexes de données ont été réalisés avec succès. Il est aussi important de souligner qu'à travers cette étude, le différend entre l'efficacité de la stratégie d'allocation en termes de temps de réponse et la nature des données manipulées a été clairement établi. En effet, les performances de notre allocateur auraient été nettement meilleures si nous n'avions pas été obligés de manipuler des champs descriptifs. Cependant, et comme indiqué précédemment, cette contrainte reste primordiale dans le contexte d'un environnement massivement parallèle dans lequel plusieurs processus partagent

des structures complexes.

4

Le Terminal du Bus Applicatif (TBA)

Sommaire

4.1 Introduction

La notion de connecteur est née de la nécessité d'une interaction entre applications elles-mêmes ou entre applications et leur environnement extérieur. Ce connecteur s'impose, donc,

dès qu'il s'agit d'établir une interaction entre deux ou plusieurs applications qui ont besoin de communiquer ou de collaborer.

Cette nécessite de réciprocité de relations est inexistante dans les applications isolées pour qui le seul motif de liaison concerne leurs interactions avec les usagers, donc uniquement avec le milieu extérieur et dans ce cas le connecteur prend un aspect particulier généralement présenté sous forme d'une IHM (Interface Homme Machine).

Dans d'autres cas, le connecteur prend la forme d'une API (Application Programming Interface) composée d'une batterie de fonctionnalités. Elle peut être portable ou machine dépendante, libre ou propriétaire, exemple :

- **WinApi** : Fonctionnalités des différents systèmes d'exploitations de la famille Windows.
- **Socket** : Interface de programmation réseau.
- **OLE Automation** : Mécanisme de communication inter-processus basé sur la notion de Component Object Model (COM).
- **ODBC (Open Database Connectivity)** : Driver de bases de données.

L'incorporation des fonctionnalités de l'API dans le corps d'une application passe, dans la majorité des cas, par une phase de compilation, dans laquelle, le code de l'API est intégré à celui de l'application.

Dans une architecture EAI (Enterprise Application Integration) ou ESB (Enterprise Service Bus), le connecteur prend une forme plus élaborée que celle d'une API et correspond à des entités plus complexes offrant ainsi une fonctionnalité plus avancée. Les premiers prototypes de ce connecteur ont vu le jour au cours du développement des architectures d'intégration point-à-point, puis dans les architectures EAI, notamment pour répondre à des problèmes de standardisation.

La standardisation du connecteur est, certes, une condition nécessaire, mais pas suffisante en vue d'une amélioration des actuelles solutions ESB/EAI. En effet, un autre facteur tout aussi important est à prendre en considération : celui du niveau d'implémentation des services de base qui lui sont directement incorporés et qui se traduit par une différence des services offerts par chaque connecteur, rendant ainsi leurs gestions de plus en plus ardue. Cette caractéristique est illustrée par la Figure 4.1

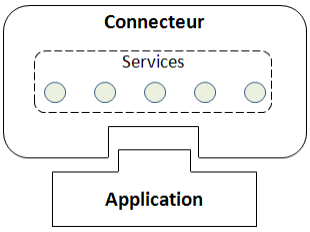

Figure 4.1 – Niveau d'implémentation des services au sein des connecteurs.

Pour remédier à ce problème, nous avons adopté une approche qui repose sur le principe d'externalisation de services, qui ne sont plus implémentés au sein du TBA, mais directement au niveau du MBA, facilitant ainsi leur gestion : l'ajout ou le retrait d'un service se fait en une seule fois, directement au niveau du MBA, pour éviter la répétition de cette même opération à chaque TBA qui ne sert que de passerelle vers le MBA et dont le rôle ne se limite plus qu'à l'invocation des services, au partage de données et à la synchronisation.

L'intégration de ce nouveau service au MBA ne signifie pas son accessibilité systématique à toutes les applications hôtes, qui restent tributaires des droits d'accès qui leur ont été affectés par l'administrateur du BA.

Par ailleurs, la granularité des unités de traitement prises en charge par le connecteur est un aspect très important qui mérite une attention particulière. En effet, avec l'introduction du parallélisme dans le traitement, les usagers ont besoin de faire collaborer des applications, mais aussi des threads, ce qui ne fait partie du cahier de charges des solutions ESB/EAI actuelles où les seules unités de traitement prises en charge par le connecteur sont les applications.

Avec l'introduction du concept de canal d'échanges parallèle, notre plateforme est suffisamment adaptée pour permettre aux différents threads d'une même application d'interagir en toute sécurité sur le BA et sans risque d'interférence de données.

Nous consacrons le chapitre suivant à la présentation du Terminal de Bus Applicatif (TBA) tout en illustrant la place prépondérante qu'il occupe dans notre plateforme. la composition de ce composant est entièrement basée sur des ressources IPC, avec toute sa richesse en termes de mécanismes de communication entre processus, qu'offre ce standard, le choix des ressources a été fait en fonction des résultats obtenus à l'issue du chapitre 2.

4.2 Définition et caractéristiques du TBA

Le Terminal du Bus Applicatif (TBA) joue le rôle de connecteur entre les tâches/applications et le BA, c'est une sorte de case d'accueil disposant des ressources qui leur sont nécessaires pour dialoguer avec leur MBA hôte, tout en leur donnant accès aux différents services du BA.

Le TBA est un connecteur universel compatible avec une large variété d'applications indépendamment de leurs domaines d'utilisation (gestion, finance,...), du système d'exploitation sur lequel elles sont déployées (Windows, Linux, ...) ou des caractéristiques matérielles sous-jacentes. Il est pensé de manière à ce que l'intégration des applications parallèle y soit réalisée de façon simple et sans pour autant bouleverser leurs codes respectifs. De plus, le TBA apporte des solutions simples aux problèmes causés par la programmation parallèle ; notamment ceux liés au partage de données et à la synchronisation des tâches.

Le cycle de vie du TBA est entièrement géré par le MBA depuis leurs créations jusqu'à leurs destructions, c'est le seul composant qui possède tous les droits d'accès sur les TBA. Pour des raisons de sécurité, les applications doivent d'abord s'authentifier (demande de LOGIN) auprès de leur MBA hôte avant de pouvoir manipuler des terminaux. Dans le cas idéal, chaque application signale explicitement sa terminaison (demande Logout) à son MBA hôte afin de lui indiquer qu'elle ne sent plus l'utilité de son terminal. En cas de Logout oublié ou d'exception à l'exécution, la récupération des ressources sera assurée par le ramasse-miettes du MBA.

4.3 Accès au TBA

Les tâches/applications n'accèdent pas directement aux ressources partagées d'un TBA, mais uniquement à travers l'API du bus applicatif tel qu'il a été indiqué par la Figure 1.15. L'API se présente sous forme d'une classe C++ outillée d'un ensemble de méthodes donnant accès aux ressources du TBA, elle doit être obligatoirement compilée avec le code de l'application avant de pouvoir accéder à ses fonctions. Cette caractéristique soulève une problématique pour les applications existantes qui n'ont pas connu notre API, et qui souhaitent utiliser la technologie du BA. Ce problème se pose notamment dans le cas où il n'est pas envisageable de procéder à une recompilcation du code des applications, dans une telle situation, l'accès au TBA se fait à l'aide de la technique de pilotage. Nous distinguons donc deux types d'applications :

4.3.1 Applications avec API incorporée

Ce type concerne toutes les applications qui ont connu l'API du BA et qui ont prévu son intégration dès la phase de développement. Le corps de l'API est donc systématiquement incorporé à celui des applications et en fait partie intégrante. L'intérêt majeur de cette approche paraît dans l'accès direct au TBA depuis l'application. Chaque fois qu'une nouvelle version de l'API est mise au point, le programmeur doit procéder à une recompilation du code afin de bénéficier de ses nouvelles fonctionnalités. Ce mode d'accès est illustré par la Figure 4.2

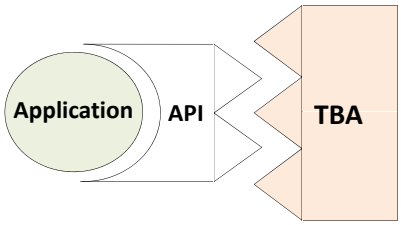

Figure 4.2 – Application avec API incorporée.

4.3.2 Applications sans API incorporée

Ce type concerne toutes les applications qui sont déjà opérationnelles alors qu'elles n'ont pas été développées avec l'API du BA. Dans notre terminologie, elles sont désignées par l'expression qualificative « applications aveugles ». Ces applications sont avantageusement prises en charge par d'autres applications tierces de type API incorporée qui leur offre l'adaptation nécessaire au BA. Une application tierce est en quelque sorte un pilote (driver) adaptant les applications aveugles au BA. Il faut signaler que la mise au point de ce procédé nécessite une double compétence de la part des programmeurs qui doivent non seulement maîtriser l'utilisation de l'API du BA pour la partie *application tierce* $\Rightarrow$ *TBA*, mais aussi l'usage d'une technologie d'automation comme OLE-Automation pour la partie *application tierce* $\Rightarrow$ *application aveugle*. Ce mode d'accès est présenté par la Figure 4.3

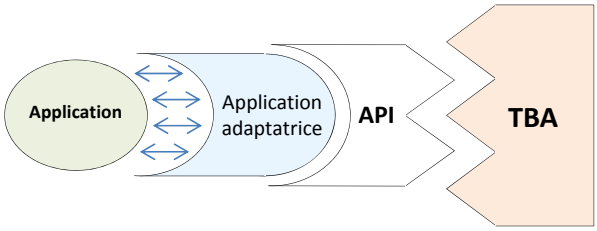

Figure 4.3 – Application aveugle pilotée par une application tierce.

4.4 Les canaux parallèles d'échange

Pour favoriser le traitement et l'échange massivement parallèles de données entre tâches, la solution la plus évidente consiste à partitionner l'espace mémoire total d'un TBA de manière à ce que chaque thread puisse disposer de son propre espace d'adressage. Un tel dispositif permet à l'ensemble des threads de la même application mère de fonctionner simultanément sur des

zones complètement disjointes. Dans notre terminologie, cette zone mémoire est désignée par l'appellation : canal d'échange parallèle qui définit la plus petite unité d'accès et de cohérence de données dans le TBA, et délimite l'espace de travail d'un thread.

Selon d'autres critères avantageux et non limitatifs de cette approche, plusieurs threads s'échangent simultanément des données, sans aucun risque d'interférence ou de violation de cohérence des données manipulées, lorsqu'un accès concurrent en lecture/écriture se présente.

Au sein d'un même TBA, chaque canal porte un identifiant unique (numéro) qui le distingue des autres. La Figure 4.4 montre ce concept de canal parallèle d'échange.

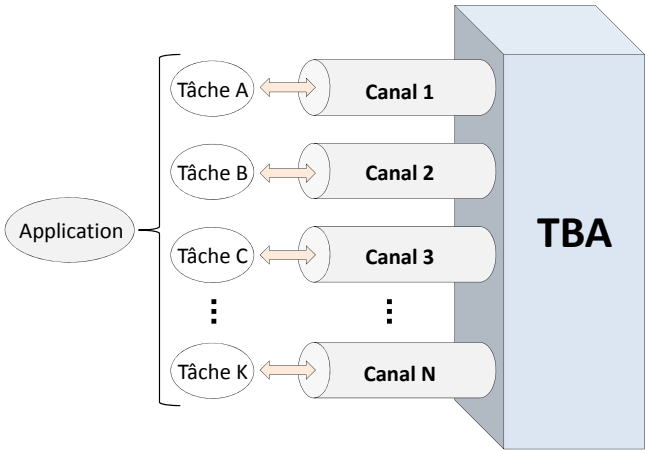

Figure 4.4 – Canaux d'échange parallèles du TBA.

Théoriquement, aucune limite particulière n'est imposée sur le nombre de canaux parallèles d'un TBA, mais sur le plan pratique elle dépend de la taille mémoire totale allouée au terminal, conformément à l'équation suivante :

$$\sum_{i=0}^{n} Taille\_Canal_i \leq Taille\_Totale\_TBA \tag{4.1}$$

L'usage des canaux parallèles du TBA se fait en exclusion mutuelle, il n'est donc pas autorisé d'utiliser un même canal par plusieurs threads simultanément. Lorsqu'un canal est sollicité par un thread donné, aucun autre thread ne pourra le réutiliser tant qu'il n'est pas libéré par le thread initial. La réservation d'un canal se fait lors du dépôt de la demande, alors que sa libération se fait après la récupération de la réponse. Il nous parait donc évident de refuser aussi les appels répétitifs, dans lesquels, un canal est sollicité par le même thread à plusieurs reprises successives, et sans qu'il n'y ait de récupération de réponses. Ce mode de fonctionnement est implémenté directement au niveau de l'API du BA garantissant ainsi une meilleure préservation des données stockées sur les canaux.
La gestion des canaux d'échanges parallèles est à la charge du MBA et comporte essentiellement les opérations suivantes : création ou suppression d'un canal, réattribution d'un canal libre existant à un autre thread de traitement, redimensionnement de la zone mémoire allouée à un canal. Ces fonctionnalités sont assurées par l'algorithme 1 décrit ci-dessous :

Algorithm 1 Gestion des canaux d'échanges parallèles.

```
if (le canal demandé n'existe pas) then
  if (la taille mémoire du canal demandé ≤ La taille mémoire disponible sur le TBA) then
    Création du canal demandé ;
    Mettre à jour le TBA (Taille mémoire disponibles, nombre de canaux,...) ;
  else
    Renvoyer un code d'erreur (Espace mémoire insuffisant) ;
  end if
else
  if (la taille mémoire du canal demandé ≤ La taille mémoire du canal existant) then
    Réattribution du canal existant à la nouvelle tâche ;
    Mettre à jour le TBA (Récupération de l'espace non utilisé) ;
  else
    if (Taille canal demandé ≤ Taille mémoire disponible sur le TBA + taille canal existant)
    then
      Redimensionnement de la taille du canal existant ;
      Réattribution du canal existant à la nouvelle tâche ;
      Mettre à jour le TBA ;
    else
      Renvoyer un code d'erreur (Espace mémoire insuffisant) ;
    end if
  end if
end if
```

4.5 Les différents types de canaux parallèles

Les canaux parallèles du TBA sont classés en deux catégories : canaux de services et canaux usagers bien distincts les uns des autres, les premiers étant propres aux services du BA et les seconds réservés au programmeur, mais quel que soit le type du canal considéré, son utilisation se fait toujours en respectant les deux règles de gestion (exclusion mutuelle, et d'appels répétitifs) mentionnées précédemment (section 4.4). La Figure 4.5 suivante illustre la répartition des canaux sur le TBA :

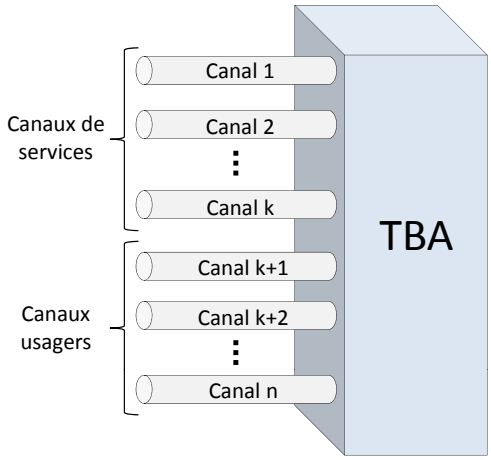

Figure 4.5 – Répartition des canaux du TBA.

4.5.1 Les canaux de services

Les canaux de services sont réservés au MBA et donnent accès aux différents services de base du Bus Applicatif. Chaque canal porte un numéro qui correspond à un service bien déterminé et qui appartient à un intervalle fixé par le MBA commençant par 1 et finissant par n, n étant le nombre total de services proposés par le BA. De plus, les correspondances *numéro de canal demandé* $\Rightarrow$ *service invoqué* sont rigoureusement identiques sur tous les MBA de la constellation. Le Tableau 4.1 suivant donne quelques exemples de correspondances :

N° Canal	Service invoqué	Description du service invoqué
1	Authentification	Authentification d'une application auprès du BA pour demander un terminal de bus
2	Récupération des versions	Récupération des versions relatives aux différents composants logiciels du BA tels que API, TBA, et MBA.
3	Demande de la liste des applications	Récupération de la liste d'applications autorisées pour l'application courante en fonction des droits attribués par l'administrateur
4	Demande de la liste des adresses	Récupération de la liste de tous les TBA connectés au BA, de manière à ce que l'application courante puisse avoir une vision globale de la plateforme
5	Demande de déconnexion	Action par laquelle l'application indique explicitement au MBA son désintérêt pour le TBA, ce qui déclenche le ramasse-miettes du MBA
6	Lancement d'une application distante	Lancement d'une application installée sur un MBA distant
7	Réception permanente	Réception à la volée de messages provenant d'autres tâches locales/distantes
8	Transfert (Upload) de fichiers	Transmission d'un fichier vers un MBA distant
9	Téléchargement (Download) de fichiers	Rapatriement d'un fichier situé sur un autre MBA distant
10	Création de référentiels	Création d'un référentiel (Alias)
11	Liste des référentiels	Récupération de la liste de tous les référentiels présents sur le système
...	...	...

