Laboratoire d'Informatique Fondamentale de Lille

Envrionnement fonctionnel distribué et dynamique pour systèmes embarqués

THÈSE

présentée et soutenue publiquement le 05 Décembre 2003

pour l'obtention du

Doctorat de l'Université des Sciences et Technologies de Lille
(spécialité informatique)

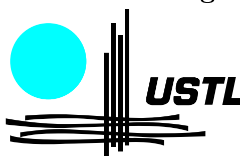
par

Abdelkader Amar

Composition du jury

Président :	Laurence DUCHIEN	Université de Lille 1
Rapporteurs :	Frédéric PETROT	Université de Paris VI
	Hervé GUYENNET	Université de Franche Comté
Examinateurs :	Marc DURANTON	Philips Research
	Henri BASSON	ULCO de Calais
	Pierre BOULET	Université de Lille 1
Directeur :	Jean-Luc DEKEYSER	Université de Lille 1

Université des Sciences et Technologies de Lille - Lille 1



À mes parents pour tout leur amour et leur soutien
je leur dois beaucoup

À Katarzyna pour tout

Remerciements

Je tiens à remercier toutes les personnes qui ont rendu cette thèse possible, par leur soutien, leur aide et leurs contributions.

J'aimerais remercier en particulier :

Laurence Duchien, Professeur à l'Université des Sciences et Technologies de Lille, pour m'avoir fait l'honneur de présider ce jury.

Frédéric Pétrot, Maître de conférence à l'Université de Paris IV et Hervé Guyennet, Professeur à l'Université de Franche-Comté pour avoir accepté de juger mon travail de thèse.

Jean-Marc Duranton et Henri Basson pour avoir accepté de faire partie de mon jury de thèse.

Jean-Luc Dekeyser et Pierre Boulet qui m'ont accueilli au sein de l'équipe *West* lors de mon arrivée à Lille. Ils ont su diriger mes recherches et je les remercie pour leurs nombreux conseils qui m'ont permis de faire évoluer ce travail et ce document.

Julien Soula [1] pour ces nombreuses aides techniques et son aide autour d'Array-OL. Je le remercie également pour son extrême gentilesse.

Les anciens thesards de *West*, Florent Devin et Emmanuel Cagniot [2] pour leur sympathie et leur bonne humeur.

Benoît Planquel pour m'avoir supporté lors de mes nombreux passages au bureau 302.

Un merci particulier pour la personne sans laquelle je n'aurais pas pu finir ma thèse dans les délais, Katarzyna [3]. Je la remercie de m'avoir supporté et soutenu tout au long de cette thèse, et surtout lors de la rédaction.

Ma famille et plus particulièrement mes parents et mon oncle qui m'ont toujours aidé au cours de mes études.

Je voudrais aussi dire merci et souhaiter bon courage pour leur « fin de thèse », à Javed Dulloo, Aranud Cucurru, Philippe Dumont, Chadi Aljundi, Alexandre Hesseman et Mickael Samyn. Je souhaite bon courage pour les nouveaux thésards de l'équipe *West* : Ouassila Labbani, Lossan Bonde, Joel Vennin et Ashish Meena.

Je n'oublie pas non plus Philippe Marquet, Jean-Luc Levaire [4] pour sa sympathie et son empreinte particulière dans le bureau, Cédric Dumoulin, Samy Meftali et Stéphane Akhoun.

[1] la mascotte de *West*

[2] Dits le « le méchant bleu clair » et le « méchant bleu foncé » respectivement

[3] Dite Kasia

[4] Dit *lulu le zen*

Je remercie également Jaymz qui m'a tenu compagnie pendant les longues heures de développement et qui m'a souvent aidé à décompresser et Loreena, Sarah et Wolfgang qui m'ont tenu compagnie pendant les heures de rédaction.

Merci enfin à tous ceux que je n'aurais pas dû oublier ...

Abdelkader [5]

[5]Dit aussi *Kadirou*, *Caliméro*, *Le Chacal*

Table des matières

Introduction **1**

Chapitre 1
Systèmes Répartis :
Architectures et Modèles

1.1 Architecture des environnements répartis 10
1.1.1 Middleware 11
1.1.2 Modèle objets distribués 13
1.2 Approche objet/composant 14
1.2.1 CORBA 14
1.2.2 Java Beans, Java-RMI et Enterprise Java Beans 18
1.2.3 Distributed COM 21
1.2.4 Comparaison entre Java-RMI, CORBA et DCOM 23
1.2.5 Autres systèmes d'objets répartis 25
1.2.6 Méta-modèles 27
1.3 Approche Grid 31
1.3.1 Legion 33
1.3.2 Globus 34
1.3.3 Harness 35
1.3.4 XtremWeb 35
1.3.5 Grid versus Middlewares objet/composant 36
1.4 Conclusions 37

Chapitre 2
Réseaux de Processus

2.1 Réseaux de Processus de Kahn 40

2.1.1 Représentation mathématique formelle des réseaux de processus de Kahn . . . 41
2.1.2 Déterminisme . . . 42
2.2 Ordonnancement des réseaux de processus de Kahn . . . 43
2.2.1 Ordonnancement demand-driven . . . 44
2.2.2 Ordonnancement data-driven . . . 44
2.2.3 Ordonnancement avec FIFOs bornées . . . 45
2.2.4 Proposition d'un ordonnancement data-driven non borné . . . 46
2.3 Environnements et implémentations des réseaux de processus de Kahn . . . 47
2.3.1 YAPI . . . 47
2.3.2 Ptolemy . . . 50
2.4 Réseaux de processus à flux de données . . . 52
2.4.1 Le modèle de Karp et Miller . . . 53
2.4.2 Flux de données synchrones . . . 54
2.4.3 Flux de données booléens . . . 55
2.5 Conclusion . . . 56

Chapitre 3
Réseaux de Processus de Kahn Distribués

3.1 Réseaux de processus de Kahn distribués . . . 60
3.1.1 Représentation mathématique . . . 61
3.1.2 Conception . . . 62
3.1.3 Ordonnancement des processus . . . 64
3.2 FIFOs distribuées . . . 65
3.2.1 Déploiement et connection . . . 67
3.3 Protocole de Transfert . . . 68
3.3.1 Demand driven . . . 68
3.3.2 Data driven . . . 68
3.3.3 Hybride . . . 69
3.3.4 Déclenchement des communications et équilibrage de charge . . . 69
3.4 Analyse des modes de transfert . . . 71
3.4.1 Vectorisation des communications . . . 73
3.4.2 Mesures de performances . . . 78
3.5 Réseaux de processus, composant et génération automatique de code . . . 84
3.5.1 Architecture du système . . . 85

3.5.2 Structure d'un composant 85
3.5.3 Référentiel d'objets 86
3.5.4 Étapes de la génération automatique du code 86
3.5.5 Description des composants 87
3.6 Exemple d'application : Décodeur JPEG distribué 87
3.6.1 Génération de code 88
3.7 Conclusion 90

Chapitre 4
Dynamicité

4.1 Introduction 93
4.2 Types de dynamicité 94
4.3 Architecture du système 95
4.3.1 Points de reprise 95
4.3.2 Restrictions 96
4.3.3 Console interactive 97
4.4 Développement incrémental de l'application répartie 99
4.4.1 Création et ajout d'un processus 99
4.4.2 Ajout d'une liaison 99
4.5 Migration et remplacement de composant 99
4.6 Reconfiguration d'un processus 100
4.6.1 Suspension/Reprise de l'exécution 100
4.6.2 Suppression d'un processus 102
4.6.3 Suppression/Modification d'une liaison 102
4.6.4 Changement des paramètres 102
4.7 Exemple : Application de traitement de signal 103
4.7.1 Description de l'application 103
4.7.2 Mesures de performance 104
4.8 Conclusion 107

Chapitre 5
ARRAY-OL **sur réseaux de processus distribués**

5.1 Introduction 109
5.2 Langage ARRAY-OL 110
5.2.1 Formalisme matriciel d'ARRAY-OL 113

5.2.2 Contraintes d'Array-OL 114
5.2.3 Transformations de code Array-OL 115
5.3 Array-OL et réseaux de processus 117
5.3.1 Déterminisme et data-parallélisme dans Array-OL 117
5.3.2 Modèle d'exécution 117
5.3.3 D'une dépendance de données à un flux de tableaux 118
5.4 Évaluation 122
5.5 Conclusion 124

Chapitre 6
Conclusions et perspectives

6.1 Bilan 125
6.2 Perspectives 126
6.2.1 Continuation des travaux entrepris 127
6.2.2 Nouveaux thèmes de recherche 128

Bibliographie **129**

Table des figures

1 Thèmes abordés dans la thèse 6

1.1 Système réparti et middleware 12
1.2 Le modèle d'objets distribués 13
1.3 Architecture de l'OMA 15
1.4 Structure d'un composant CORBA Component Model 18
1.5 Le modèle *Enterprise JavaBean* 20
1.6 Architecture de DCOM 22
1.7 Un objet réparti partagé dans Globe 26
1.8 Le modèle objet réparti partagé de Globe 27
1.9 Architecture des composants dans RM-ODP 28
1.10 Transformation de modèles dans le MDA 30
1.11 Architecture du Grid 32

2.1 Représentation graphique d'un réseau de processus 41
2.2 Transformation d'un graphe non borné en un graphe borné 46
2.3 Les processus producteur/consommateur en YAPI 49
2.4 Le réseau de processus contenant les processus producteur/consommateur en YAPI 50
2.5 Réseau de processus hiérarchique en YAPI 51
2.6 L'environnement de simulation de Ptolemy : Vergil 52
2.7 Flux de données synchrones 54
2.8 Réseaux de processus synchrones inconsistent 55
2.9 Flux de données booléen 56

3.1 Exemple d'un réseau de processus distribué 62
3.2 Diagramme de Classe 63
3.3 Structure d'une FIFO locale 66
3.4 Structure des FIFOs distribuées 67
3.5 Les opérations read et pre-read dans les FIFOs distribuées 70
3.6 Les opérations write et post-write dans les FIFOs distribuées 70
3.7 Diagramme d'état de la FIFO de sortie 72
3.8 Diagramme d'état de la FIFO d'entrée 73
3.9 Temps de transfert de sequences de short 75
3.10 Temps de transfert de sequences de double 76

3.11 Gestion séparée des communications . . . 77
3.12 Sans vectorisation/sans notification . . . 79
3.13 Sans vectorisation/avec notification . . . 80
3.14 Avec vectorisation/sans notification . . . 80
3.15 Avec vectorisation/avec notification . . . 81
3.16 Sans vectorisation/sans notification . . . 81
3.17 Sans vectorisation/avec notification . . . 82
3.18 Avec vectorisation/avec notification . . . 83
3.19 Avec vectorisation/sans notification . . . 83
3.20 Architecture du système de génération de code . . . 85
3.21 DTD du format de description du référentiel d'objets . . . 86
3.22 Représentation XML d'un objet de traitement de signal (VBL : Veille Large Bande) . . . 86
3.23 DTD du format de description des composants . . . 88
3.24 Structure du décodeur JPEG . . . 89
3.25 Programme principal du décodeur JPEG . . . 90
3.26 JFIF distribué (programme principal) . . . 90
3.27 Décodeur JPEG distribué . . . 91
3.28 Description du composant FrontEnd . . . 91

4.1 Exemple d'un réseau de processus sans point de reprise . . . 97
4.2 Diagramme de classe de la Console . . . 98
4.3 Scénario de remplacement (Étape 1) . . . 100
4.4 Scénario de remplacement (Étape 2) . . . 101
4.5 Scénario de remplacement (Étape 3) . . . 101
4.6 Description de l'application . . . 103
4.7 Distribution de l'application VBL . . . 104
4.8 Performance de l'exécution distribuée - charge des trois tâches . . . 105
4.9 Évolution de la longueur des demi-FIFOs durant la redistribution . . . 106

5.1 Le modèle global d'Array-OL . . . 111
5.2 L'opération d'ajustage dans Array-OL . . . 112
5.3 L'opération de pavage dans Array-OL . . . 113
5.4 Exemple d'une tâche Array-OL . . . 113
5.5 Exemple de recouvrement : les motifs opérandes de taille (1×3) se recouvrent de deux points suivant la dimension *Dim1* . . . 115
5.6 Application originale . . . 116
5.7 Fusion des trois tâches en une tâche hiérarchique . . . 116
5.8 Application Array-OL et flux de motifs . . . 119
5.9 Phénomène du corner turn . . . 119
5.10 Structure des motifs produits et consommés . . . 120
5.11 Tâche Array-OL avec du recouvrement et sa transformation en tâche hiérarchique et en flux de tableaux . . . 121
5.12 Tâche Array-OL avec du « corner turn » et sa transformation en tâche hiérarchique et en flux de tableaux . . . 121

5.13 Description de l'application Array-OL par un réseau de processus (dont un data-parallèle) 122
5.14 Performances de la parallélisation d'un processus de la VBL sur un quadri-xéons. 123

Introduction

Contexte et présentation de la thèse

L'avènement d'Internet et le développement des technologies réseaux ont, non seulement démocratisé l'informatique, mais ont ouvert la voie à de nouveaux types d'applications que l'on appelle « les applications réparties ». Celles-ci partagent des ressources se trouvant sur des sites différents. Les gros calculateurs parallèles ont été donc confrontés à la concurrence de systèmes distribués offrant des puissances de calcul plus grandes et surtout à moindre coût.

De plus, le développement des technologies réseaux, associé à la montée en puissance des ordinateurs, a engendré :

- une complexité grandissante des systèmes informatiques ;
- une évolutivité du logiciel qui est devenu un critère essentiel pour assurer sa réussite ;

Pour gérer l'accroissement de la complexité des systèmes informatiques, différents modèles ont été définis afin d'uniformiser le développement de telles applications réparties. L'expérience a montré que les développements doivent s'appuyer sur la modularité, la réutilisabilité et l'extensibilité. La technique objet qui présente les notions d'encapsulation et d'héritage qui permettent de construire un code respectant les trois caractéristiques citées, a ainsi émergé parmi les autres techniques. Elle a par la suite donné naissance à un concept de granularité plus grande, celui du composant. Un composant permet d'encapsuler plusieurs objets pour offrir des services et d'intégrer du code non-fonctionnel tel que la communication avec les autres composants ou la politique de sécurité. Ainsi les composants sont destinés à faciliter le développement des applications réparties, en mettant la notion de modularité à un niveau plus haut et en intégrant de façon plus concrète le mécanisme de composition.

Quand à l'évolutivité du logiciel, elle a imposé (et construit la réussite) des technologies non propriétaires, telles CORBA, Java ou DCOM, qui sont devenues des standards. Elle a aussi montré la nécessité de construire des passerelles entre les systèmes existants.

Parallèlement à ces bouleversements, les besoins en évolutivité des applications réparties ne concernent plus seulement le développement de celles-ci, mais aussi la nécessité d'évoluer pendant l'exécution. C'est le cas par exemple des applications critiques ou des applications qui requièrent une qualité de service et qui s'exécutent dans un environnement dont on ne maîtrise pas tous les paramètres. Ces changements peuvent être dûs à diverses raisons : évolution matérielle, évolution logicielle, changement des technologies de programmation, etc. Ainsi, les systèmes répartis doivent être évolutifs, ouverts et dynamiques. La différence entre un système ouvert et un système dynamique, est que le

changement apporté au système se fait pendant l'exécution dans un système dynamique et à l'arrêt dans les systèmes ouverts. Des d'applications réparties très variées ont des besoins de dynamicité, parmi lesquelles :

- les applications de calcul scientifique nécessitant des puissances et des temps de calcul très grands. Ces applications sont généralement confrontées à une disponibilité variable des ressources de calcul et à des problèmes d'équilibrage de charge et d'efficacité. Une reconfiguration dynamique par remplacement ou migration d'une partie de l'application, peut résoudre certains de ces problèmes.
- les applications de simulation telles les applications de traitement de signal auxquelles nous nous intéresserons dans nos travaux, qui sont amenées parfois à s'exécuter indéfiniment. Une exécution infinie est toujours sujette à des pannes ou à des variations de charge et disposer d'un moyen de remplacement à l'exécution autoriserait l'application à continuer l'exécution sans s'interrompre.
- les applications du Web où on ne maîtrise pas tous les paramètres d'exécution (trafic réseau, charge des service, etc.) et pour lesquelles une certaine qualité de service est requise en continu.

En plus de ce qui vient d'être évoqué, un point crucial est à noter dans le développement des systèmes informatiques : la conception et la réalisation ne se font plus entièrement par une seule partie, sauf dans certains domaines critiques comme la défense ou le transport, où une vérification de l'ensemble du code est obligatoire. Dans le domaine en pleine expansion des « Systèmes embarqués », la réalisation de SoC (pour « System-On-Chip ») est souvent une opération coûteuse et complexe, car elle nécessite l'utilisation de différents composants matériels vendus par différents fournisseurs. Pour réduire le coût et la complexité de la mise en œuvre, les industriels sont souvent amenés à faire des simulations avant de finaliser leurs choix. Dans un tel contexte de simulation, les contraintes de propriété industrielle et de localité doivent être gérées pour protéger la propriété intellectuelle (ou *IP*) des composants et des applications : le fournisseur de composant ne doit pas avoir accès au code de l'application simulée et réciproquement, le développeur d'application ne doit pas avoir accès au code du simulateur. Cet ensemble de simulateurs contenant des « composants virtuels » (ou *VC*), couplés via le réseau Internet, permet d'obtenir une « cyber-entreprise » [BDD+03], où différents services sont disponibles tout en respectant les propriétés industrielles de chacun, que ce soient les fournisseurs des composants ou les développeurs des applications. La réalisation d'une telle cyber-entreprise pose des problèmes d'interopérabilité entre les simulateurs et les applications. L'hétérogénéité matérielle des environnements d'exécution et des langages de programmation, ainsi que la définition des interfaces des différents composants, doivent être gérées.

Les applications de traitement de signal sont elles, spécifiées généralement sur des modèles à base de graphes, ou sur des langages spécifiques. Le modèle le plus répandu est le modèle des réseaux de processus, qui fût proposé par Kahn et MacQueen [KM77]. Ce modèle de calcul, dans lequel les processus communiquent par des files d'attente de type *first-in, first-out* ou *FIFO*, est bien adapté pour modéliser les architecture matérielles auxquelles on s'intéresse dans un environnement de simulation de type cyber-entreprise. C'est aussi un modèle de choix pour décrire les applications de traitement de signal et les applications à flux de données en général (telles les applications audio/vidéo). En effet, dans ces applications, des flux de données (éventuellement infinis), sont traités par

plusieurs processus. Le modèle exprime explicitement l'aspect calcul et communication et permet une construction d'application par composition. Il fournit ainsi un excellent modèle pour la construction d'applications réparties dynamiques et plus particulièrement d'applications dynamiques à flots de données dans un environnement réparti.

Contribution

Cette thèse a pour objectif de définir et de concevoir un support d'exécution pour des applications réparties et pour la simulation d'applications de traitement de signal écrites en ARRAY-OL [DLB+95] en particulier. Dans un contexte de cyber-entreprise [6], ce support est basé sur l'interconnexion de composants multithreadés par des files d'attente. Une proposition du modèle des réseaux de processus distribués est définie et implémentée en utilisant l'architecture CORBA [Obj01]. On s'est intéressé par la suite aux aspects de la dynamicité que peut procurer un tel support.

Ainsi l'originalité de notre approche peut être résumée dans les points suivants :

1. Décharger le programmeur de la gestion des communications entre les processus, qui sont complètement transparentes. Ainsi le programmeur ne s'occupe que de la partie traitement qu'effectue le composant.
2. Définir un protocole de communication bidirectionnel, qui permet de tenir compte des contraintes de charge mémoire et de disponibilité des données. L'utilisation de seuils sur le nombre d'éléments dans les files d'attente permet de mieux gérer l'espace mémoire utilisé et d'augmenter la disponibilité des données.
3. Autoriser une évolution dynamique du réseau de processus au cours de l'exécution, en offrant des mécanismes de remplacement, de migration et de reconfiguration des processus.
4. Générer automatiquement des composants logiciels qui forment le réseau à partir d'une description basée sur le langage XML [Xml]. Ces composants générés peuvent par la suite être intégrés automatiquement à une application existante.
5. Prouver l'adéquation de ce support d'exécution distribué pour des applications ARRAY-OL, par construction d'un modèle d'exécution particulier et par sa mise en œuvre.

Justificatif de notre démarche

Sur chaque point énoncé ci-dessous, nous allons expliquer notre démarche et la justifier.

Architecture du système

Dans le domaine des systèmes répartis, les deux approches, *middleware objet* et grid, sont certainement les meilleures candidates pour la construction de supports d'exécution.

[6]Dans le cadre du projet ITEA 99038 Sophocles

Pour l'architecture de notre système, nous avons préféré suivre une approche *middleware objet*. Les raisons d'un tel choix sont les suivantes :
- les systèmes de grid présentent l'inconvénient d'être peu flexibles, et de ce fait obligent les applications à s'adapter à un environnement figé ;
- la réutilisabilité, la modularité et la flexibilité doivent être favorisés par rapport des performances. Dans un contexte de simulation distribuée, les contraintes de performance sont moins fortes, néanmoins nous avons optimisé les coûts des communications tout en restant à un niveau de programmation élevé ;
- l'hétérogénéité des ressources et l'évolution de leur disponibilité dans un environnement répartis doivent être gérées.

Parmi les trois standards dans les « middlewares objets » que sont CORBA, DCOM et Java, notre choix s'est porté sur l'architecture CORBA [Obj01] (*Common Object Request Broker Architecture*). Ce choix a été motivé par les avantages que présente cet environnement :
- environnement dont les spécifications sont standardisées (par l'OMG[7]) ;
- affranchissement des solutions purement propriétaires ;
- plate-forme multi-systèmes et multi-langages ;
- bon support pour l'intégration de logiciels existants ;
- conception modulaire ;
- architecture logicielle claire et simple.

Modèle d'exécution

Le modèle d'exécution des réseaux de processus de Kahn est bien adapté pour modéliser et exécuter les applications de simulation de systèmes embarqués. En plus, il permet de résoudre le problème de la définition des interfaces des composants rencontré dans le cadre d'une cyber-entreprise. En effet, les interactions entre les différents processus se font par l'intermédiaire de files d'attente en local et de demi files d'attente dans un environnement de simulation distribué.

Les notions d'assemblage et de modularité qu'on rencontre dans le domaine des composants logiciels s'appliquent avec succès au modèle. Nous allons montrer que, conjointement au modèle objet/composant, le modèle des réseaux de processus de Kahn est aussi un modèle d'exécution efficace pour les applications réparties.

Évolutivité

L'architecture des applications réparties est faite d'un ensemble de composants logiciels coopérants. Ces composants logiciels ainsi que les ressources matérielles sur lesquelles ils s'exécutent sont physiquement séparés, et de ce fait, l'application est amenée durant son cycle de vie à évoluer pour diverses raisons : modification de l'environnement d'exécution, évolution du matériel, amélioration du code des composants, etc. Nos objectifs sont d'assurer cette évolution durant l'exécution de l'application, de manière complètement transparente au programmeur. Notre approche se base sur :

[7]Object Management group

- les points de reprise au sein du code du programme pour autoriser la migration ou le remplacement d'un composant pendant l'exécution sans devoir l'arrêter. Dans notre système, les points de suspension/reprises peuvent être définis par le programmeur et puis générés automatiquement par un préprocesseur de compilation, procurant ainsi une plus grande facilité de programmation ;
- le contrôle des composants de l'application se fait par l'intermédiaire d'une console interactive. L'utilisation d'une console interactive présente deux avantages principaux :
  - autoriser le développement de l'application distribuée de façon incrémentale, ce qui implique qu'elle peut être conçue en plusieurs parties indépendantes ;
  - autoriser la reconfiguration des composants à l'exécution.

La génération automatique de composants est basée sur le langage XML. Ce langage est devenu le langage standard pour l'échange des données entre les systèmes distribués, indépendamment des plate-formes d'exécution. Notre approche se base sur la composition d'objets et le fait que pour développer une application distribuée, seule la partie traitement et la description des liens entre les composants sont nécessaires pour générer automatiquement toute la gestion des communications. Notre modèle permet ainsi de générer des composants à partir de la description de leur fonction de calcul, des interfaces implémentées et exportées, et des interfaces importées d'autres composants. Ceci libère le développeur de toute la gestion de bas niveau des flux de données.

Projection vers ARRAY-OL

Le langage Array-OL [DLB+95] (*Array Oriented Language*) est un langage orienté traitement de signal (TS). Il permet de spécifier l'algorithme de calcul et les dépendances de données sans se soucier de l'ordonnancement de l'application. La compilation de ce langage et les besoins de simulation visent des architectures hétérogènes qui vont des stations Unix à des systèmes répartis, en passant par des systèmes multiprocesseurs embarqués dédié à Array-OL. Notre d'approche qui associe le modèle Array-OL et le modèle des réseaux de processus distribués, permet d'obtenir un support dédié pour la simulation d'applications de traitement de signal sur des composants distribués et de répondre aux besoins de simulation hétérogène. Des études précédentes [SMDD01, Sou01] ont permis de définir des transformations de code Array-OL en exploitant le parallélisme de données seulement. Notre support permet d'exploiter le parallélisme de tâches et le parallélisme de données qu'offrent respectivement le modèle des réseaux de processus de Kahn et le modèle Array-OL, permettant ainsi de mieux tirer profit des capacités de calcul des ressources disponibles.

Plan du manuscrit

Les cinq thèmes principaux abordés dans la thèse sont : les systèmes répartis, le modèle des réseaux de processus de Kahn, les systèmes dynamiques, le traitement de signal intensif

et les systèmes embarqués. La figure 1 montre l'enchaînement dans le traitement des différents thèmes et les relations entre eux.

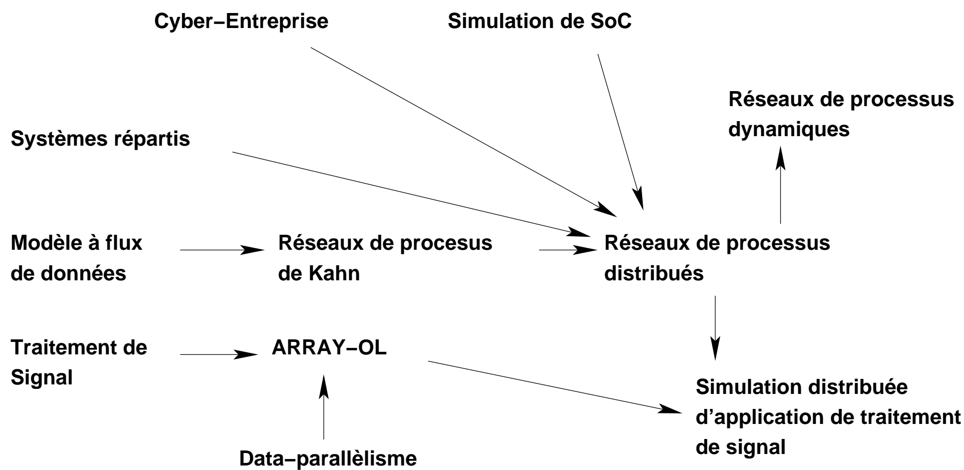


FIG. 1 – Thèmes abordés dans la thèse

Les paragraphes suivants présentent un peu plus en détails l'organisation du document.

Chapitre 1 : Systèmes répartis : Architecture et modèles

Ce chapitre présente une vue d'ensemble des systèmes distribués et présente les deux approches les plus utilisées pour construire de tels systèmes. Fortes de leurs succès, l'approche « Objet/Composant » et l'approche « Grid » sont présentées, ainsi que quelques systèmes de chaque approche.

Chapitre 2 : Réseaux de processus

Ce chapitre présente le modèle des réseaux de processus de Kahn et ses caractéristiques. Ensuite, l'ordonnancement du modèle et les implémentations existantes sont décrits. Des sous modèles du modèle de Kahn (Karp et Miller, réseaux synchrones, réseaux booléens) sont présentés, car ils sont bien adaptés à certaines applications de traitement de signal et peuvent être utilisés pour construire des ordonnanceurs plus performants que ceux du modèle de base.

Chapitre 3 : Réseaux de processus de Kahn distribués

Dans ce chapitre, on présente le modèle des réseaux de processus distribués que nous proposons. Ce modèle basé sur celui de Kahn, utilise la notion de demi-FIFO (ou de FIFO distribuée), et est basé sur un protocole de communication transparent et bidirectionnelle. Le protocole de communication entre les demi-FIFOs est présenté, puis comparé à certains protocoles classiques pour évaluer ses performances.

Chapitre 4 : Dynamicité

Dans ce chapitre, nous présentons l'aspect de la dynamicité appliqué au modèle des réseaux de processus distribués. La dynamicité est vue selon plusieurs aspects, qui peuvent être regroupés en trois types : changement d'implantations (migration ou remplacement), reconfiguration et dynamicité de développement (développement incrémental). L'apport de la dynamicité dans les simulations distribuées est essentiel pour permettre l'évolution matérielle et logicielle, et améliorer les performances.

Chapitre 5 : Array OL sur réseaux de processus distribués

Ce chapitre présente brièvement le langage ARRAY-OL et propose un schéma d'exécution pour les applications écrites en ce langage dans un environnement de simulation distribué. A partir du support présenté dans les chapitre 3 et 4, le langage ARRAY-OL est utilisé pour pouvoir exécuter des applications de traitement de signal distribuées dynamiques.

Conclusion

En conclusion, nous présentons le bilan de nos travaux et des résultats obtenus, et discutons des perspectives et des extensions qui nous semblent intéressantes à proposer.

1

Systèmes Répartis : Architectures et Modèles

Sommaire

1.1 **Architecture des environnements répartis** 10
1.2 **Approche objet/composant** 14
1.3 **Approche Grid** 31
1.4 **Conclusions** 37

Certaines choses s'améliorent... mais les thèmes fondamentaux devraient se révéler dans la continuité.

Stephen J. Gould.

Le développement des applications informatiques a toujours été lié au développement des technologies matérielles. La montée en puissance des capacités des microprocesseurs et des tailles mémoires a permis de développer des applications plus performantes, plus puissantes et plus ergonomiques. La fin du siècle dernier a connu le développement des technologies réseaux. Le développement a été plus rapide que celui des technologies matérielles, au point que les réseaux informatiques sont actuellement omniprésents, et l'Internet a rendu cette technologie accessible à tout le monde. Ces réseaux forment ce qu'on appelle les *systèmes distribués* et regroupent l'infrastructure matérielle et logicielle qui les composent.

Le développement des systèmes distribués a permis la création de nouveaux types d'applications qu'il n'était pas possible de concevoir en utilisant un seul système informatique. C'est le cas des applications partageant des données géographiquement distantes ou nécessitant de très grandes puissances de calcul.

Dans ce chapitre, nous présentons les architectures des systèmes distribuées et les modèles logiciels sur lesquels ils sont basés. Nous présenterons le concept de *middleware* puis nous décrirons les quatre modèles des *middlewares* tels qu'il sont présentés dans [AST02].

Les deux sections suivantes présentent les deux approches les plus utilisées pour construire de telles applications : l'approche orientée objet (ou orientée composant) et l'approche grid. Ces deux approches se basent sur deux philosophies différentes et visent chacune des types d'applications bien distincts. L'une fournit un modèle de communication et de distribution, alors que l'autre fournit les services de base pour utiliser de manière uniforme les ressources du système. L'approche orientée objet/composant vise un large type d'applications, alors que l'approche grid vise généralement les applications de calcul distribué et de calcul scientifique.

Malgré les différences qui existent entre les deux approches, ceci n'empêche pas un certain rapprochement. Certains systèmes basés sur l'approche orientée objet/composant visent les applications du grid, et réciproquement des systèmes de grid se basent sur l'approche orientée objet.

1.1 Architecture des environnements répartis

Un système informatique réparti peut être défini de plusieurs façons différentes, néanmoins, tout système réparti doit satisfaire la condition suivante : mettre en interaction des ressources matérielles et logicielles indépendantes reliées par un réseau de communication, afin d'apparaître aux utilisateurs comme un seul système. La principale motivation de la construction des systèmes distribués est le partage de ressources, des ressources qui peuvent être matérielles comme les ressources de calcul ou de stockage, ou logicielles telles des pages Web ou des bases de données.

Plusieurs types de systèmes répartis existent. Les systèmes d'exploitation répartis permettent de gérer des grappes d'ordinateurs hétérogènes ou des ordinateurs multiprocesseurs. Les systèmes d'exploitation réseaux qui, contrairement aux précédents, permettent de gérer des systèmes informatiques hétérogènes. Chaque ordinateur y possède son propre système d'exploitation (éventuellement différent) et est connecté aux autres ordinateurs formant ainsi un réseau où les services disponibles sont partagés.

Les systèmes répartis actuels sont généralement construits à partir d'une couche logicielle intermédiaire au dessus du système d'exploitation réseau pour masquer l'hétérogénéité du système et rendre plus transparent les accès aux services distants. Ces systèmes sont construits en général en se basant sur un modèle spécifique (appel de procédures distantes, notification d'événements, objets répartis, documents distribués, etc.).

La construction des systèmes distribués pose de nouveaux défis :

L'hétérogénéité : peut être vu selon plusieurs points de vue, matériel, systèmes d'exploitation, langages de programmation. TCP/IP permet de masquer l'hétérogénéité des réseaux, et les middlewares gèrent les autres aspects de cette hétérogénéité.

La transparence : un système d'exploitation doit rendre l'accès aux ressources réparties transparent, en déchargeant les applications de certains aspects (accès, représentation des données, localisation).

L'extensibilité : elle permet de faciliter l'intégration de nouvelles entités dans le système sans affecter ce dernier. Une des approches les plus utilisées est celle de la séparation entre les interfaces des services et leur implémentation.

La sécurité : la sécurité est un point important dans un système distribué. Elle permet d'assurer le bon fonctionnement des applications et de respecter les droits sur les ressources disponibles. Par exemple elle doit :
- assurer l'intégrité des données ;
- gérer les droit d'accès et la confidentialité des utilisateurs ;
- gérer l'accès aux ressources de façon sécurisée.

La tolérances aux pannes : l'une des caractéristiques des systèmes distribués est qu'il sont fortement exposés aux pannes. L'un des défis des concepteurs des systèmes distribués est de mettre en œuvre des mécanismes qui permettent selon la qualité de service requise de gérer au mieux les éventuels accidents qui risquent de se produire (indisponibilité d'une ressource de calcul, échec d'une communication, endommagement d'un fichier, etc.)

La passage à l'échelle : l'un des principaux problèmes qui se posent lorsqu'un système distribué évolue (en nombre de ressources) est la dégradation des performances. Cette dégradation est dûe à plusieurs paramètres tels les coûts des communications (surtout si les sites sont géographiquement éloignés), ou les accès aux services du système.

La gestion des accès concurrents : ce problème déjà présent dans les applications parallèles ou multithreadées doit aussi être géré dans les systèmes distribués. Les accès concurrents ne concernent plus seulement les accès aux applications mais aussi aux ressources du système.

Le modèle client/serveur Le modèle client/serveur est le modèle de programmation le plus répandu dans les systèmes répartis. Ce modèle est né du besoin de la séparation des rôles entre les différentes composantes d'un système réparti. Les principes de base dans ce modèle sont les suivants [CDK01] :
- le client est l'initiateur de l'invocation au serveur ;
- le serveur implémente les services ;
- le client utilise une référence vers le serveur pour pouvoir l'invoquer.

Ce modèle possède plusieurs variations, car dans une application répartie réelle, cette séparation des reponsabilités entre les différentes entités n'est pas toujours suffisante pour définir son architecture. Un client peut jouer le rôle d'un serveur et le serveur peut devenir à son tour un client (par exemple lors d'un *callback*).

1.1.1 Middleware

Le terme *middleware* désigne la couche logicielle intermédiaire qui fournit un haut niveau d'abstraction de programmation permettant de masquer l'hétérogénéité des réseaux de communication, des ressources matérielles, des systèmes d'exploitation et des langages de programmation. CORBA en est un exemple. D'autres *middlewares*, tel Java-RMI n'autorisent qu'un seul langage de programmation. La plupart des middlewares sont basés sur TCP/IP, qui permet déjà de masquer les différences des réseaux.

Les *middlewares* fournissent un ensemble de services aux applications à travers des

interfaces de programmation d'application (APIs), pour réduire la complexité des développements.

L'avantage des *middlewares* peut être résumé dans les points suivants :

1. la portabilité et réutilisabilité des codes sur différents environnements ;
2. le développement d'applications est indépendant des infrastructures, ce qui réduit le temps du développement ;
3. l'affranchissement des solutions purement propriétaires ;
4. les services fournis réduisent la complexité des systèmes distribués.

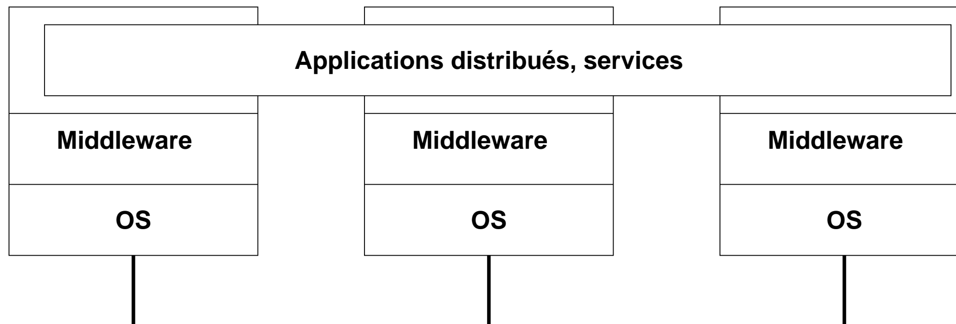


FIG. 1.1 – Système réparti et middleware

Les middlewares ne permettent pas seulement de masquer l'hétérogénéité, mais fournissent aussi un modèle de distribution et de communication uniforme pour les programmeurs d'applications distribuées. Par exemple, le modèle CORBA est basé sur l'invocation d'objet distant. D'autres modèles existent, tels la notification d'évènement ou les traitement de transactions distribuées. Les quatre principaux modèles [AST02] sur lesquels sont basés les *middlewares* sont :

Appel de procédures à distance : ce modèle permet l'appel de procédures implémentées sur des machines distantes de la même manière qu'un appel de procédure implémentée sur la machine locale, en masquant les communications réseaux.

Objets répartis : le principe de l'appel de procédures distante est appliqué dans ce modèle pour pouvoir invoquer des objets distants de façon transparente. Il se base sur le fait qu'un objet implémente une interface et que seule la description de cette dernière est nécessaire pour pouvoir invoquer l'objet.

Systèmes de fichiers distribués : dans ce modèle simple (basé sur un principe introduit par UNIX), tout est considéré comme étant un fichier.

Documents distribués : dans ce modèle, l'information est organisée en documents se trouvant sur des machines distribuées à travers le réseau. Le Web est l'exemple le plus populaire de ce type de modèle, Lotus Notes en est un autre.

Les *middlewares* sont devenus un choix incontournable pour le développement d'applications réparties, et des environnements d'implémentation offrant tout un ensemble de mécanismes pour rendre au maximum transparents les communications et la répartition sont apparus. Hormis les applications du web, les *middlewares* basés sur les objets répartis, que nous allons présenter dans les sections suivantes, sont devenus des standards.

1.1.2 Modèle objets distribués

Un objet distribué est une entité logicielle qui possède les mêmes caractéristiques qu'un objet « classique », mais qui possède par rapport à ce dernier une certaine autonomie de fonctionnement, qui lui permet d'interopérer avec d'autres objets distribués.

La caractéristique principale des objets est l'encapsulation des données et des opérations sur ces données. Les méthodes d'un objet sont accessibles par l'intermédiaire d'une (ou plusieurs) interface(s). Cette séparation entre l'implémentation d'un objet et son interface est très importante dans les systèmes distribués, car elle permet d'un côté de bien définir les services fournis par une application, et d'un autre côté de se libérer des détails d'implémentation de l'objet distant en ne se préoccupant que de son interface.

Lorsqu'un client invoque un objet distant, une implémentation de l'interface de l'objet, appelé *souche* permet d'empaqueter les paramètres de la méthode invoquée et d'appeler l'objet distant en passant en premier par les squelettes qui font l'opération inverse (dépaquetage des données), et transmet l'appel à l'interface de l'objet. Au retour, les données sont empaquetées par les squelettes et transmis au client par l'intermédiaire de la souche qui dépaquette les résultats fournis.

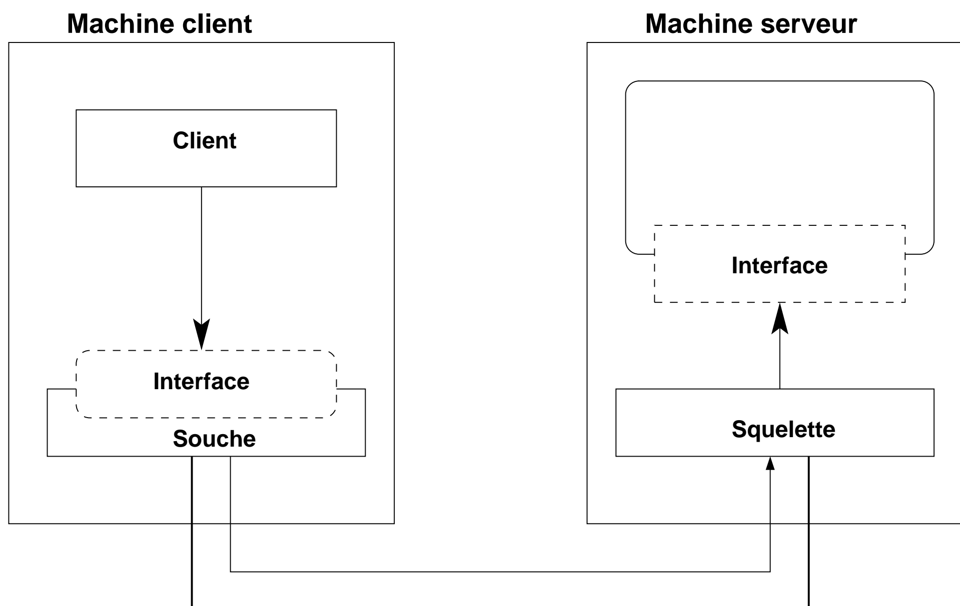


Fig. 1.2 – Le modèle d'objets distribués

Les avantages de l'approche objet (i.e. encapsulation des données et des méthodes, granularité, extensibilité, réutilisabilité, modularité) restent valables dans un contexte distribué. Viennent s'ajouter à ceux-ci d'autres avantages, qui sont surtout liés aux caractéristiques de l'environnement dans lequel les objets distribués s'exécutent. Ainsi, les objets distribués supportent généralement la portabilité entre plates-formes hétérogènes et permettent l'interopérabilité entre les applications.

1.2 Approche objet/composant

La technologie logicielle en général et la programmation en particulier, se sont toujours intéressées aux notions de la décomposition et la modularité. C'est ainsi qu'après la décomposition des programmes en fonctions et procédures, sont apparus les « bibliothèques », « paquetages », « DLLs » et autres « modules ». La programmation orientée-objet n'a fait qu'amplifier cette orientation vers plus de modularité dans les programmes.

La réutilisabilité s'est généralisée dans le développement logiciel, et le terme « composant » est apparu pour désigner ces entités logicielles autonomes qui peuvent être intégrées dans les applications.

Composant La notion de composant étend la notion d'objet et se place à un niveau plus haut de modularité. Cette notion est apparue après le constat que l'approche objet a une granularité très fine, qui la rend très peu adaptée aux systèmes complexes et aux systèmes distribués en particulier. Ces derniers sont composés de plusieurs entités coopérantes entre elles, et une fine granularité rendrait la conception et la maintenabilité difficiles, surtout si le développement de l'application est réalisé par plusieurs parties.

Une définition précise du terme « composant » ne pourrait être valide pour tous les cas où ce terme est employé. A partir des différentes définitions qu'on peut trouver, on peut extraire certaines caractéristiques qui sont communes à la plupart d'entre elles et qui nous semblent les plus pertinentes. Un composant peut être défini comme suit :
- c'est un objet dans le sens où il possède les propriétés d'encapsulation, d'héritage et de polymorphisme ;
- il peut encapsuler plusieurs objets ;
- il possède des interfaces bien définies au travers desquelles il peut être manipulé ;
- c'est une entité logicielle indépendante qui sait interopérer. Un composant offre des services et est capable d'appeler d'autres composants ;
- il peut être combiné à d'autres composants pour former l'application. La réutilisabilité qui existe déjà dans les objets a été étendue pour donner lieu à la notion de composition, mécanisme plus transparent pour construire des applications à partir d'entités logicielles possédant des fonctionnalités particulières ;

Les notions d'objet et de composant qui viennent d'être présentées, ont servi de modèle architectural pour la construction de plusieurs *middlewares*. Trois d'entre-eux se détachent par le succès qu'ils ont eu : CORBA, Java-RMI et DCOM.