Tableau 4.1 – Correspondance entre le numéro du canal sollicité et le service invoqué.

4.5.2 Les canaux usagers

Les canaux usagers sont destinés aux programmeurs, ils sont généralement utilisés pour l'échange de données entre les différentes ULT et permettent aux usagers d'une part, de définir leurs propres protocoles d'échange suivant un jeu de canaux bien déterminé, d'autre part, d'implémenter un mécanisme de gestion du flux standard : entrée, sortie et erreur de leurs applications indépendamment des canaux de base du MBA.

Par ailleurs, aucune contrainte particulière n'est imposée sur le choix des numéros de canaux

utilisés pourvu que ces derniers soient en dehors de l'intervalle réservé aux canaux de services. Les usagers ne sont donc pas obligés de respecter un ordre quelconque de nommage des canaux qu'il soit croissant ou décroissant.

La grande différence entre les canaux usagers et les canaux de services réside essentiellement dans la nature des données échangées. En effet, dans le cas des canaux usagers, le MBA ne confère aucune sémantique particulière aux messages manipulés. son rôle ne consiste ni à analyser ni à interpréter le contenu des messages usagers mais tout simplement à les acheminer vers leurs bonnes destinations (de toute façon, seules les tâches/applications sont en mesure d'interpréter ce contenu). Cette caractéristique augmente la généricité des canaux usagers en les rendant indépendants de la nature des données en leur possession.

4.6 Dictionnaire des services du BA

A priori, les tâches/applications ne connaissent pas (au moins pour la première fois) les correspondances *numéro du canal sollicité* $\Rightarrow$ *service invoqué* initialement établies par leur MBA hôte. De plus, elles ignorent complètement les arguments nécessaires à l'invocation d'un service particulier. Pour répondre à cette problématique, le MBA met à disposition des programmeurs deux dictionnaires qui doivent nécessairement être consultés respectivement avant chaque demande de service :

- **Dictionnaire des services :** Ce dictionnaire décrit les caractéristiques des différents services tels que : numéro du canal associé, nom du service, arguments requis, ordre des arguments, etc.
- **Dictionnaire des formats :** Ce dictionnaire représente le format du message (demande) attendu par le MBA pour la réalisation du service en question. Le programmeur doit respecter l'ordre des arguments composant la demande conformément à sa description par le dictionnaires des services.

Avant de solliciter un canal de service, l'application doit d'abord passer par la consultation du dictionnaire de services afin de récupérer la description du service désiré, puis ensuite le format du message (demande) imposé par le MBA depuis le dictionnaire des formats. Une fois la demande formulée, elle est déposée sur le canal correspondant au service concerné. Toute demande mal formulée ou incomplète est systématiquement rejetée par le MBA.

Ce système de référencement par dictionnaire facilite la gestion et la maintenance des différents services proposés par le BA. Chaque fois qu'un nouveau service est implémenté, l'administrateur du BA se charge de le référencer dans le dictionnaire des services ainsi que dans celui des formats afin de le rendre accessible aux tâches/applications.

4.7 Composition du TBA en termes de ressources IPC

Le TBA est un composant logiciel entièrement constitué de ressources IPC. Comme nous l'avons vu dans le chapitre 2 de la partie I, ce standard offre tout un ensemble de mécanismes de partage de données et de synchronisation entre processus (mémoires partagées, files de messages, sémaphores, ...). Ces mécanismes diffèrent selon leurs fonctionnalités, leurs niveaux de performances, et leurs limites systèmes autorisées. La composition du TBA doit être donc élaborée en fonction de tous ces aspects afin d'optimiser le rendement rendement. Ainsi, nous considérons les trois caractéristiques suivantes avec les interrogations qu'elles suscitent :

- **Fonctionnelle :** le mécanisme choisi est-il en mesure de répondre à la fonctionnalité souhaitée ?

- **Performance :** quel est le niveau de performance du mécanisme choisi par rapport à tous ceux qui sont disposés à remplir la même fonctionnalité ?
- **Limite système :** les limites systèmes autorisées pour ce mécanisme sont-elles contraignantes pour la suite de nos implémentations ?

Par ailleurs, Il serait regrettable de repartir sur une nouvelle base de conception de l'architecture des TBAs, sans exploitation des fonctionnalités offertes par notre Mécanisme d'Allocation Dynamique de Mémoire Partagée (ADMP) que nous avons mis en œvre dans le chapitre 3, notamment pour réduire les taux de fragmentation interne/externe, c'est pour cette raison que l'architecture du TBA a été élaborée également, en parfaite correspondance avec les caractéristiques de notre ADMP [BBA13c].

4.7.1 Limites systèmes autorisées

Le Tableau 4.2 présente un récapitulatif des divers limites systèmes autorisées et ce, en fonction du SE considéré. Bien entendu ces limites ne sont pas figées et peuvent être ajustées depuis des fichiers de configuration systèmes, souvent sans recompilation du noyau (sur simple redémarrage). Cependant, cette manipulation présente le risque de faire chuter le niveau de performance du système, et les difficultés du paramétrage constituent un handicap pour les usagers qui n'ont pas forcement de compétences systèmes avancées.

Ressource IPC / SE	Mémoires partagées	File de messages	Sémaphores
Fedora 14	4096	1649	32000
Mac-OS	4096	978	32000
Debian	1024	484	32000
Windows	Supérieur à 65636	Supérieur à 65636	Supérieur à 65636

Tableau 4.2 – Limites systèmes relatives aux ressources IPC.

4.7.2 Performance

Avec les résultats de l'étude présentée dans le chapitre 2, nous faisons une classification des différents mécanismes du standard IPC en fonction de leurs niveaux de performance, ce qui nous facilite le choix du mécanisme d'échange le plus rentable. Cette classification est présentée par la Figure 4.6

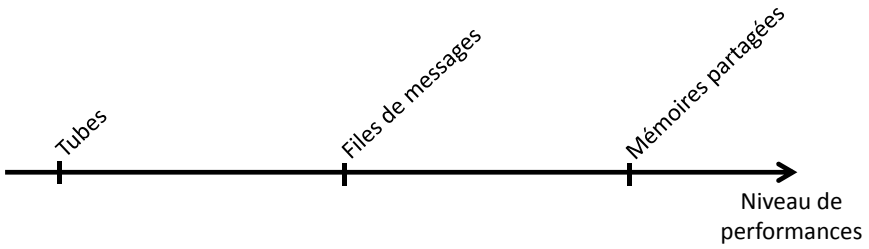

Figure 4.6 – Classification des mécanismes IPC par ordre croissant de performance.

4.7.3 Fonctionnalités

D'un point de vue purement fonctionnel, les tubes (pipes), les files de messages et les mémoires partagées répondent toutes les trois au même besoin de partage de données. Cependant, la classification présentée par la Figure 4.6 montre que le mécanisme de mémoires partagées est le plus performant, suivi juste après par celui des files de messages, la dernière place revenant à celui des tubes, un résultat d'autant plus confirmé par notre analyse alors que nous savons qu'il a pour support d'accès le disque. Il est donc préférable de favoriser l'utilisation des mémoires partagées dès que l'occasion se présente.

4.7.4 Composition

D'après le Tableau 4.2, nous constatons que les systèmes linux imposent des limites plus contraignantes que ceux de Windows, et les valeurs les plus basses ont été identifiées sur Debian avec seulement 484 files de messages et 1024 segments de mémoire partagée, par conséquent, la composition du TBA doit être établie sur cette dernière base. Les limites en termes de sémaphores (32000) ne posent quant à elles aucun problème puisque ceux-ci sont réutilisables.

D'un point de vue purement performance, la Figure 4.6 montre qu'il est toujours de favoriser l'utilisation des mémoires partagées, cependant, ces ressources sont chères et doivent donc être manipulées avec grande délicatesse. En effet, d'après le tableau 4.2, nous remarquons que le système Debian ne dispose que de 1024 segments de mémoire partagée, il serait trop coûteux d'assigner à chaque canal du TBA un bloc mémoire complet, d'autant plus que :

1. L'espace mémoire d'un bloc n'est généralement pas exploité dans sa totalité par un canal ;
2. Les données manipulées sont éphémères ;
3. Le mécanisme d'allocation dynamique de mémoire que nous avons mis en œuvre supporte le maintien de plusieurs éléments dans un seul bloc mémoire.

Le mécanisme d'allocation dynamique de mémoire que nous avons proposé dans le chapitre 3 repose sur l'utilisation de deux blocs mémoires, le premier étant destiné à la TDI, le second à la TDD. Parmi les solutions envisageables nous citons celle qui consiste donc à assigner pour chaque TBA deux couples de TDI/TDD, le premier étant réservé aux canaux de dépôt de demandes, le second destiné aux canaux de retrait de réponses. L'avantage principal de cette solution se joue au niveau de la mémoire partagée où son réalisés tous les échanges. Néanmoins, elle n'est pas sans inconvénient bien significatif, étant donné qu'elle réduit considérablement le nombre total de TBAs susceptibles d'être déployés sur le SE. En effet, sur une base de 1024 segments de mémoire (Debian) et pour une composition de 4 segments par TBA, nous atteignons le plafond de 256 TBAs (1024/4) sur une machine Debian.

Par ailleurs, nous remarquons que le rapport entre le nombre des files de messages et celui des mémoires partagées est bien supérieur à 1/2, ce qui nous laisse assigner à chaque file de messages deux segments de mémoires partagées pour le couple TDI/TDD. la nouvelle composition du TBA est alors la suivante :

- **Une file de messages :** Appelée file de réception générale simulant l'ensemble des canaux de demandes sur une même queue de messages sur laquelle les tâches demanderesses déposent leurs requêtes ;
- **Deux segments de mémoire partagée :** Servant à maintenir les canaux de réponse, le premier est réservé pour la TDI, le second pour la TDD. D'un point de vue plus pratique, nous associons pour chaque canal de réponse une ou plusieurs entrées (éléments) au niveau de la TDI, qui donneront à leurs tours accès aux différentes pages correspondantes au sein de la TDD. Ce mode de gestion est à la charge du MBA.

• **Un ensemble de sémaphores :** Utilisés pour la synchronisation des tâches ;
La composition du TBA en termes de ressources IPC est illustrée par la Figure 4.7 :

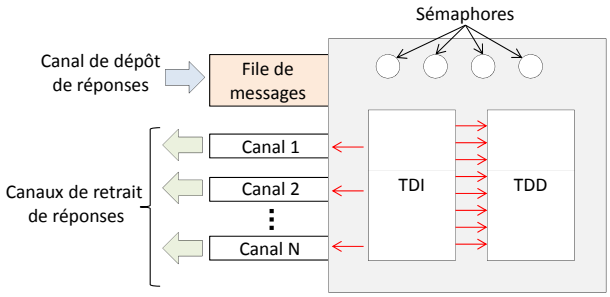

Figure 4.7 – Composition du TBA en termes de ressources IPC.

Avec une telle organisation, sur une base de 1024 segments de mémoire, 484 files de messages (Debian), pour une composition de 2 segments mémoires et une file de messages par TBA, nous pouvons atteindre un plafond de : 484 TBA sur une machine Debian, un de 978 TBAs sur une machine Mac-OS et un autre de 1649 TBAs sur un machine Fedora 14, dépassant ainsi le nombre de TBA plafonné par la solution précédente à 256 TBAs. Nous allons nous attarder sur cette nouvelle solution pour élucider les deux points suivants :

1. Étant donné que les demandes sont déposées sur la même file, elles passent toutes nécessairement par la même entrée, les tâches doivent donc être en mesure d'indiquer le numéro du canal de demandes désiré afin que le MBA hôte puisse réaliser le service adéquat. La solution que nous proposons consiste à rajouter au contenu des demandes (messages) un argument contenant le numéro du canal service désiré. Cette opération est réalisée de manière soit explicite, soit implicite en fonction du type du canal demandé : service ou usager : lorsque la demande concerne un canal de service, les tâches passent obligatoirement par la fonction de l'API correspondant au service désiré qui se charge d'ajouter de manière implicite le numéro du canal relatif au service demandé au contenu du message. Dans le cas où la demande concerne un canal usager, c'est le programmeur lui-même qui doit indiquer explicitement à l'API le numéro du canal qu'il sollicite.
2. Puisque toutes les demandes passent par la même file de messages, cette dernière doit être évacuée au plus vite par le MBA afin d'éviter toute forme de saturation. Pour assurer une telle contrainte, nous avons implémenté un mécanisme de traitement multithread dans lequel nous associons à chaque demande un nouveau thread destiné à son traitement de manière à accélérer la ventilation du contenu de la file de demandes.

Ainsi conçue, la composition du TBA propose donc, non seulement un usage judicieux des ressources IPC, mais aussi une exploitation du mécanisme d'allocation dynamique de mémoire précédemment élaboré.

4.8 Utilisation du TBA : la Parallel Instruction (P-Instruction)

Le principe d'utilisation du TBA repose sur la notion de la *Parallèle Instruction (P-instruction)* qui se déroule en deux parties : la première concerne le dépôt de demandes, et la seconde le retrait de réponses. Plus précisément, une P-instruction est un couple de deux sous fonctions, à savoir Fct-Demande et Fct-Réponse.

4.8.1 Fct-Demande

Cette fonction est non-bloquante, elle porte toujours le préfixe *Ba* (exemple : BaLogin, BaConnect, ...), la tâche demanderesse s'en sert pour déposer sa demande sur la file de réception générale de son TBA, en précisant, bien sûr, le numéro du canal désiré. Dans la foulée, elle met automatiquement en place un sémaphore bloquant, qui sera exploité ultérieurement par la fonction Fct-Réponse pour se synchroniser par rapport à la présence du résultat de traitement, marquant ainsi la fin d'exécution du service invoqué. Pour des raisons de traçabilité, chaque dépôt de demande est accompagné par un compte rendu indiquant le bon/mauvais déroulement de l'opération.

Le MBA surveille en permanence la file de réception générale et lorsqu'il détecte la présence d'une nouvelle demande, il met systématiquement en place un nouveau thread pour la traiter, ainsi le contenu de la file est instantanément ventilé. Le principe de fonctionnement de la Fct-Demande est présenté dans la Figure 4.8

Figure 4.8 – Fonctionnement de la Fct-Demande.

4.8.2 Fct-Réponse

Cette deuxième sous fonction est bloquante, elle porte toujours le même nom que celui de la Fct-Demande complété par le suffixe *Ret* (exemple : BaLoginRet, BaConnectRet, ...), la tâche demanderesse l'utilise pour se synchroniser par rapport à la fin de l'exécution du service, avant récupération du résultat de traitement. Si les demandes sont déposées au niveau de la file de réception générale, les réponses quant à elles sont stockées dans les canaux de retour en mémoires partagées (couple TDI/TDD).

La synchronisation est réalisée via le sémaphore installé lors de l'étape précédente (Fct-Demande), qui est systématiquement libéré par le MBA hôte, après réception et mise en place du résultat de traitement dans le canaux de réponses, ce qui déclenche le déblocage de la fonction Fct-Réponse. La Figure 4.9 montre ce principe de fonctionnement.

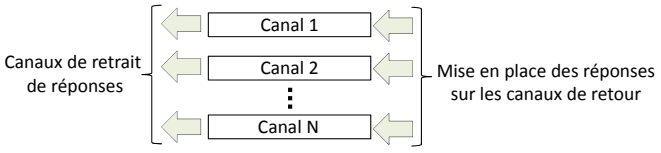

Figure 4.9 – Fonctionnement de la Fct-Réponse.

4.9 Quelques fonctions de l'API

Le Tableau 4.3 présente quelques fonctions de l'API : Celles qui ne portent pas de suffixe *Ret* sont de type Fct-Demande alors que celles qui le portent sont de type Fct-Réponse.

Tableau 4.3 – Quelques fonctions de l'API du BA.

Nom de la fonction	Description
BaLogin	Logeage dans la constellation du BA
BaLoginRet	Récupération du résultat du logeage
BaConnect	Connexion d'une ULT au TBA
BaConnectRet	Récupération du résultat de la connexion
BaListRef	Demande de la liste des référentiels de la machine hôte
BaListRefRet	Récupération de la liste des référentiels
BaVers	Demande des versions relatives aux composants logiciels : MBA, TBA et API
BaVersRet	Récupération des versions
BaDonTe	Demande de mise en place d'un nouveau terminal esclave
BaDonTeRet	Récupération de l'identifiant du nouveau Te
BaAdr	Demande de la liste des adresses relatives aux MBA de la constellation
BaAdrRet	Récupération des adresses
BaMenu	Demande de la liste des applications disponibles sur la constellation
BaMenuRet	Récupération de la liste des applications disponibles
BaCreateApp	Demande de lancement d'une application distante
BaCreateAppRet	Récupération du compte rendu du lancement
BaFicRcv	Demande de téléchargement d'un fichier
BaFicRcvRet	Récupération du résultat du téléchargement
BaFicEnv	Demande d'émission d'un fichier
BaFicEnvRet	Récupération du résultat de l'envoi
TTCmd	Demande de transmission de commande de terminal à terminal
TTCmdRet	Compte rendu de l'envoi de la commande
TTRecep	Réception d'une commande de terminal à terminal
TTRecepRet	Récupération du résultat de la réception de la commande

4.9.1 Discussion

Dans cette section, nous analysons le mode de fonctionnement de la P-Instruction dans le but d'illustrer l'intérêt majeur qu'elle suscite pour la résolution des problèmes de synchronisation entre tâches. Si nous faisons abstraction de la composition du TBA en termes de ressources IPC, et nous considérons uniquement le scénario d'échange entre une tâche demanderesse et le MBA, le fonctionnement général de la P-Instruction se résume par la Figure 4.10

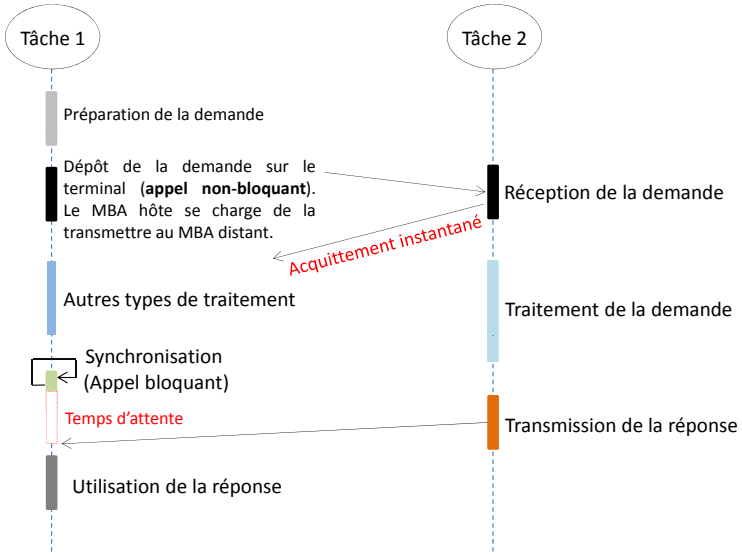

Figure 4.10 – Fonctionnement générale de la P-Instruction.

Nous avons volontairement divisé la P-Instruction en deux parties de manière à séparer le dépôt de demande du retrait de réponse, ainsi la tâche demanderesse est désormais indépendante de la durée réelle d'exécution du service demandé. Nous remarquons également, que la fonction Fct-Demande est non-bloquante, ce qui permet à la tâche appelante pour continuer son exécution tant qu'elle n'a pas besoin du résultat de traitement, notamment pour anticiper le traitement sur les parties du code qui ne nécessitent pas d'interaction avec le BA. La fonction Fct-Réponse, quant à elle est bloquante et joue en faveur de la synchronisation de la tâche appelante avec le résultat final (voir Figure 4.11).

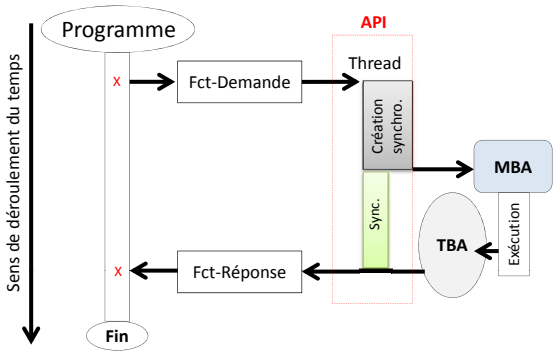

Figure 4.11 – Les deux sous-fonctions de la P-Instruction.

Avec un tel procédé, l'intervalle de temps séparant le dépôt de demande du retrait de réponse devient un paramètre modifiable, géré par le programmeur en fonction de son besoin.

La programmation anticipée est l'un des critères avantageux et non limitatifs de cette solution. En effet, le programmeur est libre de soumettre autant de Fct-Demandes qu'il souhaite, de préférence *au plus tôt*, c'est à dire avant même que le résultat du traitement ne se fasse nécessaire, sans qu'il ne soit contraint de respecter le même ordre de Fct-Réponses en retour.

Ce mécanisme à été exploité dans un système de gestion de boites de dialogues. C'est la tâche demanderesse qui prend la part active dans cette opération, elle demande à une autre tâche distante d'ouverture d'un dialogue en mode caché en attendant son utilisation. Au moment de sa réelle utilisation, le dialogue bascule en mode visible, donnant ainsi à l'usager l'impression d'instantanéité d'ouverture. La synchronisation par rapport à la validation du formulaire contenu dans le dialogue est réalisée par la fonction Fct-Réponse.

Considérons maintenant deux tâches : T1 et T2, et supposons que T1 souhaite faire une P-Instruction dont le traitement est réalisé par T2, deux sénarios d'exécution sont alors possibles :

1. T1 dépose une Fct-Demande, sachant que la durée d'exécution de T2 est supérieure à celle anticipée par T1, et dans ce cas, T1 est obligée d'attendre la fin de T2 pour pouvoir continuer son exécution : la phase de synchronisation (Fct-Réponse) est alors plus ou moins importante ;
2. T1 dépose une Fct-Demande, et la durée d'exécution de T2 est inférieure à celle anticipé par T1, et dans ce cas, l'opération de retour de T1 est instantanée puisque la réponse est déjà en place, la durée de synchronisation étant nulle.

Nous comprenons donc pourquoi il est préférable de déposer le plus tôt les Fct-Demandes. L'idéal serait de soumettre toutes les Fct-Demande incessamment, ainsi le temps d'attente relatif aux Fct-Réponses est réduit au maximum et les unités de traitement sont mieux exploitées. En anticipant les appels, nous augmentons les opportunités d'utilisation des ressources systèmes aux périodes creuses et par conséquent, nous diminuons l'engorgement dans les pics d'activité de la machine.

4.10 Types de TBAs

D'après la description présentée dans la section 4.4, nous constatons que le TBA est partagé entre l'ensemble des tâches qui composent l'application, chacune d'elles s'approprie un certain nombre de canaux en fonction de ses besoins. Peine perdu, nous nous sommes rendu compte que ce modèle de partage présente un inconvénient majeur dans la manipulation des canaux de services, en effet, comme ils sont utilisés en exclusion mutuelle, nous ne sommes pas en mesure de satisfaire la demande en masse d'un même canal service par plusieurs threads qui doivent obligatoirement se mettre en attente pour être servis l'un après l'autre : la capacité d'acceuil d'un canal service étant par unité. En revanche, les canaux usagers ne posent pas problème puisque les tâches choisissent des plages de canaux différentes.
Nous ne devons donc pas nous limiter au modèle de partage avec un seul TBA par application, mais plutôt donner la possibilité aux différentes tâches d'avoir leurs propres terminaux, d'où l'intérêt du concept de *terminal esclave* que nous proposons. Nous distinguons donc deux types de terminaux : maître et esclave.

4.10.1 Terminal Maître (TM)

Le terminal maître (TM) fait référence au TBA associé à l'application mère, c'est-à-dire celle qui a sollicité le BA pour la première fois et dont l'ensemble des tâches qui la composent se

partage le même TM.
La demande de mise en place d'un TM nécessite une authentification de la part de l'application mère auprès du BA afin qu'il vérifie son identité. Le service d'authentification est toujours accessible via le canal N°1. Exemple :

- **BaLogin (partie Fct-Demande) :**
  int iDem=pTerminal->BaLogin (cLogin, cPasswd) ;
  Le retour est immédiat avec iDem=0 si l'opération est en cours, 1 si le MBA est absent, et 2 si une autre demande est déjà en cours. La valeur -1 indique un mésusage.
- **BaLoginRet (partie Fct-Réponse) :**
  Lorsque le retour est requis par le programme appelant, il est obtenu par la fonction BaLoginRet.
  int iRep=Terminal->BaLoginRet () ;
  Si l'authentification est réussie iRep=0 ; dans le cas contraire, un code d'erreur est retourné.

Dans le cas où l'application est autorisée à accéder au BA, le MBA lui met en place un nouveau TM, et lui transmet ses identifiants sous forme d'un triplet composé par : le descripteur de la file de réception générale, le descripteur de la mémoire réservée à la TDI, et le descripteur de la mémoire partagée réservée à la TDD. Lorsque l'application n'a plus besoin de son TBA maître, elle dépose une demande de déconnexion (Logout) sur le canal N°5 de manière à ce que le MBA puisse récupérer proprement les ressources allouées au TM.

4.10.2 Terminal esclave (TE)

Nous parlons de terminal esclave (TE) lorsqu'un TBA est entièrement réservé à une tâche. D'un point de vue structurel, il n'y a aucune différence entre la composition d'un TM et celle d'un TE, de même pour leurs fonctionnalités, qui sont pratiquement les mêmes, à l'exception du canal Login qui représentera celui du Connect. A la différence du service Login, celui du Connect ne nécessite pas une phase d'authentification puisque les tâches héritent des mêmes droits que ceux de l'application mère.
Initialement, les tâches ne connaissent que le TM de l'application mère. Pour obtenir un nouveau TE, la tâche dépose une demande sur le canal de service numéro 12 de son TM, et dont le traitement par le MBA hôte aboutit à la création d'un nouveau TE avec des références stockées dans le canal de retour correspondant à la demande. En cas d'échec de l'opération (Ex : ressources systèmes insuffisantes), un code d'erreur est retourné. Ces références sont utilisées par la tâche afin de se connecter au nouveau terminal esclave. Exemple de code :

- **BaDonTe (partie Fct-Demande) :**
  int iDem=pTerminal->BaDonTe() ;
  Le retour est immédiat avec iDem=0 si l'opération est en cours, 1 si le MBA est absent, et 2 si une autre demande est déjà en cours. La valeur -1 indique un mésusage.
- **BaDonTeRet (partie Fct-Réponse) :**
  int iRep=Terminal->BaLoginRet(&TerminalEsclave) ;
  Si l'opération est réussie iRep=0 et la variable TerminalEsclave passée en argument est initialisée avec le triplet de descripteurs (file de réception générale, TDI, TDD) fourni par le MBA. En cas d'échec de l'opération, un code d'erreur est retourné.

Avec un tel dispositif, plusieurs tâches d'une même application sont en mesure d'invoquer les mêmes services simultanément en utilisant les canaux de leurs TM respectifs, opération irréalisable avec uniquement des TM.

4.11 Canal de réception permanente

Nous réservons cette section à la description du principe de fonctionnement du canal de réception permanente (canal de service numéro 7) car son mode d'utilisation est légèrement différent de celui des autres canaux de services, qui d'après la description présentée par la section 4.8 passe par deux étapes :

- **La demande** : dans laquelle la tâche dépose une demande sur un canal service/usagers ;
- **La réponse** : durant laquelle la tâche se synchronise au canal de retour correspondant à celui de la demande, le temps que la réponse soit disponible.

Le mode de fonctionnement du canal de réception permanente ne rentre pas dans ce cas de figure, en effet, il a été pensé de manière à permettre aux tâches de recevoir à la volée des messages venant d'autres tâches locales/distantes, et sans dépôt préalable de demandes. Avec cet arrangement une tâche est tenue en permanence informée des événements en cours par ses congénères, et sans aucune initiative particulière de sa part.

Pour illustrer l'intérêt du canal de réception permanente, nous donnons quelques exemples de messages recensés sur notre plateforme :

- **Notification** : informer la tâche de la présence d'un événement imprévu ;
- **Commande** : piloter une tâche distante en lui donnant des ordres de traitement ;
- **Consigne** : personnaliser certains paramètres d'affichage d'une tâche : couleurs, titres, etc ;

Étant donné qu'il est probable qu'une tâche reçoit plusieurs messages simultanément sur le canal de réception permanente de son TBA, en provenance d'autres tâches locales ou distantes, il est nécessaire de mettre en œuvre un mécanisme de gestion d'accès concurrent assurant la cohérence des données manipulées. Pour résoudre ce problème, nous incluons au TBA un couple de sémaphores (Sem1, Sem2) fonctionnant en alternance pour garantir une exclusion mutuelle entre les opérations de lecture et celles de l'écriture. Sem1 et Sem2 sont utilisés comme suit :

1. Lorsque le MBA reçoit un message dédié au canal de réception permanente d'un TBA, il commence par vérifier si ce canal n'est pas déjà en cours d'utilisation, pour cela, il se synchronise avec Sem1 ;
2. La tâche se synchronise avec le sémaphore bloquant Sem2 pour savoir si de nouvelles données sont disponibles ou non ;
3. Après avoir mis en place le données reçues dans le canal de réception permanente, le MBA bloque Sem1, puis libère Sem2 afin d'informer la tâche de la présence d'un nouveau message ;
4. Sem2 est désormais passant, la tâche peut donc récupérer le contenu du canal de réception permanente. Pour indiquer au MBA que l'opération de lecture est terminée, la tâche libère Sem1, ainsi le cycle est bouclé pour être repris à l'état initial 1 avec la réception de nouveaux messages. Ce système d'échange est illustré par la Figure 4.12

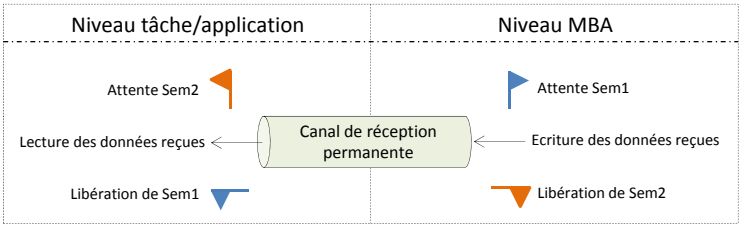

Figure 4.12 – Principe de fonctionnement du canal général.

4.12 Structure hiérarchique des TE (arbre de terminaux)

Le processus de création des TE est récursif, il est possible de demander un TE à partir de n'importe quel terminal : maître ou esclave, formant ainsi une structure arborescente appelée arbre de terminaux. Si les applications mères avaient été exclusivement composées de threads, nous aurions systématiquement obtenu un arbre de niveau de profondeur 1, mais ce n'est pas toujours le cas, celle-ci pouvant donner naissance à une application fille (cas du fork) elle aussi composée d'un ensemble de threads qui, à leurs tours, déclenchent de nouveaux fork, et ainsi de suite ... Il est donc possible d'avoir un arbre de terminaux de niveau de profondeur supérieur à 1. La Figure 4.13 montre un exemple de cette hiérarchie arborescente.

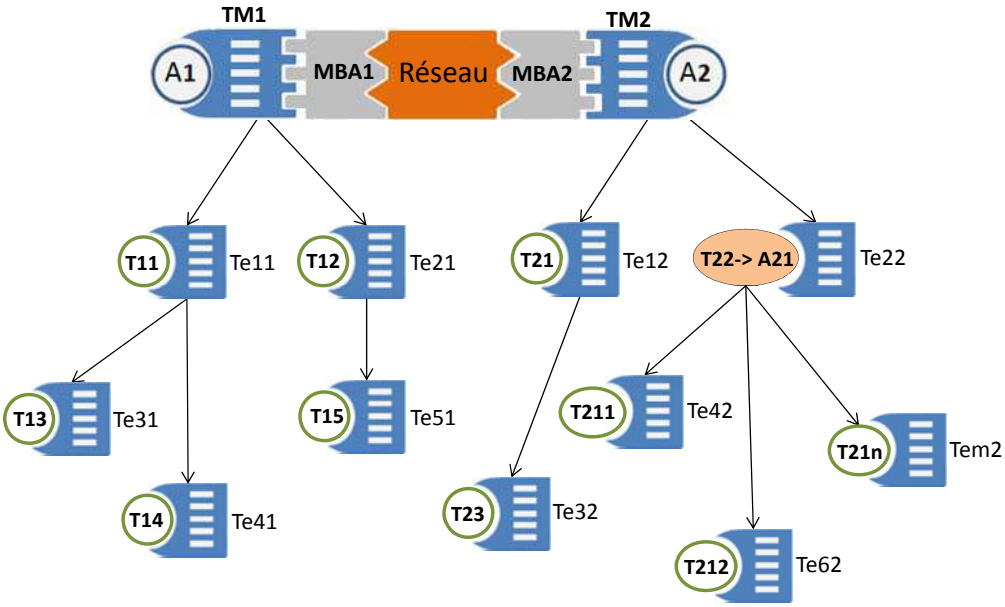

Figure 4.13 – Arbre des terminaux.

- *TM1* : Terminal maître de l'application A1.
- *TM2* : Terminal maître de l'application A2.
- *Tij* : Thread j de l'application i.
- *Tijk* : Thread K de l'application fille j de l'application i.
- *Teij* : Terminal esclave i de l'application j.
- $\rightarrow$: Déclenchement d'un nouveau terminal esclave.

Dans cet exemple, le thread 2 de l'application 2 a fait un fork donnant ainsi naissance à l'application fille A21, composée d'un ensemble de tâches parallèles (T211, T212, ..., T21n). Chacune de ces tâches est à même de demander à travers le terminal de son application mère (TE22) la mise en place d'un nouveau TE qui lui est exclusivement dédié, nous nous retrouvons donc avec un arbre de niveau 2, et si, à leurs tours elles font hes font à leurs tours un fork nous obtenons un arbre de niveau 3, ainsi de suite... Théoriquement, aucune limite particulière n'est imposée au niveau de profondeur maximale d'un arbre de terminaux, cependant, sur le plan pratique, cette limite dépend du nombre maximum de terminaux autorisé sur la machine.