1.2.1 CORBA

CORBA est un *middleware* orientée objet qui est indépendant des langages de programmation. Il est basé sur l'*Object Management Architecture (OMA)* et le *Core Object Model (COM)*. L'architecture définie par l'OMA identifie les composantes essentielles d'une application distribuée. Elle se compose principalement de cinq parties [GGM99, AMR99] :

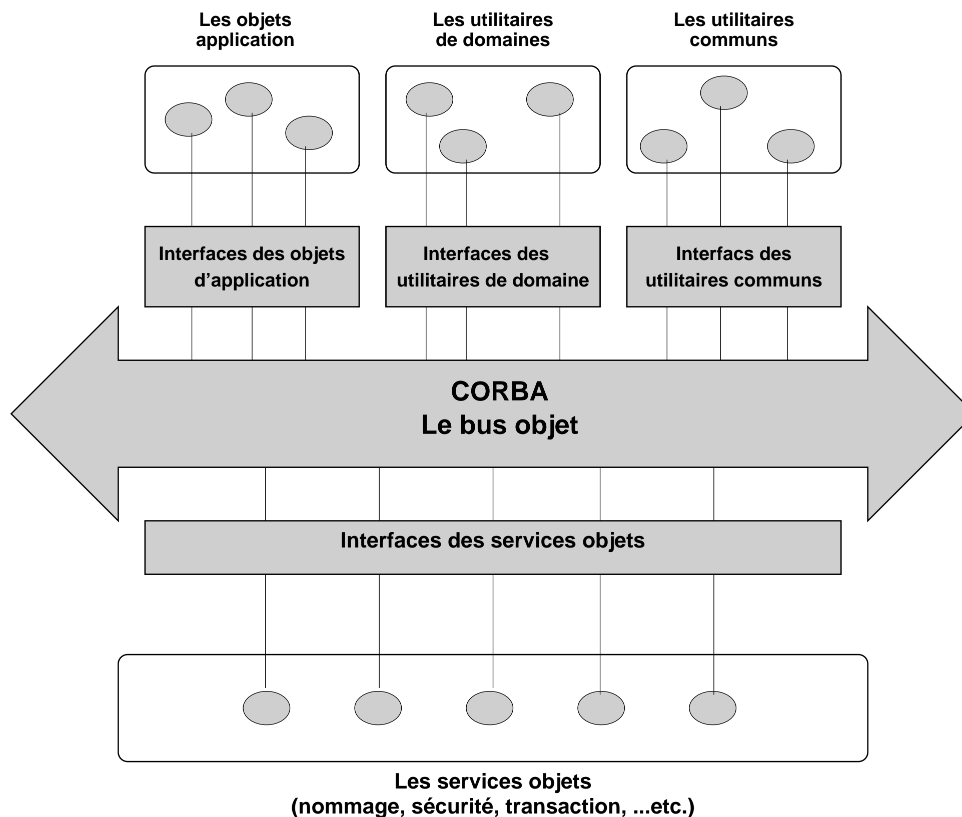


Fig. 1.3 – Architecture de l'OMA

Le bus objet *(l'Object Request Broker ou ORB)* est la partie principale de l'architecture. L'ORB prend en charge les communications entre les différents acteurs présents dans l'architecture. Il permet aussi de masquer tous les détails de communication entre le client et l'objet serveur [Vin97] :

- la localisation de l'objet serveur : le client accède à l'objet serveur de la même façon, que ce dernier se trouve dans le même espace adressage que le client, ou sur une machine accessible à travers le réseau.
- l'implémentation de l'objet serveur : le détail concernant le langage d'implémentation, du système d'exploitation ou de l'architecture matérielle sont transparents au client.
- l'état de l'objet serveur : l'ORB permet d'activer automatiquement l'objet serveur (si nécessaire) lors d'une invocation du client.
- les mécanismes de communication : l'ORB peut utiliser des protocoles de communication spécifiques de façon transparente pour le client.

Les services objets *(CORBAservices)* définissent les objets systèmes qui permettent d'élargir les fonctions de l'ORB par des services supplémentaires tels le service de nommage, d'évènement, de transaction, ...etc.

Les utilitaires communs *(CORBAfacilities)* définissent les outils utilisés par les applications. Ces outils se placent à un niveau d'abstraction supérieur par rapport aux « Services objets », et prennent en charge des besoins communs à un grand nombre

d'applications. On peut citer par exemple, les gestionnaires d'impression, les outils de gestion documentaire ou les gestionnaires de messagerie.

Les interfaces de domaine ou *Domain Interfaces* définissent les interfaces des « applications metier » pour différents domaines. Elles couvrent des besoins spécifiques comme par exemple les domaines de finance, de santé ou de télécommunication.

Les objets d'application sont les applications développées par les utilisateurs. Contrairement aux quatre catégories précédentes, ces applications ne sont pas standardisées, car elles dépendent des besoins des développeurs.

CORBA n'étant pas limité à un seul langage de programmation, deux aspects ont été introduits :

- L'interface de l'objet a été séparée de son implémentation. L'interface spécifie toutes les méthodes et attributs publiques ;
- Un langage de définition d'interface (IDL) a été introduit. Ce langage est utilisé pour spécifier les interfaces indépendemment des langages de programmation.

Dans ce modèle, le client et le serveur (qui est un objet CORBA) communiquent par des appels de méthodes distantes. L'ORB permet de masquer la localisation de l'objet, son implémentation, son état d'exécution et les mécanismes d'exécution. L'ORB fournit deux mécanismes d'invocation de méthodes :

- Les invocations statiques sont utilisées quand les objets sont connus à la compilation. Le client utilise les souches IDL et le serveur les squelettes statiques. Ces souches et squelettes sont générées par compilation des interfaces du client (si elles existent) et du serveur ;
- Les invocations dynamiques sont utilisées lorsque les objets sont inconnus à la compilation. Le client et le serveur, utilisent respectivement les interfaces d'invocation dynamique et les interfaces de squelette dynamique. Ces mécanismes utilisent un composant CORBA normalisé,l*InterfaceRepository*, qui garde une représentation des interfaces IDL. C'est grâce à ces deux mécanismes et à l'*Interface Repository* que la norme CORBA supporte la réflexivité. Par rapport au mode d'invocation statique, ce mode présente l'avantage de pouvoir supporter des environnements dynamiques, par contre il est moins efficace en temps de communication.

Pour les communications inter-objets, l'ORB utilise un protocole de communication haut niveau indépendant des protocoles de transport, appelé GIOP (*General Inter-ORB Protocol*). Ce protocole définit une représentation externe des données appelée CDR. L'implémentation de GIOP sur le protocole TCP/IP est appelée IIOP (*Internet Inter-ORB Protocol*).

Le modèle objet de CORBA

Le modèle objet de CORBA est similaire au modèle décrit dans 1.1.2, mais le client n'est pas forcément un objet et peut être un programme quelconque qui envoie des requêtes et reçoit les réponses. Dans ce modèle, le terme objet CORBA est utilisé pour référencer les objets distants dont l'implémentation est dans le même espace d'adressage que le serveur.

Dans CORBA, la description des interfaces se fait en IDL et il existe des règles de traduction du langage IDL vers chaque langage cible. D'autres systèmes comme DCOM

ou Globe, utilisent une approche différente basée sur les interfaces binaires. Avec ce type d'interface, les règles de traduction vers les langages cibles ne sont plus nécessaires, car il utilise des pointeurs vers des tables de fonctions virtuelles. On peut noter qu'un objet CORBA peut être implémenté avec un langage qui n'est pas objet [8] et la notion de classe n'existe pas dans le langage CORBA IDL.

Le modèle composant de CORBA : CCM

Avec *CORBA 3*, l'OMG a introduit plusieurs nouvelles capacités parmi lesquelles celle du modèle de composants peut être considérée comme la plus importante. Les autres spécifications concernent principalement :

- RMI sur IIOP ;
- les pare-feux ;
- la spécification d'un CORBA minimal ;
- la gestion de qualité de service ;
- la spécification d'un langage de script.

Le modèle composant de CORBA CCM (*Corba Component Model*) qui est compatible avec les EJB décrit le déploiement, la configuration et la composition des composants. Deux types de composants sont définis dans ce modèle : les composants basiques qui sont une mise en forme sous forme de composants des objets CORBA classiques et les composants étendus.

L'accès au composant se fait par l'intermédiaire d'une référence de base et d'une interface de navigation. Le modèle définit aussi les fabriques (*Home*) qui gèrent le cycle de vie des composants.

La spécification du modèle définit plusieurs sous-modèles sur l'architecture et le fonctionnement des composants [Dan02] :

Le modèle abstrait : Le modèle abstrait de composant permet de définir la structure intrinsèque du composant, il décrit les ports et les interfaces de la fabrique à l'aide d'un langage IDL étendu.

Le modèle d'implantation : Le modèle d'implantation décrit les comportements du composant, en prenant en compte la relation de composition de composants, ainsi que leurs caractéristiques (gestion des transactions par exemple). Le langage CIDL (*Component Implementation Description Language*) permet de décrire ce modèle.

Le modèle de programmation par conteneur : Ce modèle décrit les composants du point de vue du client ainsi que l'API mise à la disposition du développeur.

L'architecture des conteneurs de composant : Ce modèle décrit le fonctionnement des conteneurs.

La création, l'assemblage et le déploiement : Ce modèle décrit la distribution, l'assemblage et le déploiement d'un composant au sein d'une architecture CCM.

L'interfonctionnement avec les EJB : Cette partie de la spécification décrit les interactions avec l'autre modèle de composants.

[8] Par exemple en C ou en Cobol

Les caractéristiques du composant sont décrites et définies principalement par les ports, qui sont de quatre types dans le modèle. On distingue :

- Les facettes (*facets*) qui définissent les interfaces IDL exportées ;
- Les réceptacles (*réceptacle*) ;
- Les sources d'évènements (*event source*) ;
- Les puits d'évènements (*event sink*) ;
- Les attributs (*attribute*).

FIG. 1.4 – Structure d'un composant CORBA Component Model

1.2.2 Java Beans, Java-RMI et Enterprise Java Beans

Le modèle de calcul de Java [AG98] est *écrire une fois, exécuter partout.* Ici, le modèle d'objets distribués est intégré dans le langage lui-même, en essayant de garder le plus possible la sémantique des objets locaux.

Java Beans Java Beans est le modèle de composants proposé par Java. Il peut être considéré comme des classes Java précompilées qui peuvent être configurées par changement des valeurs des attributs exportés. La construction de l'application avec les Java Beans se fait par composition (en utilisant des outils graphiques) sans accès au code source du composant. Les composants Java Beans présentent les avantages du langage Java. Ils sont multiplateformes car l'exécution se fait dans la machine virtuelle Java, peuvent être chargés dynamiquement sur le réseau et l'aspect réflectif de Java permet de déterminer dynamiquement les interfaces.

Le modèle présente un nombre limité d'interactions et ne fournit pas d'autres services d'interaction à part l'invocation de méthodes. La composition qui se fait au moment de la conception et non à l'exécution, ne permet pas de créer ou de définir de nouveaux Java Beans à partir de plusieurs autres, le développeur étant obligé d'écrire le code qui permet de décrire les caractéristiques du *Bean* composé. Il faut noter aussi que l'utilisation de

l'API de réflection est 100 fois plus lente qu'un appel direct, et que ce modèle qui permet d'exécuter des applications s'exécutant sur un seul espace d'adressage est utilisé avec le modèle Java-RMI pour construire le modèle de composants distribués Java : *L'Enterprise Java Beans.*

Java-RMI Le modèle Java étant incapable de spécifier comment initier des calculs dans un espace d'adressage distant, *Sun* a définit un mécanisme appelé RMI [WRW96] (Remote Method Invocation), qui permet au programmeur d'appeler des objets distants. RMI est étroitement lié à la sérialisation d'objets dans Java, par laquelle il empaquette et dépaquette les données. Et comme la sérialisation d'objets Java est une spécifité du langage Java, RMI n'est utilisable qu'avec le langage Java seulement. L'architecture de RMI se compose de quatre couches :

la couche application : c'est cette couche qui contient l'implémentation du client et du serveur.

la couche proxy : c'est à travers cette couche que l'application communique. Tous les appels vers les objets distants, l'empaquetage et le dépaquetage des paramètres et le retour des résultats se font dans cette couche. Les souches et squelettes, qui représentent respectivement les côtés client et serveur sont des classes compilées par le compilateur RMI (RMIC).

la couche référence distante : c'est elle qui assure l'aspect fonctionnel de l'application. Elle assure notamment une référence persistante vers l'objet distant (avec reconnection éventuelle si besoin est).

la couche transport : c'est elle qui assure la connexion entre le client et serveur, la gestion et la localisation de tous les objets distants répertoriés. Elle assure les fonctions suivantes :
- Connexion entre les espaces d'adressage ;
- Suivi des connexions en cours ;
- Écoute et réponse aux invocations ;
- Constitution d'une table de tous les objets distants.

Elle repose sur TCP, mais pourrait aussi utiliser UDP ou SSL (*Secure Socket Layer*) grâce au *Socket Factory.*

Enterprise Java Beans Ce modèle [MH98, Rom99] étend le modèle Java Beans pour supporter le modèle composants répartis. Un EJB encapsule une partie de l'application globale, et est exécuté dans un « conteneur » qui gère les accès concurrents, les transactions, la sécurité et le protocole d'invocation des méthodes. Ce modèle contrairement à son modèle de base, ne supporte pas la composition graphique d'EJB, mais permet au développeur de configurer l'environnement d'exécution de son conteneur. Le modèle présente aussi l'inconvénient d'être réactif ce qui limite son domaine d'application. Le protocole de communication utilisé par les EJBs est Java-RMI.

L'architecture du modèle EJB est décrite dans la figure 1.5 et consiste en :
- Le conteneur EJB : permet de gérer l'exécution d'une ou plusieurs classes EJBs et sert d'intermédiaire entre les clients et les Beans. En effet, les interfaces distantes

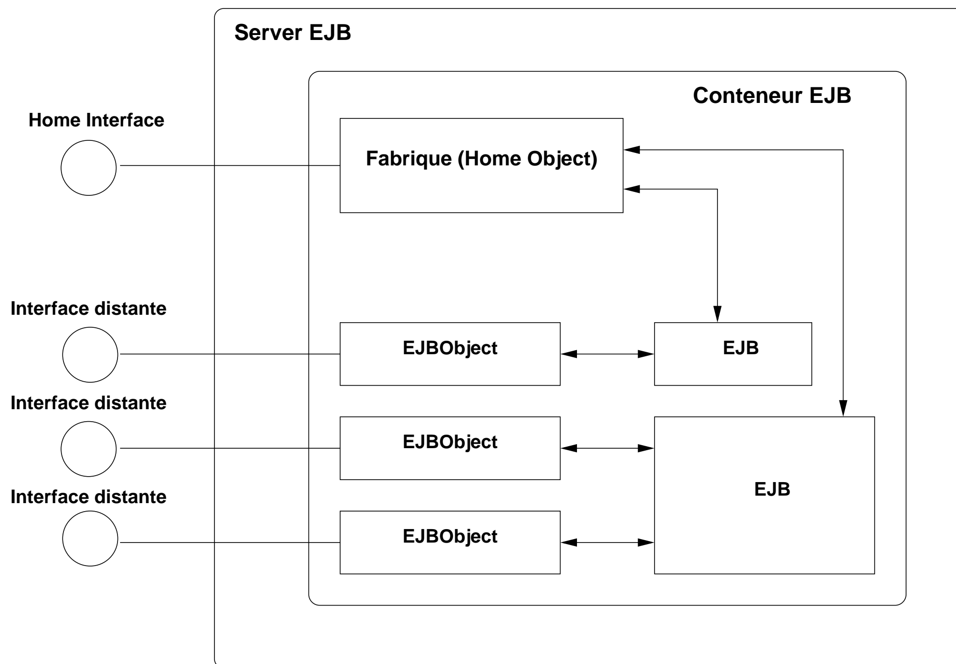


FIG. 1.5 – Le modèle *Enterprise JavaBean*

ne sont pas implémentées par les objets de l'EJB, mais générées automatiquement par les outils de développement du conteneur sous forme de classes `EJBObject`.

- Le serveur EJB : fournit un support d'exécution pour un ou plusieurs conteneurs et permet de gérer les ressources du système.
- La *Home Interface* : definit les méthodes de l'interface de l'objet qui sert de fabrique pour les classes EJBs. La définition de cette interface se fait par le développeur de l'EJB, alors que son implémentation est générée automatiquement par les outils de développement.
- Les *Interfaces distantes* et les *EJBObject* : comme déjà mentionné, l'accès aux EJBs réel, ne se fait pas directement mais par l'intermédiaire du conteneur. C'est grâce à ces objets *EJBObject* qui implémentent les interfaces distantes et qui sont générés automatiquement, que le client peut invoquer les EJBs. Autrement dit, les objets *EJBObject* représentent le point de vue du client de l'instance de l'EJB.
- Les classes EJBs : les Enterprise JavaBeans réels sont contenus à l'intérieur du conteneur. Ce sont les classes qui fournissent les fonctionnalités de l'EJB, mais elles n'implémentent pas les interfaces distantes pour ne pas autoriser l'accès direct au Bean.
- Les clients
- Les systèmes et services auxilliaires (Nommage-JNDI, JTS, sécurité)..

Java et CORBA L'une des principales limitations du modèle d'objets répartis Java est de ne pas pouvoir s'ouvrir à d'autres environnements et d'être toujours dépendant du langage. Avec l'apparition des EJBs, ce besoin d'ouverture est devenu indispensable pour pouvoir exploiter des ressources logicielles réparties hétérogènes. Pour cela l'*OMG* et *SUN*

ont définit un ensemble d'éléments qui permettent d'ouvrir les applications CORBA au monde Java :

- l'application Java utilise le bus logiciel de CORBA pour communiquer ;
- le protocole de communication de RMI est substitué par le protocole IIOP ;
- l'interface JNDI (*Java Native Directory Interface*) définie par *Sun* fournit une API qui permet de manipuler de façon unifiée n'importe quel type d'annuaire et en particulier celui de CORBA. La liaison entre l'annuaire et l'API JNDI se fait à travers une interface intermédiaire SPI (*Service Provider Interface)* ;
- l'*OMG* a défini les règles de traductions qui permettent de générer la description IDL correspondante à une application RMI.

Cette passerelle permet aux applications Java de profiter des nombreux services qu'offre la norme CORBA, l'ouverture aux autres langages de programmation, ainsi que de la robustesse et des performances des ORBs (généralement plusieurs fois plus rapide que RMI).

1.2.3 Distributed COM

DCOM [EE98, Red97] est une extension du modèle de Microsoft COM (*Component Object Model*) [Dal96], qui est lui même une extension du modèle OLE (*Object Linking and Embedding*). Ce dernier est un outil pour la manipulation de « documents composés » comme des composants. COM a été créé pour permettre la communication entre les applications Windows dans un environnement à base de composants. DCOM a étendu cette communication interprocessus à travers le réseau. Il se base sur le protocole DCE RPC pour implémenter un système d'appel de procédures vers des objets distants.

La figure 1.6 montre l'architecture globale de DCOM. Comme CORBA, DCOM crée les souches et squelettes pour permettre la communication entre client et serveur. L'un des éléments essentiels de l'architecture de DCOM est la « base de registre ». Cette base de registre sert à collecter toutes les informations de configurations qui paramètrent le système et en particulier les application DCOM. L'utilisation de cette base de registre et le fait que DCOM soit une extension de COM, rendent exploitable les applications COM dans un environnement réparti. Un autre élément essentiel dans l'architecture de DCOM est le *Service Control Manager* qui permet de gérer les composants du système (localisation, activation [9], enregistrement de référence).

COM se base sur la structure d'une table de fonctions C++ virtuelles pour représenter les interfaces. Les interfaces COM sont représentées au client comme un pointeur vers une table virtuelle. COM ne supporte pas l'héritage multiple, par contre, il permet l'implémentation de plusieurs interfaces par un seul objet.

DCOM utilise (contrairement à CORBA) des interfaces binaires, qui peuvent être considérées comme des tables de pointeurs vers les implémentations des méthodes de l'interface. L'avantage de l'utilisation de telles interfaces réside dans le fait que le système est indépendant du langage de programmation, car autrement (comme dans le cas de CORBA), un mapping des spécifications IDL vers chaque nouveau langage à supporter, doit être standardisé.

[9]Ce mécanisme qui n'est pas spécifié dans la norme CORBA permet d'activer un composant seulement à la demande

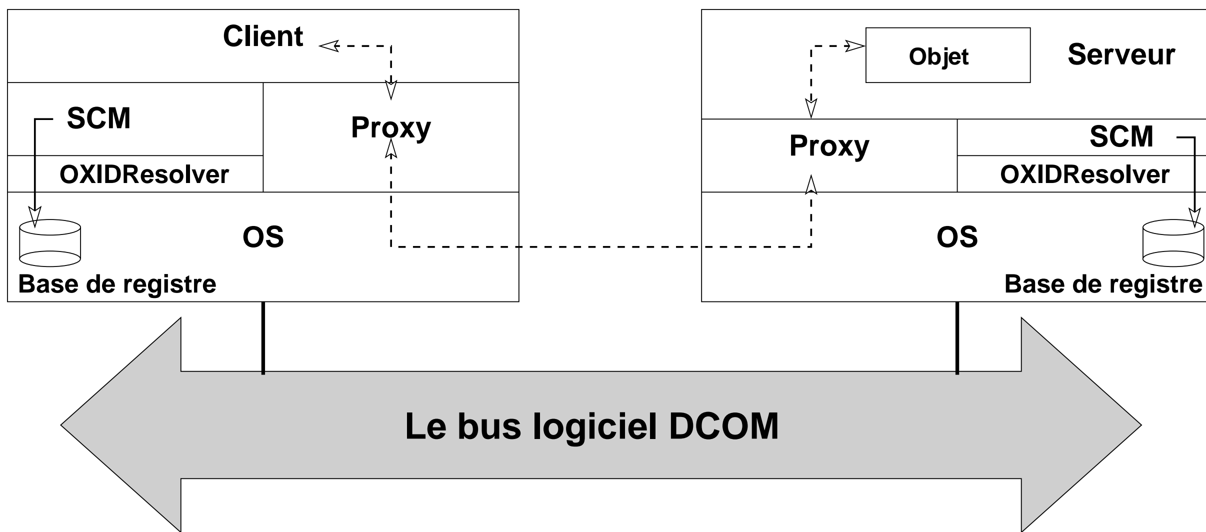


FIG. 1.6 – Architecture de DCOM

Dans ce modèle, tous les objets distribués sont temporaires et sont détruits dès qu'il n'y a plus aucun client qui les référence. Le nombre de clients référençant l'objet se fait par appel aux méthodes `AddRef` et `Release` qui font partie de l'interface *IUnknown*. Cette interface standard est implémentée par tous les objets et fournit les méthodes qui permettent de récupérer la référence de l'interface et sa description.

Le langage utilisé pour décrire les interfaces COM est appelé MIDL (*Microsoft Interface Description Language*). Comme le reste de COM, ses spécifications sont incomplètes et il ne possède pas de BNF formelle. Le langage MIDL est une extension du DCE RPC IDL qui est basé sur le langage C, et ainsi il supporte tous les types de données du C. DCOM apporte deux extensions notables au modèle :

1. le MTS(*Microsoft Transaction Server*) qui offre les mêmes caractéristiques qu'un conteneur CORBA CCM (création de lots, sécurité, persistance, transactions, évènements).

2. le MSMQ (*Microsoft Message Queue*) qui peut être considéré comme un service d'évènements plus évolué, car les évènements sont stockés dans une base de données et sont exploités en mode transactionnel.

Les communications en DCOM peuvent créer des surcoûts, car lors d'une requête d'un client vers le serveur, ce dernier *ping* le client jusqu'à la fin du traitement de la requête. Ceci est dû au protocole ORPC utilisé par DCOM qui utilise par défaut le protocole de communication non connecté UDP.

Avec des dizaines de millions de stations de travail tournant sous le système d'exploitation de Microsoft et sachant que chacune de ces stations est susceptible d'abriter des applications DCOM, ce dernier peut être considéré comme le système le plus répandu dans le monde et sera sûrement utilisé longtemps encore.

1.2.4 Comparaison entre Java-RMI, CORBA et DCOM

Les trois systèmes présentés ci-dessus sont actuellement les systèmes orientés-objet les plus largement utilisés et chacun possède des avantages et des inconvénients. Ils ont été développés avec des philosophies différentes. Java-RMI s'appuie sur le langage Java qui permet une exécution multiplatforme car c'est un langage interprété, et tire avantage de la sérialisation d'objet en Java pour permettre la construction de systèmes distribués dynamiques. CORBA est le résultat de la standardisation de nombre de sociétés pour fournir une plateforme logicielle hétérogène (logicielle et matérielle contrairement à Java) et s'appuie sur un modèle bien défini et de nombreux services. DCOM est lui la réponse de Microsoft et son principal but est de fournir une solution, compatible avec les anciennes versions (OLE, COM), pour construire un système distribué hétérogène (ce qui explique que son modèle objet distribué manque de spécification). L'utilisation très large du système d'exploitation Windows et le portage de DCOM vers d'autres plateformes (même si le système est fortement lié à l'API Win32) le rendent virtuellement le plus utilisé.

La description des interfaces se fait à l'aide de langages de description d'interface (IDL pour CORBA, MIDL pour DCOM), tandis qu'elle se fait à l'aide du langage Java dans Java-RMI. Pour la compatibilité entre les programmes serveur et client :

- les interfaces sont projetées vers les langages cibles dans CORBA ;
- les interfaces dans DCOM sont spécifiées au niveau binaire et vues par le client comme un pointeur vers une table de fonctions virtuelles (comme en C++) ;
- les programmes client et serveur sont écrit en Java, donc le problème ne se pose pas.

Le mécanisme de localisation des objets distribués reposent sur les même principes. Chaque système possède une sorte de service de nommage et une entité capable de localiser les objets et de rendre les accès transparents au client.

- Dans CORBA, c'est l'ORB qui est reponsable de la localisation des objets , à l'aide d'une référence que le client doit récupérer au préalable (généralement par l'intermédiaire du service de nommage). L'objet CORBA interagit avec l'ORB via l'interface de ce dernier ou par l'intermédiaire de l'adaptateur d'objet (BOA -*Basic Object Adapter*-, ou POA -*Portable Object Adapter*).
- Dans DCOM, c'est le SCM (*Service Control Manage*) qui gère la localisation en utilisant la base de registre.
- Dans Java-RMI, la localisation des objets est la responsabilité du *RMIRegistry* qui doit s'exécuter sur la machine de l'objet serveur.

CORBA et DCOM fournissent un grand nombre de services (nommage, sécurité, transactions, ...etc.), alors que Java-RMI en fournit relativement moins. D'un autre côté, DCOM peut s'appuyer aussi sur l'ensemble des services fournis par son environnement Windows.

Les trois systèmes possèdent des performances comparables avec un léger handicap pour Java-RMI par rapport aux deux autres. Des études [JR98, JZR98] ont montré qu'avec un environnement de développement basé sur le langage Java [10], CORBA possède un léger avantage par rapport à Java-RMI. Dans une application à grande échelle, CORBA est aussi mieux adapté et possède de meilleures performances [JRH00].

[10]Dans une situation où un autre langage est utilisé, comme C++, l'avantage de CORBA par rapport à Java-RMI est considérable

	Java/RMI	CORBA	DCOM
Plateforme	Multiplateforme : peut tourner sur n'importe quelle plateforme possédant une implémentation de la JVM	Multiplateforme : peut peut tourner sur n'importe quelle plateforme possédant une implémentation de l'ORB	Multiplateforme : peut peut tourner sur n'importe quelle plateforme possédant une implémentation des services DCOM
Langage de programmation	Java uniquement : RMI est basé sur la sérialisation d'objet en Java	Multilangage : CORBA étant une spécification, tout langage de programmation possédant une spécification et une implémentation peut être utilisé	Multilangage : plusieurs langage de programmation peuvent être utilisés car la spécification n'est pas basé sur un langage spécifique
Identification de l'objet serveur	A l'exécution à l'aide de l'*ObjID*	A l'exécution à l'aide de la référence de l'objet	A l'exécution à l'aide du pointeur d'interface
Génération de la référence de l'objet	Par l'appel d'une méthode	Par l'adaptateur d'objet	Par l'*Object Exporter*
Protocole de communication	JRMP (*Java Remote Method Protocol*	IIOP (*Internet Inter-ORB Protocol*)	ORPC (*Object Remote Procedure Call*)
Protocole de transport	TCP, UDP	TCP	TCP, UDP, IPX/SPX,... etc.
Mécanisme d'encodage	Sérialisation	CDR	NDR
Description d'interface	Java	IDL CORBA	IDL DCOM ou MIDL
Client dynamique	Oui (la réflection)	Oui (Le mécanisme DII)	Oui (Le mécanisme d'automation)
Serveur dynamique		Oui (Le mécanisme DSI)	Non
Localisation	Responsabilité de la JVM	Responsabilité du courtier (ORB)	Responsabilité du Gestionnaire de contrôle de service (*Service Control Manage* ou SCM)
Description des interfaces	Par auto-introspection, car c'est l'objet qui contient les informations de sa description	Par consultation du référentiel d'interface (*Interface Repository*)	Par consultation du *Type Library*

Tab. 1.1 – Comparaison entre Java/RMI, CORBA et DCOM

D'un autre côté, la comparaison entre les performances de CORBA et de DCOM montrent généralement des résultats comparables [APP00].

1.2.5 Autres systèmes d'objets répartis

Après avoir vu les trois systèmes objets les plus répandus, nous présenterons deux systèmes basés sur la même approche, qui sont certes moins utilisés, mais qui présentent des caractéristiques et une architecture intéressants. Le premier XCAT est motivé par l'exploitation du parallélisme et du calcul haute performance dans un environnement à base de composant. Le second se distingue par son modèle d'objets répartis partagés.

XCAT et CCA

XCAT [GKC+02, GKC+03, BCD+00, KBG+01] est un projet de plusieurs universités et laboratoires du département américain de l'énergie. Le but est de construire des applications et des composants pour des infrastructures massivement parallèles et distribuées. L'idée de base du XCAT est la construction de l'application par composition de composants. Cette composition se fait en liant les ports des composants entre eux. Les ports sont le moyen pour le composant d'offrir des fonctionnalités aux autres composants et de représenter les fonctionnalités dont le composant a besoin. XCAT se base sur les spécifications du CCA (*Common Component Architecture*) et est implémenté avec XSOAP (qui est l'implémentation du modèle Java-RMI en Java et en C++) comme protocole de communication, alors que l'implémentation précédente était basée sur HPC++ et NexusRMI. Ceci pour tirer profit des avantages du modèle composant et des services web.

Dans ce système, les ports représentent les interfaces importées et exportées par le composant. Comme CORBA et DCOM qui utilisent un langage de description d'interface, le SIDL (*Scientic Interface Dénition Langage*) est utilisé pour décrire les interfaces des ports. Ce langage permet en particulier de spécifier la distribution des données entre les objets parallèles. Avec l'implémentation XCAT qui est fortement liée au langage Java, la description des ports peut aussi se faire avec le langage Java ou en XML.

Ce système offre un service d'enregistrement basé sur LDAP pour l'enregistrement des composants et utilise le protocole de transfert HTTP.

Comparé aux autres systèmes présentés précédemment, ce projet met en avant le parallélisme de données et l'aspect performances. Comme beaucoup de projets académiques, celui-ci manque de pérennité. Après la première implémentation CCAT (*Common Component Architecture Toolkit*), une orientation vers les applications du grid et les services Web, a été adoptée lors de mise en œuvre de l'implémentation XCAT.

Globe

Globe [HvST99] est un système réparti orienté objet, qui est un système de Grid et un *middleware* au dessus d'Internet. Le système n'est pas forcément dédié aux applications de calcul scientifique, mais permet de supporter des applications réparties à grande échelle, contrairement aux autres systèmes orientés objet comme CORBA ou DCOM, qui généralement sont utilisés pour des applications distribués sur réseaux locaux.

Le modèle objet de Globe est différent du modèle utilisé dans la plupart des systèmes. Alors que le modèle objet de ces derniers se base sur le modèle objet réparti décrit dans la section 1.1.2, Globe définit un modèle objet où l'objet réparti est physiquement distribué [HvST96, BvST99] sur plusieurs sites, ou plusieurs processus. Ce modèle est appelé « le modèle d'objets répartis partagés » (ou DSO pour *Distributed Shared Object*).

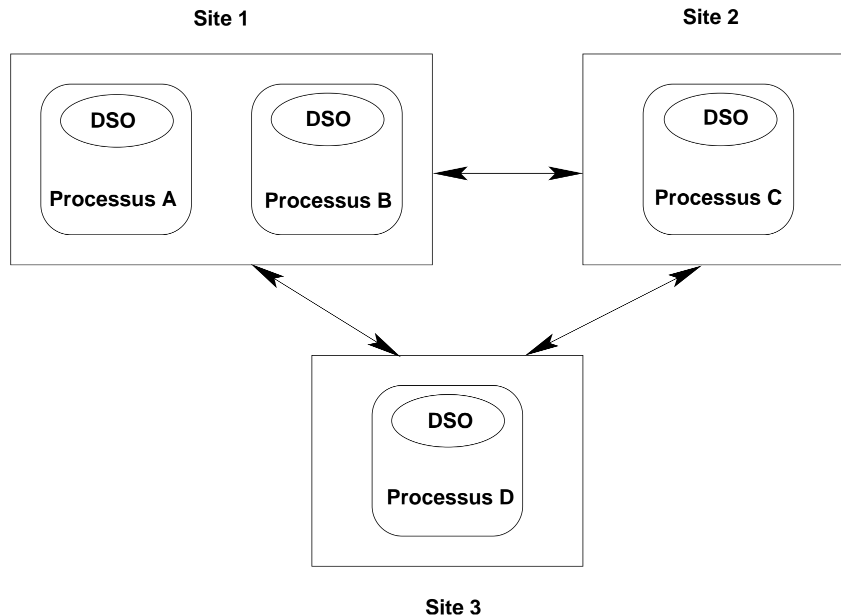


FIG. 1.7 – Un objet réparti partagé dans Globe

Dans ce modèle, un processus qui manipule un « objet distribué partagé » possède une copie locale de cet objet, appelée « objet local ». La figure 1.7, montre un « objet réparti partagé » sur quatre processus et chacun de ces processus possède une copie de l'implémentation de l'objet. Néanmoins, le modèle objet de Globe ne se résume pas seulement à une duplication d'un objet en plusieurs copies, car les différents objets locaux représentent le même objet sémantiquement.

Un DSO est un objet composé de quatre objets primaires (figure 1.8 :

1. un objet fonctionnel qui décrit et exécute les fonctionnalités de l'objet ;
2. un objet de communication qui envoie et reçoit des messages d'autres objets ;
3. un objet de contrôle qui gère l'objet fonctionnel. L'accès à l'objet fonctionnel se fait par l'intermédiaire de l'objet de contrôle qui fournit la même interface.
4. un objet de duplication qui permet de gérer les copies des objets fonctionnels. Cet objet, en collaboration avec l'objet de contrôle et l'objet de communication, gère aussi la consistance de l'objet partagé. En effet, dans un tel schéma, les invocations qui se font localement peuvent changer l'état de l'objet, donc de l'objet partagé. Grâce à l'objet de contrôle qui intercepte les requêtes vers l'objet fonctionnel et à l'objet de communication qui fournit un protocole de multicast complètement ordonné, les requêtes (au moins celle qui modifient l'état de l'objet) sont envoyées à toutes les copies de l'objet.

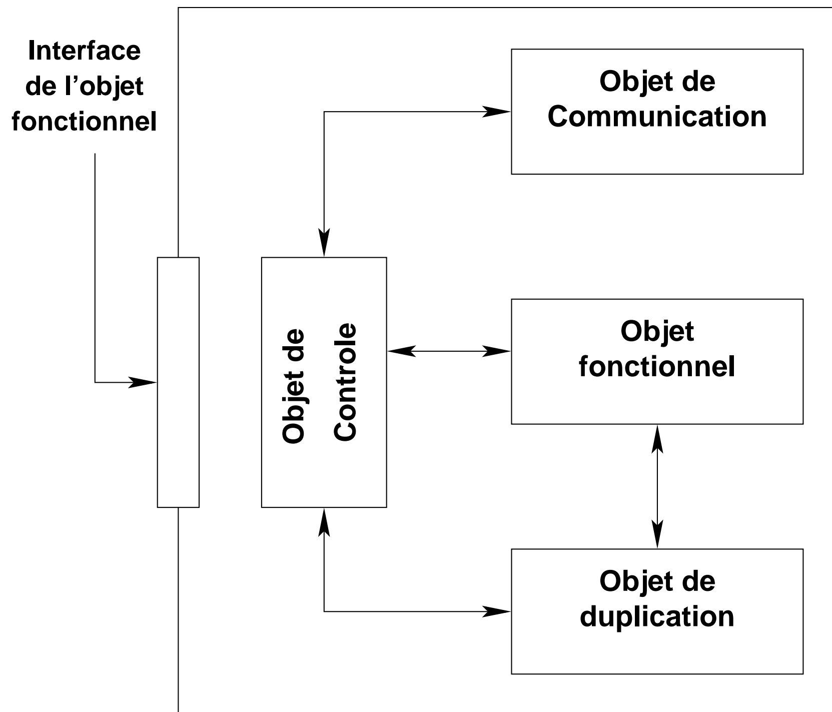


FIG. 1.8 – Le modèle objet réparti partagé de Globe

Chaque objet primaire fournit des services par l'intermédiaire d'interfaces que les autres objets primaires peuvent utiliser. Les interfaces dans Globe sont des interfaces binaires comme dans DCOM. Un autre point commun avec ce dernier, est le fait que tous les objets locaux implémentent une interface standard appelée *SOinf* (*IUnknown* dans DCOM) qui permet en particulier, à partir de l'identificateur de l'interface de récupérer le pointeur de son implémentation. Comme tous les autres systèmes utilisant des interfaces, Globe utilise un langage de description d'interface.

Un objet réparti partagé n'est pas forcément entièrement dans le même espace d'adressage et les objets primaires qui le composent peuvent être dans des espaces d'adressage différents.

1.2.6 Méta-modèles

La complexité croissante des systèmes et le besoin d'avoir un moyen formel de les modéliser pour garantir leur cohérence, ont engendré la naissance d'un nouveau domaine qui est celui de la modélisation. Ce domaine permet de couvrir les premières étapes de construction des applications, mais permet aussi de faciliter la maintenabilité des systèmes en ayant une vue de haut niveau. Dans les paragraphes suivants, nous présenterons trois méta-modèles de systèmes distribués (le MDA n'est pas spécialement conçu pour modéliser les systèmes distribués).

Le modèle RM-ODP

Le modèle RM-ODP (*Reference Model - Open Distributed Processing*) [FLdM95, dM95] est un standard défini au début des années 90, par l'ISO (*International Standardization Organization*) et l'ITU (*International Telecommunication Union*), pour définir les concepts architecturaux des systèmes répartis. Ce modèle ne définit pas une plateforme standardisée comme CORBA, mais définit le modèle générique qui permettra de construire des plateformes réparties, en normalisant les *points de vue* d'analyse, en définissant les modèles pour spécifier chaque point de vue, et en définissant les fonctions nécessaires à la construction d'applications réparties.

Pour spécifier les applications, RM-ODP définit cinq points qui permettent de fournir chacun une abstraction particulière des caractéristiques du système. Ces cinq points de vue sont :

1. le point de vue d'entreprise : pour déterminer les activités du système à créer.
2. le point de vue d'information : pour décrire l'information stocké.
3. le point de vue traitement : décrit l'ensemble des objets du système (identité, interfaces, état, comportement) et leur interaction (opération, flots, signaux). Il peut être décrit en utilisant des langages de description d'architecture (ADL), ou par des méthodologies de modélisation comme UML ou OMT.
4. le point de vue ingénierie : définit les concepts relatifs aux aspects de répartition tout en permettant de décrire l'environnement considéré. Ces aspects permettent de décrire de manière abstraite tout environnement d'exécution réparti.
5. le point de vue technologique : décrit l'infrastructure physique utilisé dans le système réparti (CORBA, Java,...etc.).

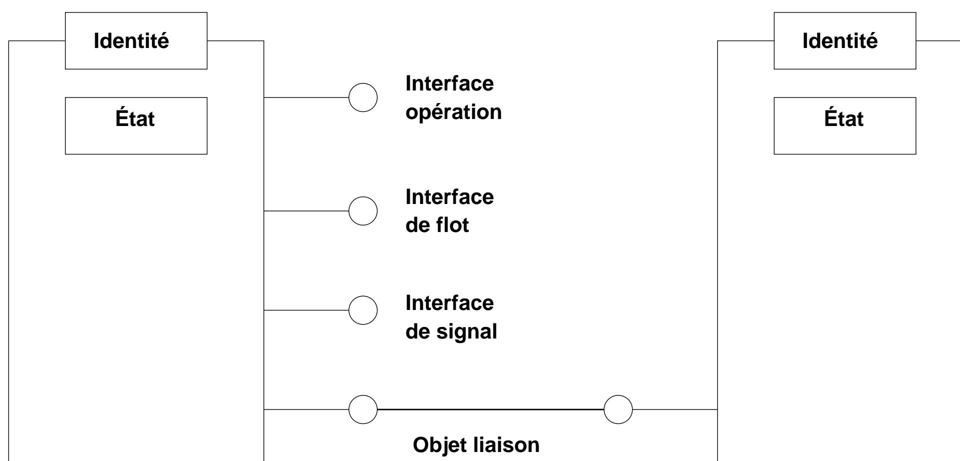


Fig. 1.9 – Architecture des composants dans RM-ODP

Le modèle (dans le point de vue traitement) décompose le système en un ensemble d'objets pouvant interagir entre eux. La figure 1.9 montre l'architecture générale d'un objet dans le modèle de référence RM-ODP. Un objet est modélisé en spécifiant

- son identité ;

- ses interfaces : qui peuvent être de trois types qui caractérisent les interactions avec les autres objets : interface opération, interface flot ou interface signal. Chacune de ces trois types d'interface définit les signatures correspondantes aux interactions. Les interfaces opérations définissent l'ensemble des méthodes invoquées par le client. Les interfaces de flots définissent les flux entre client et serveur, ainsi que le rôle de l'objet (producteur ou consommateur). Les interfaces de signal définissent les signaux qui sont des opérations de communication élémentaires ;
- son état ;
- son comportement qui caractérisent l'ensemble des états dans lesquels l'objet peut se trouver.

Le modèle RM-ODP est une norme de standardisation des concepts liés aux systèmes répartis. Seulement, même si le modèle fournit une vision globale des étapes de production logicielle, le modèle ne définit que des recommandations et n'aborde pas (ou que brièvement) la formalisation des concepts présents dans le modèle.

UMM : Unified Meta-Object Model

UMM [Raj00] est un méta-modèle orienté composant pour modéliser des systèmes distribués ouverts. Cette approche a deux objectifs principaux :

- fournir un méta-modèle adaptable aux systèmes existants permettant d'intégrer leurs concepts et caractéristiques ;
- enrichir le méta-modèle par d'autres concepts pour augmenter sa puissance.

Ce méta-modèle définit trois parties qui sont : les objets, les services et les collaborations, ainsi que les relations entre ces trois types d'entités.

1. Les objets sont entités autonomes. UMM ne fait aucune hypothèse sur le modèle d'implémentation, mais définit les caractéristiques d'un objet, à savoir : l'état, l'identificateur, le comportement, les interfaces et les implémentations. En plus trois aspects décrivent les objets qui sont :
   – l'aspect traitement : décrit l'aspect fonctionnel de l'objet (en utilisant une méthode formelle et une méthode textuelle) ;
   – l'aspect coopératif : décrit les relations entre différents objets (détection des autres objets, coût de service, négociations inter-objets,...etc.) ;
   – l'aspect auxiliaire : décrit d'autres caractéristiques que peut avoir un objet, telles la mobilité, la tolérance aux pannes ou la sécurité.

   UMM autorise l'intégration de systèmes répartis existants et pour y arriver,deux types d'objets sont introduits, les objets *Head-hunter* et les objets *Translator*. Les *head-hunter* sont des objets qui permettent de détecter les autres objets et de servir d'intermédiaire entre les objets demandeurs de services et les objets fournisseurs de services. Ce type d'objet ne permet pas par contre d'établir une communication directe entre deux objets, car ceci exigerait un modèle homogène de tous les objets et c'est la raison pour laquelle le méta-modèle a introduit les objets *translators* . Les objets *translators* servent d'intermédiaire entre deux objets de deux modèles différents, ainsi l'aspect fonctionnel de ce type d'objet est divisé en deux partie, une partie par modèle.

2. Un service dans UMM est défini comme étant l'ensemble des opérations qu'un objet peut exécuter (calcul, accès aux ressources).
3. Après avoir défini les services, le méta-modèle définit l'aspect collaboratif des objets.

La philosophie du méta-modèle UMM suit celle de l'approche MDA qui sera présentée dans la section suivante. Pour les concepteurs d'UMM, la conception des systèmes distribués, qui sont les applications visées par ce méta-modèle, doit passer par la spécification haut niveau, la génération automatique de code et l'intégration automatique ou semi-automatique des systèmes existants. Les derniers travaux autour d'UMM montrent un rapprochement entre ce dernier et l'approche MDA. Néanmoins, UMM reste un méta-modèle destiné exclusivement aux systèmes distribués et aborde certains problèmes non abordés dans l'approche MDA (tel les qualités de service). Il est orienté objet/composant, alors que le MDA est orienté modèle.

MDA (Model Driven Architecture)

Le MDA [Boa01] est actuellement la nouvelle orientation de l'OMG, pour répondre au problème de l'évoluvité et du couplage des systèmes informatiques qui deviennent de plus en plus complexes.

Le principe de base du MDA est l'élaboration de modèles indépendants des plates-formes (PIM ou *Platform Independant Model*) et de modèles dépendants des plates-formes (PSM) et la transformation du premier modèle vers le second pour aboutir ainsi à la génération d'un code exécutable. L'approche fait donc appel à trois techniques principalement : des techniques de modélisation (pour la description des différents modèles), des techniques de transformation de modèles (passage d'un modèle à un autre) et des techniques de projection de modèles vers les systèmes cibles (génération de code).

L'initiative MDA met le focus, non plus sur l'approche orientée objet/composant (longtemps défendu par l'OMG, qui a utilisé cette approche pour définir un standard d'infrastructure), mais sur une approche plus globale orientée modèle. Ce passage à un niveau plus haut d'abstraction, a pour but l'élaboration de modèles en favorisant les approches transformationnelles paramétriques vers les plates-formes technologiques.

Les transformations entre modèles ne se limitent pas seulement à des transformations de PIM vers PSM. Elles peuvent être de quatre types (figure 1.10) :

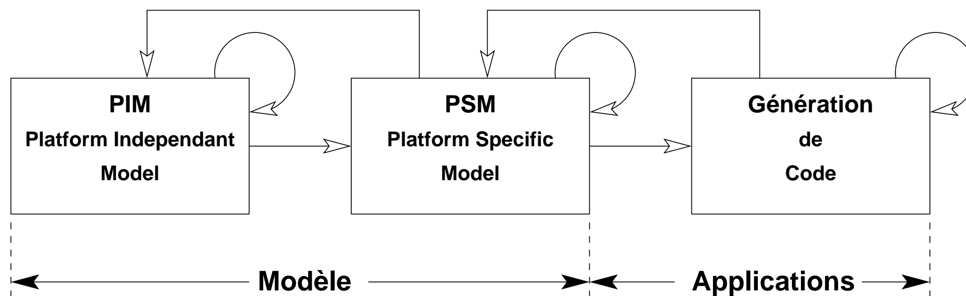


FIG. 1.10 – Transformation de modèles dans le MDA

1. PIM vers PIM : le passage entre modèles permet d'ajouter ou de soustraire des informations au modèle.

2. PIM vers PSM : c'est la transformation qui permet de passer d'un haut niveau d'abstraction (qui définit le modèle), à un niveau d'abstraction dépendant de la plate-forme (qui décrit le code exécutable).
3. PSM vers PSM : ces transformations permettent d'effectuer des reconfigurations ou des optimisations. Elles sont aussi nécessaires dans le cas où la transformation PIM vers PSM n'est qu'une étape pour obtenir un formalisme intermédiaire avant la génération de code.
4. PSM vers PIM : ces transformations permettent de construire les modèles à partir de systèmes déjà existants. Ces transformations sont essentielles pour pouvoir intégrer les applications existantes.

L'approche MDA est très prometteuse, actuellement elle est encore au stade initial et beaucoup de travail reste à faire. En particulier, il y a un réel besoin d'une théorie des modèles qui permettrait la manipulation et les transformations de modèles.

Un autre défi que cette approche devra relever et par lequel va dépendre probablement son succès, est la simplification des fonctionnalités pour le développeur. L'approche développée par l'OMG, correspond à une demande des industriels pour résoudre leurs problèmes dé génie logiciel et doit apporter des réponses performantes, opérationnelles et précises pour s'imposer.

1.3 Approche Grid

Le terme « *grid* » est utilisé pour désigner la technologie permettant de mettre en liaison un très grand nombre (qui peut atteindre plusieurs millions) de ressources, reliées par le réseau Internet et servant à résoudre des problèmes de calcul scientifique. Ce terme a été choisi par analogie avec le réseau électrique. Cette approche, connue aussi sous le nom de *métacomputing* a beaucoup évolué. Elle n'est plus seulement utilisée pour le calcul haute performance, mais a aussi donné naissance à plusieurs approches plus ou moins spécialisées comme le calcul Pair à Pair ou le calcul global.

L'architecture du Grid [FKT01] peut être décrite en identifiant les composants nécessaires pour l'exécution des applications cibles. Une description haut niveau de cette approche est la description en couches de la figure 1.11.

Les cinq composantes principales du Grid décrites dans [FKT01] sont :

1. ***Fabric*** : cette couche contient toutes les ressources partagées. Ces ressources n'incluent pas seulement les ressources de calcul (PCs, SMPs, cluster), mais aussi leurs systèmes d'exploitation et de gestion, les ressources de stockage de données, les base de données et des outils scientifiques comme des capteurs ou des télescopes.
2. ***Connectivity*** : cette couche fournit les protocoles de communication et de sécurité. Le protocole de communication permet l'échange de données entre les différentes ressources, alors que le protocole de sécurité (basé sur les services du premier protocole) fournit les mécanismes d'identification des ressources et des utilisateurs, et la sécurisation des échanges. Le service de sécurité peut être basé sur un standard existant pour réduire la complexité du système.

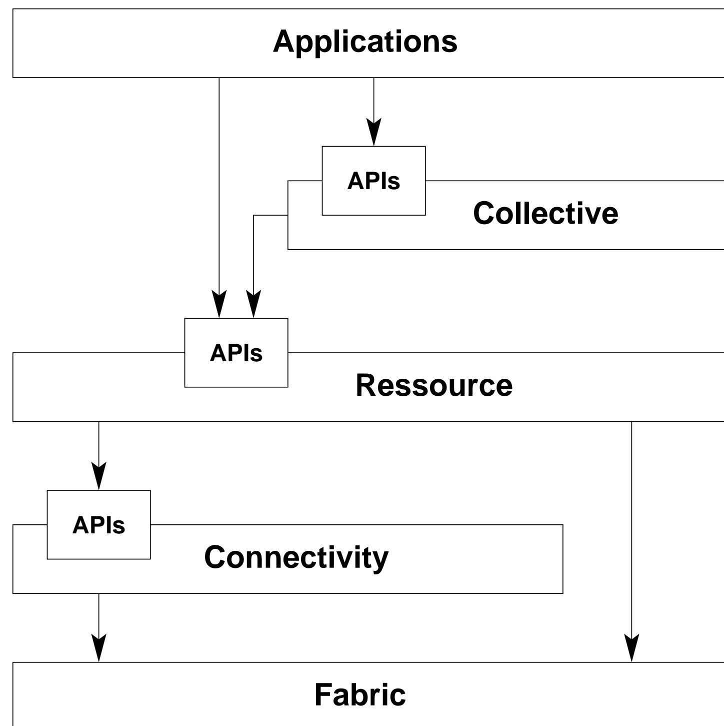


Fig. 1.11 – Architecture du Grid

3. ***Ressource*** : cette couche utilise les services communication et sécurité, pour permettre de construire le protocole sous forme d'APIs de deux services sécurisés :
   - le service d'information qui permet d'obtenir des informations sur les ressources partagées et leur état ;
   - le service de gestion qui permet de gérer l'accès au ressources partagées : allocation, accès et droit, qualité de service, monitoring, contrôle, ... etc.
4. ***Collective*** : cette couche est associée à l'ensemble des ressources partagées et collecte les interactions qui existent entre elles. Le modèle ne spécifie pas l'ensemble des services de cette couche. En pratique, on retrouve un grand nombre de services dont :
   - allocation et ordonnancement des ressources ;
   - monitoring et analyse des applications ;
   - service de réplication de données ;
   - service d'exploration de ressources connu généralement sous le nom de *Directory Service*.

   D'autres services existent dans cette couche, on peut citer un service comparable au *Directory Service*, mais qui permet d'explorer les ressources logicielles disponibles, et le service de programmation qui permet d'utiliser les modèles de programmation dans le système (passage de message, maître/esclave).
5. ***Application*** : cette dernière couche contient les applications utilisateurs. Ces applications font appel aux services des couches précédentes qui sont accessibles via des APIs bien définies.

Dans cette section nous présenterons les systèmes distribués du Grid. Ces systèmes fournissent les mécanismes de base (bas niveau généralement) du métacomputing. Certains sont à mis chemin entre les middlewares et les systèmes de Grid, en fournissant un modèle de calcul distribué haut niveau (objet par exemple pour Legion), ou des environnements de programmation haut-niveau.

Tous les systèmes de grid doivent néanmoins fournir les services suivants :

- localisation et allocation de ressources impliquant des espaces de stockage pour gérer les informations sur les ressources ;
- création et exécution des processus, avec une politique d'ordonnancement et d'allocation de ressources ;
- couche de communication entre les processus.

Partant de ces services, d'autres services supplémentaires sont construits pour construire un système de Grid puissant et complet. De tels systèmes étant utilisés par un grand nombre de personnes et exécutant des applications à durée très longue, deux services s'imposent comme étant essentiels, ce sont les services de sécurité et de tolérance aux pannes.

1.3.1 Legion

Legion [GNTW, GWF+94] est un système de métacomputing se basant sur une approche orientée objet. Toutes les ressources matérielles et logicielles dans le système sont représentées par des objets Legion, qui sont des processus actifs. Le résultat de ce projet est une machine virtuelle extensible qui gère : la mise à l'échelle, la tolérance aux pannes, la sécurité et la programmation haut niveau. Legion définit le mécanisme et le format des messages à transférer, et les protocoles haut niveau d'interaction entre objets. Deux moyens sont utilisés pour assurer une exécution haute performance de l'application distribuée : l'exécution purement parallèle de l'objet, ou l'exécution des objets sur des sites sélectionnés. Pour implémenter ce parallélisme, Legion utilise plusieurs outils et mécanismes :

- les bibliothèques de passage de messages (PVM, MPI) ;
- les langages parallèles (Fortran, Java, ou MPL [11]) ;
- l'encapsulation d'applications existantes (séquentielles ou parallèles) dans des objets ;
- la construction d'applications de flux de données ou de graphes.

Legion permet l'utilisation de différents langages de programmation en se basant sur le principe d'interface. Deux IDLs peuvent être utilisés pour décrire ces interfaces, l'IDL de CORBA et le langage MPL (*Mentat Programming Language*). L'allocation des ressources dans Legion est basée sur la négociation entre demandeurs et fournisseurs de ressources. Les trois types de ressources dans Legion sont les ressources de calcul, les ressources de stockage et les ressources réseaux.

Le système est devenu un produit commercial et porte actuellement le nom d'*Avaki*.

[11] Mentat Programming Langage, un C++ parallèle, utilisé pour écrire Legion

1.3.2 Globus

Plutôt qu'un modèle de programmation, tel le modèle orienté objet, Globus fournit un ensemble de services, à partir desquels les développeurs d'outils ou d'applications peuvent construire leurs applications en faisant appel à ces services ou en construisant d'autres services. Cette approche qui se base sur l'approche *Toolkit* [FK97] a pour but de fournir les services de base en autorisant l'utilisation de plusieurs types d'applications et de modèles de programmation. Cette méthodologie est possible si les services sont bien distincts et leurs APIs bien définies, pour être incorporés dans les applications de façon incrémentale.

L'architecture de Globus [FK98] est une architecture à niveaux, où des services de haut niveau sont construits à partir des couches bas niveau. Le toolkit Globus est modulaire et l'application peut exploiter seulement certains services. Le toolkit est composé des services suivants :

Couche de communication cette couche regroupe les modules dédiés aux communications et constitue la partie la plus importante de *Globus*.
- *Nexus* : cette bibliothèque [FKT96, FGT96] de communication qui est le composant le plus important dans Globus, fournit une interface de communication. Cette bibliothèque définit cinq abstractions : les nœuds, les contextes, les threads, les liens de communication et les requêtes de services distants. Cette interface fournit une interface qui est utilisée pour construire d'autres services ;

Couche de gestion des ressources cette couche est utilisée pour localiser, identifier, allouer et ordonnancer les ressources du métasystème, en se basant sur le langage RSL (*Resource Specification Language*).
- *Globus Ressource Allocation Manager (GRAM)* : cette couche fournit divers mécanismes de gestion des ressources (expression des besoins en ressources, identification des ressources disponible et leur ordonnancement) ;
- *Advanced Ressource Reservation and Allocation (GARA)* et *Ressource Broker* fournissent deux services de gestion de ressources plus évolués ;

Couche de sécurité : cette couche traite de l'un des aspects les plus importants qui est celui de la sécurité et de l'authentification.
- *Grid Security Service (GSS)* service de sécurité pour l'identification des utilisateurs et des ressources, ainsi que le cryptage/décryptage des messages ;
- *Global Security Infrastructure (GSI).*

Gestion de fichier :
- *Global Access to Secondary Storage (GASS)* : gère l'accès distant aux fichiers et aux ressources distribuées de stockage. Ce système permet d'utiliser les fonctions standards d'entrée/sortie en C pour pouvoir lire et écrire dans des fichiers distants, en utilisant un cache de fichier où sont copiés les fichiers distants auxquels accède le processus ;
- *Data Catalogue and Replica management*

Couche d'information : cette couche regroupe les services d'information. Elle peut être considérée comme un service de nommage étendu car elle ne se limite pas seulement au stockage de références mais fournit aussi des informations sur les ressources et les applications.

– *Metacomputing Directory Service (MDS)* : ce framework permet de regrouper les informations nécessaires et utiles aux applications distribuées telles que des informations sur les ressources, sur les performances ou les applications ;
– *Heart Beat Monitor (HBM)* : vérifie l'état et l'intégrité des composantes du système, pour pallier aux pannes et fournit un mécanisme de monitoring de l'exécution des différents processus.

Couche de gestion des processus :
– *Globus Executable Manager* : initialise les traitements par la création de l'environnement, le lancement des exécutables, le passage des paramètres, gestion de la terminaison, ... etc. ;

Cette description de *Globus* sous forme de liste (non exhaustive) illustre bien la philosophie de ce système qui a pour objectif de fournir un ensemble de services aux programmeurs. Les concepteurs de *Globus* justifient cela par le fait que leurs services sont distincts et possèdent des interfaces bien définies. À partir de là, les utilisateurs peuvent construire leurs applications de façon incrémentale, par inclusion des services requis, et les développeurs peuvent construire d'autres services et les inclure dans le système. C'est ce qui explique l'enrichissement de Globus par l'ajout de nouveaux services.

1.3.3 Harness

Par rapport aux autres systèmes de Grid, Harness [MDGS98, DFG+98] est un framework de métacomputing expérimental, qui a pour objectif de fournir un support d'exécution dynamiquement reconfigurable. Il est le fruit d'une collaboration entre Emory University, UTK et ORNL, et est basé sur IceT, un support d'exécution réparti implémenté en Java. L'aspect dynamique du framework est achevé par le mécanisme *plugin*, par lequel des services et des ressources peuvent être insérés ou retirés du système.

Harness permet de définir une ou plusieurs machines virtuelles. Une machine virtuelle, est un serveur d'état et un ensemble de noyaux s'exécutant sur les machines du système. Un noyau par machine virtuelle est exécuté sur chaque machine du système, et joue le rôle d'interface entre les applications et le système. Il a trois fonctions principales :

1. Fournir des services aux applications.
2. Interagir avec le serveur d'état.
3. Communiquer avec les autres noyaux de la machine virtuelle.

Par l'intermédiaire du mécanisme de plugin, le système permet de rajouter ou supprimer des services, tout en gardant la cohérence de l'environnement de programmation. Cette cohérence est assurée par la classification des services en deux catégories, les services du noyau qui sont permanents, et les services supplémentaires ajoutés par les *plugins*.

1.3.4 XtremWeb

XtremWeb [CDF+02] est un environnement de calcul global et de calcul Pair à Pair. Le calcul global est défini comme une branche du métacomputing et des systèmes de GRID. A la différence de ces deux derniers, le calcul global se différencie par l'implication

d'un grand nombre de ressources appartenant à des particuliers, connectées uniquement par Internet, pour exécuter de grandes applications de calcul scientifique. L'utilisation du réseau Internet impose un modèle de calcul avec peu de communication entre les sites. On retrouve ce type de partage de ressources aussi dans les systèmes Pair à Pair, mais ces derniers concernent essentiellement le partage de documents entre les différents utilisateurs. Une autre différence par rapport aux systèmes de calcul global est le fait que les machines d'un système Pair à pair peuvent être client et serveur à la fois selon que la ressource soit disponible localement, ou demandée aux autres utilisateurs.

Ce système se différencie des autres par le fait qu'il vise un grand nombre de types d'application, alors que généralement, les systèmes de ce genre, tel SETI@HOME, sont dédiés à une application particulière.

Le système est basé sur le langage Java, le langage de script Perl, et le serveur web Apache. Il utilise le protocole XML RPC et RMI, et le gestionnaire de base de données MySQL.

L'architecture du système est centralisée et basée sur le modèle Maître-Esclaves, où les Maîtres sont les *serveurs*, et les Esclaves les *workers*. Un worker permet de fournir les ressources disponibles que l'utilisateur a autorisées à être partagées. Il permet aussi l'exécution de l'application fournie par le serveur. Son architecture est composée de quatre entités, « une réserve de travail » qui reçoit le code binaire des tâches à exécuter, un « gestionnaire de tâches » qui gère les ressources et contrôle l'exécution, un « gestionnaire de communication » qui sert de lien entre le serveur de calcul ou le collecteur de résultat, et les tâches exécutées, et enfin une entité de monitoring. Durant tout le calcul, le worker envoie périodiquement des signaux, pour signaler son activité, et permettre au serveur de détecter les éventuelles pannes qui sont fréquentes dans ce genre de systèmes.

1.3.5 Grid versus Middlewares objet/composant

L'approche middleware orientée objet/composant, et l'approche grid, présentent toutes les deux des caractéristiques qui les rendent bien adaptées au développement d'applications et de support d'exécution pour la simulation répartie. Ce chapitre nous a permis de voir leur architecture logicielle et quelques systèmes utilisant les deux approches.

Les systèmes de grid sont « plus spécialisés », car ils sont destinés en premier lieu aux applications de haute performance et de calcul scientifique, et de ce fait, ils ne conviennent pas à tous les types d'applications réparties. De surcroît les outils de gestion et d'administration sont généralement lourds à gérer du fait de l'échelle très grande supportée par ces systèmes. Les middlewares orientés objet/composant sont plus « généralistes » et permettent de construire un plus large type d'applications.

Les aspects modèle de programmation, transparence et spécification de base sont des points sur lesquels les systèmes de grid sont en retard. Seulement quelques systèmes de grid sont basés sur un modèle de programmation plus ou moins bien spécifié, ce qui implique des contraintes de développement et d'entretien du code plus difficile à gérer. Dans un contexte de simulation distribuée comme celui auquel on s'intéresse, la réutisabilité est un paramètre primordial qui fait défaut à ces systèmes. Avoir un modèle basé sur la modularité est essentiel pour construire des simulations distribuées à base de composants virtuels. De plus, il faut gérer l'hétérogénéité des plateformes et l'interopérabilité entre les

simulateurs.

La pérennité de ces systèmes fait aussi défaut, par rapport au systèmes de middlewares où quelques standards ont réussi à s'imposer. Il en résulte aussi l'apparition d'un grand nombre de systèmes sans qu'aucun d'entre eux ne réussise à s'imposer.

Néanmoins, l'aspect performance qui est une priorité dans les systèmes de grid, reste un atout majeur de ces systèmes. D'un autre côté, les middlewares orientés objet/composant privilégient les aspects modularité, réutilisabilité, programmation haut niveau, etc. Des travaux autour de ces systèmes s'intéressant aux aspects performances et parallélismes [Kea99, KG97, PR98, DPP03] ont été effectués, et des passerelles entre les deux types de système sont mises en œuvre.

1.4 Conclusions

Dans ce chapitre, nous avons présenté les deux types d'environnement de développement de systèmes répartis : les middlewares orientés objet/composant et les systèmes de Grid. De cette présentation, on peut en déduire que les performances et le passage à l'échelle sont les deux principaux atouts des systèmes de Grid. En même temps, les middlewares orientés objet/composant s'imposent de plus en plus comme plateforme de développement dans le milieu industriel et académique.

Le chapitre suivant présente le modèle de calcul des réseaux de processus de Kahn, que nous allons utiliser comme modèle de base pour définir notre environnement d'exécution distribué.

2

Réseaux de Processus

Sommaire

2.1 Réseaux de Processus de Kahn 40
2.2 Ordonnancement des réseaux de processus de Kahn 43
2.3 Environnements et implémentations des réseaux de processus de Kahn 47
2.4 Réseaux de processus à flux de données 52
2.5 Conclusion 56

La forme doit être en rapport avec la fonction.
Le Corbusieur.

Un modèle de calcul est l'ensemble des choix « théoriques » qui permettent de construire un modèle d'exécution pour un langage donné. Il permet de définir le comportement et les mécanismes d'interaction des entités qui le composent. Ces entités peuvent être considérées comme des modules monolithiques (souvent appelés processus ou tâche). À partir de cela, un modèle de calcul permet de définir :

- comment chaque module (processus ou tâche) exécute ses traitements ;
- comment les modules s'échangent les données entre eux (mémoire partagée, passage de message bloquant/non bloquant) ;
- comment exprimer la concurrence entre les modules : cette caractéristique peut être exprimée de deux façons différentes : *data-flow* ou *control-flow*. Dans la première, ce sont les données échangées entre les modules qui expriment « implicitement » le parallélisme, alors que dans l'autre, l'ordre d'exécution des processus est explicite dans la spécification du système.

Certains modèles ne définissent pas l'aspect traitement relatif au module, mais seulement les aspects communication et/ou concurrence.

Parmi les modèles de calcul existants, celui de la « programmation à flux de données » est un modèle de programmation où l'accent est mis sur les mouvements des données

manipulées par le programme. Dans ce modèle, un programme peut être spécifié par un graphe orienté où les nœuds représentent les traitements ou processus, et les arcs les données.

Pour ce modèle, qui peut être graphique, le parallélisme est exprimé d'une manière naturelle car les différents processus peuvent s'exécuter concurremment les uns des autres. Les communications se font généralement à l'aide de buffers qui sont traités (théoriquement) comme étant non bornés. Les opérations de lecture dans les buffers sont destructives, les données sont enlevées des buffers après leur lecture.

Les applications de prédilection de ce modèle de programmation sont les applications où il existe un flux continu de données. C'est le cas des applications de traitement de signal et des applications audio/vidéo. On retrouve ce modèle aussi dans l'ingénierie électrique pour modéliser les circuits et les systèmes de contrôle. Au cours de son évolution, ce paradigme s'est enrichi avec plusieurs modèles qui ont exploré ses différents aspects et caractéristiques.

2.1 Réseaux de Processus de Kahn

Le modèle des réseaux de processus a été proposé par Kahn et MacQueen [Kah74, KM77] pour représenter des applications parallèles. Dans ce modèle, des processus qui s'exécutent indépendamment les uns des autres communiquent par des canaux de communication de type « premier entré, premier sorti » (FIFO). Ces canaux sont (théoriquement) de taille non bornée. Les processus sont séquentiels et déterministes : ils ne peuvent tenter de lire dans plusieurs canaux en concurrence, et un processus qui tente de lire dans un canal de communication vide est suspendu. Le réseau est formé en interconnectant les processus par des canaux. Pour chaque canal de communication, il ne doit y avoir qu'un seul processus producteur et au plus un processus consommateur. Chaque élément de la FIFO (ou jeton) est lu exactement une fois dans le canal (lecture destructive) et écrit exactement une fois.

Le déterminisme est la propriété fondamentale des réseaux de processus : le résultat du traitement (production et consommation des jetons dans les canaux de communication) ne dépend pas de l'ordre d'exécution. Les deux opérations possibles sur un canal sont écrire ou lire (si le canal n'est pas vide). Comme dit précédemment, un processus est suspendu s'il tente de lire dans un canal vide jusqu'à ce qu'une donnée soit écrite dans ce dernier. En particulier, un processus ne peut tester si un canal est vide.

Les processus peuvent être définis individuellement dans des langages hôtes tel C ou C++ avec la sémantique des réseaux de processus. Néanmoins, il faut faire attention lors de l'écriture des programmes avec les langages de programmation. Il peut être tentant d'utiliser des variables globales, pour ne pas passer par des canaux de communication, mais ceci peut entraîner un non-respect du modèle de Kahn et aboutir à un système non déterministe. Avec un peu de discipline, il n'est pas difficile d'écrire un programme déterministe.

Le principal désavantage du modèle est l'absence de réactivité comme les interactions avec l'utilisateur. Ceci est dû à l'absence d'évènement non déterministes dans les processus.

2.1.1 Représentation mathématique formelle des réseaux de processus de Kahn

Dans ce qui suit, nous allons voir la représentation mathématique formelle de Kahn des réseaux de processus. Les canaux de communication sont représentés par des suite de données, et les processus par des fonctions . À partir de cela, on peut décrire les réseaux de processus par des équations mathématiques.

DÉFINITION 1 (FLUX)
Les flux de données qui représentent les FIFOs dans le modèle peuvent être représentés par une suite ordonnée de données. Le modèle suppose que cette suite $X = [x_1, x_2, x_3, ...]$ peut être infinie. A ces séquences, on peut appliquer un ordre sur les préfixes, où la séquence X *précède* la séquence Y (notés $X \subseteq Y$), si X est un préfixe de (ou égal à) Y. Par exemple la séquence $X = [0, 1]$ est un préfixe de la séquence $Y = [0, 1, 2, 3]$, qui est à son tour préfixe de $Z = [0, 1, 2, 3, 0....]$. Le flux vide représenté par le symbole $\perp$, est un préfixe de toutes les séquences : $\forall X, \perp \subseteq X$. Toute suite croissante $\overrightarrow{X} = (X_1, X_2, ...)$ avec $X_1 \subseteq X_2 \subseteq ....$ a une borne supérieure $\cup\overrightarrow{X}$ qui peut être interprétée comme la limite :

$$\lim_{i \to +\infty} X_i = \cup\overrightarrow{X}$$

DÉFINITION 2 (PROCESSUS)
Un processus est une projection fonctionnelle de flux de données entrants (FIFOs d'entrée) à des flux de données sortants (FIFOs de sortie). Pour chaque processus, on peut écrire une équation décrivant cette association.

DÉFINITION 3 (RÉSEAU DE PROCESSUS)
En utilisant les représentation mathématiques des canaux de communication et des processus, on peut décrire un réseau de processus comme un ensemble d'équation. L'exemple de la figure 2.1 peut être représenté par les équations suivantes :

$$z = f(x, y)$$

$$(s_1, s_2) = g(z)$$

ou seulement par l'équation suivante :

$$(s_1, s_2) = g(f(x, y))$$

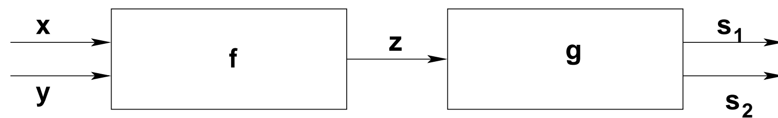


FIG. 2.1 – Représentation graphique d'un réseau de processus

Les fonctions décrivant les processus sont continues et monotones, car pour une suite de séquences $X_1 \subseteq X_2 \subseteq X_3 \subseteq ...$, on a :

1. continuité : $f(\lim_{i\to\infty} X_i) = \lim_{i\to\infty} f(X_i)$
2. monotonie : $X \subseteq Y \Rightarrow f(X) \subseteq f(Y)$

2.1.2 Déterminisme

Un réseau de processus est *déterministe* si le résultat du traitement (production et consommation des jetons dans les canaux de communication) ne dépend pas de l'ordre d'exécution. Tout ordre d'exécution qui obéit à la sémantique des réseaux de processus produit le même résultat.

Kahn a utilisé les points suivants pour prouver que le modèle est déterministe :

- **L'ensemble X est partiellement ordonné :**
  - l'ensemble $X = [x_1, x_2, x_3, ...]$ muni de la relation d'ordre partiel $\subseteq$ est un ensemble partiellement ordonné car :
    1. Réflexivité : $\forall X_i \in X, X_i \subseteq X_i$;
    2. Antisymmétrie : $X \subseteq Y$ *et* $Y \subseteq X \Rightarrow X = Y$;
    3. Transitivité : $(X \subseteq Y$ *etY* $\subseteq Z) \Rightarrow X \subseteq Z$.

    Le mot "partiellement" signifie que deux éléments quelconques ne sont pas nécessairement comparables.
- **L'ensemble X est un ordre partiel complet (CPO) :** $(X, \subseteq)$ forme un ordre complet car :
  1. X est un ensemble partiellement ordonné.
  2. X admet un plus petit élément $\perp$.
  3. l'ensemble X a une « borne supérieure » $\cup\overrightarrow{X}$, d'après le théorème de *Zorn*. Pour rappel, voici le théorème de Zorn :
     Soit $M \neq \emptyset$ un ensemble partiellement ordonné tel que tout sous-ensemble N de M, constituant une chaîne, admet une borne supérieure. Alors M admet un élément maximal.
- **Continuité** La fonction f décrivant le réseau est une fonction continue.
- **Monotonie** La fonction f décrivant le réseau est une fonction monotone.
- **Le point fixe de l'équation du réseau est unique**

  Théorème 1 (Théorème de Tarski)
  Si S est un ordre partiel complet, et $f : S \longrightarrow S$ une fonction continue, alors f admet un point fixe minimal (i.e., $\exists x \in S$ tel que $f(x) = x$ et si $f(y) = y$ alors $x \leq y$).

  De plus, la fonction décrivant le réseau étant monotone, ce point fixe minimal est unique.
  Donc, la solution de l'équation $X = f(X)$, qui est le point fixe de l'équation, est unique, quelque soit l'ordonnancement utilisé. Cette solution correspond ainsi au résultat fourni par le réseau de processus. Le théorème de *Scott* qui donne une méthode

itérative pour calculer ce point fixe à partir de la chaîne vide $\perp$, peut être appliqué au modèle des réseaux de processus de Kahn. En effet ce théorème dit que si f est une fonction continue d'un ordre partiel complet dans lui-même, son plus petit point fixe peut se calculer en itérant à partir de $\perp$: $\perp$, $f(\perp), f(f(\perp)), f(f(f(\perp))), ...$, et en prenant le la plus petite borne supérieure (ou supremum)

$$point_{fixe} = sup_i(f^i(\perp))$$

où $f^i(x)$ note $\underbrace{f(f(f(...f(x))))}$ i fois.

Pour les réseaux de processus finis, la méthode se terminera au bout d'un certain nombre d'itérations j pour lequel $f(X) = X$, avec $X = f^j(\perp)$. Par contre, pour les réseaux de processus qui s'exécutent infiniment, la méthode itérative est infinie elle aussi.

Non déterminisme Le non déterminisme peut être introduit dans le modèle de Kahn de plusieurs façons différentes :

- utiliser une fonction d'un processus non déterministe : pour garantir le déterminisme, la fonction d'un processus est une fonction séquentielle. Une fonction parallèle peut être non déterministe et rendre tout le réseau non déterministe. Introduire l'aléatoire dans la fonction du processus est un autre moyen qui invalide le déterminisme du modèle ;
- autoriser les communications entre processus par un autre mécanisme que les FIFOs (variables partagées, échanges d'attributs, ...etc.) ;
- partager une FIFO entre deux ou plusieurs processus : si deux processus ou plus lisent (resp. écrivent) dans la même FIFO, l'historique des jetons est aléatoire car aucune contrainte sur l'ordonnancement des processus n'est imposée par le modèle. La caractéristique d'ordre dans les FIFOs est ainsi perdue ;
- autoriser l'opération de lecture non bloquante : ceci peut se faire en autorisant la lecture sur deux canaux au choix (opération *select*) ou en introduisant l'opération de *lire au plus n éléments*, mais aussi ou en testant le nombre de jetons dans la FIFO à un instant donné. En effet avoir accès au nombre de jetons dans la FIFOs permet de construire les deux opérations précédentes. Pour éviter ce problème, le nombre de jetons dans la FIFOs ne doit pas être accessible dans les implémentations du modèle.

2.2 Ordonnancement des réseaux de processus de Kahn

L'ordonnancement du modèle des réseaux de processus de Kahn est confronté au problème du *write* non bloquant, qui implique des FIFOs non bornées. Avant d'aborder le problème d'efficacité, l'ordonnancement doit avant tout pouvoir exécuter le modèle avec une mémoire physique qui elle, est bornée. Trois ordonnancements principaux existent :

- demand-driven : proposé par Kahn et MacQueen, cet ordonnancement est basé sur un schéma demand-driven pour éviter l'accumulation des jetons dans les FIFOs que peut entrainer le mode data-driven ;

- data-driven : cet ordonnancement permet d'activer les processus qui ne sont pas bloqués sur une opération de lecture. Autrement, c'est l'ensemble des processus qui peuvent lire ou écrire des données qui peuvent être activés ;
- bornés : le plus utilisé, il a été proposé par Parks, et introduit le *write* bloquant.

2.2.1 Ordonnancement demand-driven

Kahn et MacQueen ont défini une stratégie d'ordonnancement des réseaux de processus basée sur une approche demand-driven. Cette approche vise essentiellement à éviter l'accumulation des jetons dans les FIFOs qu'engendre un ordonnancement data-driven. Pour cela, un processus n'est activé que s'il écrit dans une FIFO marquée. Une FIFO est marquée si un processus a effectué une opération de lecture sur cette dernière, qui a suspendu son exécution.

Dans cette stratégie, l'ordonnancement et l'activation des processus sont guidés par un processus. Ce processus est généralement celui qui produit les résultats finaux du réseau. Quand un processus tente de lire dans une FIFO vide, il est naturellement suspendu, la FIFO est marquée, et le processus écrivain dans cette FIFO est activé. Ce changement du processus actif est répété jusqu'à ce que les données requises soient produites par un des processus. Quand un processus produit des données dans une FIFO marquée il est immédiatement suspendu et le processus demandeur est activé pour consommer les données qui viennent d'être produites. Le changement du processus actif ne se fait que lorsqu'un processus écrit dans une FIFO marquée. Une opération d'écriture dans une FIFO non marquée ne suspend pas l'exécution du processus.

Le principal inconvénient de cette stratégie est le manque de performance. En effet, une implémentation multithreadée du modèle implique des changements de contexte répétitifs, avec généralement un effet d'aller/retour entre les processus actifs. Pour y remédier, Kahn et MacQueen définissent un « facteur d'anticipation ». Ce facteur représente le nombre de jetons qu'un processus peut écrire dans une FIFO marquée avant qu'il ne soit suspendu, alors le processus demandeur est activé. L'utilisation de ce facteur diminue le nombre de changements de contexte.

2.2.2 Ordonnancement data-driven

Dans ce type d'ordonnancement, un processus est candidat à être activé dès la disponibilité des données nécessaires à son exécution. L'ordre précis dans lequel les processus prêts sont activés n'est pas spécifié et plusieurs variantes peuvent être définies pour l'ordonnancement data-driven.

Comme les tailles des FIFOs ne sont pas bornées (dans le modèle au moins), il faut gérer la liste des processus prêts à un instant donné pour éviter l'explosion des tailles des FIFOs. Un ordonnancement simple consisterait à exécuter le processus jusqu'à ce qu'il se bloque sur un *read*, et activer un autre processus parmi les processus prêts (incluant ceux qui étaient bloqués et qui sont devenus prêts). En procédant ainsi, on risque d'avoir une explosion des tailles de certaines FIFOs. En effet un exemple simple est celui d'un processus qui s'exécute indéfiniment, et qui ne lit dans aucune FIFO.
Un ordonnancement data driven avec temps partagé entre les processus permet de garantir

une certaine équité entre les processus, mais ne tient pas compte de la fréquence des lectures/écritures dans les FIFOs et peut conduire à une accumulation des jetons dans certaines FIFOs.

2.2.3 Ordonnancement avec FIFOs bornées

Cet ordonnancement (proposé par Parks [Par95]), permet une exécution des réseaux de processus avec des FIFOs bornées. Il est basé principalement sur deux points :

Transformation d'un graphe non borné en un graphe borné : cette transformation est similaire à d'autres transformations dans d'autres modèles à flux de données. Elle a été inspirée des transformations de graphes définies dans [PA85, PA86], et dans le modèle des flux de données statiques [Den91], et permet de transformer un graphe représentant un réseau de processus en un graphe équivalent borné [12].

À chaque arc (FIFO) reliant deux processus est ajouté un arc dans le sens inverse, ou «FIFO inverse». Le processus écrivain (resp. lecteur) dans une FIFO devient le lecteur (resp. écrivain) dans la «FIFO inverse» correspondante. Dans chaque «FIFO inverse», n (qui définit la borne des FIFOs) jetons sont placés au départ, et à chaque opération de lecture (resp. d'écriture) est ajoutée une opération d'écriture (resp. de lecture) dans la «FIFO inverse». Le rajout de ces FIFOs inverses permet de bloquer le processus s'il tente d'écrire plus de n jetons dans une FIFO. À chaque instant, il ne peut y avoir plus de n jetons dans l'ensemble formé par la FIFO et sa FIFO inverse, car si P_1 et P_2 sont respectivement les processus écrivain et lecteur dans une FIFO F (voir la figure 2.2), alors ils sont aussi respectivement lecteur et écrivain dans la FIFO inverse F' qui contient au départ n jetons. À chaque instant, si $w(P_1)$ représente le nombre de jetons écrits par le processus P_1 dans F et $r(P_2)$ le nombre de jetons lus par le processus P_2, alors il y a $w(P_1)$ jetons lus par P_1 et $r(P_2)$ jetons écrits par P_2 dans F'. Donc dans les deux FIFOs F et F', on a respectivement $len(F)$ et $len(F')$ jetons, avec :

$$\begin{cases} len(F) = 0 + w(P_1) - r(P_2) \\ len(F') = n - w(P_1) + r(P_2) \end{cases}$$

Les deux FIFOs ne pouvant contenir toutes les deux plus de n jetons, un processus qui tenterait d'écrire plus de n jetons se bloquerait sur une opération de lecture sur la FIFO inverse.

Résolution des *deadlocks* : la limitation des capacités des FIFOs peut introduire des *deadlocks* artificiels. Lors de la détection de ce dernier, l'algorithme d'ordonnancement de Parks tente de le résoudre en incrémentant les capacités des FIFOs pleines. C'est la FIFO pleine de plus petite capacité qui est augmentée.

Basten montre dans [HB01] et [GB03] que la stratégie de Parks pour résoudre les *deadlocks* est incomplète, et présente un ordonnancement data-driven [13] avec une politique

[12] Un graphe borné est un graphe représentant un réseau de processus dont la taille de toutes les FIFOs est bornée par un entier n, pour n'importe quel ordre exécution.

[13] L'algorithme d'ordonnancement de Parks ne spécifie pas le type d'ordonnancement, et peut utiliser un ordonnancement data-driven, demand-driven ou hybride

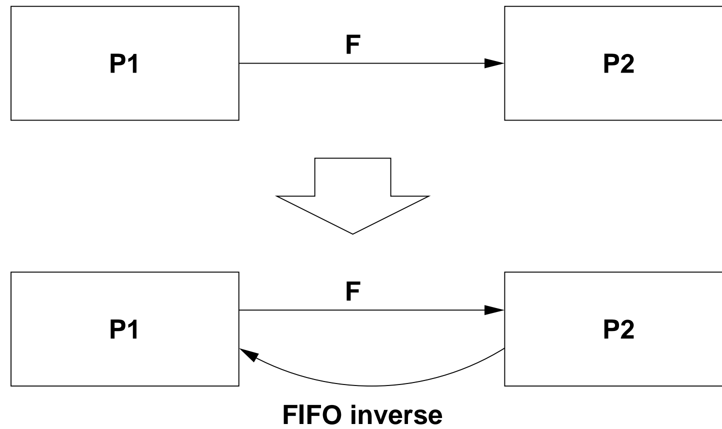


FIG. 2.2 – Transformation d'un graphe non borné en un graphe borné

de résolution des *deadlocks* légèrement différente, qui tient compte des *deadlocks* locaux [14].

2.2.4 Proposition d'un ordonnancement data-driven non borné

Partant des inconvénients de l'ordonnancement data-driven qui peut conduire à une accumulation des jetons dans certaines FIFOs, nous proposons un ordonnancement basé sur le même schéma d'exécution. Notre proposition permet de prendre en compte les tailles des FIFOs pour définir l'ordre d'activation des processus, évitant ainsi l'accumulation des jetons dans les FIFOs si ce n'est pas nécessaire.

Soit un réseau de processus $Pn = \{P, F\}$, où $P = \{p_1, ..., p_n\}$ est l'ensemble des processus, et $F = \{f_1, ..., f_m\}$ l'ensemble des FIFOs. On définit les fonctions suivantes :

- $w(f) = p$ si p est le processus écrivain de la FIFO f ;
- $r(f) = p$ si p est le processus lecteur de la FIFO f ;
- $Out(p) = \{f/w(f) = p\}$, l'ensemble des FIFOs dont p est l'écrivain ;
- $In(p) = \{f/r(f) = p\}$ l'ensemble des FIFOs dont p est le lecteur ;
- $len(f)$ est le nombre de jetons dans la FIFO.

On définit aussi la fonction qui calcule la moyenne des jetons des FIFOs dont le processus p est écrivain :

$$C(p) = Max\{len(f)/f \in Out(p)\}$$

On fixe pour les réseau Pn un seuil $s \geq 0$. Dans un ordonnancement data-driven classique, un processus est soit actif, soit bloqué, et un processus actif est toujours candidat à l'exécution. Pour éviter l'accumulation des jetons dans les FIFOs, nous proposons une autre classification qui tient compte de l'état (actif ou bloqué) du processus, mais aussi du nombre de jetons écrits dans les FIFOs. Ainsi un processus est classé soit bloqué (s'il a tenté de lire dans une FIFO vide), soit actif d'ordre 1 s'il n'est pas bloqué et $\frac{C(p)}{s} \leq 1$, soit actif d'ordre 2 s'il n'est pas bloqué et $1 < \frac{C(p)}{s} \leq 2$, et plus généralement, un processus est actif d'ordre i si $i - 1 < \frac{C(p)}{s} \leq i$. On obtient alors l'ensemble des processus bloqués

[14]ensemble de processus formant un cycle, où le dernier processus est en attente du premier

B, et une suite d'ensembles des processus actifs $A_1, A_2, ...A_n$. Un ensemble A_i est appelé ensemble d'ordre i. L'ensemble A_{min} est l'ensemble non vide d'ordre minimal.

La stratégie d'ordonnancement consisterait à exécuter tous les processus de l'ensemble A_{min}, (qu'on appellera dans ce cas l'ensemble actif) jusqu'à ce qu'ils soient tous bloqués, ou qu'ils passent au niveau suivant. Dans ce cas, rechercher les processus actifs appartenant à l'ensemble A_{min} et les exécuter. Si l'ensemble A_{min} est vide, répéter la même procédure pour les ensembles des processus d'ordre supérieur à min. Quand l'ensemble actif devient vide, classer tous les processus de nouveau, et choisir l'ensemble non vide d'ordre minimale comme ensemble actif.

Cet ordonnancement préserve la sémantique du modèle, et n'introduit pas de *deadlock artificiel* en bornant les tailles des FIFOs. L'accumulation des jetons dans une FIFO ne se fait que lorsqu'il n'y a aucun processus actif qui peut écrire dans une FIFO de taille inférieure. Une accumulation des jetons dans une FIFO ne se produit que lorsque tous les autres processus sont bloqués.

2.3 Environnements et implémentations des réseaux de processus de Kahn

Plusieurs implémentations des réseaux de processus existent. Ces implémentations sont généralement basées sur l'algorithme d'ordonnancement de Parks, qui prend en considération une mémoire de taille bornée.

2.3.1 YAPI

La bibliothèque YAPI [15] est écrite en C++ par Philips pour la simulation d'applications de traitement de signal digital.

Pour définir un réseau de processus, YAPI introduit les notions de type de processus et de type de données. Chaque type de processus possède un ensemble de ports d'entrée et de ports de sortie. Les ports sont typés, et chacun peut être connecté à une seule FIFO. Ainsi un réseau de processus peut être défini comme un ensemble de processus et un ensemble de FIFOs. Chaque FIFO sert à relier le port de sortie d'un processus au port d'entrée de même type d'un processus (différent ou le même). Les réseaux de processus dans YAPI sont hiérarchiques, un réseau de processus peut contenir d'autres réseaux de processus.

Trois opérations sont introduites dans YAPI :
- l'opération *read* qui lit des données du port d'entrée (ou plus précisément de la FIFO connectée au port d'entrée), et les stocke dans la variable locale du processus ;
- l'opération *write* qui écrit les valeurs de la variable locale dans le port de sortie ;
- l'opération *select* [16] qui introduit le non-déterminisme dans le modèle. Elle permet de choisir un port d'entrée (ou un port de sortie) parmi plusieurs. Le choix du port dépend de la disponibilité des données pour une opération de *select* sur des ports

[15] dont la syntaxe, et certains aspects sémantiques seront repris pour l'implémentation distribuée présentée dans 3.1

[16] cette opération est disponible seulement dans certaines implémentations de YAPI

d'entrée), et de la taille des FIFOs pour une opération de *select* sur des ports de sortie.

L'introduction du *select* dans YAPI est motivée par les applications simulées par cette bibliothèque, où la sélection d'un port pour la lecture de données est nécessaire (en particulier dans les applications de décodage MPEG). L'utilisation de cette primitive permet d'étendre le modèle déterministe des réseaux de processus de Kahn en introduisant des évènements non déterministes. Cette approche est basée sur celle de [Mar85] qui a introduit des primitives de sondage dans le modèle des « processus séquentiels communicants ». Dans ce modèle, les processus communiquent par des canaux de communication non bornés à l'aide de « rendez-vous ». Ainsi deux processus communiquant exécutent l'action de communication de façon synchrone. Si l'un des deux atteint le point de rendez avant l'autre, il se bloque jusqu'à ce que l'autre processus soit prêt à communiquer. Dans l'approche de Martin, une opération de sondage permet de savoir si un autre processus est bloqué parce qu'il a initié la communication sur le canal. Martin a démontré dans [Mar90] qu'avec l'utilisation des opérations de sondage, la sélection de canaux de communication peut être effectuée.

Un formalisme a été présenté dans [dKES+00] pour définir la sémantique des trois opérations *read*, *write*, et *select*.

Sémantique de YAPI En considérant que p est un port, alors à chaque instant :
- $c(p)$ est le nombre de jetons transféré par p ;
- $n(p)$ est le nombre de jetons écrits dans p ;
- $m(p)$ est le canal de communication connecté à p ;
- $v(p, k)$ est la valeur du k-ième élément dans p ;

A partir de ces définitions, pour un port p rattaché à un canal $m(p)$, à chaque instant on a les propriétés suivantes :
- $n(p) = 0$: un port ne peut pas contenir de jetons ;
- $c(m(p)) \leq c(p)$: le nombre de jetons transférés par un port est toujours supérieur au nombre de jetons transférés par son canal ;
- $\forall i, 0 \leq i < c(m(p)), v(m(p), i) = v(p, i)$: le .i-ième jeton transféré par p est le i-ième jeton du canal.

A partir de ces définitions, les opérations *read*, *write*, et *select* sont définies comme suit :

read si p est un port d'entrée de type t, x un tableau de type t, et n le nombre de jetons, alors, l'opération $read(p, x, n)$ est définie par :
- la précondition : $c(p) = N$;
- et la postcondition : $c(p) = N + n \wedge \forall i,\ 0 \leq i < n,\ x[i] = v(p, N + i)$

write si p est un port de sortie de type t, x un tableau de type t, et n le nombre de jetons, alors, l'opération $write(p, x, n)$ est définie par :
- la précondition : $c(p) = N \wedge \forall i,\ 0 \leq i < n,\ x[i] = v(p, N + i)$;
- et la postcondition : $c(p) = N + n \wedge \forall i,\ 0 \leq i < n,\ x[i] = v(p, N + i)$

select si k est un entier positif, $p_1, ..., p_k$ k ports, $n_1, ..., n_k$ k entiers positifs indiquant les nombres de jetons demandés (en lecture ou en écriture), alors l'opération $s = select(p_1, n_1, ..., p_k, n_k)$ est définie par :

– la précondition $\forall i,\ 0 \leq i < k,\ c(p_i) = N_i$;
– la postcondition $\forall i,\ 0 \leq i < k,\ c(p_i) = N_i \wedge 0 \leq s < k \wedge \diamond(N_s + n_s < c(m(p_s)))$ où le symbole $\diamond$ signifie « eventuellement » dans la théorie de la logique temporaire ;

Programmation Un programme YAPI se compose d'un ensemble de processus, de FIFOs, de réseaux de processus et de ports liant les processus aux FIFOs.