4.13 Gestion d'identifiants et adressage

La gestion des identifiants relatifs aux TBA est à la charge du MBA qui associe à chaque TBA nouvellement créé, qu'il soit maître ou esclave, un identifiant local unique sous forme de numéro. Cet identifiant qui ne représente aucunement la référence globale du terminal est juste un descripteur interne, valable uniquement sur le SE hôte. Lorsqu'un terminal est détruit suite à une demande de déconnexion (canal de service numéro 5) son identifiant est automatiquement récupéré par le MBA pour un usage ultérieur.

Pour avoir l'adresse globale d'un TBA, il faut d'abord localiser le SE sur lequel il est hébergé, en d'autres termes, la référence du MBA de la machine cible, en suite connaître l'identifiant du terminal désiré, ainsi, les adresses globales se présentent sous forme d'un doublet *partie1@partie2* de telle façon que partie1 représente l'identifiant du TBA et partie2 la référence du MBA. Voici un exemple concret d'une adresse globale : 1@MBA010094001072Ridha-Cnam

- 1 : représente l'identifiant du terminal.
- MBA010094001072Ridha-Cnam : désigne la référence du MBA.

Si un message est déposé à destination d'une adresse ne comportant pas de référence MBA, c'est une indication qu'il est destiné à une tâche locale. Par ailleurs, le message envoyés à une adresse ne contenant pas de référence TBA est systématiquement rejetés par le MBA hôte. Enfin, chaque fois qu'un message est déposé sur la file de réception générale d'un TBA, l'identifiant de ce dernier est automatiquement rajouté à l'en-têtes de ce message tout juste avant sa transmission sur le réseau, de manière à ce que l'entité réceptrice puisse connaître l'identifiant du TBA qui est à l'origine du message et répondre ainsi à la bonne destination.

Ce mode d'adressage encapsule toutes les couches systèmes (TCP/IP, socket, IPC, ...) nécessaires à la transmission des données, et facilite le mécanisme de routage des données pour le programmeur.

4.14 Mise en correspondance des TBA

Chaque application, et en fonction de ses besoins, est en droit de demander un certain nombre de terminaux esclaves qui ont initialement une visibilité locale, et ne sont connus que de leur application mère. Cependant, pour faire collaborer plusieurs tâches simultanément, il faut qu'elles connaissent leurs TBA respectifs, d'où la nécessité de présence d'un mécanisme de mise en correspondance entre TBAs et dont la mise en œuvre passe par un protocole d'échange ouvrant la voie à un TBA donné pour se faire reconnaitre auprès des autres TBAs de la constellation.

Nous proposons dans cette section un mécanisme de découverte et d'échange d'adresses entre terminaux. Pour arriver à cette fin, il est d'abord important d'introduire les concepts suivants :

1. **Initiateur** : ce champs s'applique uniquement sur les applications distantes, et il représente l'identifiant du TBA de l'application mère qui est à l'origine de leurs lancements. Dans notre terminologie, ce TBA est désigné par le terme *Init*.
2. **Correspondant** : ce champs représente l'identifiant du TBA distant avec lequel le TBA courant est directement lié, et il prend l'appellation : *Cor*
3. **Identifiant** : ce champs représente l'identifiant du TBA courant.

Pour bien comprendre le contexte d'utilisation de ces sigles, nous allons prendre l'exemple d'un système simple, formé de deux applications A1 et A2, composées chacune d'un ensemble de tâches parallèles. Nous supposons que A1 est à l'origine du lancement de A2. Ce système est présenté par la Figure 4.14

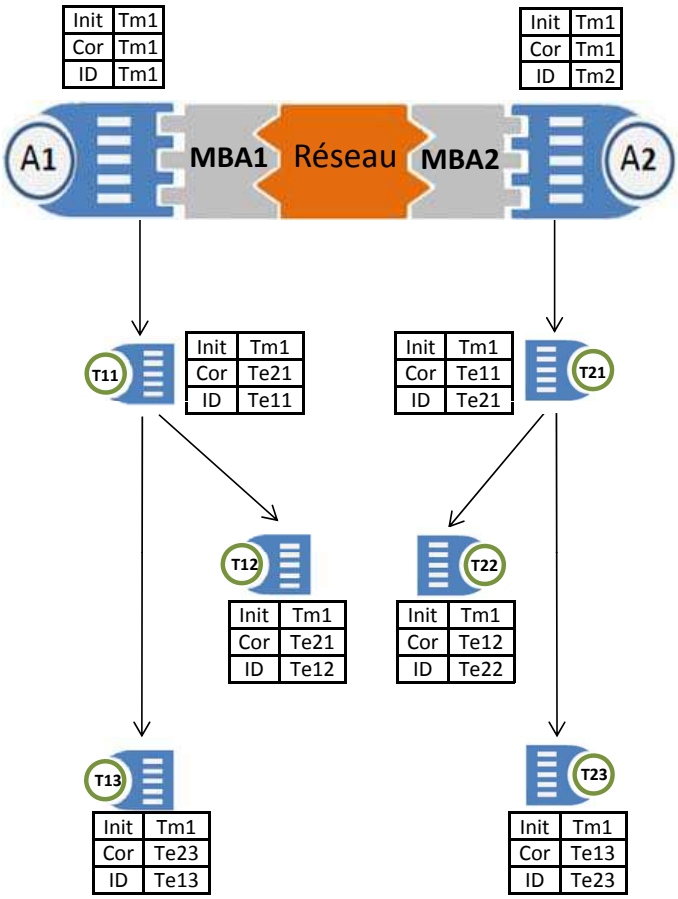

Figure 4.14 – Mise en correspondance des TBAs.

La séquence suivante décrit les différentes étapes à travers lesquelles transite le système avant d'arriver à la configuration présentée par la Figure 4.14

1. A1 dépose sur le canal de service numéro 6 de son TM (Tm1) une demande à destination du MBA distant (MBA2) lui demandant de lancer l'application A2.
2. Avant de lancer l'application A2, le MBA2 lui crée un nouveau TM (Tm2) et lui prépare l'environnement d'exécution.
3. MBA2 lance l'application A2 à travers un fork (cas Linux) ou un createprocess (cas Windows).
4. A2 récupère depuis son environnement d'exécution l'identifiant de TM2 créé précédemment, et se connecte à lui via le canal de demande numéro 1, par la suite, elle récupère depuis le canal de retour numéro 1 les informations suivantes :
   - L'identifiant de son terminal : TM2 ;
   - L'identifiant du terminal initiateur : TM1 ;
   - L'identifiant du terminal correspondant : TM1 ;

   L'application A2 est maintenant en mesure de communiquer avec A1 en prenant TM1 pour identifiant ;
5. En utilisation l'adresse de MBA1, MBA2 transmet à A1 le compte rendu de lancement de A2 (Echec/Succès) qui sera stocké sur le canal de retour numéro 6 de TM1.

En cas de succès de l'opération, A1 récupère l'adresse complète du terminal associé à l'application distante A2 (Tm2). Tm1 aura donc pour correspondant Tm2. Dans le cas contraire, un code d'erreur sera indiqué.

6. La tâche T21 de A2 dépose sur TM2 une demande de création d'un nouveau TE.
7. MBA2 crée un nouveau terminal esclave TE21 dont il stocke l'identifiant sur le canal de retour associé à ce service (canal de service numéro 12).
8. T21 se connecte au nouveau terminal esclave TE21 via son canal de service numéro 1.
9. Te21 hérite des mêmes droits que ceux de TM2 et garde les mêmes propriétés que lui, elle aura donc comme initiateur TM1. La tâche T21 utilise l'adresse de TM1 pour communiquer avec l'application A1.
10. T21 transmet à l'application A1 (adresse héritée de Tm2) une demande de lancement d'une nouvelle tâche parallèle T11, qui doit disposer à son tour de son propre terminal esclave.
11. Avant de lancer la tâche T11, A1 récupère l'adresse du terminal distant (Te21) qui est à l'origine de cette demande. Le correspondant de Te11 sera alors Te21.
12. A1 dépose par la suite sur le canal 12 de TM1, une demande de création d'un nouveau terminal esclave (TE11).
13. MBA1 crée le nouveau terminal esclave Te11 dont il stocke l'identifiant dans le canal de retour numéro 12, et initialise ses propriétés :
    - Adresse du terminal correspondant de Te11 = Te21.
    - Adresse du terminal initiateur de Te11 = Tm1 (par héritage).
14. T11 dépose un message d'acquittement à destination du terminal correspondant (TE21) via son terminal TE11.
15. T21 reçoit l'acquittement de T11, elle récupère alors l'adresse du terminal qui lui correspondant c'est-à-dire TE11.

A ce niveau, les deux tâches parallèles T11 et T21 disposent de deux terminaux esclaves indépendants TE21 et TE11 par lesquels elles communiquent en parallèle indépendamment des autres tâches de leurs applications respectives. Ce même processus s'applique aussi sur le reste des tâches : T22, T23, ... donnant ainsi la configuration présentée par la Figure 4.14

4.15 Destruction des Terminaux

Lorsqu'une tâche/application n'a plus besoin de son TBA, elle doit le signaler explicitement à son MBA en déposant une demande de déconnexion sur le canal de service numéro 5 : canal LogOut. Si le TBA en question ne possède aucun terminal esclave, il reste le seul concerné par le processus de destruction, en revanche, s'il possède plusieurs TE, le processus de destruction se fait en cascade de manière à supprimer tous les TE fils qui n'ont plus de raisons d'exister suite à l'arrêt de leur application mère. Cette façon de procéder se justifie non seulement pour des raisons fonctionnelles, mais aussi techniques liés aux fonctionnement des threads :

- D'un point de vue fonctionnel, une éventuelle décision de l'application mère de mettre terme à son exécution dénote qu'elle n'a plus besoin de ses threads fils.
- D'un point de vue purement technique, un thread ne peut jamais évoluer seul, sans l'apport logistique de son application mère. Autrement dit, aucun thread orphelin ne subsiste sur un SE.

Après notre analyse et pour toutes les raisons que nous avons évoquées, nous avons conclue que les TEs d'une application mère doivent être extirpés avec la fin de son exécution. Cette caractéristique constitue l'un des atouts majeur du ramasse miettes du MBA qui assure une récupération propre des ressources IPC allouées aux TBAs. La Figure 4.15 montre ce principe de destruction en cascade.

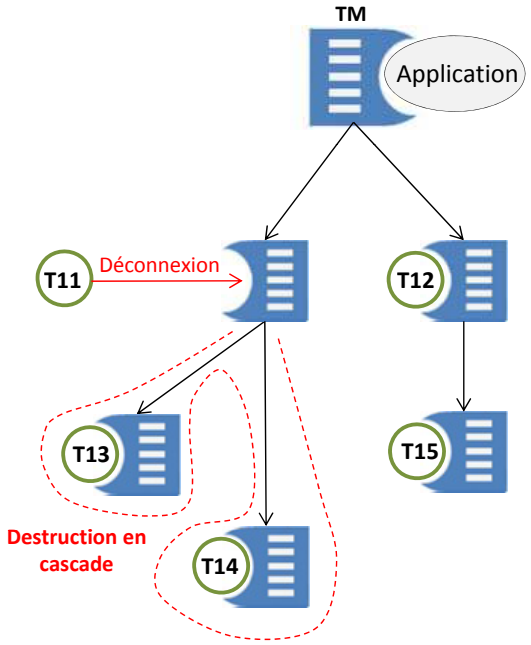

Figure 4.15 – Destruction en cascade des terminaux.

5

Le MBA et la gestion des communications

Sommaire

5.1 Introduction

Nous avons présenté dans la Section 2 de la Partie I toute une gamme de bibliothèques de communication, classées selon leurs niveaux d'interaction (intermédiaire, applicatif, ...) ainsi que leur paradigme de programmation (échange de messages, appel de procédure à distance, etc). Nous nous intéressons maintenant à la couche de communication du bus applicatif.

Comme nous l'avons déjà mentionné précédemment, cette couche est assurée par un composant logiciel nommé multiplexeur du bus applicatif (MBA) dont le rôle est d'assurer l'acheminement de tous les messages circulant entre tous les acteurs (application, services, threads, ...) de toutes les machines de la constellation vers leur destination. L'inadaptation et l'insuffisance des solutions existantes (voir section 2.10) vis-à-vis de nos besoins en termes de communication sont à l'origine de notre motivation à élaborer notre propre mécanisme de transfert de données.

Ce chapitre est organisée comme suit. Nous commençons par quelques descriptions des grandes briques de base qui composent notre mécanisme de communication, et nous enchaînons par une partie consacrée à la description détaillée du système de multiplexage/démultiplexage que nous avons mis en œuvre.

5.2 Le multiplexeur du bus applicatif et la gestion des communications

Le multiplexeur du bus applicatif (MBA) est le composant logiciel responsable de la gestion des ressources IPC partagées (mémoires partagées, files de messages, sémaphores, ...) dont le réseau standard TCP/IP à travers un service de multiplexage/démultiplexage de données qui d'une part, offre un meilleur contrôle sur la circulation du flux de communication, d'autre part, optimise l'utilisation de la bande passante. Il s'agit d'un service de multiplexage/démultiplexage de niveau applicatif, ce qui entraîne une certaine indépendance vis-à-vis du protocole de transport utilisé (TCP, UDP), et des contraintes matérielles sous-jacentes (Ethernet, fibre optique, ...), ainsi, nous restons compatibles avec toute sorte de solution impliquant un autre protocole de communication de niveau plus bas (exemple : Madeleine).

En plus du service de multiplexage/démultiplexage de données, le MBA ordonnance les données à transmettre en fonction de leurs niveaux de priorité et non pas selon leur ordre d'arrivée. Pour cela, le MBA assigne à chaque message une valeur numérique comprise entre 0 et 3 indiquant son niveau de préférence, ce qui lui permet d'assurer une meilleure gestion du flux de communication circulant sur le réseau, par exemple, il aura le pouvoir de contrôler les transferts de fichiers volumineux qui risquent de monopoliser la bande passante en les découpant en plusieurs parties pour être avantageusement transmises et de manière discontinue.

Pour assurer la communication entre les différentes SPU communicantes, la solution la plus naïve consiste à établir une nouvelle liaison réseau (TCP,UDP, ...) chaque fois qu'il y a des données à transmettre. Cependant, même si cette connexion a une duré de vie éphémère (juste le temps de transmission des données), nous risquons facilement d'atteindre les limites systèmes autorisées (nombre de ports, nombre de sockets). Il faut donc être en mesure de multiplexer plusieurs communications sur une même liaison réseau. Nous estimons que cette approche est plus bénéfique que celle citée précédemment pour deux raisons :

- Le temps d'accès à la mémoire RAM est négligeable par rapport à celui des périphériques (opération E/S) qui se mesure en millisecondes. Le gain en termes de performances augmente si nous arrivons à transmettre le maximum d'informations dans un nombre réduit d'opérations E/S.
- Le multiplexage/démultiplexage de données sur les mêmes liaisons offre un meilleur gain en termes de ressources système, ce qui permet d'envisager la montée en charge de toute l'infrastructure du BA en faisant littéralement exploser le nombre des tâches communicantes.

Il est à noter que le fonctionnement du MBA est multitâche et comporte deux phases : une phase de multiplexage/démultiplexage assurant l'agglomération/distribution des données et une phase de transmission responsable de l'envoi/réception des données sur le réseau, toutes deux indépendantes l'une de l'autre et peuvent s'exécuter en parallèle.

Pour chaque fonctionnalité considérée (multiplexage et démultiplexage), nous abordons les trois points suivants :

- Les entrés : Représentent le flux des données entrantes.
- Le traitement : Fonction de multiplexage/démultiplexage à appliquer sur les données.
- Les sorties : Représentent le flux des données sortantes.

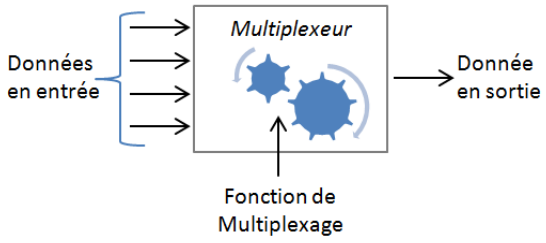

Figure 5.1 – Le multiplexage.

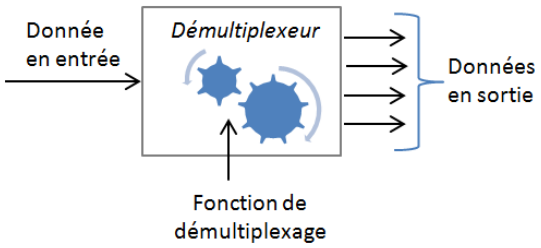

Figure 5.2 – Le démultiplexage.

5.3 Le multiplexeur

5.3.1 Les entrées du multiplexeur

Les tâches/applications sont connectées au BA via leurs terminaux. Lorsqu'une application/-tâche s'apprête à transmettre un message sur le réseau, elle doit le déposer sur la file de messages de son terminal en indiquant l'adresse de sa destination, il est ensuite pris en charge par un thread de surveillance de la file qui crée systématiquement un nouveau thread destiné à son traitement de manière à ce que la file soit vidée le plus rapidement possible. La Figure 5.3 illustre ce principe.