Processus Pour chaque type de processus, une classe est définie. Elle hérite de la classe `Process`. La figure 2.3 montre la déclaration et l'implémentation de deux processus : producteur/consommateur.

```
class Producer : public Process
{
 public:
  Producer(const Id& n,
           Out<int>& o);
  const char* type() const;
  void main();
 private:
  OutPort<int> out;
};
```

```
Producer::Producer(const Id& n,
                   Out<int>& o) :
                   Process(n),
                   out( id("out"), o)
{ }
const char* Producer::type() const
{
  return "Producer";
}
void Producer::main()
{
  std::cout << "Producer started"
     << std::endl;
  const int n = 1000;
  write(out, n);
  for (int i=0; i<n; i++)
     write(out, i);
  std::cout << type() << " " << fullName()
     << ": " << n << " values written"
     << std::endl;
}
```

```
class Consumer : public Process
{
  public:
     Consumer(const Id& n, In<int>& i);
     const char* type() const;
     void main();
  private:
     InPort<int> in;
};
```

```
Consumer::Consumer(const Id& n, In<int>& i) :
  Process(n),
  in( id("in"), i)
{ }
const char* Consumer::type() const
{
  return "Consumer";
}
void Consumer::main()
{
  int n,j;
  std::cout<<"Consumer started"<<std::endl;
  read(in, n);
  for (int i=0; i<n; i++)
      read(in, j);
  std::cout << type() << " " << fullName()
      << ": "<< n << " values read" <<
      std::endl;
}
```

FIG. 2.3 – Les processus producteur/consommateur en YAPI

Réseaux de processus C'est à l'intérieur des objets de type « réseaux de processus » que peuvent être créés les processus et les FIFOs. À l'intérieur d'un objet processus, seuls les ports sont instantiables . La figure 2.4 montre le réseau de processus de l'exemple précédent (producteur/consommateur).

```
class PC : public ProcessNetwork
{
  public:
    PC(const Id& n);
    const char* type() const;
  private:
    Fifo<int> fifo;
    Producer prod;
    Consumer cons;
};
```

```
PC::PC(const Id& n) :
    ProcessNetwork(n),
    fifo( id("fifo")),
    prod( id("prod"), fifo),
    cons( id("cons"), fifo)
{ }
const char* PC::type() const
{
  return "PC";
}
```

FIG. 2.4 – Le réseau de processus contenant les processus producteur/consommateur en YAPI

Réseaux de processus hiérarchiques Les objets de type `ProcessNetwork` peuvent être hiérarchiques, et donc contenir d'autres réseaux de processus. Ceci permet de diminuer la complexité et d'utiliser la réutilisabilité non pas au niveau processus mais au niveau réseau de processus. La figure 2.5 montre un exemple d'un tel réseau.

2.3.2 Ptolemy

Ptolemy [Lee03] est plus qu'une implémentation des réseaux de processus. Il propose un environnement de modélisation, de simulation hétérogène, et de construction d'applications concurrentes, dédié aux systèmes embarqués. Par hétérogène, on entend l'hétérogénéité des modèles de calcul, et non l'hétérogénéité logicielle ou matérielle. Cet environnement orienté objet développé à l'université de Berkeley est basé actuellement sur le langage Java (alors que la version antérieure 1.x était basée sur le langage C++). Pour manipuler plusieurs modèles de calcul, le système se base sur la notion de domaine. Chaque domaine a sa sémantique. Néanmoins, tous les domaines utilisent le modèle graphe à flux de données comme modèle de base. Les entités de calculs sont les nœuds, et les arcs les relations entre ces entités. Pour la plupart des domaines, les entités sont des « acteurs » (sauf pour le domaine FSM où les entités sont des états), et les relations représentent les communications entre acteurs (pour le domaine FSM, des transitions). Chaque domaine est géré par un « directeur ».

CSP (*Communicating Sequential Process*) Dans ce domaine, les processus communiquent par des points de synchronisation appelés « rendez-vous ». Lorsque deux processus atteignent un point de synchronisation, ils communiquent par des actions atomiques.

CT (*Continious Time*) Dans ce domaine, c'est un signal continu qui représente le temps qui permet aux acteurs d'interagir. Il est bien adapté pour modéliser les circuits analogiques et les systèmes mécaniques. Les acteurs dans ce modèle peuvent être décrits par les équations différentielles suivantes :

$$\frac{dx}{dt} = f(x, y, t)$$

$$y = g(x, u, t)$$

Avec x : l'état du système, u son entrée, y sa sortie, et $\frac{dx}{dt}$ la dérivé de x par rapport au temps.

```
class PN_H : public ProcessNetwork
{
  public:
    PN_H(const Id& n);
    const char* type() const;
  private:
    Fifo<int> fifo_front;
    Fifo<int> fifo_back;
    FrontEnd frontend;
    BackEnd  backend;
    ScaleXY  scale;
};
```

```
PN_H::PN_H(const Id& n) :
    ProcessNetwork(n),
    fifo_front(id("fifo_front")),
    fifo_back(id("fifo_back")),
    frontend( id("frontend"), fifo_front),
    backend( id("backend"), fifo_back),
    scale(id(("scale"),fifo_front, fifo_back))
{ }
const char* PC::type() const
{
  return "PC";
}
```

```
class ScaleXY : public ProcessNetwork
{
  public:
    ScaleXY(const Id& n,
            In<int>& i,
            Out<int>& o);
    const char* type() const;
  private:
    InPort<int>  in;
    OutPort<int> out;
    Fifo<int>    f;
    ScaleX       x;
    ScaleY       y;
};
```

```
ScaleXY::ScaleXY(const Id& n,
                 In<int>& i,
                 Out<int>& o) :
   ProcessNetwork(n),
   in (id("in"), i),
   out(id("out"), o),
   f   (id("f")),
   x   (id("x"), in, f),
   y   (id("y"), f, out)
};
. . . . . . . . . . . .
```

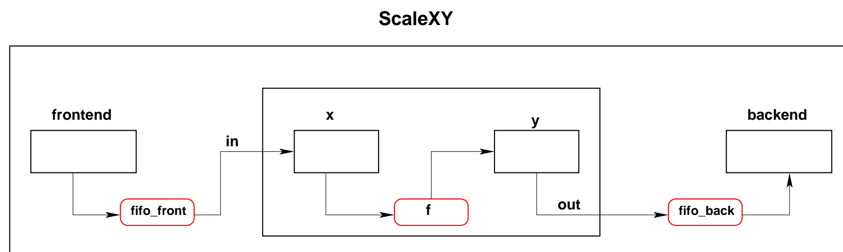


Fig. 2.5 – Réseau de processus hiérarchique en YAPI

DE (*Discret Events*) Dans ce domaine, qui est utilisé pour spécifier du matériel, et des systèmes de télécommunication numériques, les acteurs communiquent par des séquences d'événements, un événement est constitué d'une valeur et d'une étiquette de temps. Quand un acteur reçoit un événement, il est activé pour être exécuté.

FSM (*Finite-State Machine*) Comme dit précédemment, ce domaine définit des états et les transitions entre ces différents états. Il permet de construire des systèmes hiérarchiques, et est utilisé généralement pour modéliser des systèmes de contrôle.

SDF (*Synchronous Data Flow*) Domaine des réseaux de processus synchrones, il ne nécessite pas le multithreading, car le nombre de jetons produits et consommés par chaque processus est fixe à la compilation, et seule une résolution des équations décrivant le système est nécessaire.

PN (*Process Networks*) Domaine des réseaux de processus de Kahn, la sémantique défini par Kahn et MacQueen est respectée, et l'ordonnancement utilisé est celui proposé par Parks.

D'autres domaines existent (*Synchronous/Reactive*, *Discret Time*, *Distributed Discret Events*), mais sont encore au stade expérimental.

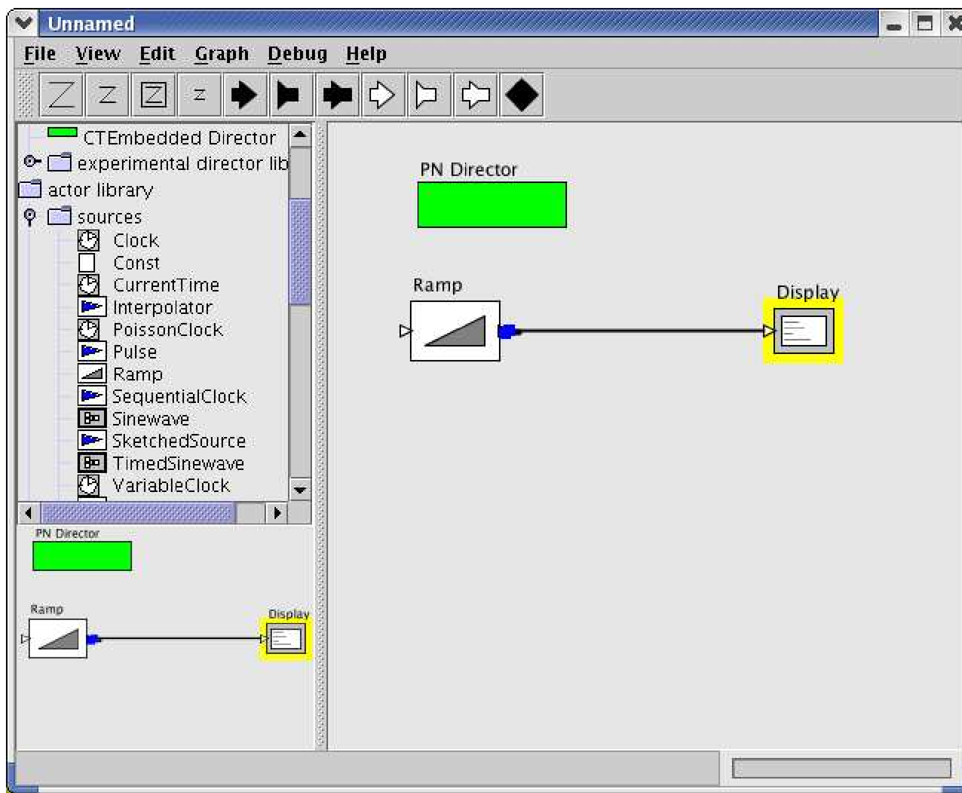


FIG. 2.6 – L'environnement de simulation de Ptolemy : Vergil

Ptolemy possède une interface graphique, *Vergil*, pour la construction des simulations, et possède un format XML pour les fichiers de description appelé MoML.

Ptolemy présente un environnement fiable pour modéliser, créer et simuler des applications de flux de données. Néanmoins il manque d'ouverture vers les autres systèmes existants, et impose des règles rigides de développement des composants. Un autre inconvénient est le manque de réutilisabilité des acteurs en dehors de l'environnement. Les interfaces peuvent ne pas être suffisantes pour les besoins de l'application.

D'autres implémentations des réseaux de processus de Kahn existent. [AE00] et [Par95] présentent une implémentation en C du modèle, avec les threads posix, mais seuls le système Jade/PAGIS [17], et [PR03] présentent une implémentation distribuée du modèle. La première ressemble plus à une implémentation du modèle d'exécution « Maître/Esclaves », qu'au modèle de Kahn, alors que la seconde est basée sur le langage Java et la sérialisation d'objets dans ce langage, pour fournir une construction de graphe dynamique.

2.4 Réseaux de processus à flux de données

Le modèle des « réseaux de processus à flux de données » [LP95] définit un sous modèle du modèle des réseaux de processus de Kahn. Comme dans le modèle de Kahn, c'est un

[17]Système de traitement d'images satellitaires

modèle de calcul où les processus communiquent par des FIFOs, avec des *Écritures* non bloquantes, et des *Lectures* bloquantes. En plus, dans ce modèle, chaque processus est une série d'activations « d'acteur », et à chaque activation du processus, un certain nombre de jetons sont produits et consommés. En divisant le processus en une série d'activations d'acteurs[18], le modèle permet d'éviter les changements de contexte qui se trouvent dans les implémentations des réseaux de processus de Kahn, et ainsi d'améliorer les performances d'exécution liées à l'ordonnancement. Les spécifications de ce modèle ne concernent pas seulement le fait que la fonction du processus est une suite de répétitions (éventuellement infinie) de fonction élémentaire, mais aussi le nombre de jetons lus ou écrits à chaque exécution de la fonction élémentaire du processus. Ce nombre de jetons est statique dans le modèle des réseaux de processus synchrones et dans le modèle de Karp et Miller, ou dépendant des valeurs des jetons d'un canal de contrôle dans le modèle des réseaux de processus booléens.

2.4.1 Le modèle de Karp et Miller

Dans ce modèle [KM66], qui est considéré comme la première référence au paradigme de flux de données, le parallélisme est exprimé par un graphe orienté, où les nœuds représentent les processus de calcul, et les arcs les flux de données. Chaque nœud est associé à une fonction qui calcule les données de sortie à partir des données en entrée. Le modèle possède d'autres restrictions, sur le nombre de jetons produits et consommés à chaque activation, et même sur l'ordre d'exécution des nœuds. Il peut être considéré comme un sous-modèle des réseaux de processus de Kahn.

Définition Le graphe G peut être défini par $G = (NF, A)$ où :
- NF est l'ensemble des tuples (n_i, f_i) avec n_i le nœud et f_i la fonction de calcul qui lui est associé ;
- A est un ensemble de septuples $(a_t, n_o, n_i, i_t, p_t, c_t, s_t)$ décrivant les arcs liants les nœuds du graphe avec
  - a_t représente la file d'attente qui lie les deux nœuds ;
  - n_o représente le nœud producteur dans la file a_t ;
  - n_i représente le nœud consommateur dans la file a_t ;
  - i_t représente le nombre de jetons initialement présent dans la file a_t ;
  - p_t représente le nombre de jetons produits par le nœud n_o à chaque exécution de f_o ;
  - c_t représente le nombre de jetons consommés par le nœud n_i à chaque exécution de f_i ;
  - s_t représente le seuil minimum de la taille de la file d'attente pour que le nœud n_j

Les caractéristiques du modèle fournissent des informations sur l'ordonnancement des nœuds du graphe. Ainsi si on considère $E_j = \{d_t / \exists (d_t, n_o, n_t, i_t, p_t, c_t, s_t) \in A\}$ représentant l'ensemble des arcs entrant dans un nœud n_j, celui-ci ne peut s'exécuter que si $\forall d_t \in E_j, size(d_t) \geq s_t$, autrement, si le nombre de jetons de chaque arc entrant dans n_j est supérieur au seuil associé à l'arc.

[18]un acteur est défini comme une unité de calcul

L'exécution du modèle peut être vue comme une séquence des ensembles finis $S_1, ..., S_i, ...$ représentant les opérations permises. Ainsi, à chaque activation, des processus vont consommer des données et en produire d'autres qui vont servir à activer d'autres processus. Un ordonnancement équitable consisterait à exécuter tous les processus présents dans un ensemble S_i.

Karp et Miller ont prouvé que le modèle est déterministe [KM66], ainsi le résultat de l'exécution ne dépend pas de l'ordre d'exécution des différents nœuds. Ils ont aussi étudié les conditions de terminaison des calculs, et les conditions pour que les files d'attente soient bornées.

2.4.2 Flux de données synchrones

Le modèle des réseaux de processus synchrones introduit par Lee et Masserschmitt [LM87], peut être considéré comme un sous modèle du modèle de Karp et Miller ou et ainsi des réseaux de processus de Kahn présenté dans la section 2.1. Dans ce modèle le nombre de jetons produits et consommés à chaque itération est fixe pour tous les processus du réseau.

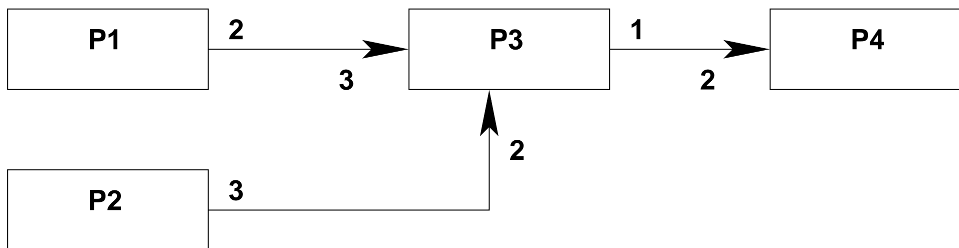


Fig. 2.7 – Flux de données synchrones

Définition Un flux de données synchrone de n nœuds et m canaux est défini par une matrice $P : n \times m$, avec $P[i,j] = w$ si le processus i écrit dans le canal j w jetons à chaque itération, et $P[i,j] = -r$ si le processus lit r jetons du canal j à chaque itération.

Partant des restrictions sur le modèle, un ordonnancement simple et efficace peut être appliqué au modèle. Il suffirait pour chaque canal qui a un processus producteur qui écrit w jetons à chaque itération, et un processus consommateur qui lit r jetons à chaque exécution de résoudre l'équation suivante : $x.w = y.r$. Pour l'ensemble du flux de données, ceci revient à trouver un vecteur qui vérifie la condition :

$$\overrightarrow{v}.P = \overrightarrow{0}$$

Cet ordonnancement statique permet d'exécuter un réseau de processus synchrones avec des canaux de communication bornés, évitant ainsi l'accumulation des jetons et les *deadlocks artificiels*. Par exemple la matrice définissant le réseau de processus de la figures 2.7 est la suivante :

$$P = \begin{pmatrix} 2 & 0 & -3 & 0 \\ 0 & 3 & -2 & 0 \\ 0 & 0 & 1 & -2 \end{pmatrix}$$

et la solution est :

$$\overrightarrow{v} = k \begin{bmatrix} 9 \\ 4 \\ 6 \\ 3 \end{bmatrix}$$

où k est un entier strictement positif quelconque, qui indique le nombre d'activations du processus. Une minimisation de la taille des canaux de communication peut être obtenue en fixant la valeur de k à 1.

La résolution de cette équation ne permet pas toujours d'obtenir une solution. Pour les graphes dits *inconsistants* (voir figures 2.8), qui contiennent un cycle (pas forcément orienté), la solution de l'équation $\overrightarrow{v}.P = \overrightarrow{0}$ est le vecteur nul. Un tel graphe n'est pas ordonnançable avec cette méthode, mais peut s'exécuter avec une mémoire non bornée. Dans la figure 2.8, c'est dans la deuxième FIFO entre P3 et P4 que les jetons risquent de s'accumuler.

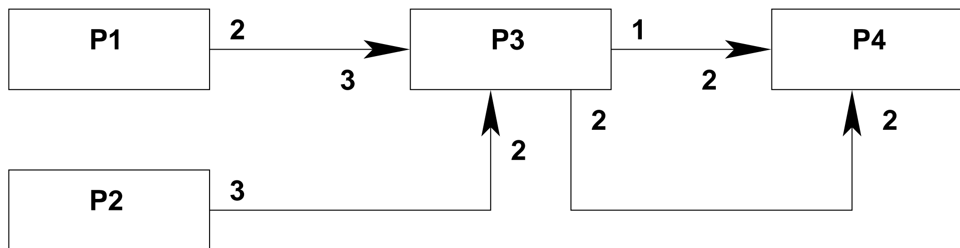


Fig. 2.8 – Réseaux de processus synchrones inconsistent

Résoudre l'équation décrivant le réseau de processus permet aussi de détecter les *deadlocks* causés par un nombre insuffisant de jetons initiaux. Mais un ordonnancement basé sur le nombre de répétitions des processus n'a d'intérêt que dans une architecture monoprocesseur. Dans une architecture multiprocesseur, ou distribuée, le modèle perd de son intérêt. La réduction des changements de contexte qui était le principal objectif de ces modèles n'a plus de sens dans de telles architectures.

2.4.3 Flux de données booléens

Les flux de données booléens de Buck [BL92, BL93, Buc93] représentent un modèle étendu du modèle des flux de données synchrones. Mais contrairement à ce dernier, le nombre de jetons lus ou écrits à chaque activation du processus n'est pas connu à la compilation pour tous les processus du réseau. En effet, pour certains canaux de communication, la valeur le nombre de jetons lus ou écrits dépend d'un autre canal de communication, et peut prendre deux valeurs.

Le canal de communication qui détermine le nombre de jetons lus/écrits est appelé « canal de contrôle », et le canal contrôlé « canal conditionnel ». A chaque activation du processus, un seul jeton (« jeton de contrôle ») est lu du canal de contrôle, et en fonction de la valeur de ce dernier (`TRUE`, `FALSE`), le nombre de jetons lus/écrits dans le canal conditionnel est déterminé.

Si le canal conditionnel est un canal d'entrée, le canal de contrôle est forcément un canal d'entrée. Par contre, si le canal conditionnel est un canal de sortie, alors le canal de contrôle peut être un canal d'entrée ou un canal de sortie. Si le canal de contrôle est un canal de sortie, un seul jeton est produit à chaque activation du processus.

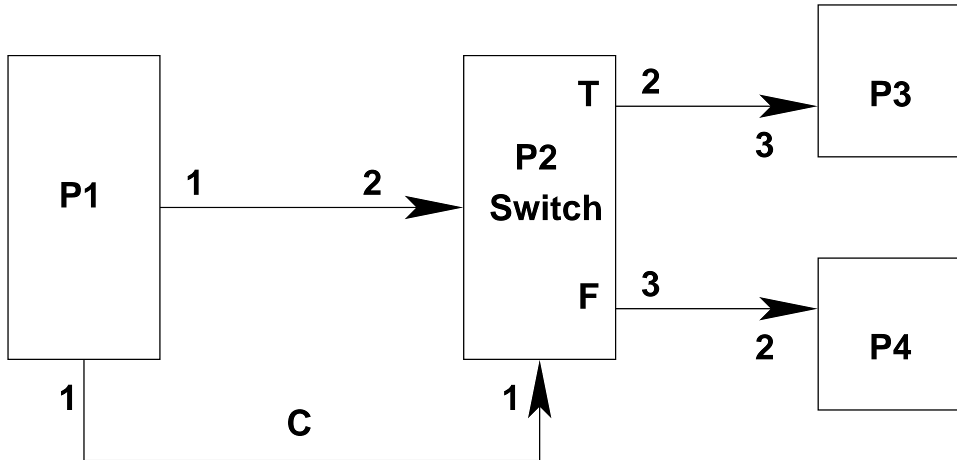


Fig. 2.9 – Flux de données booléen

La figure 2.9 représente un réseau de processus booléen. Le processus P_3 implémente la fonction « switch ». Si la valeur du jeton lu dans le canal de contrôle C est égale à `TRUE`, deux jetons sont écrits dans le canal reliant P_2 à P_3. Dans le cas contraire (valeur du jeton égale à `FALSE`), trois jetons sont écrits dans le canal reliant P_2 à P_4.

2.5 Conclusion

Comme nous l'avons vu dans la première partie de ce chapitre, le modèle des réseaux de processus de Kahn est un modèle **déterministe** pour représenter les applications parallèles à flux de données. Il est utilisé dans divers domaines d'application, tels le traitement au signal, la simulation de systèmes embarqués ou les applications vidéo.

Le modèle est assez large pour définir un ordonnancement efficace pour n'importe quelle application. C'est principalement pour cette raison, et en partant des caractéristiques des applications cibles, que des sous modèles ont été définis permettant ainsi d'avoir un ordonnancement efficace. Ces caractéristiques concernent la structure des fonctions de calcul des processus qui est généralement constituée d'une répétition d'une fonction élémentaire, et qui à chaque itération, consomme et produit un nombre fixe de données.

Dans le cadre de la simulation distribuée de type cyber-entreprise, on doit résoudre les problèmes suivants :

- La construction de l'application doit se faire par composition pour construire des applications plus complexes de façon modulaire. La notion de hiérarchie (qui permet de regrouper plusieurs composants dans une seule entités) permet d'augmenter le niveau de réutilisabilité.
- La distribution de l'application doit être masquée au programmeur, pour permettre

une plus grande facilité de programmation, et la réutilisabilité des composants indépendemment du contexte de simulation (distribué ou local).
- La définition des interfaces des composants est un des problèmes qui résultent du fait que les fournisseurs des composants sont différents.
- Les interactions entre composants doivent aussi être définies pour assurer le bon fonctionnement de l'ensemble des composants assemblés.

Le modèle des réseaux de processus de Kahn est un bon modèle pour simuler des systèmes embarqués. Outre les limitations des modèles des réseaux de processus synchrones et booléens et les autres variantes du modèle de Kahn, ces modèles ne sont pas adaptés à une exécution distribuée, car les contraintes liées à la synchronisation des différents processus disparaissent dans ce cas là. De plus, le modèle permet de résoudre les problèmes cités précédemment. En effet le modèle des réseaux de processus de Kahn :
- supporte la composition et la hiérarchie de façon naturelle ;
- définit implicitement les interfaces des composants. Ces interfaces sont représentées par les FIFOs qui sont le seul moyen de communication autorisé par le modèle ;
- définit implicitement les interactions (mécanismes de lecture/écriture dans les FIFOs) entre composants.

Pour résoudre le problème de la distribution, nous introduisons dans le chapitre suivant le modèle des réseaux de processus distribués. Notre approche est de définir un modèle qui procure cette répartition, tout en la masquant au maximum au programmeur. De plus, nous traitons aussi les aspects performances et équilibrage de charge.

3

Réseaux de Processus de Kahn Distribués

Sommaire

3.1 Réseaux de processus de Kahn distribués 60
3.2 FIFOs distribuées 65
3.3 Protocole de Transfert 68
3.4 Analyse des modes de transfert 71
3.5 Réseaux de processus, composant et génération automatique de code 84
3.6 Exemple d'application : Décodeur JPEG distribué 87
3.7 Conclusion 90

Restez simple : aussi simple que possible, mais pas plus.

A. Einstein.

Dans le domaine des systèmes embarqués, les industriels sont de plus en plus amenés à procéder à des simulations avant de passer à l'étape finale de conception. Lors de cette étape de simulation, les concepteurs des systèmes embarqués et les fournisseurs de composants sont confrontés à deux contraintes principales :

- la localité : les simulateurs de composants et les applications de simulation ne sont pas forcément au même endroit ;
- la propriété industrielle : le fournisseur doit protéger le simulateur du composant et le développeur doit protéger l'application simulée.

De là est née l'idée de la « cyber-entreprise » [BDD+03] qui permet d'interconnecter un ensemble de simulateurs, couplés via le réseau Internet, pour fournir un ensemble de services.

Le modèle des réseaux de processus de Kahn présenté dans le chapitre précédent, est l'un des modèles utilisé pour la modélisation d'applications de simulation de systèmes

embarqués. En se plaçant dans un contexte de simulation distribuée tel qu'il est défini dans une cyber-entreprise, ce chapitre présente les réseaux de processus distribués et leur implémentation en se basant sur l'architecture CORBA. Il se décompose en quatre sections. La première décrit le modèle des réseaux de processus distribués et sa conception. La deuxième présente les FIFOs distribuées et le déploiement de l'application. La troisième détaille le protocole de communication des FIFOs distribuées et le compare à d'autres protocoles. Enfin la quatrième section présente notre système de génération de code automatique.

3.1 Réseaux de processus de Kahn distribués

Le modèle des réseaux de processus de Kahn permet l'expression du parallélisme et la modélisation de diverses applications à flux de données, dont les applications de simulation pour systèmes embarqués. Ce modèle, resté longtemps comme un modèle de simulation locale, possède aussi des caractéristiques que les systèmes distribués doivent présenter, à savoir, la modularité, la composition, l'indépendance des différentes composantes (processus) et la séparation des aspects calcul et communication (gérés dans le modèle respectivement dans les processus et les canaux de communication).

Lors de la distribution, on est confronté au problème d'accès aux données distantes. Deux choix principaux sont possibles :

1. Mettre le canal de communication dans l'espace d'adressage d'un des deux processus, producteur ou consommateur, et accéder à distance au canal pour écrire ou lire les données.
2. Utiliser deux FIFOs (qu'on appellera « demi-FIFOs » ou « FIFOs distribuées ») pour remplacer la FIFO partagée par deux processus ne se trouvant pas sur le même espace d'adressage, et ajouter un mécanisme qui fait circuler les données de la demi-FIFO du processus producteur vers la demi-FIFO du processus consommateur.

La première solution a l'avantage d'être facile à mettre en œuvre, car il suffit de rediriger les opérations de lecture et d'écriture vers la FIFO distante. Néanmoins, aucun asynchronisme ne peut être utilisé dans ce cas, le processus doit attendre la fin de l'exécution de l'opération sur la FIFO distante pour pouvoir continuer son exécution. Cette solution s'avère très pénalisante dans un environnement distribué. Même avec des réseaux de communication qui sont de plus en plus rapides, les temps de transfert restent relativement grands par rapport aux temps d'accès mémoire. En plus, un certain déséquilibre est créé entre les deux processus lecteur et écrivain, puisqu'un des deux processus doit effectuer une requête réseau pour accéder à la FIFO alors que l'autre réalise un accès mémoire.

La deuxième solution permet un asynchronisme entre les traitements et les transferts de données d'une demi-FIFO à une autre. Le processus effectue l'opération sur la demi-FIFO qui se trouve dans son espace d'adressage, et seules la disponibilité des données (pour une opération de lecture) et la taille de la demi-FIFO (pour une opération d'écriture) déterminent le résultat de l'exécution. Par contre, elle nécessite un mécanisme de transfert

des données entre les deux demi-FIFOs. Ce transfert entre les deux demi-FIFOs peut être réalisé de façon asynchrone au processus de calcul, et indépendemment des opérations de lecture/écriture que ce dernier effectue sur la demi-FIFO. Cette solution, que nous avons adoptée pour l'implémentation de notre système, a l'avantage de garantir un recouvrement des communications par les calculs. Elle permet aussi d'augmenter la disponibilité des données, car le transfert des données entre les deux demi-FIFOs est indépendant des opérations qu'effectue le processus sur sa demi-FIFO.

Un problème persiste, celui de la charge mémoire des deux processus. Elle reste, comme dans la première solution, concentrée du côté d'un seul processus. Pour remédier à ce problème, nous proposons une approche basée sur des communications entre les FIFOs asynchrones et indépendantes des calculs, mais qui tient aussi compte de la charge mémoire de chaque demi-FIFO pour réguler les transferts, et répartir la charge entre les deux demi-FIFOs de manière plus équitable. Pour cela, des seuils sur le nombre de jetons sont utilisés.

Pour résumer, notre approche de distribution des réseaux de processus qui est basée sur les deux « demi-FIFOs » présente les avantages suivants :

1. Recouvrement des calculs par les communications : l'indépendance des communications et des calculs permet d'effectuer les transferts de données entre les deux demi-FIFOs de façon asynchrone par rapport au traitement dans les processus. De cet asynchronisme résulte un recouvrement des communications par les calculs.
2. Équilibrage de la charge mémoire entre les deux demi-FIFOs : l'utilisation de seuils sur le nombre de jetons de la demi-FIFO permet de contrôler les transferts de données entre les deux demi-FIFOs.
3. Augmentation de la disponibilité des données : en plus de l'équilibrage de la charge mémoire, l'utilisation de seuils permet aussi d'augmenter la disponibilité des données des processus consommateurs. Ainsi la demi-FIFO du consommateur possède un seuil minimal qui lui permet d'anticiper les demandes de données pour éviter le phénomène de famine.

On présentera dans ce qui suit le modèle distribué basé sur les FIFOs distribuées. L'implémentation proposée en C++ [Str01] utilise la syntaxe de YAPI, pour procurer une compatibilité qui garantit la réutilisabilité des applications existantes, et utilise la technologie CORBA pour fournir un système hétérogène, indépendant des plates formes et des langages de programmation.

3.1.1 Représentation mathématique

Un réseaux de processus distribué peut être représenté de manière identique aux réseaux de processus. La distribution du modèle introduit une séparation des canaux de communication dits « distribués » en deux canaux, un côté processus producteur, et l'autre côté processus consommateur. Ces deux « demi FIFOs » doivent contenir successivement les mêmes données dans le même ordre. Un opérateur définissant la distribution de la FIFO distribuée en deux demi FIFOs peut être ajouté au formalisme. Par exemple le réseau de processus distribué de la figure 3.1 peut être représenté par les équations suivantes :

$$(F_3, F_4) = P_1(F_1, F_2)$$

$$F_5 = P_2(F_3, F_4)$$

$$F_7 = P_3(F_6)$$

$$F_8 = id(F_5)$$

$$F_9 = id(F_7)$$

$$(Y_1, Y_2) = P_4(F_8)$$

$$Y_3 = P_5(F_9)$$

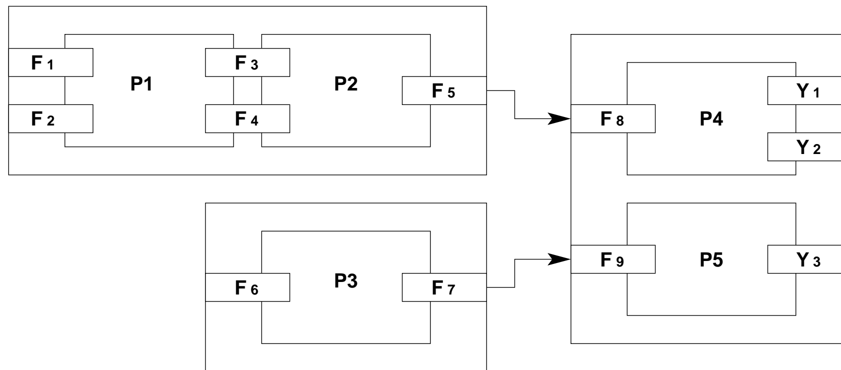


FIG. 3.1 – Exemple d'un réseau de processus distribué

3.1.2 Conception

La conception du réseau de processus a été réalisée en ayant pour objectifs :

1. de permettre aux programmeurs de développer leurs réseaux de processus rapidement et efficacement : les communications entre les deux demi FIFOs doivent être recouvertes par les calculs, et le parallélisme exprimé par le modèle doit être exploité.
2. de garder la compatibilité avec la bibliothèque YAPI (pour l'implémentation en C++) : cette bibliothèque développée par Philips [dKES+00] implémente le modèle de réseaux de processus pour une exécution locale, et garder la compatibilité avec l'implémentation distribuée implique une transparence complète de la distribution. Pour arriver à cela, il faut rendre transparents :
   – les objets CORBA qui gèrent les communications ;
   – les types de données de la FIFO et les interfaces de communication car les « templates » C++ utilisés dans YAPI n'existent pas dans la norme CORBA ;
   – les liaisons des demi-FIFOs.
3. de masquer les transferts de données au programmeur, et ainsi le décharger de la gestion des communications : seules les opérations *read* et *write* sont visibles, et toutes les opérations d'envoi ou de demande de données entre les demi-FIFOs doivent être transparentes.

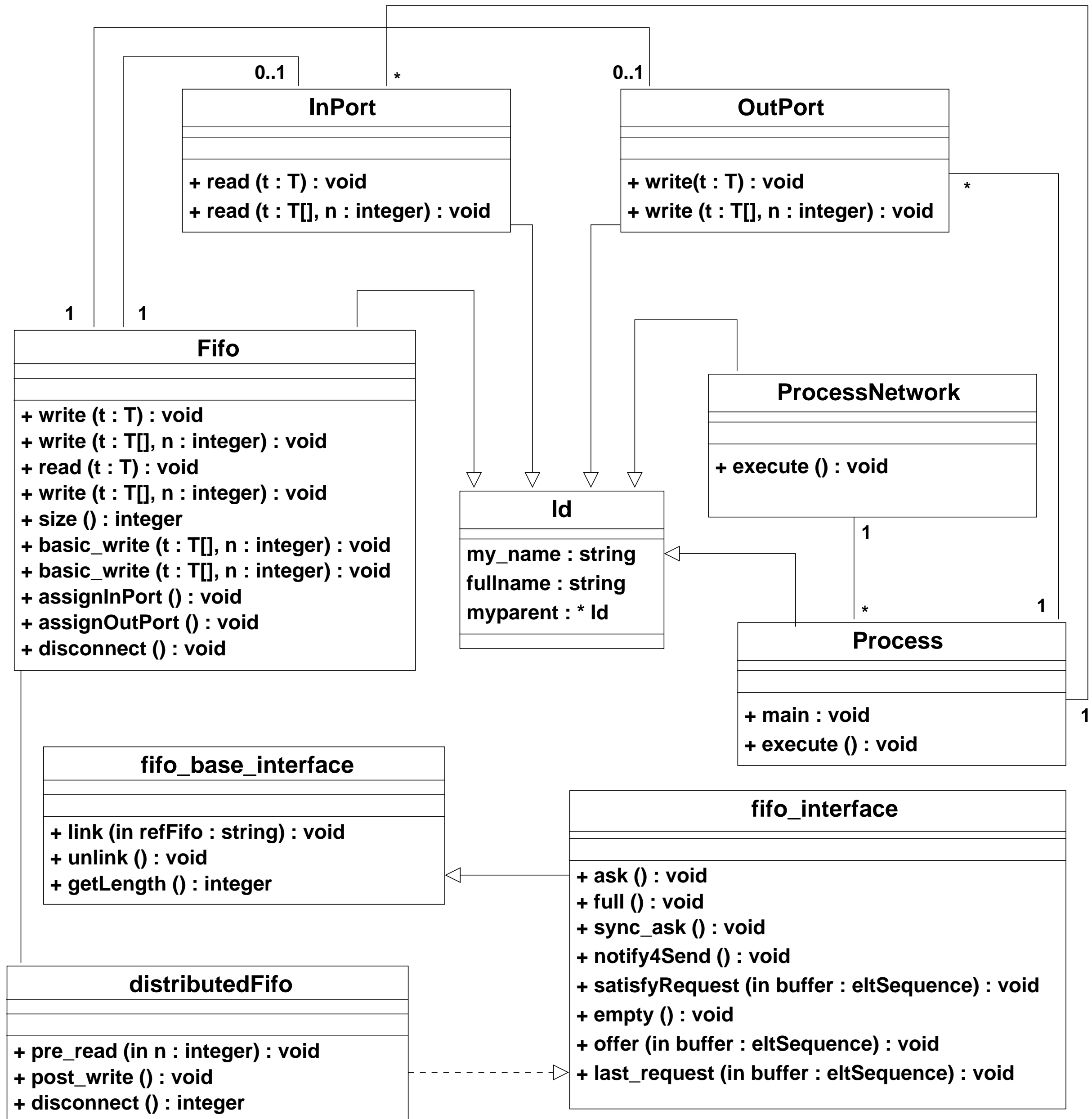


Fig. 3.2 – Diagramme de Classe

L'implémentation est orientée objet, et les communications sont gérées sans l'intervention du programmeur, qui peut configurer s'il le désire les paramètres de transfert de données.

Le diagramme de classes est représenté sur la figure 3.2. La classe de base pour toutes les autres classes est la classe *Id*. Cette classe sert à sauvegarder le nom de l'objet et une référence vers son parent. Elle permet ainsi d'obtenir le nom complet de l'objet (concaténation des noms des parents et du nom de l'objet). Les ports sont représentés par deux classes : *OutPort* et *InPort*. Chaque port est associé à une seule FIFO, pour respecter les règles de Kahn (au plus un seul processus lecteur et un seul processus écrivain pour une FIFO). Les *InPort* et *OutPort* sont utilisés respectivement, pour lire et écrire dans les FIFOs. La classe *Fifo* représente le canal de communication. Comme il n'y a aucune différence entre les FIFOs locales et les FIFOs distribuées pour le programmeur, une seule classe représente les deux types de FIFO, et c'est à l'exécution que le support d'exécution détermine si la FIFO est distribuée ou pas selon le nombre de port qui lui sont rattachés.

La classe *Process* définit les processus. Pour exploiter le parallélisme du modèle, chaque processus est exécuté dans un thread. La fonction principale du processus est dans la fonction *main* qui est appelée automatiquement lors du lancement du processus. La classe *ProcessNetwork* représente le réseau de processus. C'est au sein de cette classe que sont créés les FIFOs et les processus. Les réseaux de processus peuvent être hiérarchiques, un objet *ProcessNetwork* peut créer d'autres *ProcessNetwork*.

Enfin la classe *distributedFifo* représente la FIFO distribuée proprement dite. Cette classe implémente deux interfaces : la première *fifo_base_interface* fournit les méthodes générales que doivent implémenter toutes les FIFOs de l'application, indépendemment du type du jeton de la FIFO. La seconde *fifo_interface* fournit les méthodes spécifiques pour la communication avec l'autre FIFO distribuée liée. Pour être générique et pouvoir supporter les *templates*, le type IDL dynamique *Any* a été utilisé.

3.1.3 Ordonnancement des processus

Parks [Par95] a défini deux exigences qu'un ordonnancement de réseaux de processus doit satisfaire à savoir :

1. Une exécution complète du réseau : ce qui correspond à atteindre le point fixe de l'équation décrivant le réseau. En général, une exécution complète correspond à un *deadlock* [19], à la fin d'exécution de tout les processus, ou à une exécution infinie du réseau pour les programmes s'exécutant indéfiniment.
2. Une exécution bornée pour pouvoir exécuter le réseau avec une mémoire physique limitée.

En plus de ces deux conditions, et du fait de la distribution du modèle sur plusieurs ressources de calcul, l'ordonnancement des réseaux de processus distribués impose de nouvelles exigences liées à la sémantique du modèle, à la synchronisation et aux ressources disponibles :

1. Sémantique : la distribution et l'ordonnancement doivent garder la sémantique du modèle, et ainsi préserver son déterminisme.

[19] à ne pas confondre avec un *deadlock* artificiel

2. Synchronisation : l'exécution concurrente et distribuée du réseau sur différents sites ne possédant pas une horloge commune impose des contraintes de synchronisation entre les événements internes et externes, et les événements externes entre eux.
3. Exploitation de l'hétérogénéité des ressources : le parallélisme exprimé par le modèle peut être mieux exploité dans des architectures particulières (Multiprocesseur par exemple), et un ordonnancement efficace doit tenir compte de ces architectures.

Ordonnanceur des processus distribués L'ordonnanceur est composé de deux parties, une partie qui gère l'exécution et l'arrêt des processus du réseau, et une partie responsable de l'initialisation et la gestion de l'environnement externe au réseau de processus. Ainsi l'initialisation et la gestion de l'environnement CORBA sont rendues transparentes au programmeur.

3.2 FIFOs distribuées

Aucune différence n'existe pour le programmeur entre les FIFOs distribuées et les FIFOs locales. Ceci est assuré par l'encapsulation des FIFOs distribuées (les objets CORBA) dans les FIFOs. Les figures 3.3 et 3.4 montrent la structure des objets FIFOs. Chaque FIFO peut être associée à deux ports au plus, un port d'entrée et un port de sortie. Ces ports qui seront utilisés par les processus servent respectivement à lire et à écrire dans la FIFO. Ils servent également à garantir le respect des règles du modèle, à savoir au plus un seul lecteur et un seul écrivain pour chaque FIFO. L'utilisation des ports fournit l'information nécessaire au support d'exécution pour déterminer si la FIFO est distribuée ou pas. Une FIFO distribuée est représentée par deux demi FIFOs, une côté processus lecteur, et une côté processus écrivain. Ces deux demi-FIFOs n'ont qu'un seul port, un port d'entrée pour celle du lecteur et un port de sortie pour la FIFO de l'écrivain. L'utilisation de cette propriété permet au support d'exécution d'activer l'objet CORBA correspondant à la FIFO distribuée seulement dans les demi-FIFOs. Ainsi la distribution des FIFOs est complètement implicite au programmeur.

L'interface IDL des FIFOs distribuées est la suivante :

```
typedef sequence<any> eltSequence;

interface fifo_base_interface
{
  void link(in string refFifo);
  void unlink();
  unsigned long getLength();
};

interface fifo_int : fifo_base_interface
{
  oneway void ask();
  oneway void full();
```

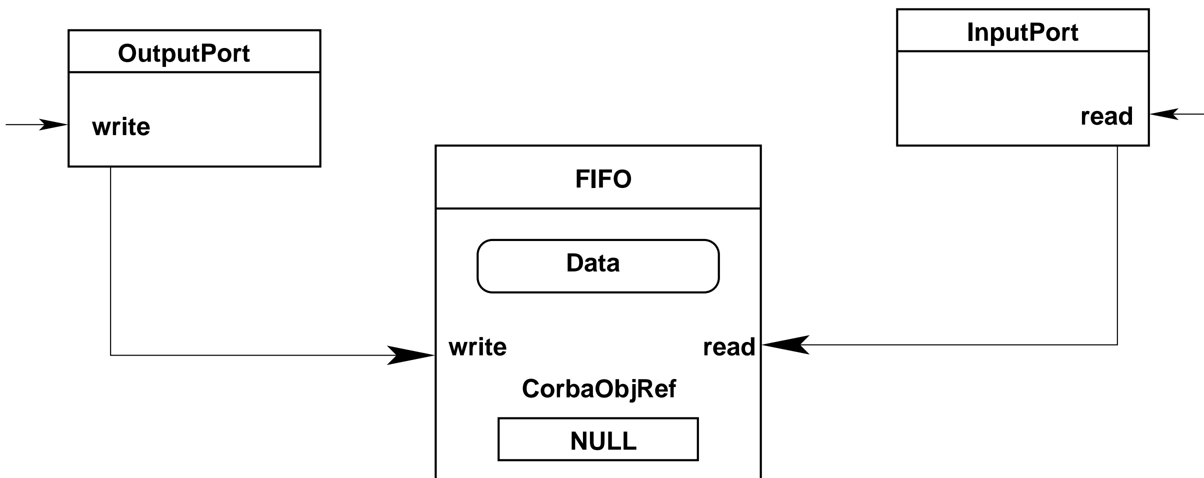


Fig. 3.3 – Structure d'une FIFO locale

```
  boolean offer(in eltSequence buffer);
  oneway void satisfyRequest
              (in eltSequence buffer);
  void empty();
  void sync_ask(in string refFifo,
              inout eltSequence buffer);
  bool notify4Send();
};
```

L'interface IDL décrite ci-dessus définit les méthodes nécessaires à la FIFO pour l'initialisation, les interactions avec l'autre FIFO, et la récupération d'information.

- Les méthodes `link` et `unlink` permettent respectivement de créer ou de supprimer un lien entre deux FIFOs. Le liaison se fait par passage de référence des objets CORBA.
- Les méthodes `ask` et `satisfyRequest` sont utilisées respectivement par les FIFOs d'entrée et les FIFOs de sortie pour demander et envoyer des données.
- La méthode `sync_ask` est la version synchrone de la requête `ask`.
- Les méthodes `full` et `empty` servent à indiquer l'état de la FIFO.
- Les méthodes `notify4Send` et `offer` sont utilisées par les FIFOs de sortie pour envoyer les données respectivement avec et sans notification.

Pour échanger les données entre les FIFOs distribuées, nous avons implémenté un protocole de communication basé sur des seuils sur le nombre de jetons disponibles dans la FIFO. Ce protocole qui sera détaillé dans la section 3.3 est un protocole hybride data-driven/demand-driven.

Type de données et génération de code

Pour des raisons de généricité, le type dynamique *Any* du langage IDL a été utilisé. Ce type permet d'envoyer et de recevoir des valeurs dont le type n'est pas connu à la compilation. Il s'avère très utile dans plusieurs situations parmi lesquelles on retrouve celle

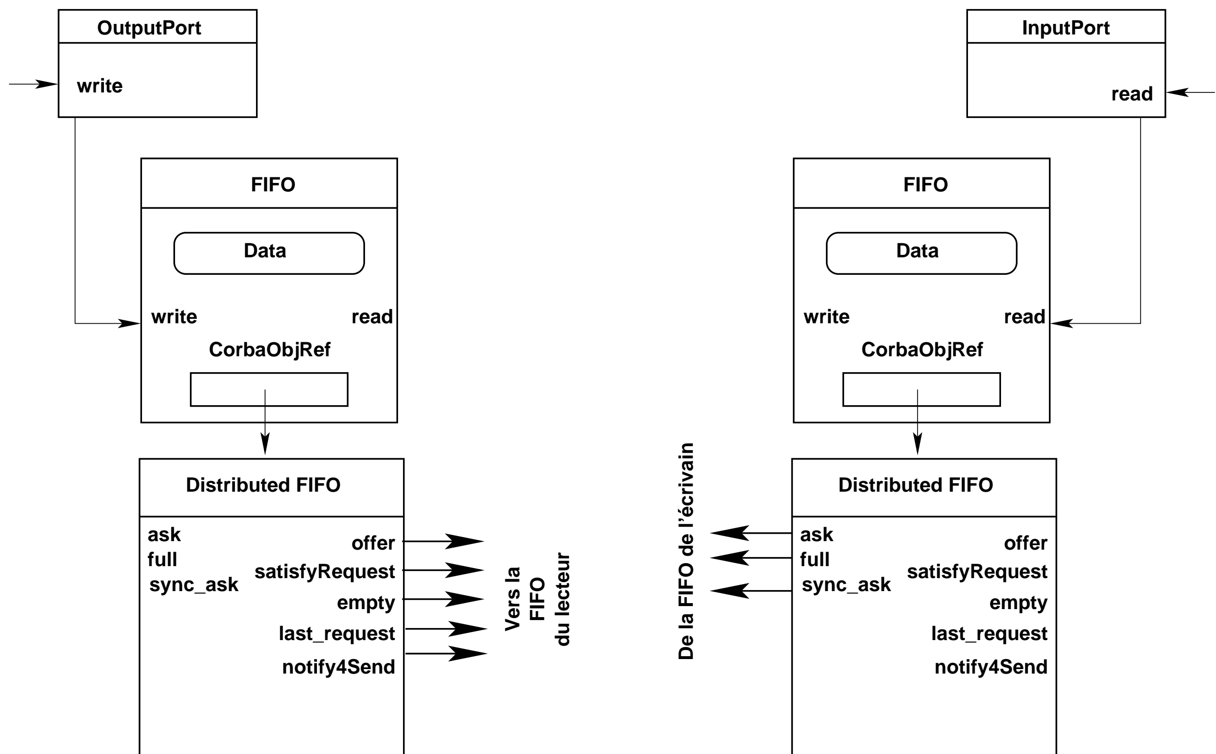


FIG. 3.4 – Structure des FIFOs distribuées

du service d'événement de CORBA qui doit transporter des données dont le type IDL est inconnu par ce service. L'utilisation de ce type permet ainsi de créer des « templates » de FIFOs distribuées.

D'un autre côté et pour des raisons de performance, un générateur de code a été développé pour produire le code source correspondant à des FIFOs manipulant un seul type de données.

3.2.1 Déploiement et connection

Pour controler les processus, une console interactive a été développée. C'est un programme qui contrôle les liaisons entre les processus et leurs exécution. L'achèvement de ces tâches est effectué à l'aide d'un langage de commande qui sera présenté plus loin. La présence d'un programme de gestion est contraire à l'aspect peer-to-peer d'un tel système réparti. Néanmoins, le rôle de la console est minimal, car tous les traitements dans les processus et les communications entre FIFOs distribuées se font directement indépendemment de cette console. En plus, l'utilisation de cette console procure une flexibilité plus grande dans le choix des liaisons, un contrôle dynamique des processus, et une plus grande liberté dans le développement des différentes parties de l'application.

Pour pouvoir contrôler les FIFOs présentes dans le système, ces dernières doivent se faire connaitre auprès de la console en s'y connectant dès le lancement des traitements. La console joue alors le rôle d'un service de nommage spécialisé.

3.3 Protocole de Transfert

Dans cette section, nous nous intéressons aux échanges de données entre les FIFOs. L'implémentation du protocole de transfert a été réalisée en ayant pour but la minimisation des communications entre les FIFOs, le recouvrement des communications par les calculs et la minimisation de la charge des processus.

Le protocole de communication est complètement distribué, sans point central de communication contrairement à Jade [WWM99]. Chaque FIFO distribuée gère la transmission de ces données indépendemment des autres processus du réseau.

Les FIFOs distribuées peuvent être de deux types : les FIFOs qui sont côté processus producteur qu'on appellera FIFOs de sortie, et les FIFOs qui sont du côté processus consommateur qu'on appellera FIFO d'entrée. Pour gérer les communications, deux seuils sur le nombre d'éléments dans la FIFOs ont été définis :

- Un seuil maximal (pour les FIFOs de sortie) pour indiquer qu'un envoi d'une partie de la FIFO est nécessaire pour éviter la surcharge ;
- un seuil minimal (pour les FIFOs d'entrée) pour indiquer qu'il est nécessaire d'envoyer une requête de demande de données à la FIFO de sortie qui lui est reliée.

L'utilisation de seuils permet de mieux gérer la taille des FIFOs pour éviter le phénomène de famine du côté consommateur et le phénomène de surcharge du côté producteur. En plus, la charge mémoire est mieux partagée entre les deux demi-FIFOs.

Dans les trois paragraphes suivants, nous présenterons notre protocole de communication hybride. Nous présenterons les stratégies demand-driven, data-driven puis la stratégie hybride. On suppose que les deux FIFOs F_{out} et F_{in} sont liées, la première étant la FIFO de sortie, et la seconde la FIFO d'entrée. Le déclenchement des opérations de communication est implicite comme décrit plus loin dans la section 3.3.4 et dépend essentiellement du nombre courant de jetons dans les FIFOs, et des seuils définis pour chaque FIFO. Pour des raisons de synchronisation, le protocole demand driven est avec notification, et le protocole data driven est sans notification.

3.3.1 Demand driven

Le transfert en mode demand driven se fait avec notification. Dès que le nombre d'éléments dans la FIFO F_{in} passe au dessous du seuil minimal défini pour cette dernière, une requête `ask` est envoyée à la FIFO F_{out} qui construit un buffer d'une taille donnée (qui dépend de certains paramètres tel le nombre d'éléments dans la FIFO, et la taille maximale autorisée pour les buffers), et répond par une requête `satisfyRequest`. Le mode demand driven sans notification n'a pas été utilisé, car il présente l'inconvénient d'être synchrone. Le déclenchement des requêtes se fait par le processus et de telles requêtes synchrones bloqueraient le traitement jusqu'à la réception des données.

3.3.2 Data driven

Quand le transfert est data-driven, il se fait sans notification. Quand le nombre d'éléments de la FIFO F_{out} dépasse son seuil maximal, celle-ci envoie des données (par l'intermédiaire d'une requête `offer`) à la FIFO F_{in} pour diminuer sa charge. Les requêtes `offer`

étant synchrones (car elles retournent un résultat booléen), un thread séparé se charge de la gestion des communications dans ce mode.

3.3.3 Hybride

Pour éviter la surcharge de la FIFO F_{in} (par l'envoi de plusieurs requêtes `offer` successives, cas qui peut se produire si les fréquences des opérations *write* dans la FIFO F_{out} sont plus élevées que celles des opérations *read* dans la FIFO F_{in}), la requête retourne un résultat. Ce résultat indique si la FIFO réceptrice peut encore recevoir ou non des données de la FIFO F_{out}. Si la FIFO F_{in} est pleine, la FIFO F_{out} n'enverra plus de données jusqu'à la réception d'une requête de demande de la FIFO F_{in}.

Terminaison Quand une FIFO F_{in} envoît une requête de demande de données *ask* à la FIFO F_{out}, les données demandées peuvent ne pas être disponibles au moment de la requête. Dans ce cas là, la réponse de la FIFO F_{out} ne contient aucune donnée, en emmettant le signal *empty*. Pour éviter la surcharge du réseau, la FIFO F_{in} n'envoît plus aucune requête de demande, jusqu'à ce que la FIFO F_{out} atteigne son seuil maximal et envoie une requête *offer*, permmettant ainsi de réactiver les communications entre les deux demi-FIFOs. Cette solution ne résout pas tous les problèmes, car dans le cas où la FIFO F_{out} n'atteint jamais son seuil maximal (fin de l'exécution par exemple), la FIFO F_{in} risque de ne jamais recevoir les données demandées, et le processus correspondant ne finira pas son exécution, entrainant une exécution incomplète du réseau entier. Pour résoudre ce problème tout en évitant la surcharge du réseau par la répétition des requête de demande, quand la FIFO F_{out} envoit un signal *empty*, elle envoît des données à la FIFO F_{in} dès leur disponibilité, sans attendre le dépassement du seuil maximal. Ceci permet ainsi à de réactiver les communications, en autorisant la FIFO F_{in} à réemmettre des demandes vers la FIFOs F_{out}.

3.3.4 Déclenchement des communications et équilibrage de charge

Les communications sont rendues implicites, sans intervention du programmeur. Seuls les liaisons entre les FIFOs déterminent les échanges de données entre processus. Pour les FIFOs distribuées, les communications sont déclenchées de deux manières :

A l'intérieur du processus : suivant que la FIFO distribuée est une FIFO d'entrée ou une FIFO de sortie le déclenchement se fait de deux façons :
- lors de l'exécution d'une opération de lecture `read` (figure 3.5), un pré-traitement (opération `pre-read`) est déclenché. Ce pré-traitement permet de demander éventuellement (si le nombre de jetons dans la FIFO est inférieur au seuil minimal) des données à la FIFO précédente.
- D'un autre côté, lors de l'exécution d'une opération d'écriture `write` (figure 3.6), un post-traitement est effectué pour éventuellement (si le nombre de jetons dans la FIFO est supérieur au seuil maximal) envoyer des données à la FIFO suivante ;

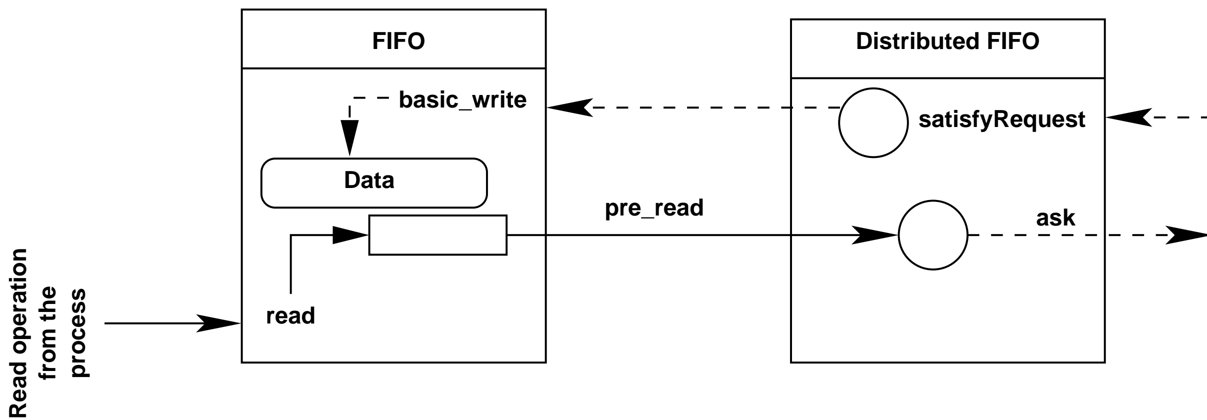


FIG. 3.5 – Les opérations read et pre-read dans les FIFOs distribuées

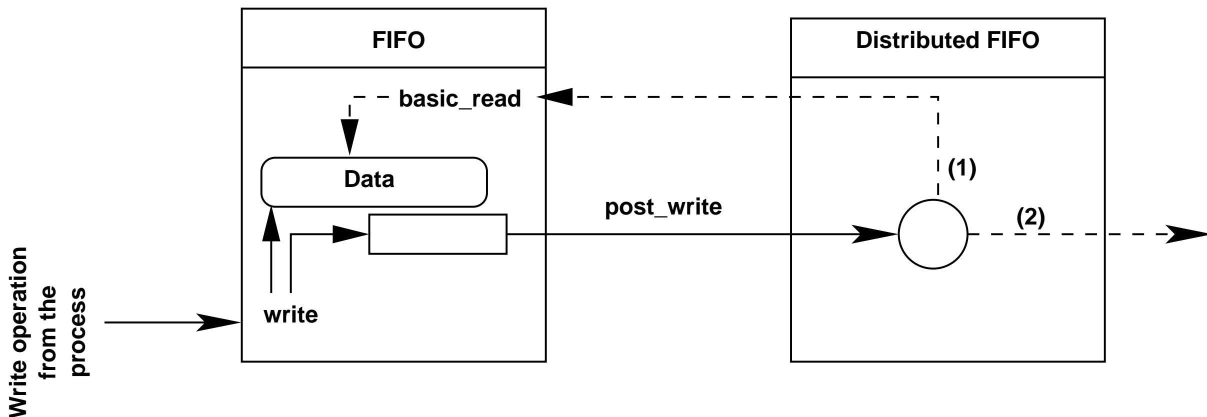


FIG. 3.6 – Les opérations write et post-write dans les FIFOs distribuées

A l'extérieur du processus : ceci se fait directement par des appels distants de la FIFO liée qui demande ou envoie des données.

Pour pouvoir adapter les communications à des applications particulières, des méthodes de configuration des communications ont été implémentées pour permettre au programmeur de régler certains paramètres : seuils minimaux et maximaux, longueur maximale des buffers envoyés, taille maximale des FIFOs.

```
void setMinTh(unsigned int _minTh);
void setMaxTh(unsigned int _maxTh);
void setCapacity(unsigned int _capacity);
void setBufferMaxLength(unsigned int _bufMax);
```

Résumé Les figures 3.7[20] et 3.8 représentent les diagrammes d'états du protocole de communication tel qu'il est considéré par les deux demi FIFOs. Dans ces figures, les

[20]La transition de Empty à Send permet de garantir la terminaison et l'exécution complète du traitement dans le cas où la FIFO de sortie n'atteint pas son seuil maximal.

événements sont des méthodes invoquées de l'autre demi FIFO connectée. Les méthodes `write` et `read` sont bloquantes, quand la FIFO est pleine pour le `write`, et quand la FIFO est vide pour le `read`. Comme la taille des données échangées est bornée par une taille maximale de paquet, le dépassement de la capacité de la demi FIFO d'entrée est détecté après chaque réception de données. Dans ce cas, la demi FIFO d'entrée signale à la demi FIFO de sortie qu'elle ne peut plus recevoir de paquet de données de taille maximale, par envoi d'un signal *full* ou par retour d'une réponse *false* à une méthode *offer*.

3.4 Analyse des modes de transfert

Avant d'analyser les temps de traitement et de communication dans les deux modes *demand-driven* et *data driven*, on va définir ceratins paramètres :

- les jetons produits sont notés par la suite d'éléments $y_1, ..., y_n$;
- les jetons consommés sont notés par la suite d'éléments $x_1, ..., x_n$;
- t_{ci} : temps (date) de consommation (traitement côté client/consommateur) du jeton x_i ;
- t_{ri} : temps (date) de la tentative de lecture par le consommateur du jeton x_i ;
- τ_{c0} : durée de consommation (traitement côté client) d'un jeton ;
- t_{pi} : temps (date) de production (traitement côté serveur/producteur) du jeton y_i ;
- τ_{p0} : durée de production (traitement côté serveur/producteur) d'un jeton ;
- τ_{t0} : durée de transfert d'un jeton ;
- τ_{notif} : durée d'une requête asynchrone sans paramètre (notification) ;
- $\tau_{t(p)}$: durée de transfert d'une séquence de p éléments ;
- α, β : les coéfficients de l'équation donnant le temps de transfert d'une séquence de longueur n ($\delta(n) = \beta + \alpha.n$) ;
- p : longueur des paquets transférés dans le cas de vectorisation des communication.

On définit aussi les trois fonctions suivantes :

- La fonction $SV_{demand}(i)$ définit le retard entre le moment de la production d'un jeton et le moment de sa lecture dans un mode demand-driven.

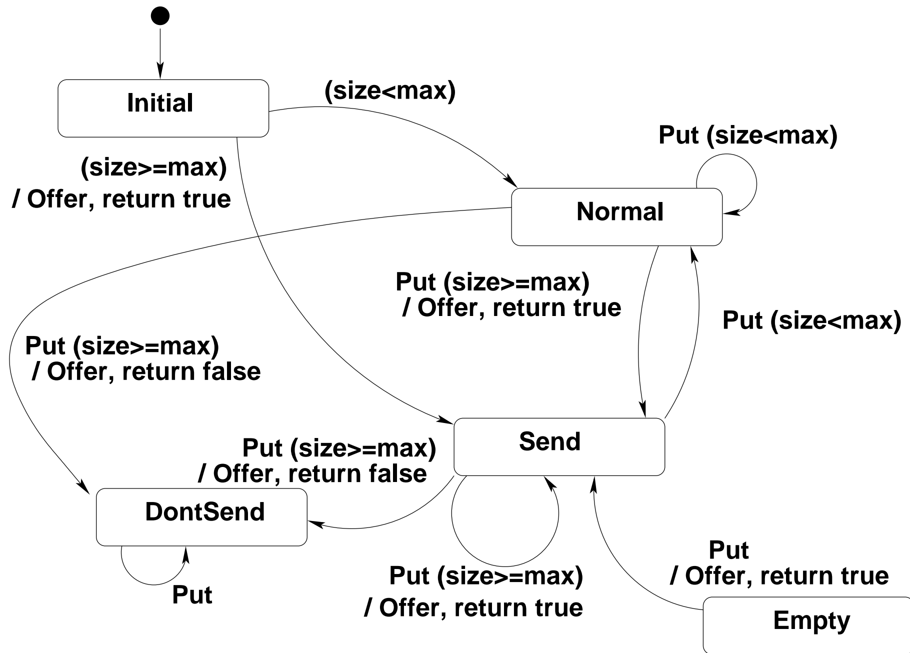
$$SV_{demand}(i) = \begin{cases} 0 & si\, t_{pi} - t_{ri} \leq 0 \\ t_{pi} - t_{ri} & sinon \end{cases}$$

- La fonction $SV_{data}(i)$ définit le retard entre le temps de la production d'un jeton et le temps de sa lecture dans un mode data-driven sans vectorisation.

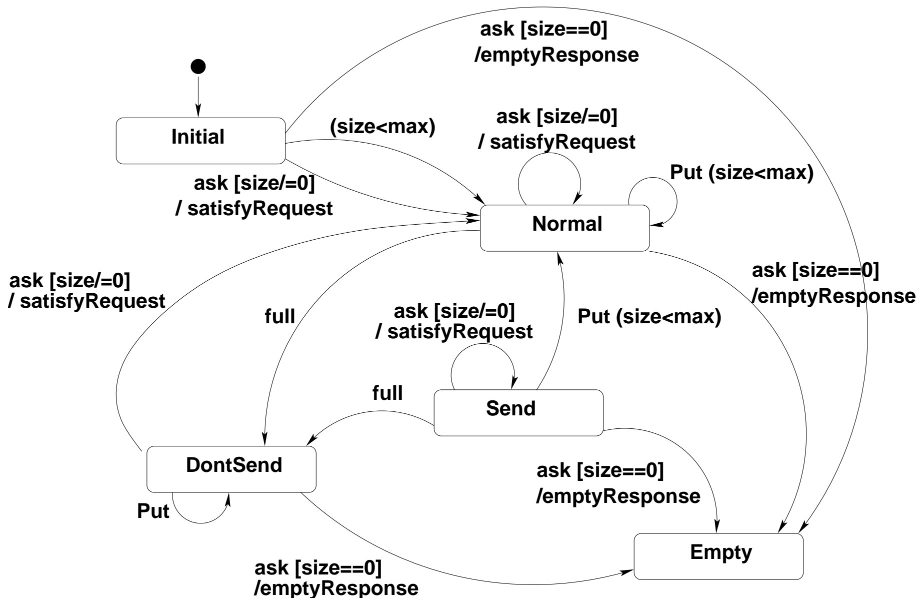
$$SVW_{data}(i) = \begin{cases} 0 & si\, t_{pi} + \tau_{t0} - t_{ri} \leq 0 \\ t_{pi} + \tau_{t0} - t_{ri} & sinon \end{cases}$$

- La fonction $SV_{data}(i)$ définit le retard entre le temps de la production d'un jeton et le temps de sa lecture dans un mode data-driven avec vectorisation.

$$SV_{data}(i) = \begin{cases} 0 & si\, t_{pi} + \tau_{t(p)} - t_{ri} \leq 0 \\ t_{pi} + \tau_{t(p)} - t_{ri} & sinon \end{cases}$$



Vue demand-driven



Vue data-driven

FIG. 3.7 – Diagramme d'état de la FIFO de sortie

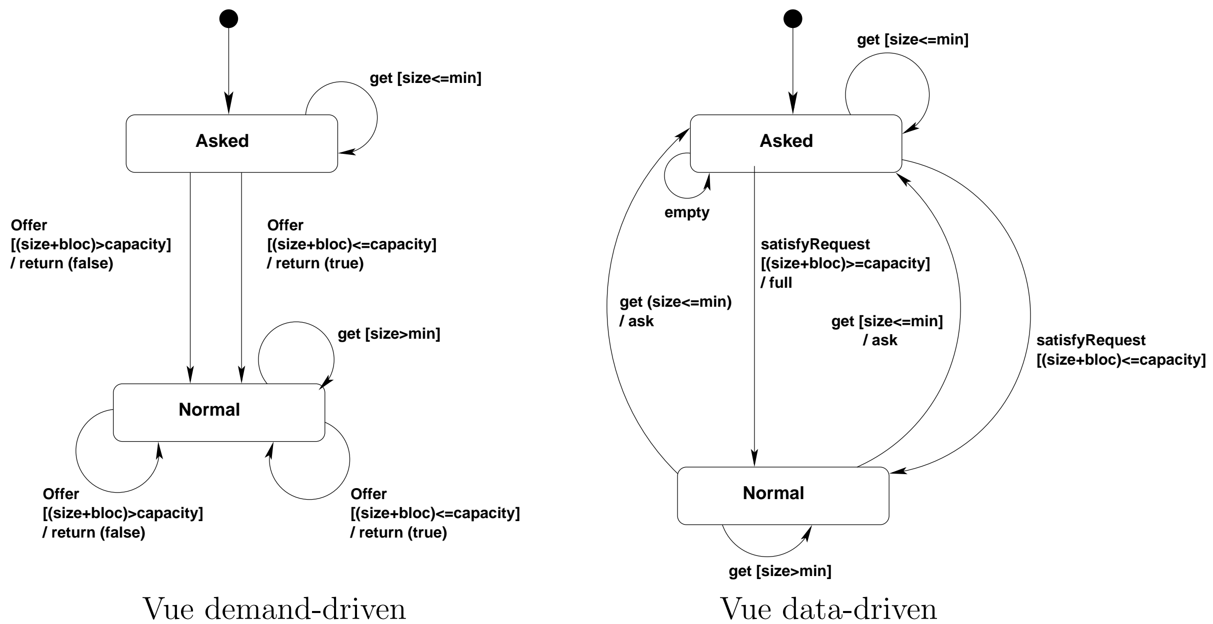


FIG. 3.8 – Diagramme d'état de la FIFO d'entrée

3.4.1 Vectorisation des communications

La vectorisation des communications est un facteur très important dans l'optimisation des performances. Par contre, dans un système distribué où les communications sont implicites et transparentes, la vectorisation pose le problème de terminaison comme il sera vu plus loin. Néanmoins, et malgré l'augmentation des vitesses des réseaux actuels, le surcoût de l'établissement de la liaison entre deux points distants, par rapport au temps de transfert des données (surtout à grain fin) reste considérable.

Résultats et analyses

Pour tester l'impact de la vectorisation des communications, une application classique de type client/sevreur a été utilisée. Aucune traitement n'est effectué dans les deux côtés, et les deux applications sont au régime continu. L'interface IDL utilisé pour ces tests est la suivante :

```
typedef sequence<short> shortSequ;
typedef sequence<double> doubleSequ;

interface perfInterface
{
  // NULL call
  void nullCall();

  // transfer short/double
  void giveShort(in short a);
  void giveDouble(in double a);
```

```
  // retrieve a short/double
  short getShort();
  void outShort(out short a);
  double getDouble();
  void outDouble(out double a);

  // transfer a sequence of short/double
  void giveShortSequ(in shortSequ sequ);
  void giveDoubleSequ(in doubleSequ sequ);

  // retrieve a sequence of short/double
  shortSequ getShortSequ(in long n);
  doubleSequ getDoubleSequ(in long n);
};
```

Les premiers tests ont été effectués sur deux machines de type PC reliées par un réseau ethernet à 10Mb/s, celle abritant le serveur est équipée d'un Pentium 4 à 2,4 Ghz, l'autre (abritant le client) est équipée d'un Duron à 1,3 Ghz.

Les deux figures 3.9 et 3.10 montrent les temps de transferts des sequences de *short* et *double* respectivement. On remarque que les temps de transferts sont linéaires par rapport à la longueur de la séquence. Ce temps peut être approximé par une fonction de la forme $\delta(n) = \beta + \tau n$, où δ est le temps en ms, et n la longueur de la séquence (nombre d'éléments). Pour les transferts de séquences de *short*, la fonction est la suivante :

$$\delta(n) = 0,00153n + 93,7$$

Pour les transferts de séquences de *double*, la fonction est la suivante :

$$\delta(n) = 0,005n + 736.4$$

Il apparaît clairement, d'après les valeurs de β et τ, que pour obtenir un protocole de communication efficace, la vectorisation des communications joue un rôle primordial dans l'amélioration des performances. Un protocole efficace sans vectorisation n'est envisageable que pour des communications à très gros grain (transfert de fichiers ou d'images volumineux par exemple).

Demand-driven

Le mode demand driven peut être avec ou sans notification, et avec ou sans vectorisation. Les temps de traitement dans ce mode peuvent être définis, selon les cas, par :

Sans vectorisation/Avec notification

$$Time(n) = n.(\tau_{c0} + \tau_{notif} + \tau_{t0}) + \sum_{i=1}^{n} SV_{demand}(i)$$

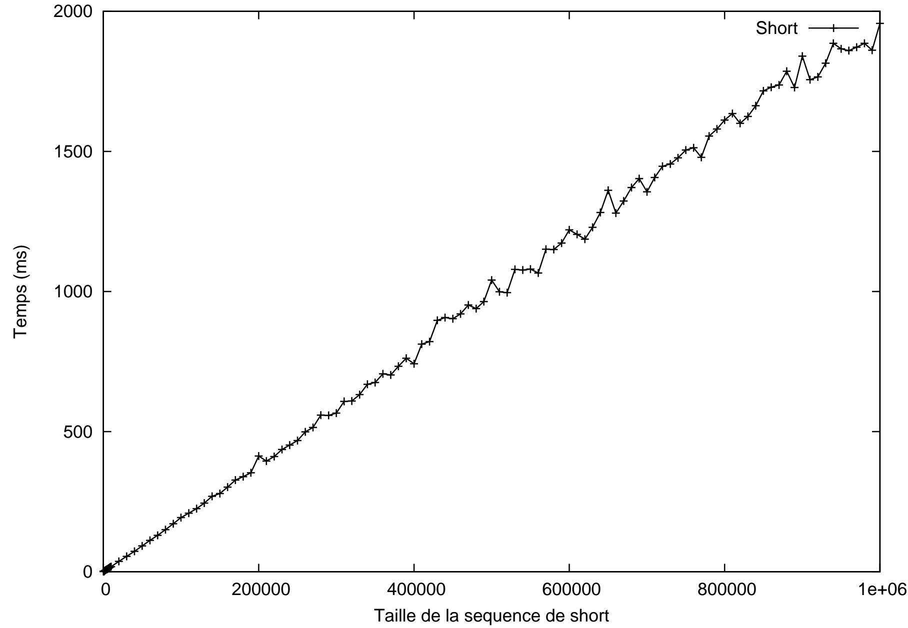


FIG. 3.9 – Temps de transfert de sequences de short

Sans vectorisation/Sans notification

$$Time(n) = n.(\tau_{c0} + \tau_{t0}) + \sum_{i=1}^{n} SV_{demand}(i)$$

Avec vectorisation/Sans notification

$$Time(n) = n.(\tau_{c0} + \alpha + \frac{\beta}{p}) + \sum_{i=1}^{\frac{n}{p}} SV_{demand}(i * p)$$

Avec vectorisation/Avec notification

$$Time(n) = n.(\tau_{c0} + \alpha + \frac{\beta + \tau_{notif}}{p}) + \sum_{i=1}^{\frac{n}{p}} SV_{demand}(i * p)$$

Évaluation du mode demand-driven Le mode demand-driven est un mode simple et facile à implémenter. Il peut être vu comme un mode client/serveur, avec des *callbacks* dans le cas du mode avec notification. En dehors du cas sans vectorisation, pénalisant dans tous les cas (demand et data driven), le principal désavantage de ce mode est le blocage du processus demandeur pendant les communications. Cette synchronisation limite considérablement les performances de ce mode, car même si les communications sont vectorisées, il n'existe aucun recouvrement des calculs par les communications.

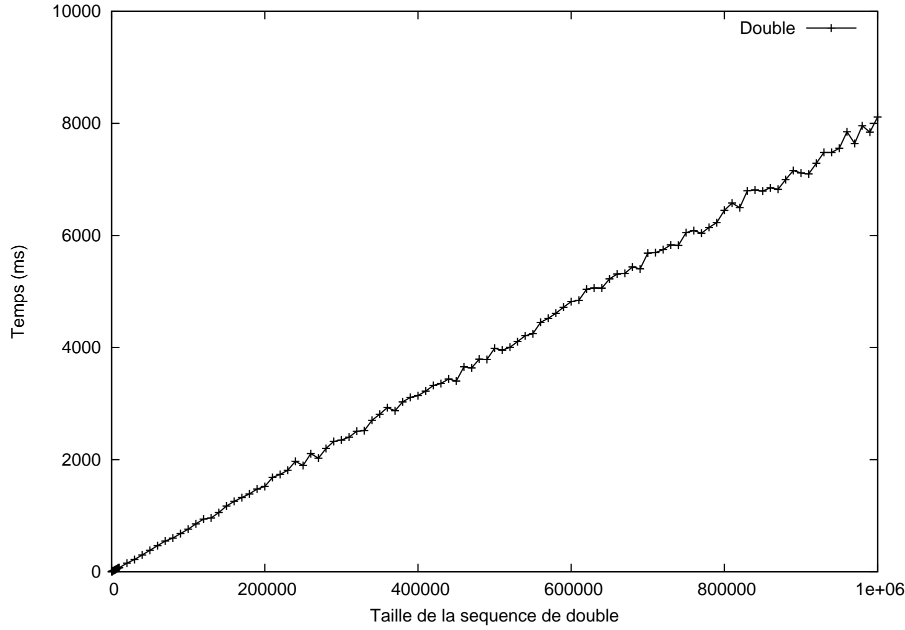


FIG. 3.10 – Temps de transfert de sequences de double

Data-driven

De la même façon, les temps de traitement en mode *data-driven* sont les suivants :

Sans vectorisation/Avec notification

$$Time(n) = n.(\tau_{c0} + \tau_{notif} + \tau_{t0}) + \sum_{i=1}^{n} SVW_{data}(i)$$

Sans vectorisation/Sans notification

$$Time(n) = n.(\tau_{c0} + \tau_{t0}) + \sum_{i=1}^{n} SVW_{data}(i)$$

Avec vectorisation/Sans notification

$$Time(n) = n.(\tau_{c0} + \alpha + \frac{\beta}{p}) + \sum_{i=1}^{\frac{n}{p}} SV_{data}(i * p)$$

Avec vectorisation/Sans notification

$$Time(n) = n.(\tau_{c0} + \alpha + \frac{\beta + \tau_{notif}}{p}) + \sum_{i=1}^{\frac{n}{p}} SV_{data}(i * p)$$

Évaluation du mode data-driven Ce mode peut être considéré comme plus performant que le mode demand-driven. En effet, la donnée est transmise directement au processus consommateur dès sa production. Comme tout mode avec notification, le mode data-driven avec notification introduit un overhead supplémentaire, mais évite la surcharge du processus consommateur si le producteur est plus rapide, cas qui peut se produire dans le mode data-driven sans notification.

Même s'il est considéré comme plus performant que le mode demand-driven, ce mode pose d'autres problèmes pour le rendre utilisable dans tous les cas. En effet, sans vectorisation, ce mode est très pénalisant (comme le mode demand-driven) sauf si les jetons de la FIFO sont de grande taille. Avec vectorisation, nous améliorons sensiblement les temps de communication, mais apparaît un autre problème qui est celui de la terminaison et de la longueur du paquet. Comment déterminer la taille du vecteur de jetons pour assurer la terminaison ? Sauf dans le cas des réseaux de processus synchrones, la longueur du paquet peut être rarement déterminée à la compilation. Une solution éventuelle consisterait à envoyer un paquet de tout ce qui reste dans la FIFO à la terminaison du processus, mais déterminer la terminaison du traitement d'un processus n'est pas toujours possible (boucle infinie par exemple). En plus, le besoin de transparence des communications nous oblige à gérer la terminaison sans intervention du programmeur.

Gestion séparée des communications Les communications que ça soit data-driven et demand-driven peuvent être réalisées séparémment de la partie calcul, pour pouvoir recouvrir ces communications par des calculs. Ceci revient à avoir un objet *Communication* qui gère les communications entre le consommateur et le producteur, et qui sera côté producteur si le mode est demand-driven et côté consommateur si le mode est demand-driven. La figures 3.11 montre l'architecture simplifié d'un tel système.

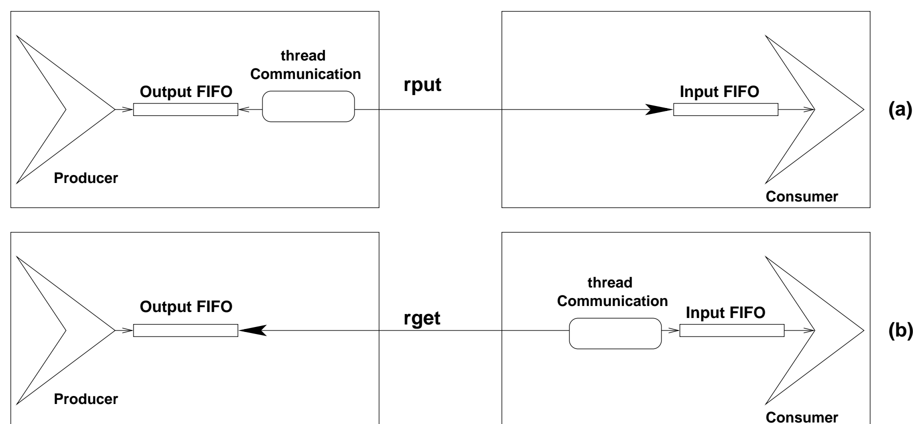


FIG. 3.11 – Gestion séparée des communications

Dans les deux modes, ceci permettrait d'éviter le blocage du processus consommateur pendant les communications, et déchargerait le programmeur de la gestion des appels distants. Néanmoins, et malgré la suppression des temps de blocage des processus, certains problèmes persistent :

- Un processus est associé à chaque FIFO distribuée, ce qui augmente la charge d'exécution.
- En mode demand-driven :
  - le problème de la surcharge (permanente) de la demi-FIFO du consommateur peut se produire au cas où ce dernier est plus rapide que le producteur. Une solution plus avancée consisterait à utiliser le mode demand-driven avec un seuil (minimal) pour éviter la surcharge. Ce mode garantit une exécution complète, une charge controlable du consommateur, mais échoue dans l'équilibrage de charge des deux côtés et peut entrainer ainsi un ralentissement de l'application ;
  - dans le mode avec notification, la requête de demande est asynchrone, ce qui nécessite une synchronisation des réponses.
- En mode data-driven
  - le problème de la terminaison : lorsque les communications sont vectorisées, la longueur des paquets pour assurer la terminaison de l'exécution reste un problème difficile à résoudre ;
  - dans le mode avec notification, il faut expliciter le mécanisme de reprise des communications, si la réponse de la notification indique un canal de communication plein.
  - dans le mode sans notification, un désiquilibre peut se produire, car la demi-FIFO du consommateur sera toujours pleine si le producteur est plus rapide, alors que la demi-FIFO du producteur sera vide tant que la demi-FIFO du consommateur n'est pas pleine

3.4.2 Mesures de performances

Protocole Hybride VS demand-driven Pour les mesures de performances, un exemple classique producteur/consommateur a été choisi.

Dans tous les cas, un fonctionnement demand-driven pur[21] a été utilisé. Les mesures ont été faites sur deux machines reliées par un réseau ethernet partagé à 10Mb/s. La premiere est équipée d'un processeur AMD Duron à 1,3Ghz, la seconde est équipée d'un Pentium IV à 2,4 Ghz. Le producteur est exécuté sur la second machine, et le consommateur est exécuté sur la seconde.

La fonction de calcul utilisée côté consommateur est la suivante :

$$f(x) = \sum_{n=1}^{n=10^5} \frac{x}{n}$$

Le producteur par contre n'a pas de fonction de calcul, et produit les données en continu.

La figures suivantes résument les caractéristiques et les mesures obtenues. Chaque graphe contient quatre courbes :

- pour chaque mode testé une courbe montrant le temps total d'exécution de l'application ainsi que les temps de calcul et de communication (temps total = temps de calcul + temps de communication) ;

[21] Demande synchrone, ou demande asynchrone puis lecture blocante.

– pour notre mode hybride la courbe montre le temps total d'exécution (calcul + communication).

Les quatre cas testés sont les suivants :

1. sans vectorisation, sans notification (figure 3.12) : on obtient des gains de performances de l'ordre de 22% avec le protocole hybride.

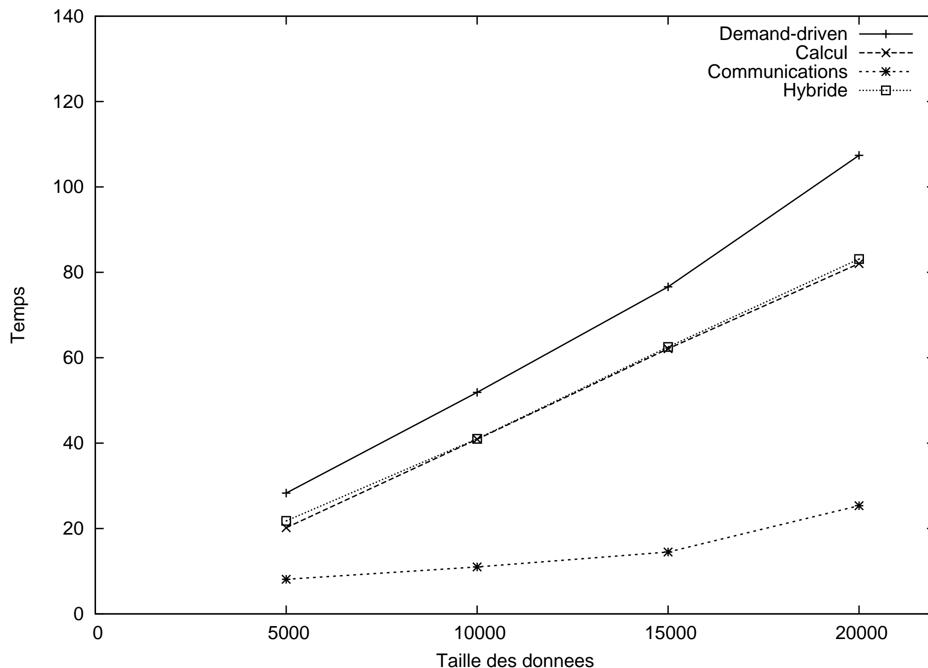


Fig. 3.12 – Sans vectorisation/sans notification

2. sans vectorisation, avec notification (figure 3.13) : c'est le scénario ou le gain est le plus important (35%). Outre la non vectorisation des communications, la notification à chaque lecture est très pénalisante.
3. avec vectorisation, sans notification (figure 3.14) : l'amélioration dûe à la vectorisation des communications permet de réduire considérablement les temps de traitements, ce qui explique que ce protocole obtient presque les mêmes performances que le notre. L'explication vient de la réduction du temps de communication qui est pratiquement divisé par 1024 (longueur des paquets transférés). En effet seules 19 opérations de communication sont nécessaires pour transférer 20000 jetons (ce qui correspond à peu près à $19 \times 8ms$).
4. avec vectorisation, avec notification (figure 3.15) on obtient pratiquement les mêmes résultats que précédémment. L'ajout de la notification n'influe pas beaucoup sur les temps de communication quand les données sont vectorisées..