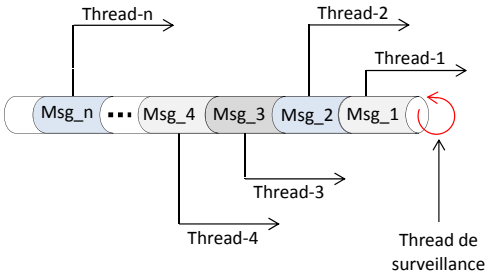

Figure 5.3 – Lancement des threads de traitement par le thread de surveillance.

Le rôle principal des threads de traitement consiste à enrichir le contenu des messages (demande-1, demande-2, ..., demande-n) par l'ajout d'informations complémentaires (headers) avant de les envoyer à l'entrée du multiplexeur, ce qui permet non seulement d'alléger la charge de travail pour le multiplexeur, mais aussi d'exploiter l'exécution parallèle des threads. L'entrée du multiplexeur sera donc composée par l'ensemble de threads de traitement lancés précédemment (Thread-1, Thread-2, ..., Thread-n) comme l'indique la Figure 5.4.

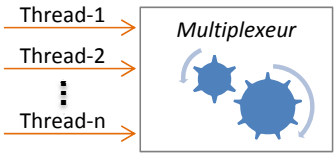

Figure 5.4 – Composition de l'entrée du multiplexeur.

L'enrichissement d'un message véhiculé par un thread de traitement consiste à lui ajouter un certain nombre d'informations complémentaires sous forme d'en-têtes, ainsi, chaque message

sera encapsulée dans une nouvelle structure qui détient toutes les informations requises à son traitement et réordonnancent au niveau du multiplexeur. Les champs les plus importants de la nouvelle structure sont les suivants :

- **La priorité :** Un numéro qui indique le niveau de privilège associé à un message. La version actuelle du MBA supporte quatre niveaux de priorité compris entre 0 (plus prioritaire) et 3 (moins prioritaire) indiquant l'ordre de préférence avec lequel les messages seront transmis sur le réseau. Pour les demandes de même priorité, le MBA applique une politique de gestion en FIFO (First in first out). Exemple :
  - Demande d'acquittement (la plus prioritaire) ⇒ Priorité=0
  - Demande de transfert de fichier (la moins prioritaire) ⇒ Priorité=3
- **Le Pointeur de Début :** Pour avoir une meilleure performance, il faut éviter au maximum les recopies mémoires. On ne doit manipuler que des références (adresses), ainsi, le pointeur est incrémenté au fur et à mesure de l'émission sur la zone mémoire à transmettre.
- **La taille totale du message :** A l'entrée du multiplexeur les données peuvent être de nature différente (texte, Binaire, ...) raison pour laquelle il est indispensable de connaitre la taille totale de la zone mémoire à transmettre.
- **La taille restante à transmettre :** Ce paramètre est utilisé pour les messages dont la taille dépasse celle de l'unité d'envoi du multiplexeur (MTU), et il est généralement utilisé pour suivre l'état d'avancement d'un transfert.

5.3.2 Mécanisme du guichet

La charge de traitement appliquée à l'entrée du multiplexeur n'est pas constante, elle peut varier de quelques dizaines de demandes à des milliers, mais dans tous les cas, le multiplexeur doit être capable de les "absorber" et les traiter de la manière la plus efficace possible. Cette opération passe d'abord par une gestion rigoureuse des demandes à l'entrée du multiplexeur : une contrainte primordiale au maintien du bon fonctionnement du système surtout avec l'augmentation du nombre de demandes déposées simultanément. Pour arriver à cette fin, nous proposons un nouveau dispositif de traitement appelé "guichet".

Un guichet est un dispositif efficace de surveillance et de traitement de demandes. Il est composé d'une entrée sur laquelle les demandes sont déposées et d'une fonction prédéfinie assurant leurs traitements. Ce dispositif diffère de celui d'un service (daemon en Anglais) qui est généralement basé sur la notion de scrutation (polling) dans laquelle l'entrée du système est surveillée en permanence en vue de détecter la présence des demandes, dont la présence conduit au déclenchement d'une fonction de traitement.

Concrètement, le polling est un concept qui consiste à interroger régulièrement un système afin de détecter l'arrivée d'un événement externe. A chaque période de temps « T », le système doit vérifier la présence de nouvelles demandes à son entrée, et déclencher par la suite le traitement adéquat. Même si ce concept présente l'avantage d'être simple à mettre en œuvre, il souffre de nombreux inconvénients, en particulier, celui qui concerne le calibrage du paramètre « T » surtout si la fréquence d'arrivée des demandes est aléatoire, ou inconnue à l'avance. En effet, Un « T » trop petit, permet une meilleure surveillance de l'entrée, et assure un traitement quasi instantané des demandes, mais le service risque d'être réveillé à plusieurs reprises sans qu'il y ait de demande, ce qui engendre une perte de ressources système (Temps CPU, mémoires, ...). Inversement, un « T » trop grand, réveille beaucoup moins le service, sauf que le traitement des demandes risque d'être différé. Pour toutes ces raisons, nous proposons un nouveau dispositif de surveillance nommé "guichet" et dont le fonctionnement est présenté dans la Figure 5.5

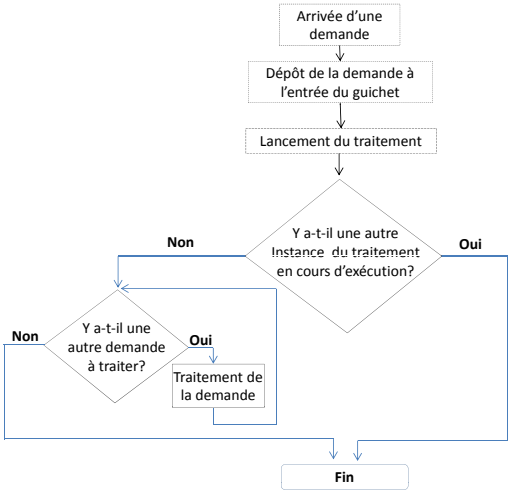

Figure 5.5 – Principe de fonctionnement du guichet.

5.3.3 Fonction de multiplexage

La fonction de multiplexage assure l'agrégation des messages déposés à l'entrée du MBA. Les paquets générés à la sortie du multiplexeur (les paquets prêts à l'émission) peuvent contenir un ou plusieurs segments à la fois, réduisant ainsi le nombre d'opérations E/S.

La composition structurelle du multiplexeur repose essentiellement sur un ensemble de listes d'attente chargées de stocker temporairement les messages déposés en entrée en vue de leur transmission sur le réseau. Le nombre de listes d'attente est égal au nombre de priorités que peut gérer le MBA (nombre de priorités=4 ⇒ nombre de listes=4). Lorsqu'un message est déposé à l'entrée du multiplexeur par un thread de traitement (Thread-n), il est systématiquement acheminé vers la liste d'attente qui correspond à son niveau de priorité, ainsi, les messages de priorité Pi peuvent être déposés de manière indépendante des messages d'une autre priorité. Ce principe de fonctionnement est présenté dans la Figure 5.6

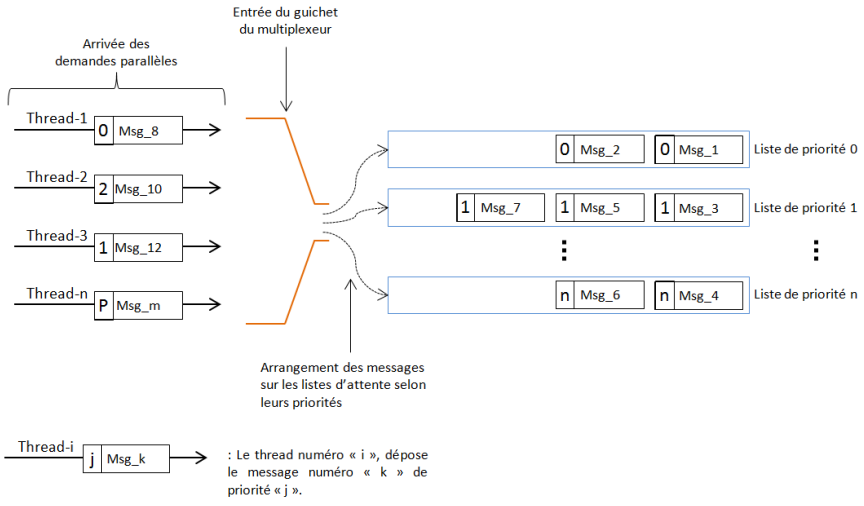

Figure 5.6 – Réorganisation des messages à l'entrée du multiplexeur.

Les listes d'attente sont implémentées sous forme de listes chainées de manière à pouvoir insérer et retirer facilement des éléments, et étant donné qu'elles ne détiennent que des références (adresses) vers la zone de mémoire décrivant le corps des messages, une recopie mémoire supplémentaire depuis l'espace d'adressage utilisateur vers celui du multiplexeur est évitée. Au sein d'une même liste d'attente, les messages sont gérés selon une politique FIFO.

Le générateur de paquets gère les listes d'attentes et offre un service d'agrégation des messages en fonction de leurs niveaux de priorité. Ainsi, ceux qui appartiennent aux listes de niveau supérieur seront servis en premier et ils sont rassemblés pour former un seul paquet à la sortie du multiplexeur. Chaque fois qu'un message est retiré d'une liste d'attente, sa structure descriptive est mise à jour, et à ce niveau deux cas de figures peuvent se présenter Si le message est totalement intégré au buffer d'émission (paquet généré à la sortie du multiplexeur), sa référence sera supprimée définitivement de la liste, sinon, le champ « Taille restante à transmettre » sera décrémenté en fonction de la taille des données réellement envoyées. Le buffer d'émission généré dans la phase d'agrégation est construit en fonction des règles suivantes :

1. **Règle 1** : Tant que la taille du buffer d'émission est inférieure à celle de l'unité de transmission, et la liste de messages de niveau de privilège supérieur n'est pas encore vide, un nouveau message est retiré, puis intégré dans le buffer d'émission.
2. **Règle 2** : S'il n'y a plus de messages dans la liste de niveau de privilège « i » et que la taille d'unité de transmission n'est pas encore atteinte, on passe à la liste de niveau « i+1 » (niveau inférieur). Si cette dernière est à son tour vide, nous passons à la liste suivante (i+2) et ainsi de suite. Ce processus sera répété jusqu'à épuisement de toutes les listes à traiter (inexistence de messages dans le système) ou égalisation de la taille du buffer d'émission avec celle de l'unité de transmission.
3. **Règle 3** : Si la taille du premier message sélectionné dépasse celle de l'unité de transmission, sa référence est directement utilisée comme adresse d'émission afin d'éviter toute recopie mémoire supplémentaire.
4. **Règle 4** : La présence d'un seul message dans une liste prioritaire est suffisant pour faire, à tout moment, basculer le générateur de paquet vers cette liste.
5. **Règle 5** : Si la taille mémoire disponible au niveau du buffer d'émission est inférieure à celle du message sélectionné, ce dernier sera fragmenté en plusieurs segments qui sont ainsi envoyés puis reconstitués ultérieurement lors de la phase de démultiplexage.

Nous souhaitons maintenant appliquer notre algorithme de multiplexage au système présenté dans la Figure 5.7 qui est composé de trois files d'attente de priorité 0, 1 et 2, tout en illustrant à chaque étape, le numéro de la règle d'agrégation qui a été appliquée.

A l'état initial, le paquet final est vide, et la file d'attente de priorité zéro (F0) contient deux messages portant respectivement les numéro 1 et 3 (le numéro ici représente juste l'identifiant du message et non pas sa priorité). F1 et F2 contiennent chacune un seul message, de son côté, le pointeur du générateur de paquets est positionné sur F0. Le scénario de notre algoithme se déroule en six étapes :

1. Application directe directe de la règle numéro 1. Le message portant le numéro 1 est retiré de F0 puis inséré dans le paquet final ;
2. Application directe directe de la règle numéro 1. Le message portant le numéro 3 est retiré de F0 puis inséré dans le paquet final juste après le message numéro 1 ;
3. F0 est vide, le générateur de paquets passe donc à F1 (règle numéro 2) retire le message numéro 2 est l'ajoute à la fin du buffer d'émission ;

4. Uun nouveau message arrive à destination F0 (message numéro 5) .
5. Le générateur de paquets détecte la présence d'un message de niveau de privilège plus élevé que celui de la file sélectionnée F1, il bascule donc vers F0, retire le message numéro 5 puis l'insère au buffer d'émission (règle numéro 4).
6. Le message numéro 4 est le dernier à rester dans le système. Il a une taille plus grande que celle disponible dans le paquet final et par conséquent, il est découpé en plusieurs segments qui seront traités dans les prochains envois (règle numéro 5).

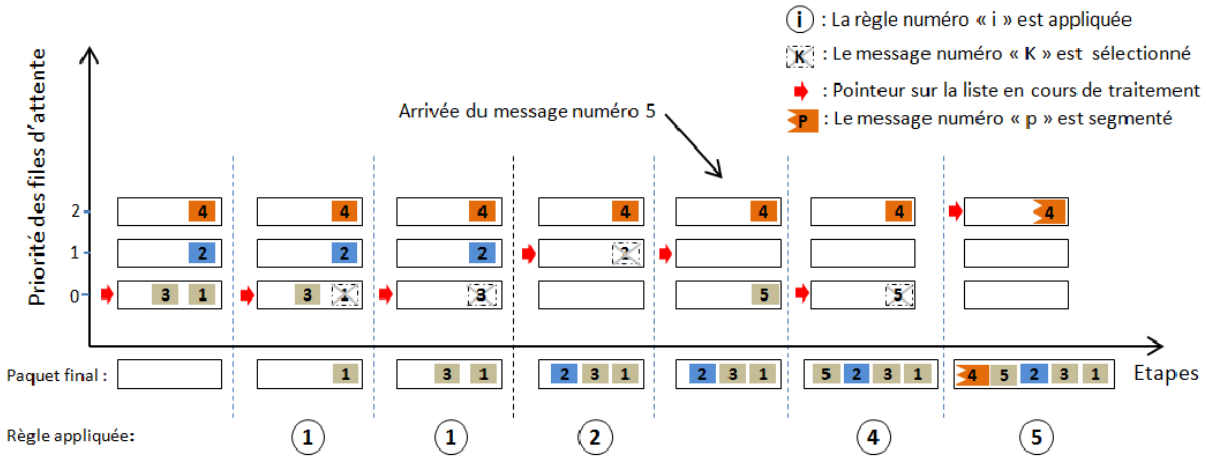

Figure 5.7 – Exemple de multiplexage.

Si nous regardons la composition du paquet final, nous apercevons que les segments ne sont pas rangés selon leur niveau de priorité, en effet, le segment numéro 2 (message 2) vient avant le segment numéro 5 (message 5) bien que ce dernier soit prioritaire. Cela ne pose aucun problème particulier puisque les segments du paquet final sont transmis d'un seul coup sur le réseau, il n'est donc pas nécessaire d'envisager une étape supplémentaire pour les réorganiser à l'intérieur du buffer de transmission.

5.4 Le démultiplexage

Le démultiplexeur est le composant logiciel qui assure le dépaquetage et la reconstitution des messages à la réception, et son exécution peut se dérouler en parallèle avec celle du multiplexeur puisqu'il n'y a aucune dépendance fonctionnelle entre ces deux composants.

Les paquets reçus à l'entrée du démultiplexeur peuvent être composés de plusieurs segments qui sont repérables par leurs en-têtes et qui jouent un rôle double : d'une part, ils marquent le début d'un segment, d'autre part, ils détiennent toutes les informations qui lui sont associées telles que : Identifiant, taille, contenu. La reconstruction des messages se fait de manière incrémentale et basée sur les deux règles suivantes :

1. **Règle 1** : Si le segment dépaqueté représente un message complet, le démultiplexeur recopie son contenu dans un buffer temporaire et met en place un nouveau thread pour son traitement.
2. **Règle 2** : Si le segment dépaqueté représente juste un fragment de message, il est stocké dans une liste temporaire en attendant l'arrivée du reste des fragments complémentaires,

quant au thread de traitement, il n'est pas déclenché tant que le message n'est pas totalement reconstruit.

Nous présentons dans la Figure 5.8 le scénario de déroulement du processus de démultiplexage en prenant en exemple le paquet multiplexé généré précédemment (Figure 5.7).

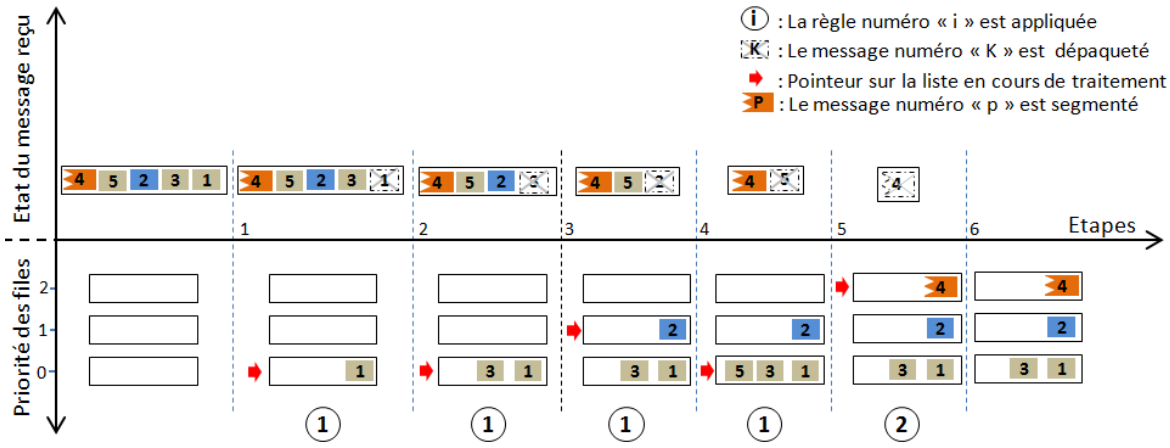

Figure 5.8 – Exemple de démultiplexage.

A l'état initial, le démultiplexeur reçoit un message composé de 5 segments (1, 3, 2, 5 et 4), la reconstitution se fait en 5 étapes :

1. Le segment 1 est extrait du message, étant donné qu'il représente un message entier, le démultiplexeur crée un nouveau thread pour le traiter (règle 1).
2. Le segment 3 sera extrait. C'est un message complet de priorité 0, la règle 1 lui est alors appliquée.
3. Le segment 2 est extrait, puis un nouveau thread est déclenché pour le traiter.
4. Le segment 5 est extrait et la règle 1 lui est appliquée.
5. Il reste un seul segment dans le système, Il correspond à un message fragmenté. Le démultiplexeur le stocke dans la liste conforme à son niveau de priorité (3) mais aucun thread de traitement ne sera appelé tant que le message n'est pas entièrement reconstitué (règle 2).

6

Évaluation de performance du Bus Applicatif

Sommaire

6.1 Introduction

L'objectif de notre étude est de déterminer la performance de l'architecture du bus en fonction de certains paramètres liés à son environnement d'exécution. A l'issue de cette étude, les indices de performance récoltés révèlent la capacité de l'infrastructure à supporter la charge, et prouvent l'efficacité du mécanisme de multiplexage/démultiplexage mis en œuvre, également, l'analyse et l'interprétation des résultats obtenus ont rendu possible d'une part, la localisation des différentes parties de l'architecture susceptibles de causer des goulots d'étranglement, d'autre part, la comparaison entre les nombreuses configurations possibles de l'architecture dans le but de fixer un meilleur ajustement des paramètres systèmes [BBA12].

6.2 Problématiques

Il existe dans la littérature de nombreuses approches pour l'évaluation de performance parmi lesquelles ont peut citer : la simulation, l'émulation, les mesures, les modèles mathématiques [Him07].

L'une des spécificité de l'architecture du BA est qu'elle se caractérise par un mode de fonctionnement collaboratif massivement parallèle qui met en jeu un nombre important de paramètres qui sont soit à caractère matériel (nombre de processeurs, taille de mémoire, etc) soit logiciel (nombre de processus, nombre de threads, ...) ce qui complique considérablement la démarche d'évaluation. Ainsi, parmi toutes les d'approches d'évaluation citées précédemment (simulation, émulation, ...) quelques unes seulement s'avèrent suffisamment adaptées pour prendre en compte l'intégralité des paramètres caractérisant l'architecture du BA.

L'autre aspect important à prendre en compte dans cette étude est la performance de la couche de communication réseau, et à ce niveau, nous ne pouvons pas faire abstraction des performances du standard TCP/IP étant donné qu'il constitue partie intégrante de notre architecture.

Le problème d'évaluation de performances réseau est classique, et de nombreux travaux de recherche on déjà traité ce sujet. Ces études ont abouti à l'élaboration de quelques modèles analytiques susceptibles de représenter et d'estimer le comportement d'un réseau [Jub07] parmi lesquels on peut citer : le modèle LogP, le modèle LogGP, le modèle pLogP, etc. Malheureusement, ces modèles sont inadaptés à l'architecture du BA car ils ne prennent pas en considération l'ensemble des couches logicielles du BA (TBA, MBA, ...)

L'influence du réseau sur les temps mesurés est négligeable pour des échanges peu fréquents et de faible taille, mais son impact peut être important lors des échanges fréquents ou volumineux. L'application d'une charge de données importante à l'extrémité du BA ne fait que surcharger le réseau, ce qui n'est pas sans influence sur le processus d'évaluation, qui dans ce cas, reviendra plutôt à une mesure de performance du réseau TCP/IP au lieu d'une évaluation de l'architecture du BA où les échanges sont calibrés de manière à ne pas surcharger la bande passante. Nous n'avons donc pas effectué de mesures pour les échanges de type transfert de fichiers dans nos tests pour les raisons suivantes :

1. Le transfert de fichiers fait intervenir la bande passante et par conséquent influe sur les temps de réponses que nous souhaitons mesurer.
2. Le transfert de fichiers fait partie des services de base du BA et ne fait pas entrer en action l'ensemble des éléments structurels.
3. Les temps de réponse mesurés lors d'un transfert de fichiers ne reflètent que la performance du réseau et non celle du BA.

Pour toutes ces raisons, nous avons opté pour une démarche d'évaluation à base de tests dans laquelle nous nous intéressons exclusivement aux temps de réponse récoltés sur une plateforme expérimentale qui utilise notre BA.

6.3 Modèle d'étude et Méthodologie

Notre modèle d'étude est composé de deux applications A1 et A2 reliées chacune au BA via un terminal. Les deux applications fonctionnent sur deux environnements différents (Windows pour A1 et Linux pour A2). Les applications lancent plusieurs tâches qui se déroulent en parallèle (T1, T2, T3, ..., Tn).

Nous menons une série d'échanges simultanés entre tâches. Le temps total d'échange est mesurée. L'application A2 lance un certain nombre de tâches parallèles en fonction de ses paramètres d'initialisation (nombre de canaux à solliciter, taille des données échangées). Chaque tâche dépose une demande à destination de la tâche destinataire et reçoit instantanément un acquittement du MBA hôte. L'application A1 joue le rôle de bouchon qui permet de mettre en

place les réponses relatives aux demandes reçues en y répondant sans aucun temps de traitement. Le modèle étudié est présenté dans la Figure 6.1

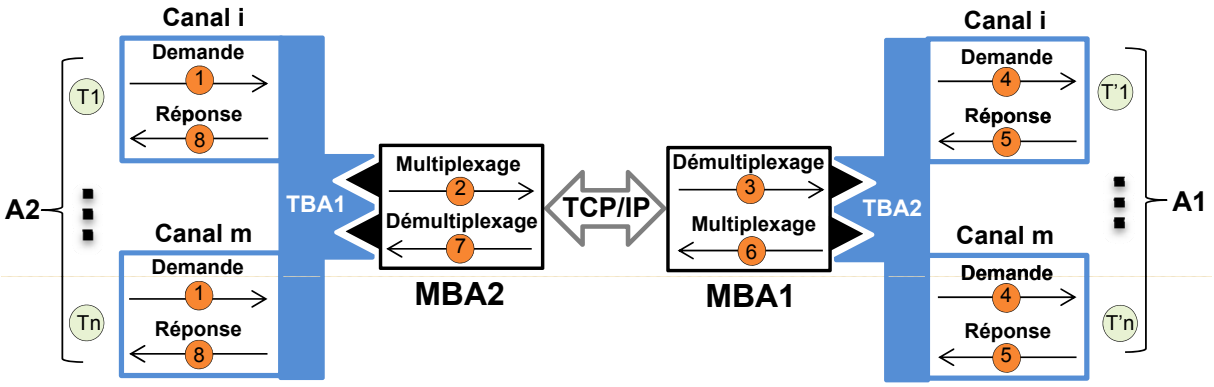

Figure 6.1 – Présentation du modèle d'étude.

Nous constatons depuis ce modèle que les tâches parallèles d'une application (A1 et A2) utilisent le même terminal. Cependant, Le BA donne aux différentes tâches la possibilité d'avoir leur propre terminal (terminal esclave). Cela ne change rien quant aux mesures faites, puisque c'est la même fonction de multiplexage qui sera appliquée sur l'ensemble des terminaux. Nous avons utilisé un modèle qui exploite les N canaux d'un terminal, au lieu d'un modèle composé de N terminaux et à canal unique. Le scénario d'échange entre les deux applications (A1, A2) est le suivant :

1. Une tâche de A2 émet une demande sur le canal de son MBA. Celui-ci acquitte le bon dépôt de la demande et libère la tâche de A2.
2. Le MBA de A2 gère sa file d'attente (en émission) suivant un procédé dépendant du type d'échange, sécurise la demande, multiplexe les éventuelles demandes émises parallèlement, et transmet le tout au MBA destinataire via le réseau.
3. Le MBA de A1 gère sa file d'attente en réception en FIFO, démultiplexe l'échange reçu, décrypte chaque demande et la (les) place en ressources partagées sur le canal du terminal destinataire et informe enfin la tâche destinataire de la présence d'une nouvelle demande.
4. La tâche destinataire traite la demande et place sa réponse sur le MBA de A1. Ce dernier acquitte la demande et libère la tâche.
5. Le MBA de A1 gère sa file d'émission, sécurise les différents messages à destination du MBA de A2, les multiplexe, et transmet l'ensemble au MBA de A2 via le réseau.
6. Le MBA de A2 gère sa file de réception en FIFO, démultiplexe et décrypte la réponse, la place sur le canal ad-hoc et signale sa présence à la tâche initiale.

Faire précéder la couche de multiplexage/démultiplexage avec la couche de cryptage/décryptage présente un intérêt majeure notamment sur les points suivants :

- Chaque usager peut avoir sa propre clef de cryptage.
- La sécurité est très renforcée, chaque paquet multiplexé comportant des cryptages différents.
- Nous favorisons l'exploitation des processeurs multi-cœurs donnant une puissance de traitement importante grâce à de nombreux threads de traitement en particulier lors du cryptage/décryptage.

6.4 Environnement expérimental

6.4.1 Architecture matérielle

L'architecture matérielle de mesure utilisée est composée de deux couples de machines. Le premier couple est composé de machines mono-cœur, le deuxième de machines multi-cœurs. La connexion entre machines de chaque couple est assurée par un réseau Ethernet (100Mb/s). Le Tableau 6.1 donne un récapitulatif des caractéristiques des deux configurations de couples précédentes :

	Couple_1 (machines mono-cœur)		Couple_2 (machines multi-cœurs)	
Système d'exploitation	Windows (Application A1)	Linux (Application A2)	Windows (Application A1)	Linux (Application A2)
Processeur	Intel® Pentium®4 monocoeur 3.06 GHZ	Intel® Pentium®4 monocoeur 2.53GHz	Intel® Pentium® Dual core CPU 3.00GHZ 2.99GHZ	Intel® Xeon(TM) Quadri-cœur CPU 3.60GHz
RAM	2Go	1Go	4Go	15Go
Version OS	Windows XP	Linux(Debian) 2.6.26-2-686	Windows 7	Linux (Fedora 14) 2.6.35.6-45.fc14.i686

Tableau 6.1 – Configuration Materielle.

6.4.2 Description de l'outil de mesure

Afin d'avoir une meilleure précision, les temps sont mesurés sur la machine Linux (Machine de A2). Il existe plusieurs fonctions de mesure de temps sous Linux (time, gettimeofday, ...). Nous avons choisi la fonction clock_gettime pour les deux raisons suivantes :

- La fonction clock_gettime fournit une précision de l'ordre de la nanoseconde.
- Cette fonction donne la possibilité de mesurer le temps CPU consommé par les threads.

La fonction clock_gettime est appelée, entre chaque début et fin d'exécution, d'un ensemble de tâches parallèles :

1. clock_gettime (CLOCK_THREAD_CPUTIME_ID, tDebut) pour mesurer tDebut.
2. Exécution d'un ensemble de tâches parallèles (T1, T2, T3, ..., Tn)
3. clock_gettime (CLOCK_THREAD_CPUTIME_ID, tFin). Pour mesurer tFin.

Le temps calculé entre tDebut et tFin représente alors la durée totale nécessaire à l'acquittement de toutes les demandes déjà déposées. Pour avoir des résultats plus pertinents, chaque test est répété 50 fois, la valeur moyenne est alors retenue. Pour des raisons de simplification de calcul, les temps seront arrondis à l'échelle de la microseconde.

6.5 Résultats des tests

6.5.1 Architecture mono-cœur avec mode d'exécution séquentiel

Les résultats présentés dans cette partie, sont récoltés sur une architecture matérielle composée de deux machines mono-cœur dont les caractéristiques sont indiquées dans l'entrée Couple_1 du tableau précédent. Le MBA fonctionne en mode séquentiel.
Lorsque le MBA fonctionne en mode séquentiel, les demandes déposées par les tâches ne sont pas traitées en parallèle. Le MBA applique donc une politique de gestion en FIFO. Les demandes sont traitées d'une manière séquentielle. Dans ce cas, la fonction de multiplexage est interrompue, A chaque message envoyé/reçu correspond une opération send/recv.

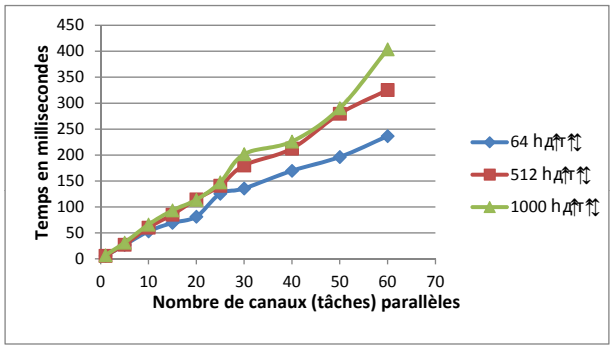

Figure 6.2 – Résultats mesurés sur le couple de machines mono-cœur en mode d'exécution séquentiel.

6.5.2 Architecture mono-cœur avec mode d'exécution parallèle

L'architecture matérielle est la même que celle décrite précédemment (couple_1). Les paramètres de tests n'ont pas changé. Nous avons gardé la même taille des messages échangés et le même nombre de canaux parallèles sollicités, sauf que les demandes déposées par les tâches sont traitées en parallèle. Le MBA fonctionne alors en mode parallèle et attribue à chaque demande déposée, un nouveau thread de traitement.

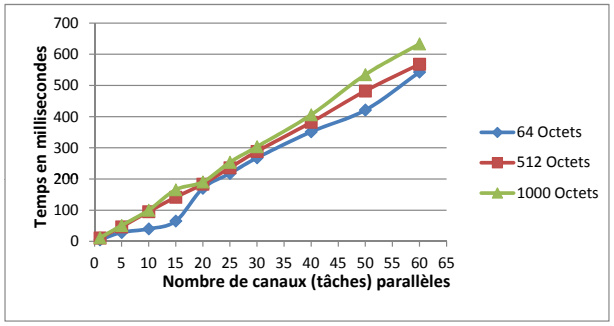

Figure 6.3 – Résultats mesurés sur le couple de machines mono-coeur en mode d'exécution parallèle.

6.5.3 Architecture Multi-cœur avec mode d'exécution Séquentiel

L'architecture matérielle est composée de deux machines multi-cœurs dont les caractéristiques sont décrites dans l'entrée « couple_2 » du tableau numéro 3. La configuration réseau est préservée. Le MBA fonctionne en mode Séquentiel.

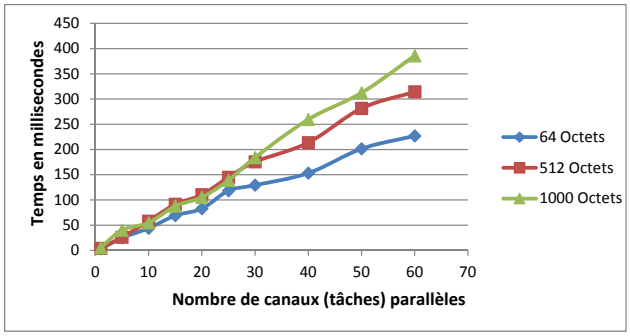

Figure 6.4 – Résultats mesurés sur le couple de machines Multi-cœurs en mode d'exécution Séquentiel.

6.5.4 Architecture Multi-cœur avec mode d'exécution Parallèle

L'architecture matérielle est composée de deux machines multi-cœurs dont les caractéristiques sont décrites dans l'entrée « couple_2 » du tableau numéro 3. La configuration réseau est préservée. Le MBA fonctionne en mode parallèle.

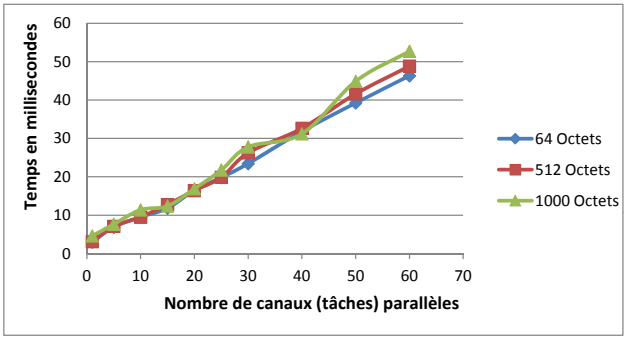

Figure 6.5 – Graphe de performances de la configuration de machines Multi-cœurs avec mode d'exécution parallèle.