Protocole Hybride VS data-driven Les mesures en mode data-driven ont été faites sur un exemple producteur/consommateur tournant sur la même configuration que précédemment. Les cas mesurés sont les suivants :

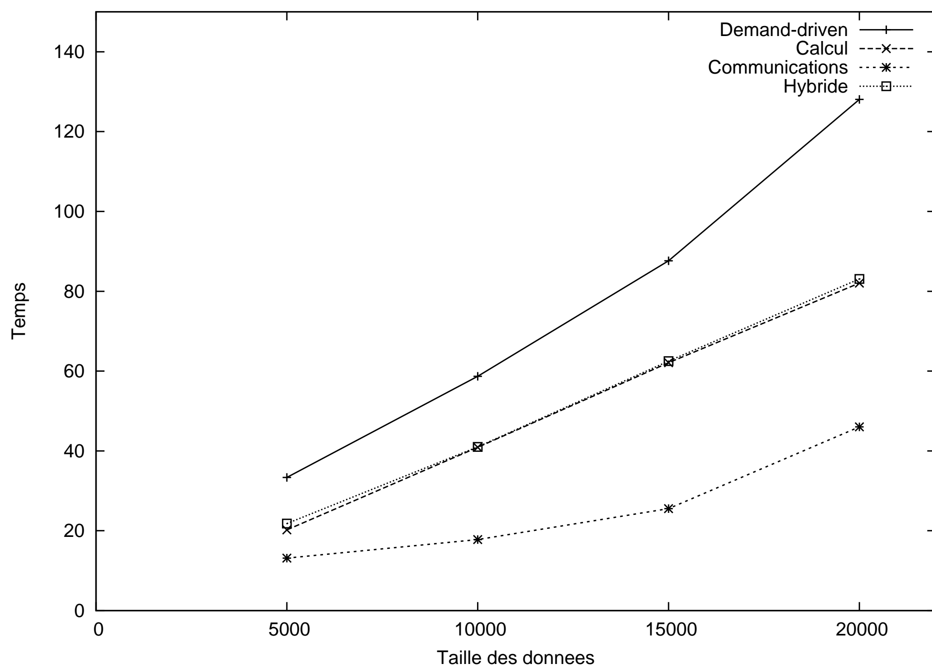


Fig. 3.13 – Sans vectorisation/avec notification

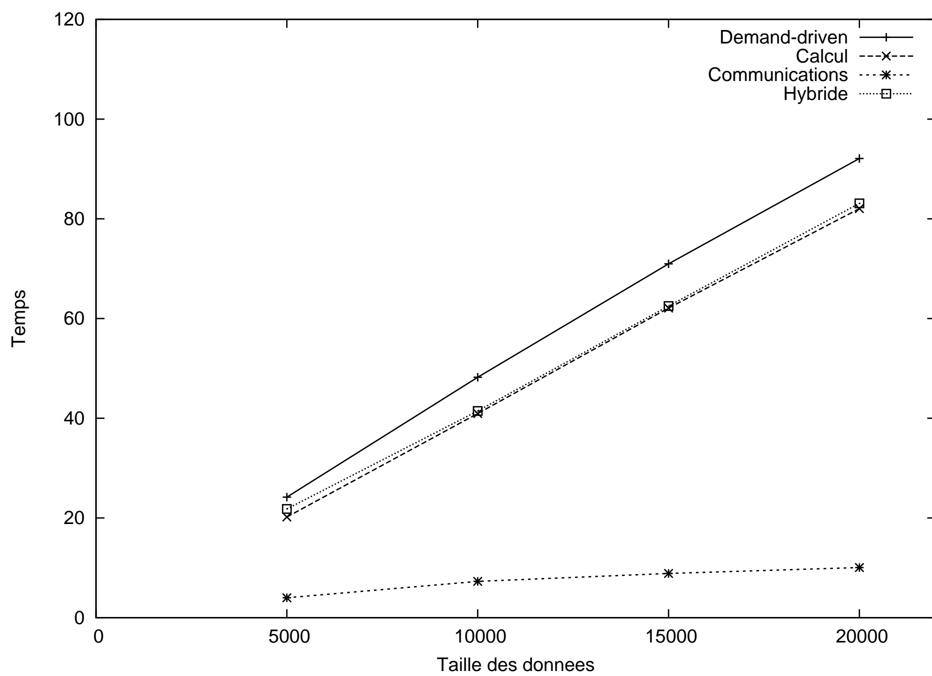


Fig. 3.14 – Avec vectorisation/sans notification

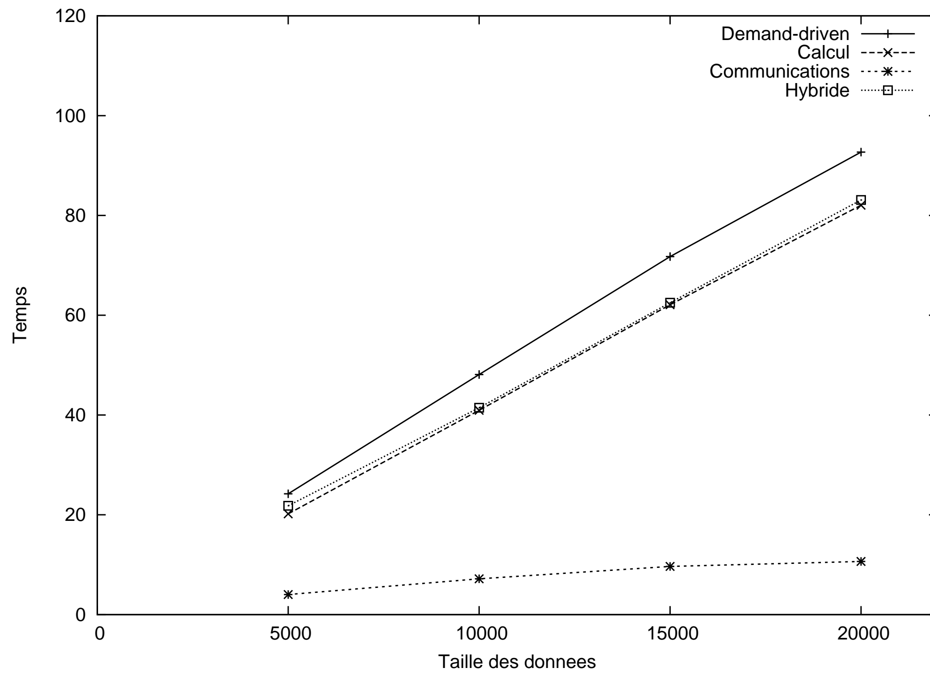


FIG. 3.15 – Avec vectorisation/avec notification

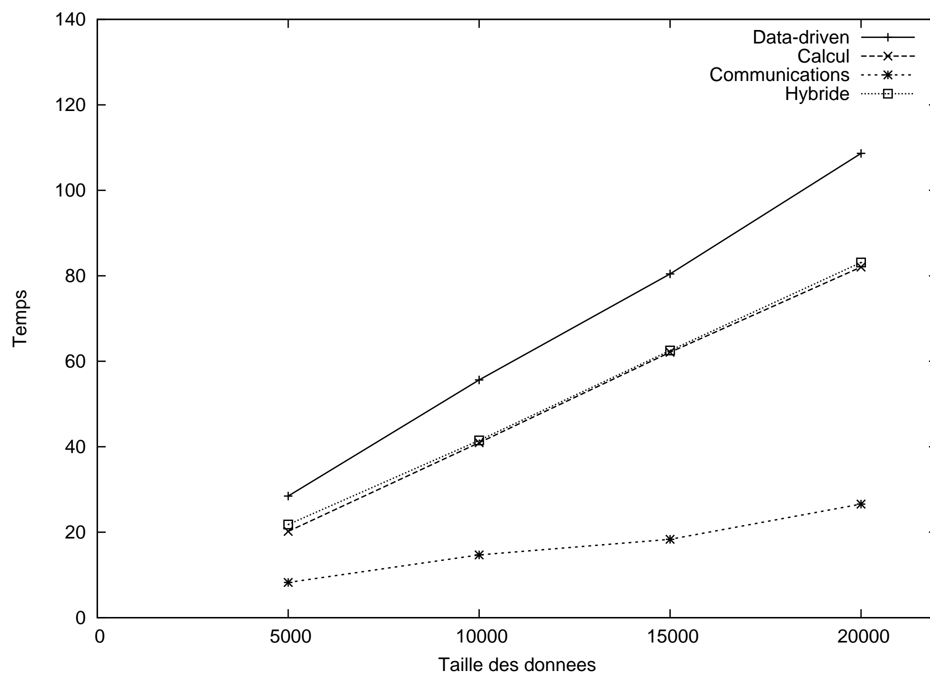


FIG. 3.16 – Sans vectorisation/sans notification

1. Sans vectorisation, sans notification ;
2. Sans vectorisation, avec notification ;

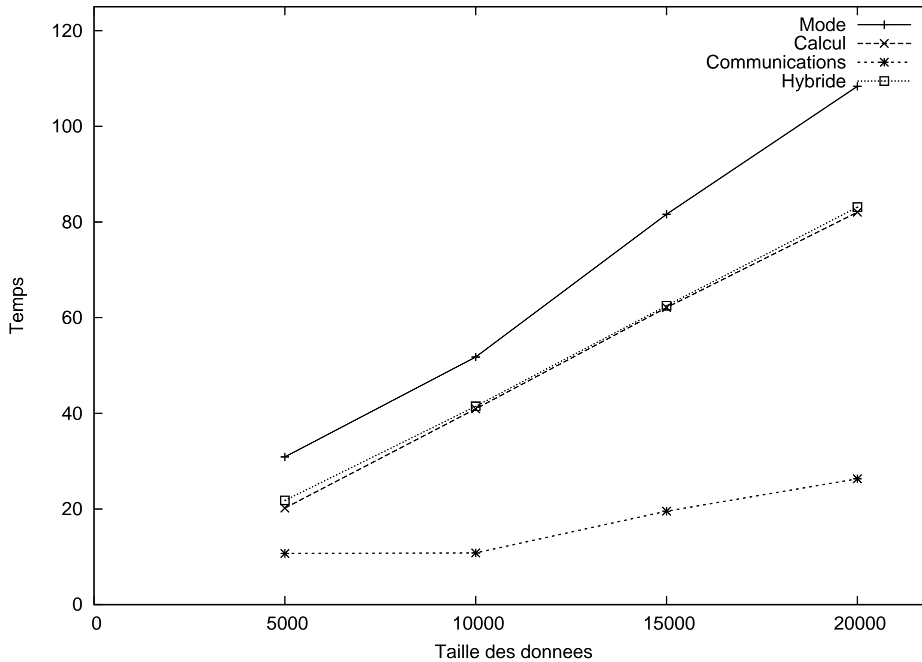


FIG. 3.17 – Sans vectorisation/avec notification

3. Avec vectorisation, avec notification ;
4. Avec vectorisation, sans notification.

Bien que le serveur soit plus rapide que le client (car il produit des données en continu), les performances du protocole data-driven où les communications se font sans vectorisation restent très médiocres. Lorsque les communications sont vectorisées, on remarque une amélioration très nette des performances, et le protocole obtient les même résultats que notre protocole hybride.

Résumé Les séries de tests effectuées montrent que notre protocole de communication est au moins aussi rapide que n'importe quel protocole demand ou data driven. On obtient des améliorations pouvant atteindre 35% dans certain cas. Lorsque les communications sont vectorisées, les protocoles demand et data driven reviennent à la hauteur de notre protocole, mais ils restent des protocoles qui ne tiennent pas compte de la charge mémoire présente de chaque côté.

Les accélérations obtenues permettent surtout d'évaluer la qualité du mode testé. La caractéristique principale qu'on peut déduire de l'analyse des différents graphes de performances est que les comunications sont recouvertes par les calculs dans notre protocole hybride. En effet, la courbe du temps de calcul est pratiquement égale à celle du temps d'exécution de l'application avec notre protocole.

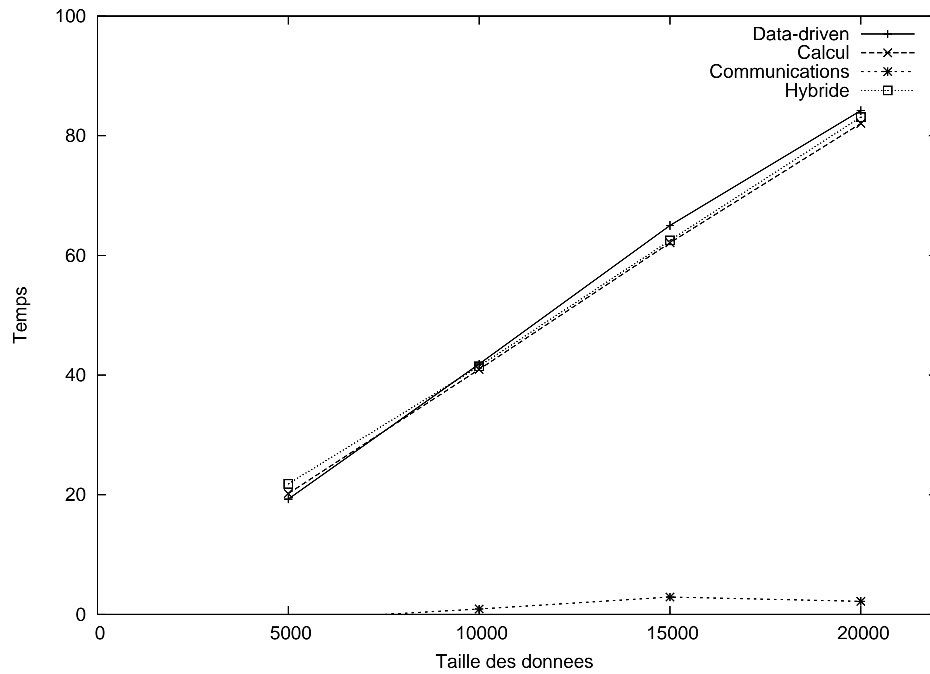


Fig. 3.18 – Avec vectorisation/avec notification

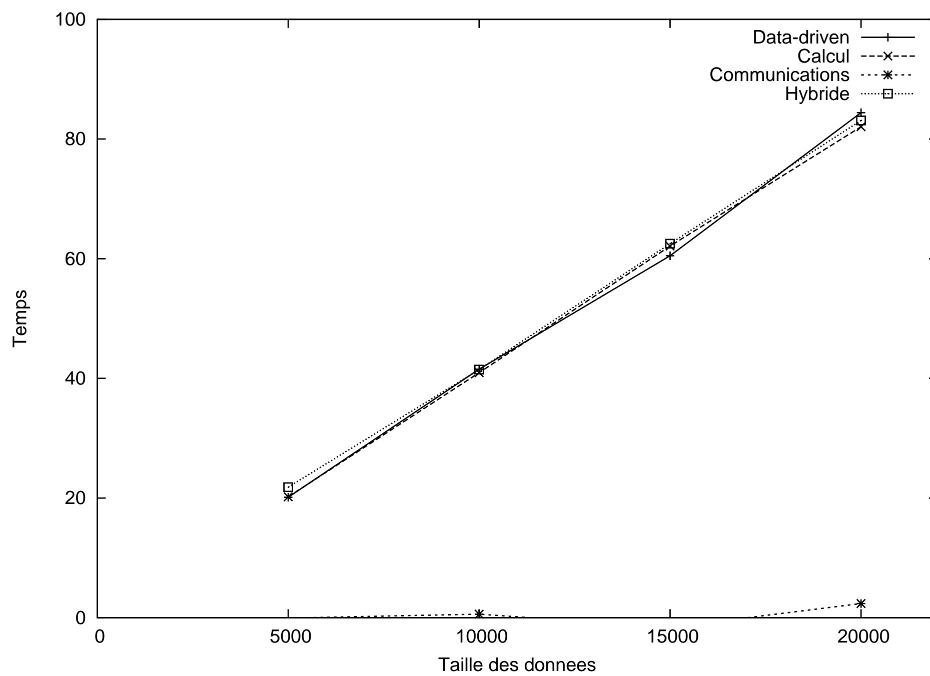


Fig. 3.19 – Avec vectorisation/sans notification

Le cas testé présente une situation où le producteur est plus rapide que le consommateur. L'application tourne en régime continu. Dans le cas où le producteur est plus lent que le consommateur, il serait difficile d'évaluer les temps de communication ou de mesurer le recouvrement des calculs par les communications. En effet dans le calcul des temps de traitement sera inclus en plus du temps de calcul et de communication les retards entre le moment de tentative de lecture du jeton et celui de sa production.

3.5 Réseaux de processus, composant et génération automatique de code

Plusieurs travaux et projets traitent de la modélisation et la génération automatique de composants distribués. L'approche MDA [Boa01] développée par l'OMG est une preuve que l'approche est acceptée par l'industrie, et plusieurs approches ont été explorées. La notion de *Wrapper* a été utilisée pour construire des composants hétérogènes dans [RLS+00]. D'autres approches plus formelles [BG97, Raj00] explorent le domaine de spécification d'algorithme.

Nous proposons ici une approche de génération automatique de code basée sur le modèle des réseau de processus distribués, en l'étendant pour pouvoir supporter un plus grand nombre de types d'applications réparties.

Cette approche se base sur la composition d'objets et le fait que pour développer une méta-application, seule la partie traitement et la description des liens entre les composants sont nécessaires pour générer automatiquement toute la gestion des communications. Notre modèle permet ainsi de générer des composants à partir de la description de leur fonction de calcul, des interfaces implémentées et exportées, et des interfaces importées d'autres composants. Ceci libère le développeur de toute la gestion de bas niveau des flux de données, que ce soit à l'intérieur du composant (gestion des FIFOs) ou entre les différents composants (échange de données et synchronisation entre composants). Le code des composants est créé à partir du code source de la fonction de traitement et la description des composants est décrite dans la section 3.5.5.

Les réseaux de processus de Kahn présentent un modèle bien adapté à la construction d'applications de type assemblage de composant, où les processus de Kahn jouent le rôle de composants réutilisables. Néanmoins, le modèle est assez rigide pour pouvoir supporter un large type d'applications réparties. Les FIFOs sont des structures adaptées aux applications où le flux de données est bien défini, et où les données sont lues et écrites par un seul processus, or un composant peut fournir des résultats à plusieurs autres tout en recevant des données où l'ordre n'est pas très important voir inexistant, tout comme il peut fournir des services difficilement réalisables par l'intermédiaire des FIFOs. C'est la raison pour laquelle on a étendu le modèle en ajoutant aux processus distribués des objets exportés qui fournissent des services aux autres composants par une autre voie que celle des FIFOs. Le fait d'utiliser un autre moyen de communication en plus des FIFOs peut entraîner la non validation du déterminisme du modèle, mais permet en contrepartie une plus grande souplesse dans la construction de l'application et une utilisation d'un large éventail de composants déjà disponibles. L'approche proposée ici reste néanmoins valable

pour générer des composants basés uniquement sur le modèle des réseaux de processus de Kahn, et ainsi de construire un système complètement déterministe.

3.5.1 Architecture du système

L'architecture du système (figure 3.20) est composée d'un référentiel d'objets où sont stockés les descriptions des objets disponibles, d'une fabrique qui permet de générer le code des composants à partir de leur description, d'un référentiel de composants, et d'une console interactive qui permet de construire le flux de données en liant les FIFOs.

Pour la description des objets, et des composants, le langage XML [Xml] a été utilisé. Ce langage est devenu le langage standard pour l'échange des données entre les systèmes distribués, indépendamment des plateformes d'exécution.

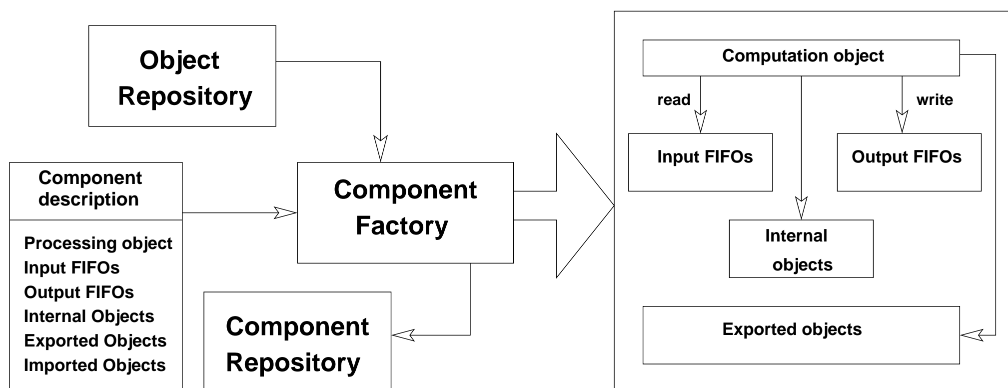


FIG. 3.20 – Architecture du système de génération de code

3.5.2 Structure d'un composant

Un composant est constitué de cinq types d'objets :

- un objet de traitement pour effectuer les traitements désirés ;
- un ou plusieurs objets FIFOs d'entrée ;
- un ou plusieurs objets FIFOs de sortie ;
- un ou plusieurs objets exportés pour implémenter les services exportés aux autres objets ;
- un ou plusieurs objets internes pour implémenter les services requis par le composant lui-même.

Seul l'objet de traitement peut lire dans les FIFOs d'entrée et écrire dans les FIFOs de sortie. Les objets internes sont des objets locaux aux composants et accessibles au travers de l'objet de traitement, alors que les objets exportés sont des objets CORBA qui fournissent des services aux autres composants. Les objets internes et les objets exportés sont fournis à partir d'un référentiel d'objets. Ce dernier fournit une description en XML de chaque objet, qui permet de connaître les services fournis par chaque objet, et le code source de ces objets. Les services requis sont décrit à travers les objets importés, mais seule l'interface de ces objets est nécessaire pour accéder à ces objets.

3.5.3 Référentiel d'objets

Pour pouvoir créer des composants par composition d'objet, un référentiel d'objet a été défini. Il permet de stocker la description des objets, les liens vers les codes sources et la hiérarchie des classes disponibles. Ce référentiel n'est pas un point central de stockage de tous les objets, et chaque fabrique peut avoir un ou plusieurs référentiels.

La figure 3.21 montre la DTD des fichiers de description des objets, on retrouve en plus du nom de la classe, du langage de programmation, et des fichiers sources, une description, de chaque fonction (avec son type de retour et ses paramètres), des attributs, et des classes de base.

```
<!ELEMENT repository:class (repository:function*,
repository:attribut*, repository:base*)>

<!ATTLIST repository:class
      name     CDATA #REQUIRED
      class_id CDATA #REQUIRED
      lang     CDATA #REQUIRED
      src_file CDATA #REQUIRED
      header_file CDATA #REQUIRED>

<!ELEMENT repository:function (repository:parameter*)>
<!ATTLIST repository:function
      name CDATA #REQUIRED
      type CDATA #REQUIRED>


<!ELEMENT repository:parameter (#PCDATA)>
<!ATTLIST repository:parameter
      type CDATA #REQUIRED
      mode (value | ref ) #REQUIRED>

<!ELEMENT repository:attribut (#PCDATA)>
<!ATTLIST repository:attribut
      name CDATA #REQUIRED
      type CDATA #REQUIRED>

<!ELEMENT repository:base (#PCDATA)>
<!ATTLIST repository:base
      base_id CDATA #REQUIRED>
```

FIG. 3.21 – DTD du format de description du référentiel d'objets

```
<!DOCTYPE repository:class SYSTEM "class_repository.dtd">
<repository:class name = "VBL" class_id = "vbl" lang = "c++"
src_file = "vbl/vbl.cc" header_file = "vbl/vbl.h">

<repository:function name ="dit2_fft" type = "void">
<repository:parameter type = "double *" mode = "ref">
</repository:parameter>
<repository:parameter type = "double *" mode = "ref">
</repository:parameter>
<repository:parameter type = "unsigned int" mode = "ref">
</repository:parameter>
<repository:parameter type = "double *" mode = "value">
</repository:parameter>
</repository:function>

<repository:function name ="c_DotCx" type = "void">
<repository:parameter type = "unsigned long" mode = "ref">
</repository:parameter>
<repository:parameter type = "double *" mode = "ref">
</repository:parameter>
<repository:parameter type = "double *" mode = "ref">
</repository:parameter>
<repository:parameter type = "double *" mode = "ref">
</repository:parameter>
</repository:function>

</repository:class>
```

FIG. 3.22 – Représentation XML d'un objet de traitement de signal (VBL : Veille Large Bande)

3.5.4 Étapes de la génération automatique du code

- Génération du code relatif aux FIFOs ;
- Analyse des objets exportés et génération du code IDL pour chaque classe, puis génération du code d'implémentation à partir du code source de la classe d'origine, mais en prenant en compte la distribution de l'objet. Le prototype actuel accepte les types simples ;
- Analyse des objets importés et génération du code IDL pour chaque classe, ceci permettra d'avoir les souches nécessaires pour l'accès à l'objet distant ;
- Génération des classes internes ;
- Génération du code relatif à l'objet de traitement ;
- Liaison des classes internes avec l'objet de traitement (récupération des références) ;
- Analyse du code de la fonction de traitement

```
Si appel vers un objet interne alors
    - ajouter une méthode du même nom dans la classe de traitement
    - dans cette fonction, rediriger l'appel vers l'objet interne
    correspondant
sinon
    si appel vers un objet distant alors
        - ajouter une méthode du même nom dans la classe de
        traitement
        - dans cette fonction rediriger l'invocation vers l'objet
        distant si la référence de ce dernier est disponible
         sinon
            - récupérer la référence de l'objet distant
            - invoquer l'objet
          fin si
   fin si
fin si
```

– Génération du code, pour créer les différents objets et le lancement du traitement.

3.5.5 Description des composants

La figure 3.23 montre la DTD des fichiers XML utilisés pour décrire les composants du système. Les cinq parties qui forment le composants sont définies :

– la partie traitement décrite par l'élément *dpn* : *processing* qui fournit l'identificateur de la fonction et son code source ;
– les FIFOs d'entrée et de sortie décrites par l'élément *dpn* : $fifo$ qui fournit le nom de la FIFO, le type des jetons qu'elle manipule, son type (entrée ou sortie), sa capacité maximale (en nombre de jetons) et son seuil ;
– les classes exportées et les classes internes : un identificateur de ces classes permet de récupérer leur description dans le référentiel d'objets ;
– les classes importées : l'identificateur des classes importées permet de récupérer la description de ces classes et ainsi l'interface implémentée par ces classes. Le nom du composant dans lequel sont implémentées ces classes sert à récupérer la référence de la classe distante.

3.6 Exemple d'application : Décodeur JPEG distribué

Nous allons illustrer notre système par un exemple d'une application de traitement d'image : un décodeur JPEG.

Structure de l'application

Le réseau de processus modélisant l'application est présenté dans la figure 3.24. Il est composé de 7 réseaux de processus (7 objets *ProcessNetwork*) qui contiennent 48 processus et 100 FIFOs.

```
<!ELEMENT dpn:component (dpn:processing, (dpn:fifo | dpn:internal | dpn:exported | dpn:imported)*)>
<!ATTLIST dpn:component  name CDATA #REQUIRED>
<!ELEMENT dpn:processing (dpn:file+)>
<!ATTLIST dpn:processing function_id    CDATA #REQUIRED>
<!ELEMENT dpn:file (#PCDATA)>
<!ATTLIST dpn:file file_name CDATA #REQUIRED>
<!ATTLIST dpn:file type (src | header) #REQUIRED>
<!ELEMENT dpn:fifo (#PCDATA)>
<!ATTLIST dpn:fifo name CDATA #REQUIRED>
<!ATTLIST dpn:fifo type CDATA #REQUIRED>
<!ATTLIST dpn:fifo direction (in | out) #REQUIRED>
<!ATTLIST dpn:fifo threshold CDATA #IMPLIED>
<!ATTLIST dpn:fifo max_capacity CDATA #IMPLIED>
<!ELEMENT dpn:internal (#PCDATA)>
<!ATTLIST dpn:internal name CDATA #REQUIRED>
<!ATTLIST dpn:internal class_id CDATA #REQUIRED>
<!ELEMENT dpn:exported (#PCDATA)>
<!ATTLIST dpn:exported name CDATA #REQUIRED>
<!ATTLIST dpn:exported class_id CDATA #REQUIRED>
<!ELEMENT dpn:imported (#PCDATA)>
<!ATTLIST dpn:imported component_name CDATA #REQUIRED>
<!ATTLIST dpn:imported name CDATA #REQUIRED>
<!ATTLIST dpn:imported class_id CDATA #REQUIRED>
```

FIG. 3.23 – DTD du format de description des composants

Le processus *FrontEnd* lit le fichier JPEG contenant l'image, et fournit un flux d'octets qui sera traité par les différents processus :

- Le réseau de processus *JFIF* décompose ce flux en informations d'entête nécessaire au traitement, et en données représentant les pixels compréssées (*MCUseg*).
- Le réseau de processus *Decode* permet d'extraire les pixels des données compressées. Il est constitué des processus *VLD* (*Variable Length Decoding*) et IQ (*Inverse Quantization*) qui utilisent la table de Huffman pour effectuer l'opération inverse de quantification. Le réseau de processus IDCT (*Inverse Disrete Cosine Transform*) réalise la transformée inverse discrète en cosinus pour produire un flux de pixels décompressé.
- Le réseau de processus *Render* permet de séparer les flux représentant la luminance (Y), crominance bleue (*Cb*) et crominance rouge (*Cr*). Ces flux sont convertis en R, V et B par le processus *Matrix*.
- Le processus *BackEnd* lit les flux R,V et B pour créer un fichier au format sun raster.

Distribution de l'application

La distribution de l'application peut se faire facilement en créant un nouveau programme pour chaque réseau de processus qu'on veut exécuter séparémment. Les figures 3.25 et 3.26 montrent respectivement, le programme principal du décodeur JPEG initial, et celui du réseau de processus *JFIF* après la distribution. Aucun changement n'est apporté au code de la classe *JFIF*, il suffit juste de créer le programme principal de lancement, ainsi que le réseau de processus qui va créer les FIFOs de *JFIF* qui étaient créées par le réseau *JPEG*.

3.6.1 Génération de code

Pour illustrer notre système de génération de code, nous utiliserons la même application du décodeur JPEG, mais modélisée à un grain plus gros par rapport à la version

Fig. 3.24 – Structure du décodeur JPEG

```
#include "jpeg.h"
#include "yapi.h"

int main()
{
  // yapi run-time environment
  RTE rte;

  JPEG jpeg (id("jpeg"));

  // start the process network and
  // wait for processes to finish
  rte.start(jpeg);

  return 0;
}
```

FIG. 3.25 – Programme principal du décodeur JPEG

```
#include "jfif-pn.h"
#include "yapi.h"

int main()
{
  // yapi run-time environment
  RTE rte;

  JFIF_PN jfif_pn (id("jfif_pn"));

  // start the process network and
  // wait for processes to finish
  rte.start(jfif_pn);

  return 0;
}
```

FIG. 3.26 – JFIF distribué (programme principal)

précédente. Dans cette version, seul six processus composent l'application. Le décodeur est composé de quatre composants principaux : *FrontEnd* qui lit l'image et récupère ses données, *IDCT* qui applique la transformée de Fourier inverse au flux de pixels, *Raster* qui produit un flux de pixels sous forme de luminance, crominance bleue et crominance rouge, et en dernier *BackEnd* qui permet de créer le bitmap correspondant à l'image JPEG en entrée. Les deux autres composants (*forkWidth* et *forkHeight*) permettent de dupliquer les données des FIFOs *imageWidth* et *imageHeight*. Les figures 3.27 et 3.28 montrent respectivement l'architecture du décodeur JPEG distribué et le fichier XML décrivant le premier composant : *FrontEnd*.

3.7 Conclusion

Dans ce chapitre, nous avons présenté la conception et l'implémentation des réseaux de processus de Kahn distribués. Notre approche basée sur l'architecture CORBA fournit un support d'exécution pour la simulation d'applications distribuées par assemblage de composants logiciels. Chaque composant logiciel représente un réseau de processus hiérarchique, qui peut être assemblé aux autres réseaux de processus en liant les demi-FIFOs entre elles.

Outre la transparence de la distribution, les communications entre les demi-FIFOs sont optimisées pour obtenir des simulations efficaces, permettant ainsi à un plus large type d'applications d'être exécutées avec ce support. Notre protocole de transfert s'est avéré au moins aussi rapide que les modes classiques de transfert (demand-driven ou data-driven), sinon plus rapide. La gestion de la terminaison d'exécution est gérée de façon transparente tout en vectorisant les communications. En plus, l'utilisation de seuils permet de mieux équilibrer les charges mémoires entre les deux demi-FIFOs. L'exécution peut ainsi s'adapter au variations de charges des ressources de calcul. Elle permet aussi d'accroître la disponibilité des données pour les processus consommateurs en anticipant

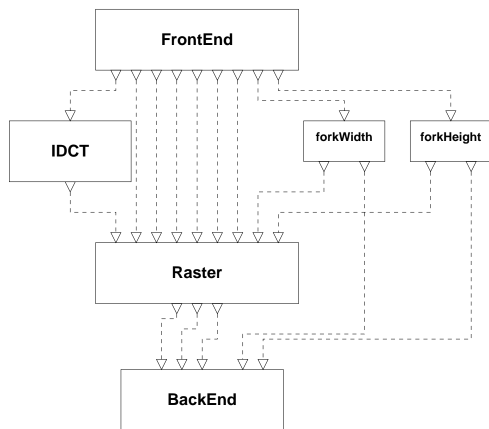


Fig. 3.27 – Décodeur JPEG distribué

```
<!DOCTYPE dpn:component SYSTEM "dpn_comp.dtd">
<dpn:component name = "FrontEnd">
<dpn:processing function_id = "frontend">
<dpn:file file_name = "frontend.cc" type = "src">
</dpn:file>
<dpn:file file_name = "frontend.h" type = "header">
</dpn:file>
</dpn:processing>
<dpn:fifo name = "imageDepth" type = "unsigned char"
direction = "out" threshold = "100" max_capacity = "2048">
</dpn:fifo>
<dpn:fifo name = "imageComponentId" type = "unsigned char"
direction = "out" threshold = "100" max_capacity = "2048">
</dpn:fifo>
<dpn:fifo name = "imageWidth" type = "unsigned int"
direction = "out" threshold = "100" max_capacity = "2048">
</dpn:fifo>
<dpn:fifo name = "imageHeight" type = "unsigned int"
direction = "out" threshold = "100" max_capacity = "2048">
</dpn:fifo>
<dpn:fifo name = "imageH" type = "unsigned int"
direction = "out" threshold = "100" max_capacity = "2048">
</dpn:fifo>
<dpn:fifo name = "imageV" type = "unsigned int"
direction = "out" threshold = "100" max_capacity = "2048">
</dpn:fifo>
................
</dpn:component>
```

Fig. 3.28 – Description du composant FrontEnd

les transfert de données pour éviter le phénomène de famine.

Enfin, la génération de code automatique permet d'accélérer le processus de développement, et la distribution d'applications existantes.

Dans le chapitre suivant, nous allons introduire la notion de dynamicité dans les réseaux de processus distribués, et présenterons sa mise en œuvre dans notre support d'exécution.

4

Dynamicité

Sommaire

4.1 Introduction 93
4.2 Types de dynamicité 94
4.3 Architecture du système 95
4.4 Développement incrémental de l'application répartie 99
4.5 Migration et remplacement de composant 99
4.6 Reconfiguration d'un processus 100
4.7 Exemple : Application de traitement de signal 103
4.8 Conclusion 107

Le problème avec le cerveau c'est que le seul outil qui permette de l'étudier et d'améliorer son fonctionnement c'est ... le cerveau lui-même.

Edmond Wells.

4.1 Introduction

Avec le développement des applications réparties, de nouveaux besoins sont apparus, parmi lesquels les besoins de maintenabilité, d'évolutivité et de dynamicité des systèmes existants. Dans ce chapitre on s'intéresse à l'aspect dynamique d'applications distribuées basées sur le modèle des réseaux de processus distribués. On présentera l'intégration de la dynamicité dans notre support dynamique d'exécution où on traitera des aspects fonctionnels de la dynamicité (migration, remplacement, reconfiguration, etc.).

Le modèle des réseaux de processus distribués présenté dans le chapitre 3 est notre modèle de référence pour construire un support d'exécution dynamique qui autorise le

remplacement et la migration des entités logicielles [22] de façon transparente pour le programmeur. Les objectifs derrière notre approche peuvent être résumés dans les points suivants :

- Exploiter au mieux les ressources (puissance de calcul, mémoire) disponibles ;
- Rendre cette dynamicité transparente au programmeur, en se basant sur l'interactivité pendant le déploiement ou l'exécution, et en fournissant des outils de précompilation ;
- Automatiser la création des applications distribuées en générant automatiquement le code correspondant.

Ce chapitre se décompose principalement en deux parties, la première partie traite de l'aspect fonctionnel de la dynamicité du support d'exécution, alors que la seconde présente l'aspect automatisation du développement d'application. La section 4.2 présente les quatre aspects de la dynamicité de notre approche. La section suivante 4.3 décrit l'architecture du système et les restrictions imposées. Les trois sections suivantes 4.4, 4.5 et 4.6 détaillent les quatre aspects de dynamicité en présentant pour chacune les mécanismes d'implémentation. Un exemple d'une application réelle de traitement de signal est présenté dans la section 4.7 qui montre l'apport de la dynamicité dans l'amélioration des performances et de la charge mémoire.

4.2 Types de dynamicité

Le concept de dynamicité de l'application est un des points essentiels de notre approche. Dans un contexte de simulation distribuée, cette dynamicité permet au développeur de tester différentes configurations de l'application, d'étudier son évolution, et de pouvoir ainsi obtenir la « meilleure configuration » en termes de performances, besoins mémoire, etc. D'un autre côté, les applications de calcul scientifique et plus particulièrement les applications de traitement de signal intensif ont une durée d'exécution longue, voir infinie si en entrée le flux de données est infini. Pouvoir améliorer les composantes logicielles ou matérielles de l'application distribuée sans être obligé de l'arrêter est l'un des critères les plus importants dans notre approche.

Cette dynamicité est vue selon quatre aspects :

Le développement incrémental

L'aspect incrémental du développement de l'application distribuée permet de construire et de déployer cette dernière en plusieurs étapes. Ainsi, les différents processus du réseau peuvent être développés indépendemment les uns des autres, et le déploiement de façon incrémentale permet de construire l'application par composition. Trois mécanismes peuvent être utilisés :

- Ajout d'un processus : ajouter un processus au cours de l'exécution pour l'intégrer dans le réseau ;

[22] Dans la suite de document, nous utiliserons le terme composant pour référencer l'entité logicielle qui représente un processus distribué

- Ajout d'une liaison : lier deux demi-FIFOs pendant l'exécution ;
- Création d'un nouveau processus : définir et fournir un système de création de processus à la volée (par génération automatique de code), et pouvoir l'intégrer au système.

La migration de processus

La migration de processus d'un site à un autre est souvent motivée par les contraintes de performance. Dans un système réparti où les ressources sont partagées par plusieurs utilisateurs, et les charges variables et imprédictibles, la migration de code vers des sites moins chargés est un moyen pour exécuter efficacement des applications distribuées. La seconde contrainte est liée à la nature dynamique de la disponibilité des ressources. En effet, de nouvelles ressources de calcul, éventuellement plus puissantes, peuvent être disponibles au cours de l'exécution de l'application, et il serait plus judicieux de les utiliser pour en tirer profit.

Le remplacement de processus

En plus de la migration, un autre mécanisme permet d'améliorer les performances des applications distribuées, surtout celles ayant un très grand temps d'exécution, comme les applications de calcul scientifique, ou les applications de traitement de signal intensif qui traitent des flux de données infinis. Dans ces applications, on est parfois amenés à remplacer un composant de l'application par un autre, pour changer la fonction de traitement ou pour utiliser un composant plus spécialisé.

La reconfiguration de processus

Cet aspect traite des interactions du processus avec son environnement extérieur, et de son exécution. On peut citer cinq mécanismes liées à la reconfiguration de processus :
- Suppression de processus ;
- Suppression des liaisons ;
- Changement des liaisons ;
- Changement des paramètres ;
- Suspension de l'exécution.

4.3 Architecture du système

L'application distribuée est basée sur les réseaux de processus distribués. La technique d'assemblage de composants est utilisée pour former le réseau de processus. Dans le système, un processus et ses FIFOs (locales et distribuées) forment le composant.

4.3.1 Points de reprise

Les mécanismes de migration et de remplacement de processus doivent se faire tout en gardant le flux et la cohérence de l'exécution. Ces mécanismes doivent se faire à des

points précis de l'exécution de l'application. Ces points sont appelés « points de reprise ».

Certains sous modèles des réseaux de processus de Kahn, comme les réseaux de processus synchrones, booléens ou périodiques, sont des répétitions d'une fonction de traitement, et de ce fait les points de reprises peuvent être naturellement pris comme les points d'entrée (ou de sortie) de ces fonctions. Pour le modèle général des réseaux de processus, ces points de reprises ne peuvent être définis que par le programmeur, qui doit les inclure dans le code source du processus.

Ces « points de reprise » correspondent à l'arrêt de l'exécution du processus, et au point de reprise du nouveau processus remplaçant. Bien sur, la sauvegarde et la restauration de l'état interne du processus, ainsi que les dépendances de données, doivent être gérés par un tel mécanisme de remplacement.

Ci-après, une description de la fonction de traitement des points de reprise (`pcp`), et de leur définition dans le code source du programme.

```
void pcp (int checkpoint)
{
    // Traitement suspendu, envoyer une confirmation à la console
    cons_var->label(checkpoint);

    if (replaced)
      {
        // Transférer l'état interne du processus
        // Récupérer la référence du nouveau processus
        // Empaqueter l'état interne du processus
        // Envoyer l'état interne au nouveau processus
        replaced = false;
      }

    sem_wait(&stop);
    printf("Computation resumed\n");
}

...

if (suspend)
  {
    pcp (x);
  }
label x:
```

4.3.2 Restrictions

Certains aspects de la dynamicité du système, qui sont liés au calcul dans les processus, tels le remplacement et la migration de composant, reposent sur la synchronisation des processus, en utilisant des points de reprise (pour la suspension des traitement). Or

synchroniser des réseaux de processus hiérarchiques imposerait plusieurs contraintes, qui augmenteraient la complexité du système. Généralement, la synchronisation de tels réseaux hiérarchiques pourrait conduire à des deadlocks.

Avant le remplacement ou la migration d'un processus, celui-ci doit être suspendu, or dans un réseaux de processus hiérarchique, ceci impliquerait de définir des points de reprise à l'ensemble des processus du réseau, pour éviter les deadlocks. Un processus peut avoir besoin de données qui devraient être produites par un autre processus qui est déjà suspendu. Dans ce cas, le processus n'atteindra jamais son point de reprise. La solution qui consisterait à définir des points de reprise globaux pour le réseau, ne peut être appliquée qu'à certaines applications particulières. Dans les réseaux de processus où les processus sont des répétitions de fonctions, définir un point de reprise global n'est pas possible, car un tel point ne dépend pas seulement de la fonction du processus, mais aussi de l'état d'exécution du réseau au moment de la suspension. La figure 4.1 montre un réseau de processus où le point de reprise du processus $P1$ ne garantit pas que le processus $P2$ va atteindre son point de repise. À chaque itération, $P1$ écrit un jeton dans la FIFO partagée, alors que $P2$ en consomme 16 à chaque itération.

```
void P1::main()
{
  while (true)
  {
    read(channel0, a);
    write(channel1, b);
    if suspend pcp(0);
  }
}
```

```
void P2::main()
{
  while (true)
  {
    read(channel1, x, 16);
    write(channel2, y, 8);
    if suspend pcp(0);
  }
}
```

FIG. 4.1 – Exemple d'un réseau de processus sans point de reprise

4.3.3 Console interactive

La console présentée dans la section 3.2.1 pour lier les demi-FIFOs et former le réseau de processus, est un élément essentiel de l'architecture du système. Comme dit précédemment, son rôle est minimal, et son langage de commande fournit les mécanismes de controle et de configuration. La figure 4.2 montre le diagramme de classe de cette console.

Langage de commande Le langage de commande de la console permet à l'utilisateur de construire et de controler interactivement le réseau de processus par les commandes suivantes :

- *list* : permet d'avoir la liste de tous les processus actifs dans le système. Une description de chaque processus comprenant sa structure (demi-FIFOs d'entrée et demi-FIFOs de sortie), son identificateur, et son état sont disponibles au travers de cette commande.
- *link* `id1 id2` : lie deux demi-FIFOs entre elle. *id1* détermine l'identificateur de la demi-FIFO de sortie, et *id2* l'identificateur de la demi-FIFO d'entrée.
- *unlink* `id` : supprime le lien que possède la demi-FIFO invoquée.

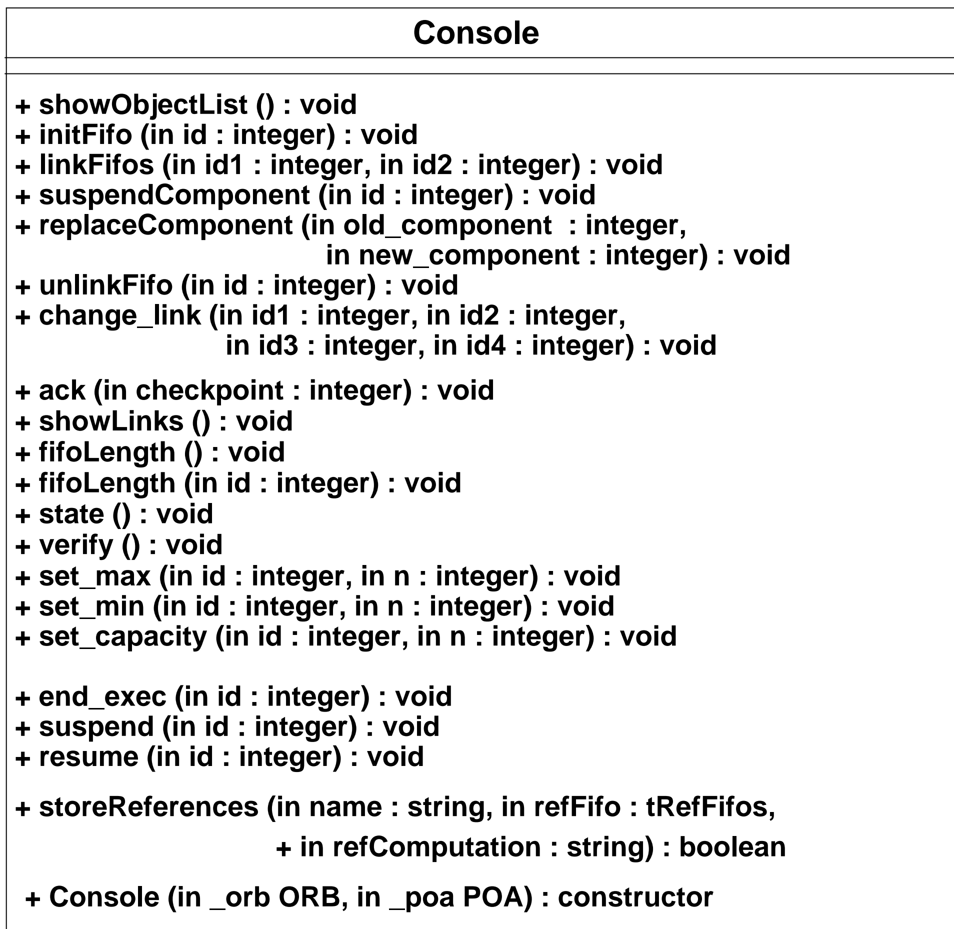


FIG. 4.2 – Diagramme de classe de la Console

- *change_link* `id1 id2 id1 id3` ou *change_link* `id1 id2 id3 id2` permet de modifier la liaison entre deux demi-FIFOs. La première syntaxe correpond au cas de changement de liaison impliquant la demi-FIFO d'entrée (la demi-FIFO `id1` devient liée à demi-FIFO `id3` en remplacement à la demi-FIFO `id2`), alors que la seconde correspond à un changement de liaison impliquant la demi-FIFO de sortie (la demi-FIFO `id2` devient liée à la demi-FIFO `id3` en remplacement à la demi-FIFO `id1`).
- *suspend* `id` : suspend l'exécution d'un processus ;
- *resume* `id` : reprendre l'exécution d'un processus suspendu ;
- *replace* `id1 id2` : remplace un procesus par un autre ;
- *end_exec* `id` : termine l'exécution d'un processus ;
- *show* : affiche toutes les liaisons entre les demi-FIFOs ;
- *length* : permet de récupérer le nombre de jetons dans toutes les demi-FIFOs du réseau ;
- *set_capacity* `id n` : permet de fixer la capacité maximale d'une demi-FIFO à n jetons ;
- *set_max* `id n` : permet de fixer le seuil maximal d'une demi-FIFO de sortie ;
- *set_min* `id n` : permet de fixer le seuil minimal d'une demi-FIFO d'entrée ;

- *fifolen* `id` : permet de récupérer le nombre de jeton de la demi-FIFO qui a l'identificateur fournit en paramètre `id` ;
- *state* : permet de récupérer des informations sur les ressources de calcul du système ;
- *check* : vérifie l'intégrité des différents processus du réseau pour détecter d'éventuelles pannes.

Dans les sections suivantes, nous allons présenter en détail les différents aspects que notre approche permet de procurer.

4.4 Développement incrémental de l'application répartie

4.4.1 Création et ajout d'un processus

L'ajout d'un processus dans l'application se fait de manière implicite, et complètement indépendante des autres composants. Deux méthodes sont utilisées pour y parvenir :

- Lancement du processus à partir d'un exécutable le représentant. Contrairement à une application classique client/serveur où le serveur doit être exécuté avant le lancement du client, aucune contrainte sur l'ordre de lancement n'est imposée. En plus, l'enregistrement du processus et de ses demi-FIFOs se fait automatiquement sans intervention du programmeur ;
- Lancement du processus en appelant la fabrique qui le gère.

4.4.2 Ajout d'une liaison

L'utilisation de la console (cf 3.2.1) permet d'ajouter des liens entre processus de manière interactive. L'ajout d'un lien consiste à lier une demi-FIFO d'entrée avec une demi-FIFO de sortie. Ces deux demi-FIFOs doivent être du même type, à savoir qu'elles doivent implémenter la même interface.