6.6 Analyse des résultats

L'analyse des résultats obtenus nécessite tout d'abord l'isolation de tous les facteurs constants qui n'ont pas d'impact sur les indices de performances, le but de cette étude étant d'évaluer les performances de l'architecture complète du BA et non celles du standard TCP/IP. Pour cela, les paramètres de test (nombre de canaux parallèles, taille des messages) ont été choisis de

manière à ne pas surcharger le réseau. La bande passante devient donc un facteur constant qui n'influe pas sur les temps de réponse. Nous considérons aussi que la mémoire système disponible est suffisante pour l'ensemble des threads afin que les mesures ne soient pas faussées par les problèmes de gestion mémoire (swapping).

Le mode séquentiel (Figure 6.2 et Figure 6.3) , et en évitant le lancement de threads, s'avère plus efficace que celui parallèle sur les anciennes configurations mono-cœurs, mais il est logiquement plus sensible à la taille des messages qui demandent un traitement proportionnel à cette taille. En effet, le mode parallèle sur ces anciennes configurations pénalise le fonctionnement du MBA à cause du lancement de nombreux threads, ce qui a pour effet secondaire de changer l'échelle du graphe et rapprocher ainsi les courbes.

Les meilleurs résultats sont obtenus sur le couple des machines Multi-cœurs. Au niveau système, la capacité de la machine à supporter l'exécution de plusieurs threads (demandes) parallèles à la fois en est la justification. Le temps de gestion système (lancement, scheduling, ...) est le même pour les deux machines, La puissance des nouveaux processeurs rapproche les courbes des tailles messages mais il apparait surtout que les temps d'exécution sont beaucoup plus réduits dans un rapport de 1 à 10 grâce au parallélisme.

Sur le plan logiciel (MBA), l'efficacité du mécanisme de multiplexage/démultiplexage dépend de la fréquence d'arrivée des demandes. Le nombre de données à multiplexer est proportionnel à celui des messages et par conséquent, les paquets générés à la sortie du multiplexeur contiendront davantage de segments, réduisant ainsi le nombre d'opérations E/S.

6.7 Comparaison des résultats

L'étude présentée dans [GJMCGS10] propose une évaluation de performance de trois ESB open-source : Fuse, Mule et Petals. Le modèle étudié est basé sur une application de type client, lancée en plusieurs instances et dont le rôle consiste à mesurer les temps de réponse relatifs aux invocations de services externes. Aucun traitement particulier n'est réalisé par ces services, et leurs rôles se résument à l'acquittement des demandes d'invocation envoyées par les clients suivant deux modes de transfert : sécurisé et non sécurisé. Les temps mesurés représente seulement la durée nécessaire pour traverser toutes les couches logicielles. Le Tableau 6.2 présente le temps de réponse moyen en ms des trois ESB mentionnés précédemment.

	Fuse	Mule	Petals
Non-Secure Web Service	15,49	7,84	11,13
Secure Web Service	31,67	40,96	25,72

Tableau 6.2 – Temps de réponse de Fuse, Mule et Petals.

La démarche d'évaluation que nous avons suivie est analogue à celle adoptée dans [GJMCGS10]. Il s'agit de la mesure du temps nécessaire pour la traversée des couches logicielles de la plateforme, avec une durée de traitement nulle au niveau de l'entité distante invoquée. Il n'y a donc aucune influence du parallélisme sur la partie purement exécution, qui elle, dépend effectivement de la puissance de la machine. Par conséquent, le gain que nous avons obtenu revient essentiellement à la *pile logicielle* du BA.

La démarche de comparaison présentée dans les sections 6.5.3 et 6.5.4 est relative à notre plateforme, sur laquelle, nous avons établi une analyse entre les résultats obtenus avec le mode de fonctionnement séquentiel (sans multiplexage/démultiplexage) par rapport à ceux du mode

parallèle (avec multiplexage/démultiplexage) pour les mêmes configurations matérielles. Cette analyse a montré naturellement l'intérêt et l'efficacité des mécanismes proposés (multiplexage, démultiplexage, canaux d'échange, ...) dans un rapport de 1 à 10 grâce au parallélisme.

Au regard des hypothèses faites dans [GJMCGS10], nous pouvons comparer le temps de réponses du BA en mode séquentiel, et qui est de l'ordre de 5ms (voir Figure) avec celui des trois ESBs (voir Tableau 6.2) : Fuse (15,49 ms), Mule (7,84 ms), Petals (11,13 ms). Nous pouvons donc dériver sur ces mêmes hypothèses que les performances de notre BA en mode parallèle sont meilleures que celles des trois ESBs étudiés dans [GJMCGS10].

6.8 Conclusion

Nous avons présenté dans cette section une étude portant sur l'évaluation de performance du BA, plus précisément, de son mécanisme de multiplexage/démultiplexage. La démarche suivie à cette fin s'inscrit dans le cadre des approches par tests. Les résultats mettent en évidence l'efficacité du mécanisme de multiplexage/démultiplexage que nous proposons mais surtout son adaptation aux nouvelles architectures multi-cœurs.

Par ailleurs, il faut signaler que le processus d'agrégation des données nécessite l'ajout d'un certain nombre d'informations complémentaires aux messages échangés. En effet, nous associons à chaque segment de données, un en-tête unique qui permet de le distinguer des autres segments du même paquet. Ces en-têtes peuvent être vus comme étant une masse d'informations supplémentaires à l'émission qui engendre un coût additionnel lors du transfert sur le réseau par rapport à un envoi direct. Cependant, les résultats montrent qu'il est beaucoup plus efficace d'utiliser des paquets à segments multiples que des envois multiples. En effet, le coût d'accès au support de communication (opération E/S) est beaucoup plus important que celui de l'agrégation des messages.

Conclusion et perspectives

Conclusion générale

Le domaine d'architectures d'intégration d'applications orientées services est un concept clé pour toute entreprise qui cherche avant tout, à gagner en flexibilité et en réactivité tout en restant compétitive, et ce, aussi bien dans un contexte intra-organisationnel qu'inter-organisationnel.

Ce domaine a été porté par l'élan de l'évolution technologique constante que connait le monde de l'informatique actuellement, notamment celle d'Internet ces dix dernières années. La planète entière est aujourd'hui un lieu d'échanges continus non seulement entre les applications mais aussi, entre l'utilisateur et les applications, et entre les utilisateurs. Cette explosion de la communication engendre des besoins de plus en plus importants en termes de puissance de traitement de données, d'où l'intérêt du parallélisme.

Depuis déjà quelques années, les constructeurs, toujours en quête de puissance accrue de traitement de données suivent la voie du parallélisme. Cependant, les architectures ESBs s'y sont mal adaptées, peinent à l'intégrer, notamment aux niveaux de leurs plateformes de déploiement, marquant ainsi un grand retard. Pour des raisons de performances, le développement de ces architectures ESBs ne peut rester en marge de l'évolution technologique des machines multicœurs et multiprocesseurs. Il est donc temps que le domaine d'architectures d'intégration d'applications dispose d'une nouvelle plateforme adaptée aux traitements massivement parallèles permettant de tirer le maximum de profit de la puissance offerte par les machines multicœurs/multiprocesseurs, d'où l'intérêt de la plateforme que nous proposons.

A travers cette étude, nous avons mis en évidence les principaux obstacles engendrés par l'intégration du traitement massivement parallèle dans la plateforme de déploiement d'une architecture ESB, qu'ils soient techniques ou fonctionnels. L'obstacle principal auquel nous avons fait face concerne l'inadaptation du mode parallèle d'exécution des traitements alors que celui de communication reste séquentiel, d'où l'extrême importance du mécanisme de multiplexage/démultiplexage que nous avons mis en œuvre. L'efficacité du Multiplexeur du Bus Applicatif est en grande partie, due à la conception du mécanisme des canaux parallèles d'échange qui permettent une bonne acquisition et préparation des données en amont, de leurs multiplexages. A cela, s'ajoute le concept de la P-Instruction qui a résolu avec succès les problèmes de synchronisation entre tâches.

Par ailleurs, cette étude a d'une part, montré la pertinence de la faisabilité des idées proposées au début, qui, une fois concrétisées ont, d'autre part, permis l'évaluation des performances de l'architecture du Bus Applicatif dans son intégralité. Les résultats des tests démontrent que le Bus Applicatif est parfaitement adapté au traitement massivement parallèle, et exploite bien la puissance de traitement fournie par les machines multicœurs/multiprocesseurs.

En conclusion, nous dirons que notre Bus Applicatif a atteint son principal objectif concrétisé dans l'intégration des applications hétérogènes et la collaboration de leurs tâches parallèles, dans un environnement distribué, à travers une API simple à utiliser. Dans l'espoir de voir un jour,

certains concepts mis en avant dans cette thèse directement intégrés aux systèmes d'exploitation, nous restons très optimistes quant à l'avenir très prometteur du parallélisme dans les architectures d'intégrations orientées services.

Perspectives

Ces travaux ouvrent de nombreuses perspectives de recherche qui ont pour ambition d'améliorer encore plus le rendement de notre plateforme, à travers une nouvelle extension appelée *Noyau du Bus Applicatif* (NBA). Ce dernier joue le rôle d'un *switch logiciel* en mesure de rejouer le niveau de priorisation des messages afin d'assurer orchestration des tâches connectées au Bus Applicatif. De part sa position centralisée, le NBA constitue l'élément pivot de l'architecture du BA. Sa mise en œuvre s'appuie essentiellement sur deux mécanismes : le premier destiné à l'ordonnancement, le deuxième à l'apprentissage.

Ainsi donc, l'architecture du Bus Applicatif fonctionnera sous deux configurations différentes : la première dite *standard* n'incluant aucun noyau, la deuxième dite *avancée* intégrant un Noyau de Bus Applicatif.

Par ailleurs, le NBA assure le routage entre plusieurs constellations de MBAs offrant ainsi, d'une part, une forte montée en charge, d'autre part, un second niveau d'optimisation des échanges sur le réseau.

Nous avons quelques pistes intéressantes pour la réalisation de cette nouvelle extension. L'idée consiste à assigner pour chaque tâche une durée d'exécution estimative, qui en fonction d'un mécanisme d'apprentissage, est mise à jour au fur et à mesure suivant la durée d'exécution réelle indiquée par le compte-rendu du BA.

Par ailleurs, les paramètres pris en compte pour l'ordonnancement au niveau du NBA peuvent être représentés sous format matriciel. Il serait donc préférable d'installer le NBA sur une machine vectorielle de la constellation, intégrant des dispositifs dédiés de traitement comme par exemple les GPGUs.

Bibliographie

[AB12] Yves Albrieux and Ridha Benosman. Procédé d'exécution parallèle d'une pluralité de tâches informatiques, January 2012.

[ACDN13] Maida Arnautovic, Maida Curic, Emina Dolamic, and Novica Nosovic. Parallelization of the ant colony optimization for the shortest path problem using openmp and cuda. In *Information Communication Technology Electronics Microelectronics (MIPRO), 2013 36th International Convention on*, pages 1273–1277, 2013.

[AHD11] Spiros N. Agathos, Panagiotis E. Hadjidoukas, and Vassilios V. Dimakopoulos. Design and implementation of openmp tasks in the ompi compiler. *Informatics, Panhellenic Conference on*, 0 :265–269, 2011.

[Alb10] Yves Albrieux. Procédé d'exécution parallèle d'un processus informatique par un bus applicatif, July 2010.

[ANAN10] A. Alghamdi, M. Nasir, I. Ahmad, and K.A. Nafjan. An interoperability study of esb for c4i systems. In *Information Technology (ITSim), 2010 International Symposium in*, volume 2, pages 733 –738, june 2010.

[Ant01] Gabriel Antoniu. *DSM-PM2 : une plate-forme portable pour l'implémentation de protocoles de cohérence multithreads pour systèmes à mémoire virtuellement partagée*. These, Ecole normale supérieure de lyon - ENS LYON, November 2001.

[Aum02] Olivier Aumage. *Madeleine : une interface de communication performante et portable pour exploiter les interconnexions hétérogènes de grappes.* Thèse de doctorat, spécialité informatique, École normale supérieure de Lyon, 46, allée d'Italie, 69364 Lyon cedex 07, France, September 2002. 154 pages. URL : `http://runtime.bordeaux.inria.fr/Download/Publis/Aum02These.ps`.

[BAN+08] E. Brunet, O. Aumage, R. Namyst, et al. Newmadeleine : ordonnancement et optimisation de schémas de communication haute performance. 2008.

[BBA12] Ridha Benosman, Kamel Barkaoui, and Yves Albrieux. Performance evaluation of a massively parallel esb-oriented architecture. In *IEEE International Conference on Service-Oriented Computing and Applications. SOCA 2012*, pages 270–273, December 2012.

[BBA13a] Ridha Benosman, Kamel Barkaoui, and Yves Albrieux. Design and implementation of a massively parallel esb. In *International Symposium on Programming and Systems (ISPS'2013)*, pages 89 – 95, April 2013.

[BBA13b] Ridha Benosman, Kamel Barkaoui, and Yves Albrieux. Exploiting concurrency for the esb architecture. In *International Conference on Engineering of Complex Computer Systems. ICECCS 2013*, July 2013.

[BBA13c] Ridha Benosman, Kamel Barkaoui, and Yves Albrieux. A new dynamic ipc-memory allocator based on a paging approach. In *International Conference on High Performance Computing and Simulation (HPCS 2013)*, pages 382–389, 2013.

[BBH+98] Henri E. Bal, Raoul Bhoedjang, Rutger Hofman, Ceriel Jacobs, Koen Langendoen, Tim R, and M. Frans Kaashoek. Performance evaluation of the orca shared object system. *ACM Transactions on Computer Systems*, 16, 1998.

[BCM04] F. Briclet, C. Contreras, and P. Merle. Openccm : une infrastructure à composants pour le déploiement d'applications à base de composants corba. *Arxiv preprint cs/0411059*, 2004.

[BDC10] K.L. Baishnab, S. Dev, and Z.H. Choudhury. An efficient heap management technique with minimum fragmentation and auto compaction. In *Computer Science and Information Technology (ICCSIT), 2010 3rd IEEE International Conference on*, volume 3, pages 145 –149, july 2010.

[Bla05] C. Blaess. *Programmation système en C sous Linux : signaux, processus, threads, IPC et sockets*. Eyrolles, 2005.

[Bru08] Élisabeth Brunet. *Une approche dynamique pour l'optimisation des communications concurrentes sur réseaux haute performance*. PhD thesis, Université Bordeaux 1, 351 cours de la Libération — 33405 TALENCE cedex, December 2008.

[BSKM11] G. Barootkoob, M. Sharifi, E.M. Khaneghah, and S.L. Mirtaheri. Parameters affecting the functionality of memory allocators. In *Communication Software and Networks (ICCSN), 2011 IEEE 3rd International Conference on*, pages 499 –503, may 2011.

[Bul03] Christopher Bull. Strategic issues in customer relationship management (crm) implementation. *Business Proc. Manag. Journal*, 9(5) :592–602, 2003.

[BZ93] David A. Barrett and Benjamin G. Zorn. Using lifetime predictors to improve memory allocation performance. *SIGPLAN Not.*, 28(6) :187–196, June 1993.

[Car] Denis Caromel. Proactive parallel suite. `http://fusesource.com/products/enterprise-servicemix/`.

[CC97] Giovanni Chiola and Giuseppe Ciaccio. Gamma : A low-cost network of workstations based on active messages. In *PDP*, pages 78–83, 1997.

[Cha04] David Chappell. *Enterprise Service Bus*. O'Reilly Media, 2004.

[CL08] Denis Caromel and Mario Leyton. Proactive parallel suite : From active objects-skeletons-components to environment and deployment. In *Euro-Par Workshops*, pages 423–437, 2008.

[Col61] George O. Collins, Jr. Experience in automatic storage allocation. *Commun. ACM*, 4(10) :436–440, October 1961.

[Coma] Fuse Open Source Community. Fuse esb. `http://fusesource.com/products/enterprise-servicemix/`.

[Comb] MuleSoft Community. Mule esb. `http://www.mulesoft.org/`.

[Comc] IBM Company. Websphere esb. `http://www-01.ibm.com/software/integration/wsesb/`.

[Cona] OW2 Consortium. Petals esb. `http://petals.ow2.org/`.

[Conb] OW2 Consortium. Talend etl. `http://www.talend.com/`.

[DDZ94] David Detlefs, Al Dosser, and Benjamin Zorn. Memory allocation costs in large c and c++ programs. *Softw. Pract. Exper.*, 24(6) :527–542, June 1994.

[Den02] Alexandre Denis. PadicoTM : un environnement ouvert pour l'intégration d'exécutifs communiquants. In *14èmes Rencontres Francophones du Parallélisme (RenPar'14)*, pages 99–106, Hammamet/Tunisie, April 2002.

[Den03] Alexandre Denis. *Contribution à la conception d'une plate-forme haute performance d'intégration d'exécutifs communicants pour la programmation des grilles de calcul.* These, Université Rennes 1, December 2003.

[DM99] George Dramitinos and Evangelos P. Markatos. Adaptive and reliable paging to remote main memory. *J. Parallel Distrib. Comput.*, 58(3) :357–388, September 1999.