4.5 Migration et remplacement de composant

Pour pouvoir remplacer un processus par un autre ou le faire migrer, il faut garantir que la cohérence de l'application soit préservée. Ceci nécessite la reprise de l'exécution au même point, la préservation des liens avec les autres processus, et le transfert de l'état interne du processus.

Algorithme Le remplacement des processus se fait interactivement à partir de la console. Le scénario du remplacement est le suivant (voir les figures 4.3, 4.4 et 4.5) :

- Les demi-FIFOs connectées au processus qui doit être remplacé sont prévenues pour ne rien envoyer, ni rien demander. Ceci permet d'éviter la perte de données, ou l'attente d'une réponse qui ne viendrait jamais ;

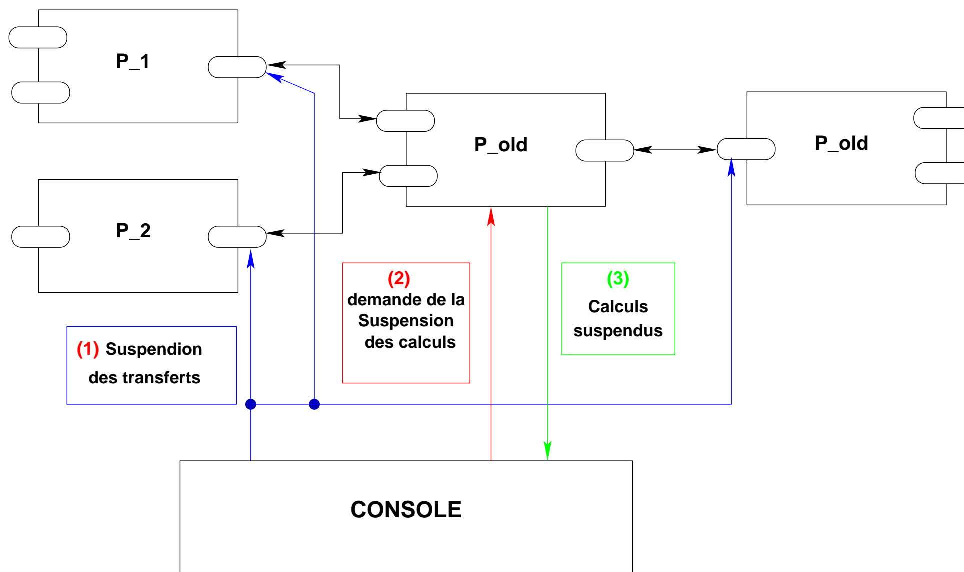


FIG. 4.3 – Scénario de remplacement (Étape 1)

- La console invoque l'objet calcul du processus qui doit être remplacé pour qu'il suspende ses traitements, en appelant la méthode `suspend_Computation`. Après réception de cette invocation et la suspension effective des calculs, l'objet de calcul envoit son état interne à l'objet calcul du nouveau processus, et une confirmation de la suspension des calculs est envoyée à la console ;
- Après réception de la confirmation de l'objet calcul du processus à remplacer, la console ordonne à toutes les demi-FIFOs de ce processus de transférer les données encore stockées vers les demi-FIFOs du nouveau processus. Chaque demi-FIFOs de l'ancien processus recevera la référence de la nouvelle demi-FIFO correspondante et procédera au transfert de données.
- Les processus qui étaient liés au processus remplacé sont liés avec le nouveau processus. Ceci est fait en restaurant la structure des liaisons des demi-FIFOs de l'ancien processus et en les projetant sur le nouveau processus ;
- La dernière étape consite à procéder au lancement du traitement dans le nouveau processus à partir du point de reprise du processus remplacé.

4.6 Reconfiguration d'un processus

4.6.1 Suspension/Reprise de l'exécution

La suspension et la reprise de l'exécution se font par une invocation de l'objet de calcul à partir de la console. Cette opération peut-être utile pour analyser le comportement de l'application, ou équilibrer les charges manuellement sans pour autant remplacer le processus. La suspension de l'exécution ne peut se faire que lorsque le processus atteint un point de reprise.

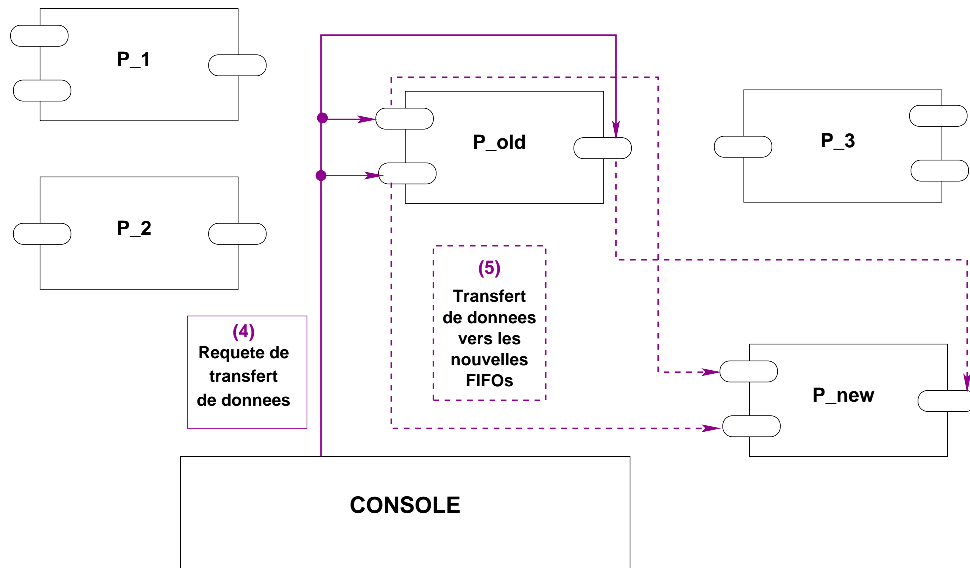


FIG. 4.4 – Scénario de remplacement (Étape 2)

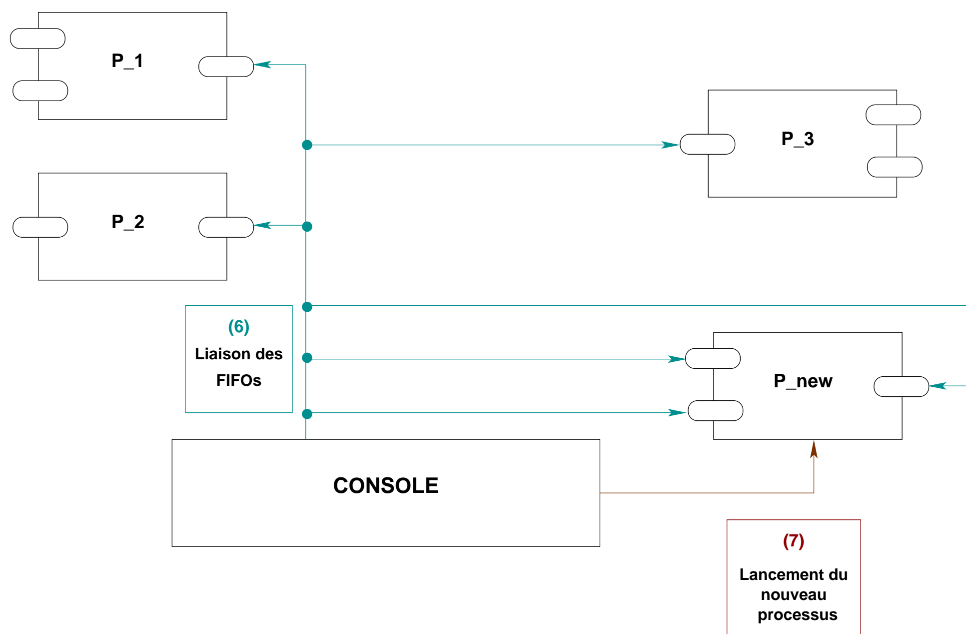


FIG. 4.5 – Scénario de remplacement (Étape 3)

4.6.2 Suppression d'un processus

La suppression d'un processus implique qu'aucun traitement ne sera effectué dans celui-ci. Elle implique aussi qu'aucune communication avec les demi-FIFOs des autres processus n'est autorisé. Cela se traduit par une suppression des liaison avec les autres processus et une restitution des données aux autres demi-FIFOs, une suspension des calcul dans le processus, et éventuellement un arrêt d'exécution du processus.

Algorithme

- Supprimer les liens qui existent entre les demi-FIFOs du processus à supprimer et les autres demi-FIFOs. Pour chacune de ces FIFOs, la console invoque la méthode `unlink`
- Suspendre le traitement à l'intérieur du processus à supprimer.
- Transférer le contenu des demi-FIFOs du processus à supprimer vers les demi-FIFOs qui étaient liées à celles-ci.
- Mettre à jour des liens dans la console.
- Arrêter l'exécution du processus, en appelant la méthode `end_processing`.

4.6.3 Suppression/Modification d'une liaison

La suppression d'une liaison consiste à supprimer le lien qui existe entre deux demi-FIFOs. Ceci se fait par invocation des deux demi-FIFOs liées entre elles pour la suppression des références qu'elles ont l'une de l'autre. Les données présentes dans chacune des deux demi-FIFOs sont gardées.

La modification d'une liaison est le changement du lien qui existe entre deux demi-FIFOs. Ce changement implique la suppression du lien entre les deux FIFOs pour éviter l'envoi et la perte de données, le transfert du contenu de l'ancienne FIFO vers la nouvelle et finalement la mise à jour des liens.

Algorithme

- Supprimer le lien qui existe entre les deux demi-FIFOs.
- Suspendre le traitement dans le processus concerné par la modification.
- Transférer le contenu de l'ancienne demi-FIFO vers la nouvelle.
- Reprendre l'exécution dans le processus.
- Mettre à jour les liens dans la console.

4.6.4 Changement des paramètres

Le changement des paramètres des demi-FIFOs peut être effectué à partir de la console. En effet, l'utilisateur peut changer pendant l'exécution la capacité maximale d'une demi-FIFO, le seuil maximal d'une demi-FIFO de sortie, ou le seuil minimal d'une FIFO d'entrée.

4.7 Exemple : Application de traitement de signal

Nous allons illustrer notre modèle par un exemple d'une application concrète de traitement de signal.

4.7.1 Description de l'application

Cette application dont le schéma est décrit sur la figure 3.22, permet d'extraire les concordances de fréquences provenant d'une même direction (donc appartenant vraisemblablement au même objet). Elle est basée sur des fonctions de traitement de signal élémentaires telles la transformée de Fourier rapide (FFT) et l'intégration discrète.

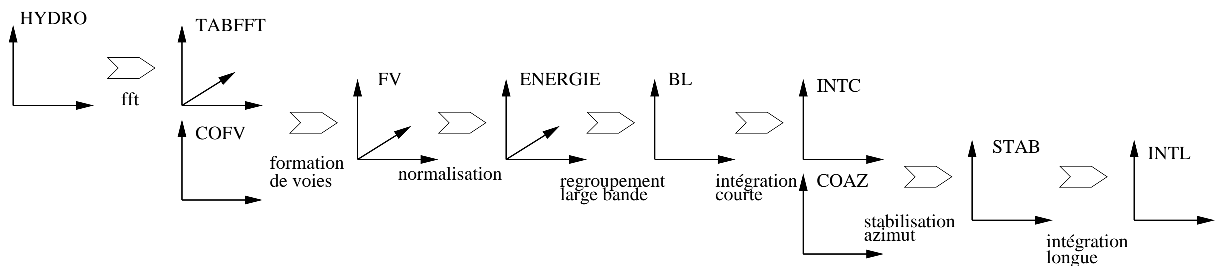


Fig. 4.6 – Description de l'application

- Les *hydrophones*, un tableau ($h = 512 \times T = \infty$), contient les données en entrée de l'application. Il représente un flux de données continu d'un ensemble de 512 capteurs.
- La première tâche calcule une FFT pour chaque capteur pour une période de 512 unités de temps. Elle produit `TABFFT`, un tableau tridimensionnel ($512 \times 256 \times \infty$).
- La seconde tâche calcule les voies pour chaque période, fréquence et l'ensemble des capteurs. Sa sortie est un tableau tridimensionnel ($t = 128 \times T = \infty \times f = 200$), FV.
- Les voies sont ensuite traitées successivement par plusieurs fonctions pour à la fin extraire les caractéristiques des fréquences. Les dimensions des tableaux en entrée et en sortie sont :

Fonction	Tableau d'entrée	Tableau de sortie
Normalisation	$(128 \times \infty \times 200)$	$(128 \times \infty \times 200)$
Regroupement large bande	$(128 \times \infty \times 200)$	$(128 \times \infty)$
Intégration courte	$(128 \times \infty)$	$(128 \times \infty)$
Stabilisation	$(128 \times \infty)$	$(128 \times \infty)$
Intégration longue	$(128 \times \infty)$	$(128 \times \infty)$

À cause de la rapidité des cinq dernières tâches relativement aux deux premières et à la puissance de calcul disponible, elles ont été regroupées en un seul composant. On

obtient ainsi trois composants formant un pipeline linéaire. La première tâche traite des tableaux de $h = 512$ (hydrophones) et ainsi de suite. La dimension du temps (qui est la dimension infinie) devient ainsi les jetons successifs produit par la première tâche. La figure 4.7 montre la distribution de l'application en trois tâches.

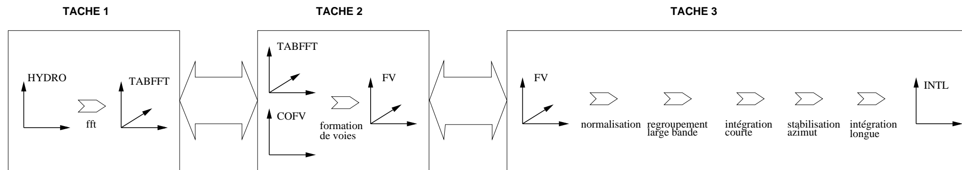


Fig. 4.7 – Distribution de l'application VBL

Le programme séquentiel utilisé pour les analyses de performances casse aussi la dimension infinie, mais une seule tâche exécute l'application entière.

4.7.2 Mesures de performance

Pour les mesures de performance, nous avons utilisé un réseau de trois machines. L'implémentation a été faite avec le langage C++[Str01], et l'ORB ORBacus [Obj00]. Plusieurs mesures ont été effectuées (voir tableau 4.1) : le programme séquentiel exécuté sur un Pentium III à 1 Ghz, trois processus s'exécutant sur deux Pentium III à 1 Ghz et un Pentium II à 266 Mhz reliés par un bus Ethernet partagé à 10 Mb/s. On note que les machines étaient utilisées légérement par d'autres processus, le réseau utilisé est lui par contre relativement chargé.

Le tableau 4.1 montre les différentes mesures de performances effectuées. Les colonnes deux et trois contiennent les temps d'exécution de la version séquentielle du programme, respectivement sans et avec utilisation de CORBA. La colonne. Les colonnes suivantes contiennent dans l'ordre, les temps d'exécution de l'application distribuée (cluster CORBA), et les temps d'exécution des trois tâches qui la composent.

# de tokens	Séquentiel P3	CORBA P3	cluster CORBA	1ère Tâche P3	2ème Tâche P3	3ème Tâche P2
5000	51 s	52 s	32 s	22 s	26 s	10 s
10000	100 s	104 s	62 s	41 s	48 s	20 s
15000	154 s	168 s	97 s	56 s	68 s	29 s
20000	203 s	214 s	123 s	75 s	91 s	38 s
25000	246 s	271 s	148 s	94 s	115 s	48 s
30000	309 s	322 s	173 s	118 s	131 s	57 s
35000	362 s	382 s	198 s	141 s	156 s	65 s
40000	408 s	430 s	230 s	166 s	180 s	74 s
45000	459 s	496 s	263 s	184 s	205 s	83 s
50000	516 s	570 s	290 s	210 s	236 s	93 s
2 500 000	26120 s	N.A.	15980 s	11270 s	12590 s	4570 s
2 750 000	27580 s	N.A.	17730 s	12910 s	13140 s	5120 s
3 000 000	29050 s	N.A.	19096 s	13550 s	13800 s	5580 s

Tab. 4.1 – Performances

Travaillant en régime continu, les temps de traitement de la version séquentielle et l'implémentation CORBA sont similaires. Le surcoût de l'utilisation de la couche CORBA sans communication est inférieur à 10% (avec la même machine monoprocesseur P3).

L'accélération obtenue avec la version distribuée nous permet d'augmenter la fréquence d'acquisition dans les mêmes proportions. Cette accélération dûe à la distribution montre bien le recouvrement des calculs par les communications. L'application globale est légèrement (22%) plus lente que la seconde tâche. La rapidité des cinq dernières tâches par rapport aux deux autres explique l'accélération de 1, 9 sur trois machines. La figure 4.8 montre les temps relatifs des tâches et le temps total de l'application.

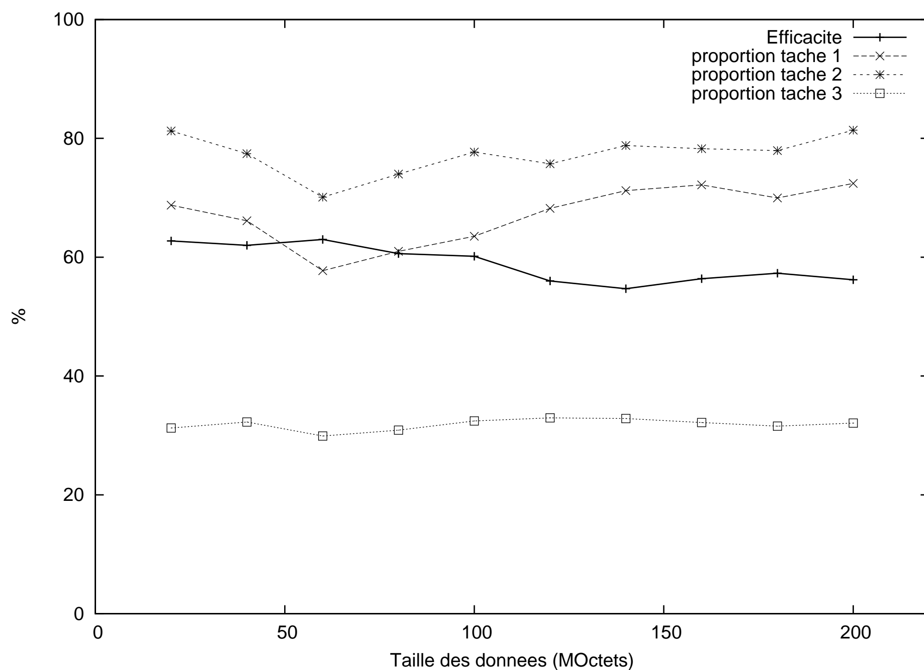


Fig. 4.8 – Performance de l'exécution distribuée - charge des trois tâches

Redistribution dynamique et l'évolution des tailles des demi-FIFOs. Le tableau 4.2 et la figure 4.9 montrent les temps d'exécution et l'évolution de la longueur des demi-FIFOs entre les deux premiers composants. Pendant l'exécution en régime continu, le second composant qui s'exécutait sur une machine de type PC équipée d'un Pentium II 266 Mhz, a été remplacé par un autre composant tourant sur une machine plus puissante équipée d'un processeur Pentium III à 1 Ghz.

Le tableau 4.2 montre les temps d'exécution de l'application dans la configuration initiale (un Pentium III pour la première tâche, un Pentium II pour la seconde tâche, et un Pentium II pour la troisième tâche). La troisième colonne montre les temps d'exécution en faisant migrer la deuxième tâche pendant l'exécution vers une machine plus puissante..

La figure 4.9 montre une diminution immédiate et significative de la longueur de la demi-FIFO, ce qui autorise une fréquence d'acquisition plus grande dans cette nouvelle configuration. On obtient ainsi une accélération de l'exécution et une charge moins importante pour les ressources de calcul.

Taille des données (MOctes)	P3-P2-P2	P3-P3-P2
20	80 s	47 s (remplacement après 30 s)
40	166 s	90 s (remplacement après 60 s)
60	258 s	158 s (remplacement après 90 s)
72	302 s	188 s (remplacement après 120 s)

TAB. 4.2 – Redistribution dynamique : la seconde tâche migre d'une machine P2 vers une machine P3 durant l'exécution

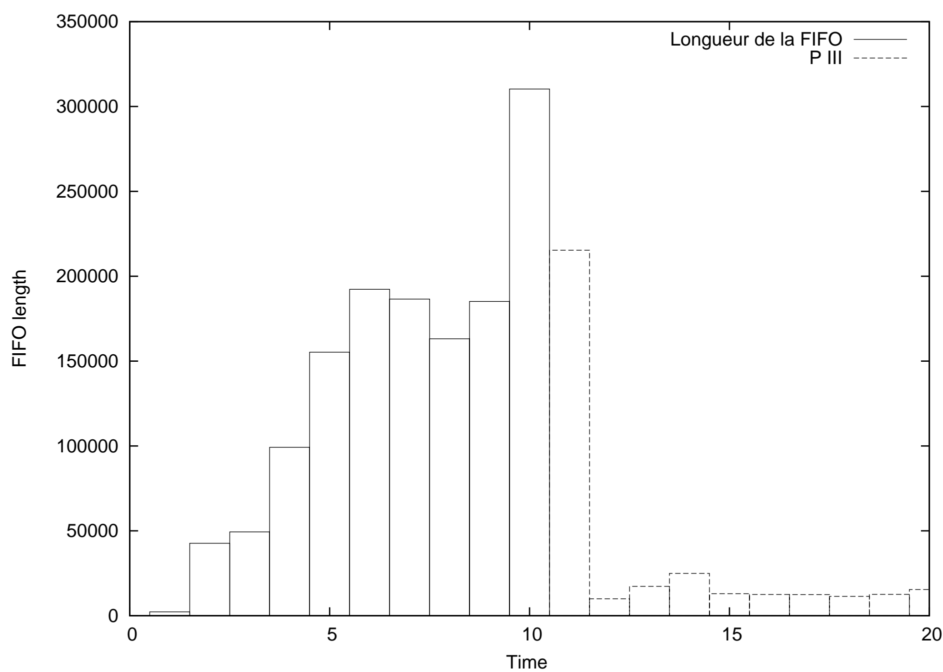


FIG. 4.9 – Évolution de la longueur des demi-FIFOs durant la redistribution

4.8 Conclusion

La dynamicité d'une application distibuée peut être présentée selon plusieurs points de vue. Ces points de vue qu'on vient de présenter dans ce chapitre, permettent de procurer une plus grande flexibilité aux développeurs d'applications distribuées basées sur le modèle des réseaux de processus. Cette flexibilité se traduit de deux façons différentes :

1. Une flexibilité de fonctionnement où le déploiement, l'exécution, et le contrôle de l'application sont gérés de manière indépendante du code de l'application.
2. Une flexibilité de développement qui permet de développer l'application de manière incrémentale.

Notre approche de remplacement et de migration de composants est basée sur les points de reprise. Ces points de reprise doivent être spécifiés explicitement par le programmeur, car le modèle des réseaux de processus de Kahn est très large et n'impose aucune restriction sur le code des processus. Dans les applications Array-OL (qui s'approchent des réseaux de processus synchrones), qu'on va présenter dans le chapitre suivant, ces points de reprise ne sont pas intrusifs et peuvent être définis automatiquement au début ou à la fin de la fonction de calcul [23]. De plus l'état interne du processus est plus facile à gérer.

[23] Même si le modèle de Kahn n'impose aucune restriction sur les fonctions des processus, en pratique la fonction de calcul des processus d'un grand nombre d'applications basées sur ce modèle est une répétition d'une fonction élémentaire

5

Array-OL sur réseaux de processus distribués

Sommaire

5.1 Introduction 109
5.2 Langage Array-OL 110
5.3 Array-OL et réseaux de processus 117
5.4 Évaluation 122
5.5 Conclusion 124

Une optimisation prématurée est la source de tous les maux.
D. Knuth.

5.1 Introduction

Array-OL (*Array Oriented Language*) est un langage de spécification d'applications de traitement de signal systématique. Dans ce type d'application, les traitements effectués sur les données par les différentes tâches sont réguliers. Array-OL permet seulement de spécifier les différentes dépendances qui existent dans une application. Ces dépendances concernent les tableaux et les tâches, ou à un niveau de granularité plus fin, les traitements effectués sur les éléments du tableau à chaque exécution de la tâche. Array-OL définit implicitement les opérations nécessaires pour développer un programme avec ce formalisme. Ceci le rend indépendant des différents langages de programmation. Par contre, il ne définit pas de modèle d'exécution précis. Le parallélisme exprimé par le langage est le parallélisme de données, où les traitements sur les différents éléments d'un tableau peuvent se faire indépendamment les uns des autres et le parallélisme de tâches.

Ce chapitre va présenter un modèle d'exécution de type « Réseaux de processus » pour ARRAY-OL. Tout d'abord, le langage ARRAY-OL est présenté, ainsi que les transformations de code qui ont été élaborées afin d'optimiser l'ordonnancement et le stockage, ainsi que pour gérer les problèmes de l'infini. Ces transformations de code permettent ainsi d'autoriser l'utilisation d'un schéma d'exécution séquentiel des tâches, tout en gardant le parallélisme de données qui existe à l'intérieur de chaque tâche. Notre démarche consiste donc à fournir un modèle d'exécution de type « Réseaux de processus » [24], où le parallélisme est exploité à deux niveaux :

- parallélisme de données : le parallélisme à l'intérieur des processus permet d'effectuer les traitement sur les éléments d'un tableau d'une façon « data-parallèle ». Le déterminisme du modèle n'est pas pour autant affecté, et l'introduction de ce type de parallélisme à l'intérieur d'un processus ARRAY-OL garde le déterminisme quelque soit l'ordonnancement ;
- parallélisme de tâches : ce parallélisme permet d'exploiter les architectures distribuées, et de créer des simulations distribuées sur des systèmes hétérogènes ;

L'avantage principal de notre approche est d'exploiter au maximum le parallélisme potentiel du modèle ARRAY-OL, dans un environnement de simulation réparti. Le modèle d'exécution d'ARRAY-OL tel qu'il a été défini dans les travaux précédents est un modèle d'exécution séquentiel/data-parallèle, car l'enchaînement des différentes tâches est séquentiel, et data-parallèle à l'intérieur de la tâche. Ce mode d'exécution ne permet pas de développer des simulations réparties, sauf en distribuant les motifs sur les différentes tâches data-parallèles.

En s'appuyant sur le support d'exécution présenté dans les chapitres 3 et 4, nous proposons et implémentons un modèle et un support d'exécution pour des applications ARRAY-OL. En plus d'exploiter le parallélisme du modèle, la dynamicité que procure notre support d'exécution est mieux exploitée car les points de reprise dans les applications ARRAY-OL ne sont pas intrusifs.

Ce chapitre se décompose en quatre sections. La première section présente le langage ARRAY-OL et les transformations de code. Puis la deuxième expose le modèle d'exécution de type « réseaux de processus » et sa mise en œuvre. Enfin, la dernière section évalue les performances de notre modèle d'exécution.

5.2 Langage ARRAY-OL

ARRAY-OL est un langage de spécification pour applications de traitement de signal. Ce langage permet de construire de telles applications en :

– spécifiant les dépendances de données ;
– utilisant les notions de réutisabilité, en séparant les fonctions de calcul de la construction de l'application ;

Du fait de ses caractéristiques, ce langage est particulièrement bien adapté à la programmation visuelle. En plus du fait d'être basé sur un modèle à graphe de dépendances, l'aspect régulier des calculs dans ARRAY-OL permet de décrire les opérations d'accès

[24]Le réseaux de processus peut être distribué ou pas

aux tableaux visuellement. C'est ainsi que plusieurs environnements de programmation visuelle ont été développés autour du langage :

- L'environnement PTOLEMY a été étendu pour spécifier graphiquement (programmation par l'exemple) une application ARRAY-OL par l'enchaînement de tâches élémentaires. Par le biais d'un *compréhenseur*, l'interface de programmation génére automatiquement le code source de l'application en partant d'une spéfication textuelle propre à ARRAY-OL.
- L'environnement GASPARD [BDL+01, DBDM02] a été développé au LIFL pour disposer d'un atelier complet de développement.

ARRAY-OL définit deux niveaux de spécification. Un niveau d'abstraction plus haut appellé *modèle global*, où l'application est définie par un graphe de dépendances entre tâches et tableaux. Le second niveau d'abstraction plus détaillé est appelé *modèle local*. Il décrit les traitements effectués sur les éléments des tableaux.

Le modèle global Dans le modèle global, l'application est décrite comme un graphe de dépendances. Le modèle définit et nomme les tâches et les tableaux. Dans ce diagramme, les nœuds du graphe représentent les tâches et les tableaux représentent les dépendances entre ces tâches. Ces dépendances sont des tableaux. La représentation du modèle globale sous forme de graphe, laisse croire que le modèle est très proche d'un graphe à flux de données mais l'interprétation est légèrement différente. En effet, le modèle ARRAY-OL, possède certaines caractéristiques différentes du modèle à flux de données :

- les arcs portent des tableaux multidimensionnels ;
- le modèle plus proche des réseaux de processus synchrones, met l'accent sur le parallélisme de données que sur le parallélisme de tâches.

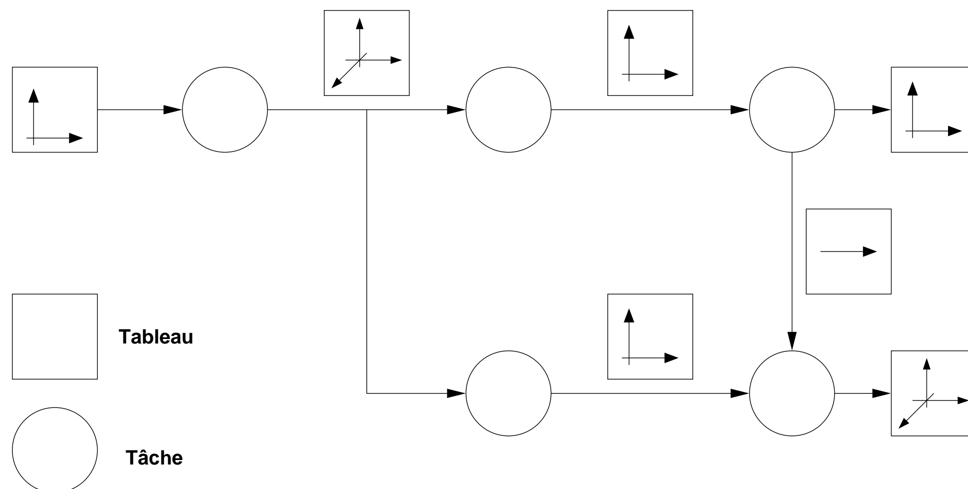


FIG. 5.1 – Le modèle global d'ARRAY-OL

Le modèle local Le modèle local spécifie pour chaque tâche la dépendance entre les éléments de ses tableaux d'entrée et les éléments de ses tableaux de sortie. Ces groupes d'éléments de tableaux auquels on applique le même calcul sont appelés « Motifs », et

l'unité de calcul appliquée à ces motifs « Tâche élémentaire » ou *TE*. Ainsi une tâche ARRAY-OL peut être considérée comme une fonction qui lit des tableaux en entrée et qui produit les tableaux résultats. Cette fonction est décomposable en calculs plus fins qui sont les instances des TE, et utilise les motifs des tableaux d'entrée pour calculer les motifs de sortie et ainsi construire le tableau résultat. Pour y parvenir, deux opérateurs ont été définis :

A. L'ajustage : permet de construire les motifs, qui sont les éléments de base pour un traitement. Ces motifs sont construits à partir d'une origine dans le tableau et d'un ensemble de vecteurs d'ajustage (autant que de dimensions dans le motif). En utilisant ces vecteurs et l'origine, une succession de décalages est efféctuée pour accéder aux différents points qui composent le motif.

Si on reprend l'exemple de la figure 5.2, les motifs n'ont qu'une seule dimension. Ils sont de taille 6 et définis par le vecteur d'ajustage $\begin{pmatrix} 1 \\ 1 \end{pmatrix}$.

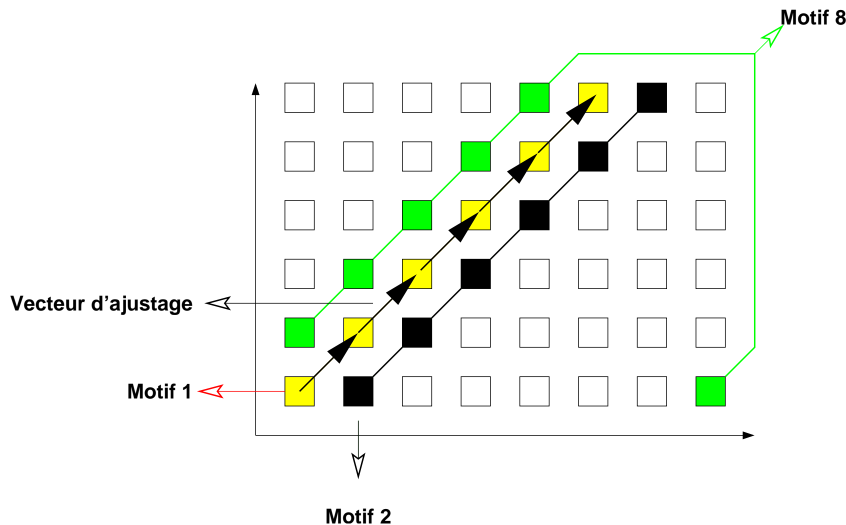


FIG. 5.2 – L'opération d'ajustage dans ARRAY-OL

B. Le pavage : le pavage permet de parcourir (en lisant ou en écrivant) le tableau de données. Il permet aussi de faire le lien entre les motifs d'entrée et les motifs de sortie. Un ensemble de vecteurs de pavage et l'origine du tableau sont nécessaires pour définir le nouveau motif. Par exemple, les vecteurs de pavage (ou matrice de pavage) du tableau opérande de la figure 5.3 sont $\begin{pmatrix} 0 & 0 \\ 1 & 2 \end{pmatrix}$. Les bornes du domaine d'itération sont $\begin{pmatrix} 2 \\ +\infty \end{pmatrix}$.

Pour le tableau résultat, la matrice de pavage est $\begin{pmatrix} 2 & 0 \\ 0 & 1 \end{pmatrix}$ et les bornes du domaine d'itération sont $\begin{pmatrix} 2 \\ +\infty \end{pmatrix}$.

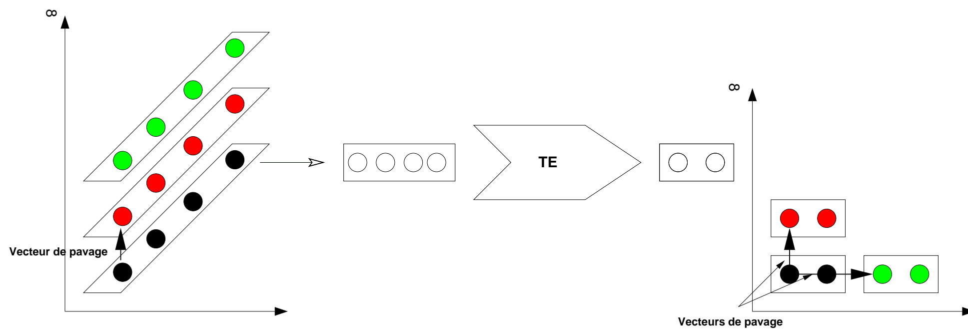


FIG. 5.3 – L'opération de pavage dans ARRAY-OL

Remarque : Les calculs étant dirigés par le tableau de sortie, le vecteur des bornes d'itération de pavage de ce dernier doit être celui des tableaux opérandes. Ainsi le tableau opérande de la figure 5.3 aurait pu avoir, sans cette contrainte, une matrice de pavage égale à $\begin{pmatrix} 0 \\ 1 \end{pmatrix}$ et un vecteur des bornes d'itération de pavage égal à $\begin{pmatrix} +\infty \end{pmatrix}$.

5.2.1 Formalisme matriciel d'ARRAY-OL

Une description formelle des spécifications ARRAY-OL est faite à l'aide d'un formalisme matriciel. Nous introduisons ici ce formalisme défini dans [DLB+95] :

Définition Soit T une tâche, et A_T^{in} et A_T^{out} désignent respectivement ses tableaux d'entrée et de sortie, alors pour tout tableau $M \in A_T^{in} \cup et A_T^{out}$:

- $\mathcal{O}_{M,T}$ définit l'origine du pavage dans le tableau ;
- $\mathcal{P}_{M,T}$ et $\mathcal{F}_{M,T}$ désignent respectivement les matrices de pavage et d'ajustage ;
- $Q_{M,T}$ et $D_{M,T}$ désignent respectivement les vecteurs des bornes d'itération de pavage et d'ajustage ;

En considérant l'exemple de la figure 5.4, et en supposant que le tableau d'entrée a deux dimensions (4×6), et le tableau de sortie deux dimension (4×3), alors la description matricielle de la tâche est la suivante :

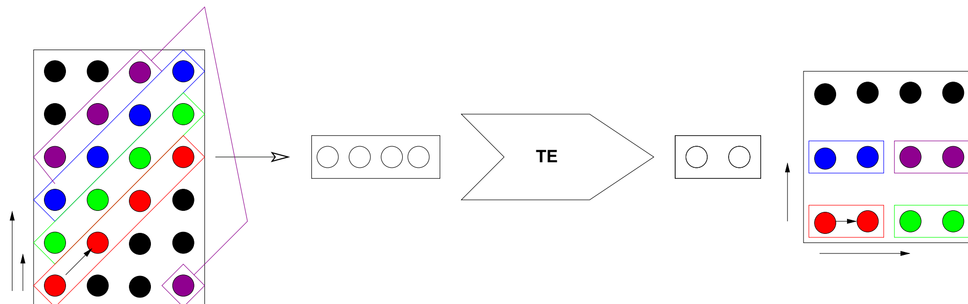


FIG. 5.4 – Exemple d'une tâche ARRAY-OL

- les motifs d'entrée sont des tranches de 4 éléments.

La matrice d'ajustage est $\begin{pmatrix}1\\1\end{pmatrix}$, et les bornes du domaine d'itération sont $\begin{pmatrix}4\end{pmatrix}$.

La matrice de pavage est $\begin{pmatrix}0 & 0\\1 & 2\end{pmatrix}$, et les bornes du domaine d'itération sont $\begin{pmatrix}c2\\3\end{pmatrix}$

– les motifs de sortie sont des tranches de deux éléments.

La matrice d'ajustage est $\begin{pmatrix}1\\0\end{pmatrix}$, et les bornes du domaine d'itération sont $\begin{pmatrix}2\end{pmatrix}$.

La matrice de pavage est $\begin{pmatrix}2 & 0\\0 & 1\end{pmatrix}$, et les bornes du domaine d'itération sont $\begin{pmatrix}2\\3\end{pmatrix}$

A partir des matrices et des vecteurs des bornes d'itération du pavage et d'ajustage et du point d'origine, on peut calculer les points constituant les différents motifs du tableaux. Ainsi, le calcul des coordonnées de l'ensemble des origines des motifs se fait par la formule :

$$\mathcal{O}_{q,M} = mod_M(\mathcal{P}_{M,T}.q + \mathcal{O}_{M,T});\ \forall 0 \leq q \leq Q_{M,T}$$

Pour chaque origine, la construction du motif correspondant est obtenue par le calcul des coordonnées de ses points :

$$\mathcal{O}_{q,M} + mod_M(\mathcal{F}_{M,T}.d);\ \forall 0 \leq d \leq D_{M,T}$$

Hiérarchie Le langage ARRAY-OL supporte la notion de hiérarchie. En effet, une tâche peut être elle même un graphe de tâches. Cette notion, très utilisée dans les graphes à flux de données, prend un autre intérêt dans le schéma d'exécution classique d'ARRAY-OL qui est un schéma d'exécution séquentiel. En effet dans un tel schéma d'exécution, l'enchaînement des tâches est séquentiel, et le parallélisme est exploité dans le traitement data-parallèle à l'intérieur de chaque tâche.

5.2.2 Contraintes d'ARRAY-OL

Les spécifications d'ARRAY-OL incluent un certain nombre de contraintes, qui concernent principalement les tableaux résultats :

1. les dimensions des tableaux ne comportent, au plus, qu'une dimension infinie ; cela représente sémantiquement un flux continu de données ;
2. les motifs sont de taille finie ;
3. le graphe de tâches d'une application ne comporte pas de cycle ;
4. tout point d'un tableau de sortie est calculé une et une seule fois (i.e les motifs de sortie ne se chevauchent pas et pavent entièrement le tableau) ;
5. les vecteurs d'ajustage et de pavage associés aux tableaux résultats sont positifs et parallèles aux axes ;
6. les tableaux résultats ne sont pas toriques à l'inverse des autres tableaux ARRAY-OL.

Les trois dernières contraintes ne concernent que les tableaux résultats, et ne sont pas obligatoires dans les tableaux en entrées.

5.2.3 Transformations de code ARRAY-OL

Les transformations de code [SMDD01, Sou01] ont été proposées pour obvier aux limitations du modèle. Ces limitations peuvent être résumées dans les points suivants :

1. la gestion de l'infini : le schéma d'exécution classique du modèle est le schéma séquentiel. Une tâche doit compléter son exécution pour que la seconde puisse être exécutée. Cet enchaînement des tâches ne permet pas d'exécuter des tâches manipulant des flux de données infinis, ce qui est très courant dans les applications de traitement de signal.
2. les structures des motifs en entrée et en sortie : les motifs résultats d'une tâche ne sont pas forcément les motifs opérandes de la tâche suivante, ce qui n'autorise pas un enchaînement des tâches en produisant un motif par exécution. Dans ce cas, la transformation proposée à pour but de produire un nombre suffisant de motifs résultats, pour construire un tableau [25] qui regroupe un ou plusieurs motifs opérandes de la tâche suivante. Dans les réseaux de processus synchrones, les flux de données ont une seule dimension, ce qui réduit le problème pour trouver le nombre d'itérations qui permet de construire un groupe de motifs résultats entiers dont la taille est égale à la taille du motif opérande de la tâche suivante. Dans le domaine des applications de traitement de signal multidimensionnelles, le problème concerne aussi la structure et la position des motifs dans le tableau.
3. la discontinuité des motifs : les motifs dans ARRAY-OL ne sont pas forcémment continus et peuvent se recouvrir. De ce recouvrement résulte une redondance des calculs assez pénalisante, surtout si le schéma d'exécution est séquentiel. En effet, les parties communes aux motifs risquent d'être calculées plusieurs fois dans un tel schéma comme le montre la figure 5.5.

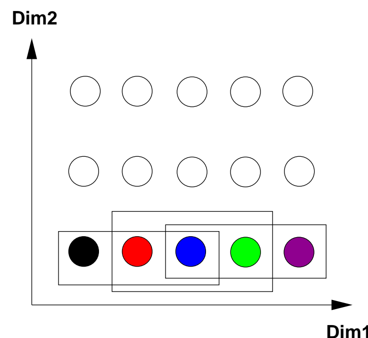


FIG. 5.5 – Exemple de recouvrement : les motifs opérandes de taille (1×3) se recouvrent de deux points suivant la dimension *Dim1*

4. la gestion mémoire : la taille des tableaux ARRAY-OL intermédiaires peut être importante, car les applications de traitement de signal manipulent un grand nombre de données en entrée pour n'en produire qu'un nombre beaucoup plus petit en sortie (filtrage, élimination de bruit, etc.). Les transformations de code permettent de controler la taille de cette mémoire, et de la réduire si possible.

[25] appelé couramment macro-motif

La fusion

La transformation de base dans ARRAY-OL est la fusion. Elle consiste à transformer une application composée de deux tâches en une seule. La nouvelle tâche qu'on appelle « tâche hiérarchique » est constituée de deux sous-tâches. L'objectif derrière cette transformation automatique est d'arriver à obtenir à chaque itération de pavage un motif de la tâche résultat.

Contrairement à la séquence initiale qui manipule trois tableaux [26], la nouvelle tâche hiérarchique manipule deux tableaux seulement, le tableau consommé par la première tâche et le tableau produit par la deuxième tâche. Le tableau intermédiaire est supprimé. À l'intérieur de la tâche hiérarchique, les deux sous tâches manipulent trois macros-motifs. Le macro-motif produit par la première sous-tâche permet à la sous-tâche suivante de s'exécuter. Il faut noter que l'application de la fusion produit des macros-motifs d'une taille minimale, et que cette transformation élémentaire peut être appliquée à plusieurs tâches par enchaînement.

L'exemple de la figure 5.7 illustre la transformation de trois tâches T_1, T_2 et T_3 en une seule tâche hiérarchique.

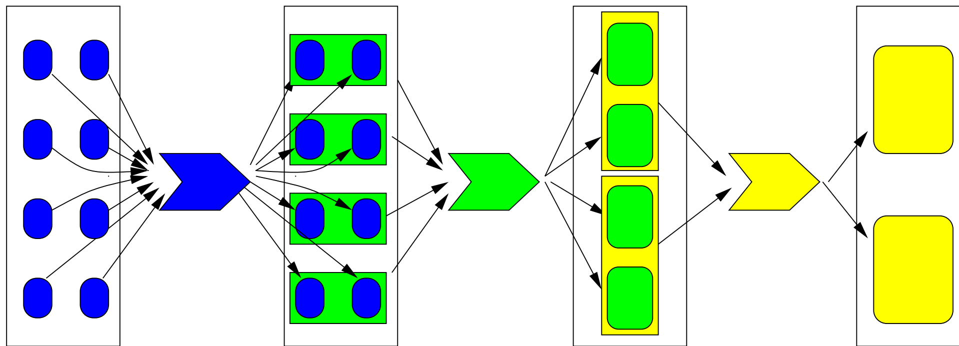


FIG. 5.6 – Application originale

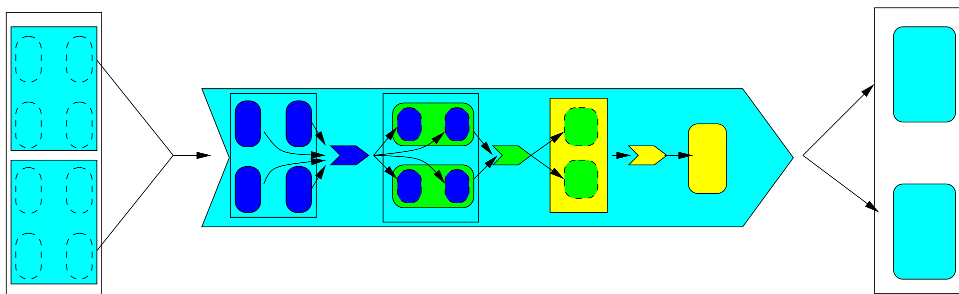


FIG. 5.7 – Fusion des trois tâches en une tâche hiérarchique

[26]Pour simplifier, nous admettons que chacune de deux tâche consomme un tableau et produit un tableau

5.3 ARRAY-OL et réseaux de processus

Nous avons vu dans le chapitre 2 que les réseaux de processus de Kahn et leurs sous modèles (SDF, BDF) sont bien adaptés pour modéliser et exécuter les applications de traitement de signal. Dans ce contexte, nous nous sommes intéressés à la projection du modèle ARRAY-OL sur le modèle des réseaux de processus distribués.