[DNR00] Vincent Danjean, Raymond Namyst, and Robert D. Russell. Integrating kernel activations in a multithreaded runtime system on top of linux, 2000.

[DRM+98] D. Dunning, G. Regnier, G. McAlpine, D. Cameron, B. Shubert, F. Berry, A.M. Merritt, E. Gronke, and C. Dodd. The virtual interface architecture. *Micro, IEEE*, 18(2) :66–76, 1998.

[EFGK03] Patrick Th. Eugster, Pascal A. Felber, Rachid Guerraoui, and Anne-Marie Kermarrec. The many faces of publish/subscribe. *ACM Comput. Surv.*, 35(2) :114–131, June 2003.

[EP01] José Esteves and J. Pastor. Enterprise resource planning systems research : An annotated bibliography. *Communications of the Association for Information Systems*, 7(8) :1–51, 2001.

[FLR98] Matteo Frigo, Charles E. Leiserson, and Keith H. Randall. The implementation of the cilk-5 multithreaded language. *SIGPLAN Not.*, 33(5) :212–223, May 1998.

[FMMA11] Tais B. Ferreira, Rivalino Matias, Autran Macedo, and Lucio B. Araujo. An experimental study on memory allocators in multicore and multithreaded applications. In *Proceedings of the 2011 12th International Conference on Parallel and Distributed Computing, Applications and Technologies*, PDCAT '11, pages 92–98, Washington, DC, USA, 2011. IEEE Computer Society.

[Fre07] Christophe Frezier. Modification de PadicoTM afin de fournir une interface de type Madeleine. Rapport Technique RT-0334, INRIA, 2007.

[FZRL08] I. Foster, Yong Zhao, I. Raicu, and Shiyong Lu. Cloud computing and grid computing 360-degree compared. In *Grid Computing Environments Workshop, 2008. GCE '08*, pages 1–10, 2008.

[GJMCGS10] FJ García-Jiménez, MA Martínez-Carreras, and AF Gómez-Skarmeta. Evaluating open source enterprise service bus. In *e-Business Engineering (ICEBE), 2010 IEEE 7th International Conference on*, pages 284–291. IEEE, nov 2010.

[GP97] I. Ginzburg and B. Plateau. Athapascan-0b : Intégration efficace et portable de multiprogrammation légère et de communications. 1997.

[Gra03a] J.S. Gray. *Interprocess Communications in Linux.* Prentice Hall PTR, 2003.

[Gra03b] J.S. Gray. *Interprocess Communications in Linux : The Nooks and Crannies.* Prentice Hall PTR, 2003.

[GZ93] Dirk Grunwald and Benjamin Zorn. Customalloc : Efficient synthesized memory allocators. *SOFTWARE-PRACTICE AND EXPERIENCE*, pages 851–869, 1993.

[GZH93] Dirk Grunwald, Benjamin Zorn, and Robert Henderson. Improving the cache locality of memory allocation. pages 177–186. ACM Press, 1993.

[hCZ11] Xiao hui Cheng and Liang Zhang. A research of inter-process communication based on shared memory and address-mapping. In *Computer Science and Network Technology (ICCSNT), 2011 International Conference on*, volume 1, pages 111 – 114, dec. 2011.

[Hen08] Michi Henning. The rise and fall of corba. *Commun. ACM*, 51(8) :52–57, August 2008.

[Her09] Fabien Hermenier. *Gestion dynamique des tâches dans les grappes, une approche à base de machines virtuelles.* These, Université de Nantes, November 2009.

[Him07] Marouane Himdi. *Performances des systàmes distribués : proposition d'une plateforme de supervision fédératrice de la supervision.* PhD thesis, Université de Rennes 1, December 2007.

[HMMR07] T. Hoefler, M. Mosch, T. Mehlan, and W. Rehm. CollGM - A Myrinet/GM optimized collective component for Open MPI. In *Proceedings of 3rd KiCC Workshop 2007*. RWTH Aachen, Dec. 2007.

[HX09] Delin Hou and Huosong Xia. Design of distributed architecture based on java remote method invocation technology. In *Environmental Science and Information Application Technology, 2009. ESIAT 2009. International Conference on*, volume 2, pages 618–621, 2009.

[Inta] Intel. Cilk plus. `http://software.intel.com/en-us/intel-cilk-plus`.

[INTb] INTEL. Tbb (threading building blocks). `http://threadingbuildingblocks.org/`.

[JcM06] Jiang Ji-chen and Gao Ming. Enterprise service bus and an open source implementation. In *Management Science and Engineering, 2006. ICMSE '06. 2006 International Conference on*, pages 926–930, 2006.

[JL04] H.T. Jensen and J.R. Leth. Automatic job resubmission in the nordugrid middleware. Technical report, Citeseer, 2004.

[Joh97] Mark Stuart Johnstone. *Non-compacting memory allocation and real-time garbage collection.* PhD thesis, 1997. AAI9824978.

[Jub07] Sylvain Jubertie. *Modèles et outils pour le déploiement d'applications de Réalité Virtuelle sur des architectures distribuées.* These, Université d'Orléans, December 2007.

[JW98] Mark S. Johnstone and Paul R. Wilson. The memory fragmentation problem : solved ? In *Proceedings of the 1st international symposium on Memory management*, ISMM '98, pages 26–36, New York, NY, USA, 1998. ACM.

[KMK08] M. Kashyian, S.L. Mirtaheri, and E.M. Khaneghah. Portable inter process communication programming. In *Advanced Engineering Computing and Applications in Sciences, 2008. ADVCOMP '08. The Second International Conference on*, pages 181 –186, 29 2008-oct. 4 2008.

[KMS08] Ehsan Mousavi Khaneghah, Seyedeh Leili Mirtaheri, and Mohsen Sharifi. Evaluating the effect of inter process communication efficiency on high performance distributed scientific computing. In *Proceedings of the 2008 IEEE/IFIP International Conference on Embedded and Ubiquitous Computing - Volume 01*, EUC '08, pages 366–372, Washington, DC, USA, 2008. IEEE Computer Society.

[KSZ05] F. Karabiber, A. Sertbas, and A.H. Zaim. Dynamic memory allocator algorithms simulation and performance analysis. *Journal of Electrical Electronics Engineering (IU – JEEE)*, 5(2) :1435–1441, 2005.

[LB10] Stephen Lewin-Berlin. Exploiting multicore systems with cilk. In *Proceedings of the 4th International Workshop on Parallel and Symbolic Computation*, PASCO '10, pages 18–19, New York, NY, USA, 2010. ACM.

[Li10] Baoan Li. Research on soa and component oriented technology in development of large system. In *Proceedings of the 2010 International Symposium on Computational Intelligence and Design - Volume 01*, ISCID '10, pages 20–23, Washington, DC, USA, 2010. IEEE Computer Society.

[Lin99] David Linthicum. *Enterprise Application Integration.* Addison Wesley, 1999.

[LK05] Nhien An LE KHAC. *Définition et évaluation d'INUKTITUT : un interface pour l'environnement de programmation parallèle asynchrone Athapascan.* These, Institut National Polytechnique de Grenoble - INPG, March 2005.

[LLKJ10] Ming Liu, Zhonghai Lu, Wolfgang Kuehn, and Axel Jantsch. Inter-process communication using pipes in fpga-based adaptive computing. *VLSI, IEEE Computer Society Annual Symposium on*, 0 :80–85, 2010.

[Lu09] Wei Lu. *Exploiting multi-core processors for the service oriented architecture paradigm : parallel xml processing and concurrent service orchestration.* PhD thesis, Indianapolis, IN, USA, 2009. AAI3344767.

[Man01] Bernard Manouvrier. *Intégration des applications d'entreprise.* HERMES, 2001.

[Mar07a] Maxime Martinasso. *Analyse et modélisation des communications concurrentes dans les réseaux haute performance.* These, Université Joseph-Fourier - Grenoble I, May 2007.

[Mar07b] Maxime Martinasso. *Analysis and modelling of concurrent communications over high performance networks.* These, Université Joseph-Fourier - Grenoble I, May 2007. URL : `http://tel.archives-ouvertes.fr/tel-00165164`.

[MBFM11] R. Matias, T. Borges Ferreira, and A. Macedo. An experimental study on user-level memory allocators in middleware applications. In *Systems, Man, and Cybernetics (SMC), 2011 IEEE International Conference on*, pages 2431–2436, 2011.

[Mor11] Stéphanie Moreaud. *Mouvement de données et placement des tâches pour les communications haute performance sur machines hiérarchiques.* These, Université Sciences et Technologies - Bordeaux I, October 2011.

[MvNV+99] Jason Maassen, Rob van Nieuwpoort, Ronald Veldema, Henri E. Bal, and Aske Plaat. An efficient implementation of java's remote method invocation. In *Proceedings of the seventh ACM SIGPLAN symposium on Principles and practice of parallel programming*, PPoPP '99, pages 173–182, New York, NY, USA, 1999. ACM.

[Nam01] Raymond Namyst. *Contribution à la conception de supports exécutifs multithreads performants.* Habilitation à diriger des recherches, December 2001.

[Nee96] Michael Shannon Neely. An analysis of the effects of memory allocation policy on storage fragmentation, 1996.

[NMH+02] Rob V. Van Nieuwpoort, Jason Maassen, Rutger Hofman, Thilo Kielmann, and Henri E. Bal. Ibis : an efficient java-based grid programming environment. In *in Joint ACM Java Grande - ISCOPE 2002 Conference*, pages 18–27, 2002.

[PKC97] Scott Pakin, Vijay Karamcheti, and Andrew A. Chien. Fast messages : Efficient, portable communication for workstation clusters and mpps. *IEEE Parallel Distrib. Technol.*, 5(2) :60–73, April 1997.

[PT98] Loic Prylli and Bernard Tourancheau. Bip : a new protocol designed for high performance networking on myrinet. In *In Workshop PC-NOW, IPPS/SPDP98*, pages 472–485. Springer-Verlag, 1998.

[RD01] Eric Renault and Pierre David. Performance and analysis of the pci-ddc remote-write implementation. In *CLUSTER*, pages 197–203, 2001.

[RDF00] Eric Renault, Pierre David, and Paul Feautrier. Pci-ddc application programming interface : Performance in user-level messaging (research note). In *Euro-Par*, pages 1201–1205, 2000.

[RKLC08] M. Ramakrishna, Jisung Kim, Woohyong Lee, and Youngki Chung. Smart dynamic memory allocator for embedded systems. In *Computer and Information Sciences, 2008. ISCIS '08. 23rd International Symposium on*, pages 1–6, 2008.

[Ron03] Marc Ronell. A c++ pooled, shared memory allocator for the standard template library. pages 247–256. Springer-Verlag, 2003.

[RYA+12] Xiaojun Ruan, Qing Yang, Mohammed I. Alghamdi, Shu Yin, and Xiao Qin. Esmpich2 : A message passing interface with enhanced security. *IEEE Transactions on Dependable and Secure Computing*, 9 :361–374, 2012.

[SAKan] Z. Siddiqui, A.H. Abdullah, and M.K. Khan. Qualified analysis b/w esb(s) using analytical hierarchy process (ahp) method. In *Intelligent Systems, Modelling and Simulation (ISMS), 2011 Second International Conference*, pages 100–104, Jan.

[Sof] Progress Software. Sonic esb. `http://www.progress.com/en/sonic/sonic-esb.html`.

[Sun90] V. S. Sunderam. Pvm : a framework for parallel distributed computing. *Concurrency : Pract. Exper.*, 2(4) :315–339, November 1990.

[TGS01] G. Torralba, V. Gonzalez, and E. Sanchis. Sci evaluation in multinode environments for computing and data-processing applications. *Nuclear Science, IEEE Transactions on*, 48(4) :1306–1312, 2001.

[Thi07] Samuel Thibault. *Ordonnancement de processus légers sur architectures multiprocesseurs hiérarchiques : BubbleSched, une approche exploitant la structure du parallélisme des applications.* These, Université Sciences et Technologies - Bordeaux I, December 2007.

[TM03] S. Tham and J. Morris. Cilk vs mpi : comparing two very different parallel programming styles. In *Parallel Processing, 2003. Proceedings. 2003 International Conference on*, pages 143–152, 2003.

[Vas09] Panos Vassiliadis. A survey of extract-transform-load technology. *IJDWM*, 5(3) :1–27, 2009.

[vEBBV95] T. von Eicken, A. Basu, V. Buch, and W. Vogels. U-net : a user-level network interface for parallel and distributed computing (includes url). *SIGOPS Oper. Syst. Rev.*, 29(5) :40–53, December 1995.

[vECGS98] Thorsten von Eicken, David E. Culler, Seth Copen Goldstein, and Klaus Erik Schauser. Active messages : a mechanism for integrating communication and computation. In *25 years of the international symposia on Computer architecture (selected papers)*, ISCA '98, pages 430–440, New York, NY, USA, 1998. ACM.

[vEV98] T. von Eicken and W. Vogels. Evolution of the virtual interface architecture. *Computer*, 31(11) :61–68, 1998.

[VS10] Shantanu Sharma Vivek Sharma, Manish Varshney. *Design and Implementation of Operating System*. Laxmi, 2010.

[WJNB95a] Paul R. Wilson, Mark S. Johnstone, Michael Neely, and David Boles. Dynamic storage allocation : A survey and critical review. pages 1–116. Springer-Verlag, 1995.

[WJNB95b] Paul R. Wilson, Mark S. Johnstone, Michael Neely, and David Boles. Dynamic storage allocation : A survey and critical review. In *Proceedings of the International Workshop on Memory Management*, IWMM '95, pages 1–116, London, UK, UK, 1995. Springer-Verlag.

[WYZQ09] Jun Wang, AiRong Yu, XiaoYi Zhang, and Lei Qu. A dynamic data integration model based on soa. In *Computing, Communication, Control, and Management, 2009. CCCM 2009. ISECS International Colloquium on*, volume 2, pages 196–199, 2009.

[YCD+09] Jianwei Yin, Hanwei Chen, Shuiguang Deng, Zhaohui Wu, and Calton Pu. A dependable esb framework for service integration. *IEEE Internet Computing*, 13 :26–34, 2009.

[YS10] Divakar Yadav and A. K. Sharma. Tertiary buddy system for efficient dynamic memory allocation. In *Proceedings of the 9th WSEAS international conference on Software engineering, parallel and distributed systems*, SEPADS'10, pages 61–66, Stevens Point, Wisconsin, USA, 2010. World Scientific and Engineering Academy and Society (WSEAS).

[Zha10] Yang Zhang. Dependable esb routing in hybrid service execution environment. *AISS*, 2(1) :83–93, 2010.

[Zid10] Meriem Zidouni. *Modeling and performance analysis of the MPI library by taking into account the hardware architecture*. These, Université Joseph-Fourier - Grenoble I, May 2010.

[ZSM11] Liyan Zhang, Yan Sun, and Jian Ma. A parallel crout algorithm based on tbb. In *Software Engineering and Service Science (ICSESS), 2011 IEEE 2nd International Conference on*, pages 239–242, 2011.

le cnam

Ridha BENOSMAN

Conception et Evaluation de Performance d'un Bus Applicatif, Massivement Parallèle et Orienté Services

le cnam

Résumé :
Enterprise Service Bus (ESB) est actuellement l'approche la plus prometteuse pour l'implémentation d'une architecture orientée services (SOA : Service-Oriented Architecture) par l'intégration des différentes applications isolées dans une plateforme décentralisée.
De nombreuses solutions d'intégration à base d'ESB on été proposées, elles sont soit open-source comme : Mule, Petals, ou encore Fuse, soit propriétaires tels que : Sonic ESB, IBM WebSphere Message Broker, ou Oracle ESB. Cependant, il n'en existe aucune en mesure de traiter, à la fois des aspects : d'intégration et de traitement massivement parallèle, du moins à notre connaissance. L'intégration du parallélisme dans le traitement est un moyen de tirer profit des technologies multicœurs/multiprocesseurs qui améliorent considérablement les performances des ESBs.
Toutefois, cette intégration est une démarche complexe et soulève des problèmes à plusieurs niveaux : communication, synchronisation, partage de données, etc.
Dans cette thèse, nous présentons l'étude d'une nouvelle architecture massivement parallèle de type ESB.
Mots clés :
Bus de services d'entreprise (Enterprise Service Bus : ESB) ; Architecture orientée services ; Traitement massivement parallèle ; Multithreading ; Systèmes distribués ; Communication inter-processus (IPC).

Abstract :
Enterprise service bus (ESB) is currently the most promising approach for business application integration in distributed and heterogeneous environments. It allows to deploy a service-oriented architecture (SOA) by the integration of all the isolated applications on a decentralized platform.
Several commercial or open source ESB-based solutions have been proposed. However, to the best of our knowledge, none of these solutions has integrated the parallel processing. The integration of parallelism in the treatment allows to take advantage of the multicore/multiprocessor technologies and thus can improve greatly the ESB performance. However, this integration is difficult to achieve, and poses problems at multiple levels (communication, synchronization, etc). In this study, we present a new massively parallel ESB architecture that meets this challenge.
Keywords :
Enterprise Service Bus (ESB) ; Service-oriented architecture (SOA) ; Massively parallel processing ; Multithreading ; Distributed and delay insensitive applications ; Inter-process communication (IPC).