Bien qu'ARRAY-OL soit plus proche du modèle des réseaux de processus synchrones, le modèle des réseaux de processus de Kahn a été préféré à celui-ci. Le modèle SDF perd de son intérêt dans un schéma d'exécution distribué ou multiprocesseurs, où il est plus judicieux d'utiliser un ordonnancement de tâches décentralisé et plus dynamique.

5.3.1 Déterminisme et data-parallélisme dans ARRAY-OL

Le modèle des réseaux de processus de Kahn est déterministe. Or, on a vu dans la section 2.1.2 du chapitre 2 que l'un des moyens d'introduire le non déterminisme dans le modèle est le parallélisme à l'intérieur des processus. Néanmoins, l'introduction d'ARRAY-OL et du data-parallélisme dans les processus garde la sémantique déterministe du modèle.

En effet l'une des caractéristiques d'ARRAY-OL est le non recouvrement des motifs des tableaux résultats. De cette caractéristique, on peut déduire que les points des tableaux résultats sont calculés une et une seule fois (assignation unique). De plus, il existe une correspondance unique entre les motifs d'entrée et les motifs de sortie qui préserve le déterminisme des calculs.

5.3.2 Modèle d'exécution

Le langage ARRAY-OL ne spécifie aucun modèle d'exécution et peut être projeté vers plusieurs schémas d'exécution selon les besoins et les objectifs. Parmi ces modèles d'exécution, on peut en citer trois principaux :

1. Séquentiel : les tâches sont exécutées les unes après les autres ;
2. SPMD : l'exécution à l'intérieur de la tâche est parallélisée pour bénéficier du fort degré de parallélisme exprimé par ARRAY-OL ;
3. Pipeline : dans ce modèle d'exécution qui nous intéresse paticulièrement, les différents tâches que comporte l'application sont pipelinés pour former un flux de tableaux.

L'exécution en pipeline par construction d'un flux de tableaux peut aussi être combinée aux deux autres modèles d'exécution. On peut obtenir ainsi un pipeline de tableaux avec certaines tâches qui sont séquentielles ou data-parallèles.

Exécuter une application ARRAY-OL dans un environnement parallèle a déjà été étudié et implémenté dans [Sou01]. Cette proposition consistait à exécuter sur une architecture à mémoire distribuée les différents itérateurs (ou tâches data-parallèles). Ce schéma d'exécution dont l'implémentation utilisait la bibliothèque de passage de message PVM, permet ainsi de traiter en parallèle les différents motifs en entrée d'une tâche. Vu de l'intérieur de la tâche, le schéma d'exécution est parallèle distribué. Par contre, au niveau global de l'application, l'enchaînement des tâches reste toujours séquentiel.

Nous proposons ici un schéma d'exécution distribué, où les tâches sont indépendantes les unes des autres formant un pipeline distribué. Ce pipeline n'a pas de tâche maître comme dans l'approche citée ci-dessus, et aucune contrainte sur l'enchaînement des tâches n'est imposée. Un tel schéma d'exécution présente les avantages suivants :

1. les tâches sont indépendantes les unes des autres, ce qui permet d'accroître la disponibilité des données. En effet, dans un schéma d'exécution séquentiel, parallèle ou distribué, tels qu'ils sont présentés dans [Sou01], à chaque instant, au plus un seul jeton (qui correspond au tableau), circule dans les arcs qui lient les tâches. Le schéma d'exécution en pipeline distribué autorise qu'il y ait plusieurs jetons dans les arcs.
2. Les contraintes de synchronisation entre tâches sont réduites, et gérées implicitement par le modèle des réseaux de processus. En effet, une tâche est rendue active dès que les données nécessaires à son traitement sont disponibles, contrairement aux schéma d'exécution existants, où les tâches s'enchaînent séquentiellement.
3. Le schéma autorise l'utilisation du parallélisme des données à l'intérieur des tâches, pour tirer mieux profit des architectures à mémoire partagée.
4. Le fait d'utiliser des FIFOs pour stocker les tableaux, et d'autoriser qu'il y ait plus d'un tableau dans une FIFO, permet l'utilisation de communications asynchrones (contrairement au schéma existant où les communications doivent obligatoirement être synchrones), augmentant ainsi le recouvrement des communications par les calculs.

5.3.3 D'une dépendance de données à un flux de tableaux

Comme décrit précédemment, le modèle ARRAY-OL permet d'établir l'enchaînement des tâches (niveau global) et les dépendances entre les éléments des tableaux consommés et ceux des tableaux produits. Pour passer d'une telle description à un modèle d'exécution de type « réseaux de processus de Kahn », nous définissons les règles suivantes :

1. « Tableau » $\mapsto$ « FIFO » : les tableaux multidimensionnels dans ARRAY-OL peuvent être facilement remplacés par des FIFOs. Le modèle ARRAY-OL impose qu'il n'y ait qu'une seule dimension infinie, la linéarisation des tableaux en FIFOs ne pose pas des problèmes particuliers. Les dimensions finies du tableau sont linéarisées en premier, et l'éventuelle dimension infinie est représentée par la taille de la FIFO, qui elle aussi est infinie. Les tableaux qui ne possèdent que des dimensions finies peuvent aussi être remplacés par des FIFOs mais l'intérêt d'une telle transformation est très limité, surtout s'il s'agit par exemple d'un tableau de coefficients qui peut être lus plusieurs fois par la tâche.
2. « TE » $\mapsto$ « Processus » : une tâche élémentaire est représentée par un processus.
3. « Tâche hiérarchique » $\mapsto$ « Réseaux de processus » : une tâche ARRAY-OL hiérarchique est un graphe de tâches et peut être représentée par un réseau de processus.

En considérant les deux niveaux de description d'une application ARRAY-OL, on peut dire que le modèle global est projeté vers un réseau de processus, alors que le modèle local est projeté vers un flux de tableaux.

Pour obtenir un flux de données, ou plus exactement un flux de tableaux, à partir d'une spécification ARRAY-OL, nous devons avant tout :

1. Définir la structure du flux : pour obtenir un flux de tableaux, une première solution consisterait à prendre un jeton égal à un motif. La figure 5.8 montre une application ARRAY-OL composée de trois tâches avec la structure des motifs consommés et produits par chaque tâche et sa transformation en réseaux de processus. Cette

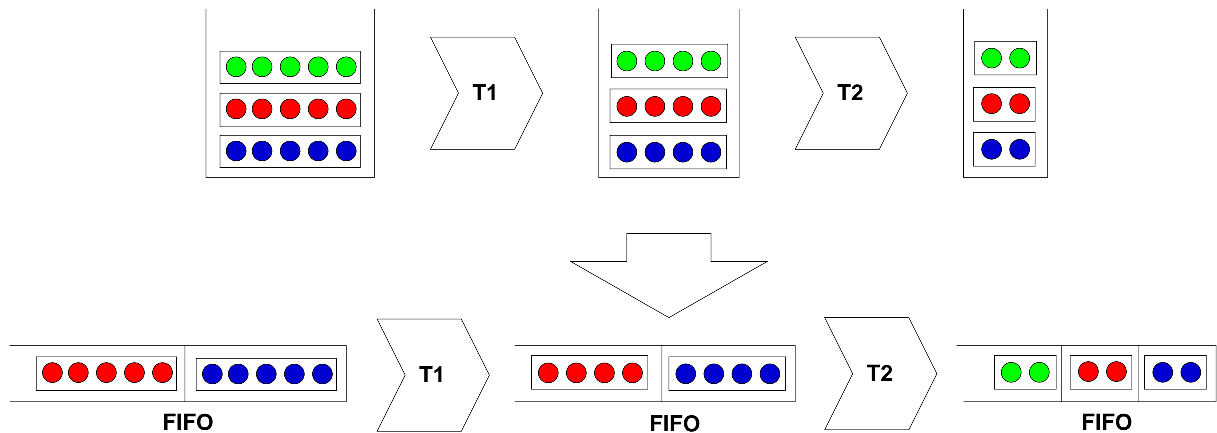


FIG. 5.8 – Application ARRAY-OL et flux de motifs

solution n'est possible que dans certaines configurations. En effet dans une configuration où le sens de la production des motifs ne correspond pas au sens de leur consommation dans la tâche suivante, la mise en place d'un flux de motifs n'est pas possible. Cette situation illustrée dans la figure 5.9 est nommée « corner turn ». Dans ce cas extrême, on est obligé de produire tout le tableau intermédiaire avant de consommer un motif quelconque de la deuxième tâche. Un autre problème est

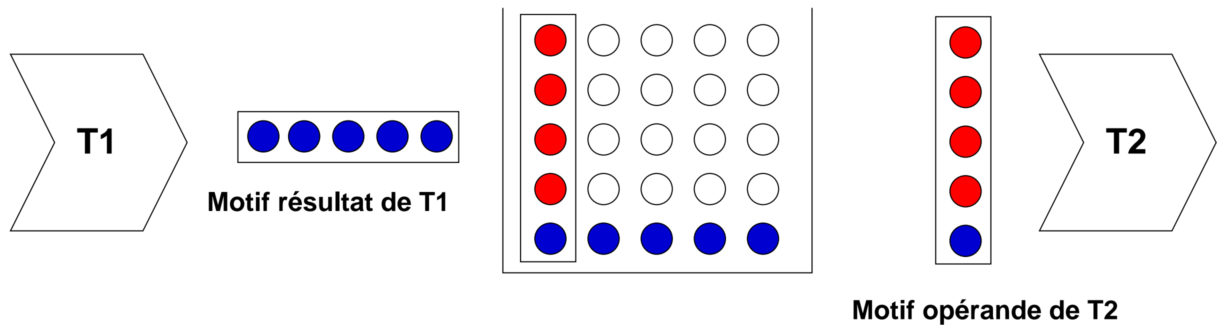


FIG. 5.9 – Phénomène du corner turn

lié à la forme des motifs consommés et produits par les différentes tâches. La forme du motif produit par une tâche n'est pas forcément le motif consommé par la tâche suivante, et l'utilisation d'un flux de motif n'est pas sans poser des problèmes de type. La figure 5.10 montre la liaison de deux demi-FIFOs, l'une manipulant des motifs (2×2), alors que l'autre stocke des motifs (2×1). Une autre solution possible serait d'utiliser une tâche intermédiaire qui effectue des conversions de type.

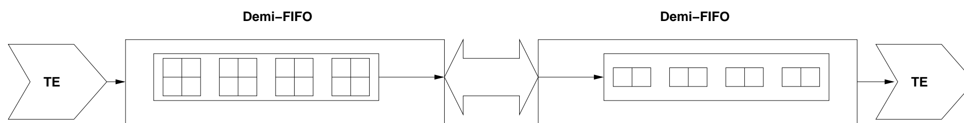


FIG. 5.10 – Structure des motifs produits et consommés

2. Gérer les lectures destructives : les lectures destructives dans les FIFOs dans les réseaux de processus de Kahn imposent au programmeur de définir les opérandes qui servent à calculer les résultats, et de libérer l'espace mémoire qui leur est alloué (implicitement par la lecture). Cette caractéristique du modèle des réseaux de processus de Kahn pose des problèmes dans certaines configurations, parmi lesquelles celle du « corner turn ». Une autre situation est celle où les motifs opérandes se recouvrent. La figure 5.5 montre un exemple d'une application où les motifs en entrée se recouvrent sur la dimension finie.

À partir de ces caractéristiques, on peut en déduire que :
- la transformation d'une application ARRAY-OL en un flux de motifs n'est possible que dans certaines configurations « simples » ;
- l'exécution d'une application ARRAY-OL avec les réseaux de processus de Kahn implique la libération de l'espace mémoire dès que possible, tout en restant dans un schéma d'exécution data-flow.

Les transformations de code ARRAY-OL définies et implémentées par J. SOULA [Sou01] permettent de répondre à ces deux contraintes. Ces transformations permettent d'obtenir des tâches hiérarchiques manipulant des macros-motifs. Ces macros-motifs représentent les opérandes des tâches, et permettent d'obtenir à chaque itération de pavage (dans un schéma d'exécution séquentiel) d'une tâche, un macro-motif qui sera le macro-motif en entrée de la tâche suivante.

L'utilisation de ces transformations dans notre schéma d'exécution permettent de résoudre les deux problèmes rencontrés lors de l'analyse du passage d'une spécification ARRAY-OL à une exécution à base de réseaux de processus. Ainsi comme le macro-motif résultat d'une tâche est le macro-motif opérande de la tâche suivante, la libération de l'espace mémoire du macro-motif est possible dès la fin du traitement sur celui-ci. La figure 5.11 montre l'exemple d'une application ARRAY-OL consommant des motifs qui se recouvrent, sa transformation en tâche hiérarchique et à partir de là sa transformation en flux de tableaux.

De plus, les problèmes liés au sens de production et de consommations des motifs opérandes sont résolus. On obtient alors un flux de tableaux, où les jetons sont des macros-motifs. De la même façon que précédemment, la figure 5.12 montre une tâche ARRAY-OL dans une configuration « corner turn », sa transformation en tâche hiérarchique et sa transformation en flux de tableaux.

L'enchaînement de tâches hiérarchiques dans notre schéma d'exécution se fait de façon beaucoup plus souple que dans le schéma d'exécution séquentiel, car les tâches sont indépendantes les unes des autres, et le fait de connaître le nombre d'itérations nécessaires sur la première tâche pour exécuter un certain nombre d'itérations de la seconde n'est plus aussi important. Seul le remplissage des tableaux résultats constitue le but principal.

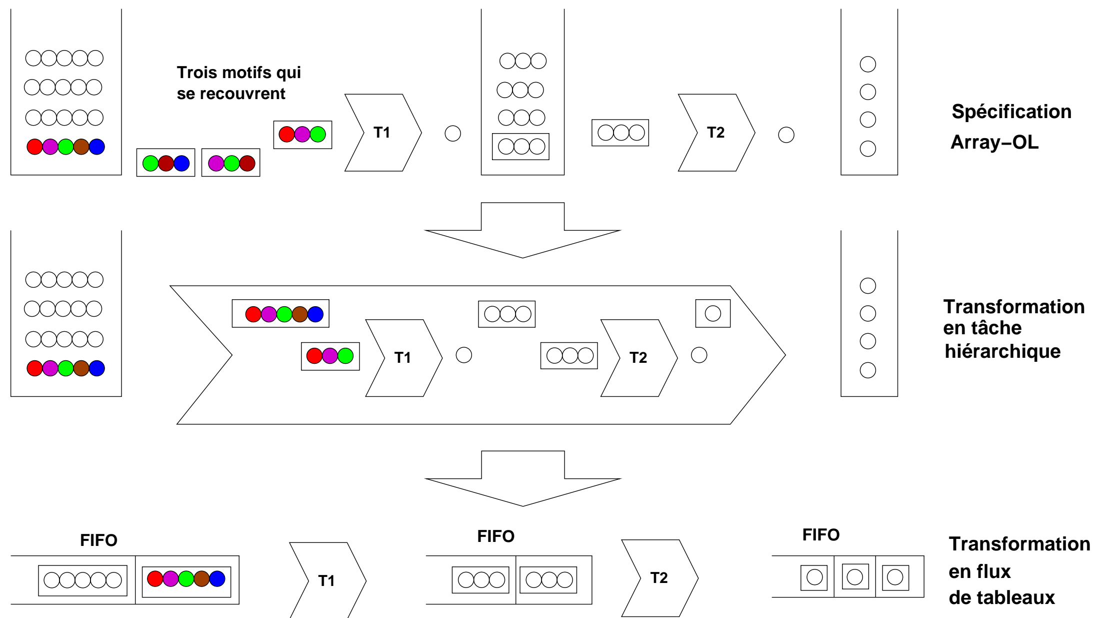


FIG. 5.11 – Tâche ARRAY-OL avec du recouvrement et sa transformation en tâche hiérarchique et en flux de tableaux

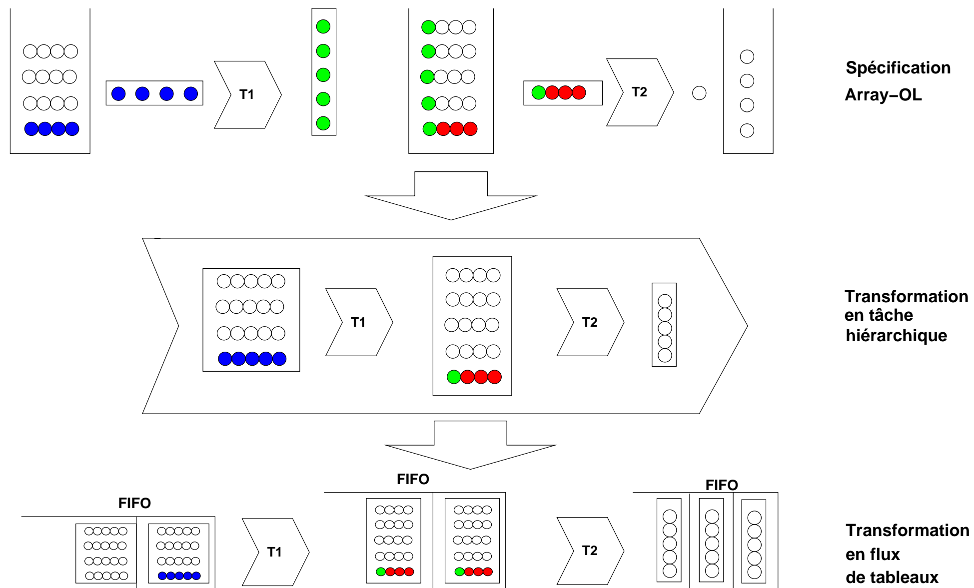


FIG. 5.12 – Tâche ARRAY-OL avec du « corner turn » et sa transformation en tâche hiérarchique et en flux de tableaux

Ainsi, on obtient par utilisation des transformations un flux de tableaux.

5.4 Évaluation

Pour évaluer notre modèle d'exécution pour ARRAY-OL, nous allons présenter une série de tests d'une application ARRAY-OL composée de deux tâches. En plus du parallélisme de tâches exprimé par le modèle global et exploité par la distribution, il y a le parallélisme de données dans la construction des tâches. Dans l'application testée, nous avons un pipeline sur la dimension infinie des tableau manipulés. Pour exploiter l'architecture multiprocesseurs de l'une des ressources de calcul, une des tâches (la FFT) est exécutée en mode SPMD.

L'application ARRAY-OL initiale est formée de deux tâches manipulant trois tableaux : un tableau $(512 \times \infty)$ en entrée de la première tâche qui produit un tableau $(512 \times 256 \times \infty)$ consommé par la seconde tâche. Les dimensions du tableau final produit par cette dernière sont $(128 \times \infty \times 200)$.

Le flux de tableau obtenu en utilisant les transformations ARRAY-OL possède les caractéristiques suivantes :

- la première tâche consomme des tableaux (512×512) et produit des tableaux (512×256) ;
- la seconde tâche consomme des tableaux (512×256) et un tableau de coefficients $(128 \times 200 \times 192)$ et produit un flux de tableaux (128×200).

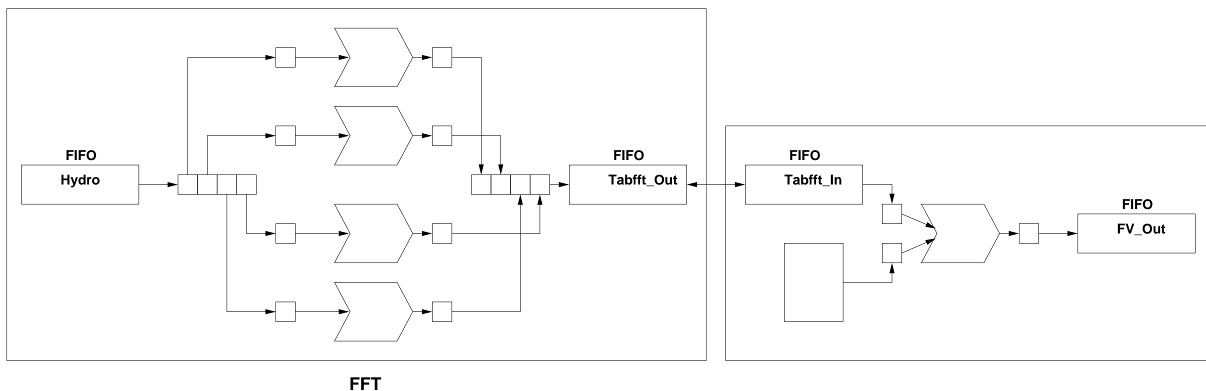


FIG. 5.13 – Description de l'application ARRAY-OL par un réseau de processus (dont un data-parallèle)

La figure 5.13 montre la structure de l'application testée (qui représente une partie de la VBL décrite dans le section 4.7). La première tâche est data-parallèle, alors que la seconde est séquentielle.

Les tests ont été effectués sur un quadri-processeurs *Xéons* à 450 Mhz, et sur une machine de type PC équipée d'un Duron à 1,3 Ghz. Les deux stations sont reliées par un réseau ethernet à 10Mb/s.

Les performaces obtenues par la parallélisation de la première tâche sont montrées dans la figure 5.14. Cette tâche a été exécutée sur le quadri-processeurs *Xéons*. Selon

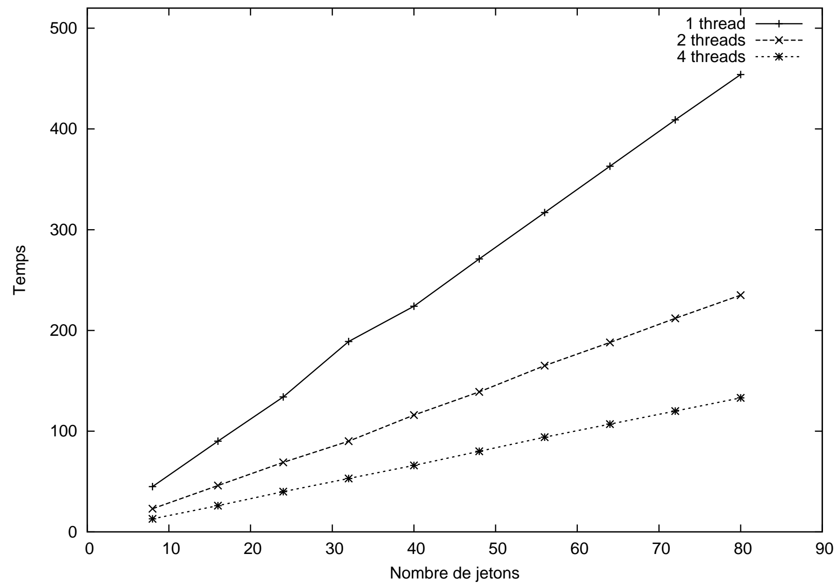


FIG. 5.14 – Performances de la parallélisation d'un processus de la VBL sur un quadri-xéons.

les configurations de test, l'espace d'itération de la tâches est partagé entre les différents threads. On obtient un rendement relativement bon puisqu'on arrive à une accélération supérieure à 3,4 pour 4 threads et 1,9 pour 2 threads.

Pour évaluer la version distribuée, nous avons exécuté la première tâche dans sa configuration avec 4 threads sur le quadri-processeurs *Xéons* et la seconde tâche sur le PC équipé d'un Duron à 1,3 Ghz. Le tableau 5.1 résume les mesures de performances obtenues. Les performances de la version distribuée montrent que les communications sont recouvertes par les calcul. Les temps d'exécution de l'application globale sont légèrement supérieurs aux temps d'exécution de la première tâche.

longueur du flux	Application distribué	1 ère Tâche (1 thread)	1 ère Tâche (2 threads)	1 ère Tâche (4 threads)	2ème Tâche
8	17	45	23	13	12
16	30	90	46	26	24
24	44	134	69	40	36
32	58	189	90	53	49
40	71	224	116	66	62
48	85	271	139	80	75
56	99	317	165	94	88
64	113	363	188	107	101
72	126	409	212	120	115
80	139	454	235	133	128

TAB. 5.1 – Performances

5.5 Conclusion

Le langage ARRAY-OL est bien adapté pour spécifier les algorithmes de traitement de signal. Nous avons présenté dans ce chapitre un modèle d'exécution pour ce langage basé sur les réseaux de processus distribués. À partir d'une spécification de données ARRAY-OL, une projection vers le modèle des réseaux de processus (distribués) a été définie. Les caractéristiques de ce schéma d'exécution sont les suivantes :

- Le parallélisme est exploité à deux niveaux : parallélisme de tâche et parallélisme de données ;
- L'exécution est pipelinée ;
- Le pipeline se fait sur des tableaux (macros-motifs), et non sur des motifs ;
- Les tableaux multidimensionnels sont projetés en FIFOs linéaires.

Pour exploiter efficacement ce support d'exécution, l'application nécessite dans certaines configurations d'être transformée. Ces mécanismes de transformation existent déjà, et permettent d'obtenir les macros-motifs optimaux. Ainsi l'association de ces outils de transformation avec le support d'exécution permet une exécution efficace, simple et distribuée d'applications ARRAY-OL sur les réseaux de processus de Kahn.

6

Conclusions et perspectives

Sommaire

6.1 Bilan 125
6.2 Perspectives 126

Films are never completed, they're only abandoned.

anonyme.

6.1 Bilan

Dans cette thèse, nous nous sommes intéressés à la conception d'un environnement d'exécution pour des applications réparties dynamiques. Nous avons défini et utilisé le modèle des réseaux de processus distribués de Kahn, comme modèle de base de notre environnement d'exécution. L'extension du modèle de Kahn pour supporter la distribution a permis de faire le lien entre les systèmes distribués et les applications des réseaux de processus de Kahn (simulation des systèmes embarqués, application de traitement de signal, traitement vidéo, ...) ouvrant ainsi la voie à la construction d'applications de simulation dans un environnement distribué.

Bien que le modèle des réseaux de Kahn soit le modèle de prédilection des applications de simulation, notre environnement n'est pas limité à ce type d'applications, et peut servir comme support d'exécution pour des applications où l'objectif derrière la distribution est l'amélioration des performances. La gestion des communications est l'un des points critiques dans de tels systèmes distribués, et notre approche ne néglige pas ce point. Elle permet d'optimiser les temps de transfert de données en utilisant la vectorisation, le recouvrement des calculs par les communications et l'équilibrage de charge. Ainsi, notre support reste aussi ouvert à des applications de haute performance dans un cadre de métacomputing.

Nos travaux couvrent essentiellement trois facettes :

1. La simulation distribuée : nous avons proposé et développé un support d'exécution capable d'assurer le fonctionnement d'une application de simulation de systèmes embarqués dans un environnement réparti. La facilité de développement se traduit dans notre approche par l'utilisation d'une méthodologie à base de composants, la transparence des communications et l'interactivité du déploiement.
2. La dynamicité des systèmes distribués : bien que l'environnement soit motivé par la simulation de composants distribués dans un contexte de cyber-entreprise, son domaine d'application ne se limite pas à la simulation distribuée. C'est ainsi qu'en plus des performances des communications et de la charge des processus, l'aspect dynamique de l'application distribuée a été pris en compte. Cette dynamicité que procure notre environnement et qui a été présentée dans le chapitre 4 est l'une des contributions principales de notre approche.
3. Le traitement de signal : le langage Array-OL est dédié aux applications de traitement de signal et plus particulièrement aux applications de traitement de signal multidimensionnel. La compilation de ce langage a été étudié par J. Soula, lors de sa thèse [Sou01], et une méthode par transformation automatique de code a été proposée. Nous avons proposé une projection du modèle Array-OL qui spécifie des dépendances de données vers le modèle des réseaux de processus qui est basé sur le flux de données. L'approche proposée combine la distribution de données et de tâches avec des exécutions de type *pipeline* et de type SPMD. De plus, l'application bénéficie de la dynamcité que procure notre support d'exécution.

Ces travaux ont donné lieu à des échanges technologiques avec les partenaires industriels du projet Sophocles [27]. Une démonstration dans le cadre de ce projet a été réalisée en collaboration avec Philips, et présentée au Workshop Sophocles lors du *Forum on specification & Design Languages (FDL '02)* en septembre 2002 à Marseille. Une seconde démonstration a été présentée lors de la revue ITEA Sophocles à Amsterdam en Octobre 2002.

6.2 Perspectives

Les perspectives ouvertes par nos travaux sont nombreuses et variées. Elles peuvent être regroupées dans deux catégories :

- la continuation des travaux : ce travail a fait l'objet de plusieurs développements (supports d'exécution, générateur de code, bibliothèque Array-OL). Toutefois, certains points ne sont pas complètement achevés, et nécessitent des améliorations.
- la reprise de ces travaux pour aborder d'autre thèmes de recherche : ces travaux se situent à l'intersection de plusieurs thématiques de recherche (systèmes distribués, parallélisme, traitement de signal, etc), et constituent un acquis qui peut être repris comme base à d'autres travaux.

Nous allons par la suite développer les perspectives.

[27]projet ITEA 99038

6.2.1 Continuation des travaux entrepris

Dynamicité et hiérarchie

Une première perspective des travaux autour de la dynamicité fonctionnelle est la généralisation de ce type de dynamicité aux réseaux de processus hiérarchiques. Une première solution consiste à ne considérer que les réseaux de processus formant un graphe acyclique, et de trouver un ordre de suspension des processus. Cet ordre doit garantir qu'un processus qui atteint son point de suspension n'est bloqué que si tous les autres processus qui lisent des données fournies par celui-ci ont déjà été suspendus. Pour la mise en œuvre d'un tel mécanisme, il faut pouvoir analyser le graphe et définir le sens de circulation des données.

Ordonnancement plus efficace des applications AOL

Cette perspective, relativement technique, concerne l'ordonnancement des applications Array-OL sur réseaux de processus de manière plus efficace. L'ordonnanceur actuel, purement dynamique, permet d'exploiter les architectures multiprocesseurs, mais ne tient pas compte du nombre de processeurs. Les applications Array-OL sont synchrones, et sont similaires aux réseaux de processus synchrones multidimensionnels [ML02]. Un meilleur ordonnancement consisterait à prendre en compte l'architecture du système (monoprocesseur ou multiprocesseurs et dans ce cas le nombre de processeurs), et les caractéristiques des processus du réseau (nombre de jetons produits à chaque itération, structure des tableaux, etc), pour définir un ordonnancement optimal du réseau.

Programmation visuelle et intégration dans Gaspard

Un autre prolongement de nos travaux est l'intégration de notre support dans l'environnement Gaspard [**?**, BDDM01](*Graphical Array Specificaton for Parallel and Distributed Computing* [28]). Cet environnement de programmation visuelle est dédié à la spécification et à l'exécution d'applications de traitement de signal en Array-OL. Outre la spécification de l'application, Gaspard est conçu pour spécifier l'architecture matérielle d'exécution. Notre support d'exécution peut être une des cibles de génération de code de Gaspard. La mise en place d'un tel environnement devrait aussi pouvoir s'appuyer sur un moteur de placement semi-automatique, voir totalement automatisé.

SystemC et distribution

Pour valider la simulation d'un SoC (« *System-On-Chip* »), il est nécessaire de passer à des niveaux de simulation plus bas niveau que la simulation fonctionnelle. Un des langages les plus utilisés est SystemC [Ope02, GLA02] pourrait constituer une cible d'implémentation. L'utilisation de SystemC est d'un grand intérêt car, en plus du fait qu'il dispose de primitives et de concepts permettant de valider un SoC du niveau fonctionnel jusqu'à un bas niveau d'implémentation, son aspect orienté objet procurera à l'environnement de simulation flexibilité et extensibilité. On disposera alors d'un environnement de simulation

[28] `http ://www.lifl.fr/west/gaspard`

distribué permettant de valider le fonctionnement d'une application de simulation de SoC à tous les niveaux d'abstraction correspondant aux différentes étapes de conception.

Placement, monitoring et équilibrage de charge

Le placement de l'application évoqué ci-dessus peut être combiné avec la dynamicité que procure notre support pour effectuer des opérations de migration semi-automatiques. Il faut alors disposer d'un outil de monitoring qui permettrait de collecter des informations sur l'exécution de l'application et l'état des ressources disponibles.

6.2.2 Nouveaux thèmes de recherche

Ordonnancement des réseaux de processus sur SMP et résolution anticipée des deadlocks

L'ordonnancement des réseaux de processus de Kahn dans le cadre des systèmes temps réel est confronté à la satisfaction des contraintes temps. Le modèle ne définit aucun mécanisme pour satisfaire ces contraintes. C'est dans le cadre de cette problématique que se déroule le travail de thèse de Javed Dulloo. Les premiers travaux [DM03] traitent du problème de l'ordonnancement des réseaux de processus de Kahn avec des contraintes de temps, ainsi que la résolution des deadlocks avant leur apparition. Notre support d'exécution qui permet d'exploiter ce type d'architecture sans surcoût (si l'exécution se fait sans distribution, tous les objets CORBA sont détruits) peut apporter une base pour le développement et l'implémentation de ce système.

Approche MDA et réseaux de processus distribués

L'approche que nous avons prise pour la génération de code est basée sur la description de l'application à partir du langage standard Xml. Cependant, elle reste assez simpliste, et nécessite plusieurs améliorations. L'objectif de ces travaux est de faciliter le développement d'applications distribuées, voir de l'automatiser. Une orientation intéressante est le développement d'une approche plus formelle basée sur une méthodologie MDA et selon les concepts du modèle « Y ». À partir d'une spécification haut niveau de l'application et de l'architecture de déploiement, l'utilisation du modèle des réseaux de processus peut être une cible de projection. Ceci passe par la spécification d'un méta-modèle d'application [29], un méta-modèle d'architecture, le méta-modèle des réseaux de processus et des règles de projection entre différents méta-modèles.

[29]Un méta-modèle d'application ISP UML est en cours de développement dans l'équipe *West*

Bibliographie

[ABD01] Abdelkader Amar, Pierre Boulet, and Jean-Luc Dekeyser. Assembling dynamic components for metacomputing using CORBA. In *Parallel Computing 2001*, Naples, Italy, September 2001. Lecture Notes in Computer Science.

[ABD03] Abdelkader Amar, Pierre Boulet, and Jean-Luc Dekeyser. Towards distributed process networks with CORBA. *Parallel and Distributed Computing Practice on Algorithms*, 2003. Special Issue on Parallel and Distributed Computing Practice on Algorithms, to appear.

[ABDT03a] Abdelkader Amar, Pierre Boulet, Jean-Luc Dekeyser, and Frans Theeuwen. Distributed process networks using half FIFO queues in CORBA. Research Report RR-4765, INRIA, March 2003.

[ABDT03b] Abdelkader Amar, Pierre Boulet, Jean-Luc Dekeyser, and Frans Theeuwen. Distributed process networks using half FIFO queues in CORBA. In *Parallel Computing 2003*, Dresden, Germany, September 2003.

[AE00] Greg Allen and Brian L. Evans. Real-time sonar beamforming on workstations using process networks and posix threads. *IEEE Transactions on Signal Processing*, 48(3) :921–926, March 2000.

[AG98] K. Arnold and J. Gosling. *The Java Programming Language (Second Edition)*. Addison-Wesley, 1998.

[AMR99] David Acremann, Gilles Moujeard, and Laurent Rousset. *Développer avec CORBA*. CampusPress, 1999.

[APP00] Dionisis X. Adamopoulos, George Pavlou, and Constantine A. Papandreou. Performance evaluation of distributed object platforms for telecommunications service engineering activities. In *Signal Processing, Communications and Computer Science. Proceedings of the 4th IMACS/IEEE World Multiconference on Circuits, Systems, Communications and Computers (CSCC '00)*, pages 131–136, Athens, Greece, July 2000. Mastorakis, ed. World Scientific and Engineering Society Press.

[AST02] Marten ven Steen Andrew S. Tanenbaum. *Distributed Systems : Principles and Paradigms.* Prentice Hall, 2002.

[BCD+00] Randall Bramley, Kenneth Chiu, Shridhar Diwan, Dennis Gannon, Madhusudhan Govindaraju, Nirmal Mukhi, Benjamin Temko, and Madhuri Yechuri. A component based services architecture for building distributed applications. In *HPDC*, pages 51–, 2000.

[BDD+03] Pierre Boulet, Jean-Luc Dekeyser, Cédric Dumoulin, Philippe Kajfasz, Philippe Marquet, and Dominique Ragot. Sophocles : Cyber-enterprise for system-on-chip distributed simulation – model unification. In *À paraître dans IP Based Design 03*, 2003.

[BDDM01] Pierre Boulet, Jean-Luc Dekeyser, Florent Devin, and Philippe Marquet. A visual development environment for meta-computing applications. In *HCI International 2001, 9th Int'l Conf. on Human-Computer Interaction*, volume 1, pages 983–987, New Orleans, LA, August 2001. Lawrence Erlbaum Associates, Publishers.

[BDL+01] Pierre Boulet, Jean-Luc Dekeyser, Jean-Luc Levaire, Philippe Marquet, Julien Soula, and Alain Demeure. Visual data-parallel programming for signal processing applications. In *9th Euromicro Workshop on Parallel and Distributed Processing, PDP 2001*, pages 105–112, Mantova, Italy, February 2001.

[BG97] Don Batory and Bart J. Geraci. Composition Validation and Subjectivity in GenVoca Generators. *IEEE Transactions on Software Engineering (special issue on Software Reuse)*, pages 62–87, 1997.

[BL92] Joseph T. Buck and Edward A. Lee. The tocken flow model. In *Data Flow Workshop*, May 1992.

[BL93] Joseph T. Buck and Edward A. Lee. Scheduling dynamic dataflow graphs with bounded memory using the tocken flow model. In *Proceedings of ICASSP'93*, Minneapolis, MN, USA, April 1993.

[Boa01] OMG Architecture Board. Model driven architecture (MDA). Technical Report ormsc/2001-07-01, OMG, 2001.

[Buc93] Joseph T. Buck. *Scheduling Dynamic Dataflow Graphs with Bounded Memory Using the Token Flow Model.* PhD thesis, University of California at Berkeley, 1993.

[BvST99] Arno Bakker, Maarten van Steen, and Andrew S. Tanenbaum. From remote objects to physically distributed objects. In *Proc. 7th IEEE Workshop on Future Trends of Distributed Computing Systems*, pages 47–52, Cape Town, South Africa, December 1999.

[CDF+02] Franck Cappello, Abderrahmane Djilali, Gilles Fedak, Cécile Germain, Oleg Lodygensky, and Vincent Néri. *Calcul réparti à grande échelle Metacomputing*, chapter XtremWeb : une plate-forme de recherche sur le Calcul Global et Pair à Pair. Lavoisier, 2002.

[CDK01] George Coulouris, Jean Dollimore, and Tim Kindberg. *Distributed Systems : Concepts and Design.* Addison-Wesly, 2001.

[Dal96] Rogerson Dale. *Inside COM, Microsoft Component Object Model.* Microsoft Press, 1996.

[Dan02] Jérôme Daniel. *Au cœur de CORBA.* Vuibert, 2002.

[DBDM02] Florent Devin, Pierre Boulet, Jean-Luc Dekeyser, and Philippe Marquet. Gaspard : a visual data-parallel programming environment for signal processing applications. In *International Conference on Parallel Computing in Electrical Engineering, PARELEC'2002*, Warsaw, Poland, September 2002.

[Den91] J.B ; Dennis. *The evolution of "static" data-flow architecture*, chapter 2, pages 35–91. Prentice Hall, Englewood Cliffs, 1991.

[DFG+98] Jack Dongarra, Graham E. Fagg, Al Geist, James Arthur Kohl, Philip M. Papadopoulos, Stephen L. Scott, Vaidy S. Sunderam, and M. Magliardi. HARNESS : Heterogeneous adaptable reconfigurable networked systems. In *HPDC*, pages 358–359, 1998.

[dKES+00] E. A. de Kock, G. Essink, W. J. M. Smits, P. van der Wolf, J.-Y. Brunel, W. M. Kruijtzer, P. Lieverse, and K. A. Vissers. YAPI : Application modeling for signal processing systems. In *37th Design Automation Conference*, Los Angeles, CA, June 2000. ACM Press.

[DLB+95] Alain Demeure, Anne Lafage, Emmanuel Boutillon, Didier Rozzonelli, Jean-Claude Dufourd, and Jean-Louis Marro. Array-OL : Proposition d'un formalisme tableau pour le traitement de signal multi-dimensionnel. In *Gretsi*, Juan-Les-Pins, France, September 1995.

[dM95] Jan de Meer. The iso reference model for open distributed processing. *Computer Networks and ISDN Systems*, 27(8) :1211–1214, 1995.

[DM03] Javed Dulloo and Philippe Marquet. Design of a real-time scheduler of Kahn process networks on multiprocessors. To appear, 2003.

[DPP03] Alexandre Denis, Christian Pérez, and Thierry Priol. PadicoTM : An open integration framework for communication middleware and runtimes. *Future Generation Computer Systems*, 19 :575–585, 2003.

[EE98] Guy Eddon and Henry Eddon. *Inside Distributed COM*. Microsoft Press, 1998.

[FGT96] Ian T. Foster, Jonathan Geisler, and Steven Tuecke. MPI on the i-way : A wide-area, multimethod implementation of the message passing interface. In *Proc. 1996 MPI Developers Conference*, pages 10–17, 1996.

[FK97] I. Foster and C. Kesselman. Globus : A metacomputing infrastructure toolkit. *Intl J. Supercomputer Applications*, 11(2) :115–128, 1997.

[FK98] I. Foster and C. Kesselman. The globus project : A status report. In *IPPS/SPDP '98 Heterogeneous Computing Workshop*, pages 4–18, Dallas, TX, USA, 1998.

[FKT96] Ian T. Foster, Carl Kesselman, and Steven Tuecke. The nexus approach to integrating multithreading and communication. *Journal of Parallel and Distributed Computing*, 37(1) :70–82, 1996.

[FKT01] Ian Foster, Carl Kesselman, and Steven Tuecke. The anatomy of the grid : Enabling scalable virtual organizations. *International Journal Supercomputer Applications*, 15(3), 2001.

[FLdM95] Kazi Farooqui, Luigi Logrippo, and Jan de Meer. The ISO reference model for open distributed processing : An introduction. *Computer Networks and ISDN Systems*, 27(8) :1215–1229, 1995.

[GB03] Marc Geilen and Twan Basten. Requirements on the execution of Kahn process networks. In *ESOP 2003*, volume 2618 of *Lecture Notes in Computer Science*, pages 319–334. Springer, 2003.

[GGM99] Jean-Marc Geib, Christophe Gransart, and Philippe Merle. *Corba : des concepts à la pratique*. Dunod Informatique, September 1999.

[GKC+02] Madhusudhan Govindaraju, Sriram Krishnan, Kenneth Chiu, Aleksander Slominski, Dennis Gannon, and Randall Bramley. Xcat 2.0 : A component-based programming model for grid web services. Technical report-tr562, Department of Computer Science, Indiana University, June 2002.

[GKC+03] Madhusudhan Govindaraju, Sriram Krishnan, Kenneth Chiu, Aleksander Slominski, Dennis Gannon, and Randall Bramley. Merging the CCA component model with the OGSI framework. In *Proceedings of CCGrid2003, 3rd International Symposium on Cluster Computing and the Grid*, May 2003.

[GLA02] T. GROTKER, S. LIAO, and AL. *System Design with SystemC*. Kluwer Academic Publishers, 2002.

[GNTW] Andrew S. Grimshaw, Anh Nguyen-Tuong, and William A. Wulf. Campus-wide computing : Early result using Legion at the university of virginia.

[GWF+94] Andrew S. Grimshaw, William A. Wulf, James C. French, Alfred C. Weaver, and Paul F. Reynolds Jr. Legion : The next logical step toward nationwide virtual computer. Technical Report No. CS-94-21, University of Virginia, June 1994.

[HB01] J. Hoogerbrugge and Twan Basten. Efficient execution of Kahn process networks. In A. Chalmers, M. Mirmehdi, and H. Muller, editors, editor, *Proc. of Communicating Process Architectures 2001*, pages 1–14. IOS Press, September 2001.

[HvST96] Philip Homburg, Maarten van Steen, and Andrew S. Tanenbaum. Communication in GLOBE : An object-based worldwide operating system. In *Proc. Fifth International Workshop on Object Orientation in Operating Systems*, pages 43–47, Seattle, Washington, USA, October 1996.

[HvST99] Philip Homburg, Maarten van Steen, and Andrew S. Tanenbaum. Globe : A wide-area distributed system. *IEEE Concurrency*, pages 70–78, Jan-Mar 1999.

[JR98] Matjaz B. Juric and Ivan Rozman. Choosing component middleware based on performance evaluation. In *ESS'98*, 1998.

[JRH00] Matjaz B. Juric, Ivan Rozman, and Marjan Hericko. Performance comparision of CORBA and RMI. *Information and Software Technology Journal*, 42(13) :915–933, October 2000.

[JZR98] Matjaz B. Juric, Ales Zivkovic, and Ivan Rozman. Comparison of CORBA and Java RMI based on performance analysis. In *MHSS'98*, 1998.

[Kah74] Gilles Kahn. The semantics of a simple language for parallel programming. In Jack L. Rosenfeld, editor, *Information Processing 74 : Proceedings of the IFIP*

Congress 74, pages 471–475. IFIP, North-Holland Publishing Co., August 1974.

[KBG+01] Sriram Krishnan, Randall Bramley, Dennis Gannon, Madhusudhan Govindaraju, Jay Alameda, Richard Alkire, Timothy Drews, and Eric Webb. The XCAT science portal. In *Proceedings SC2001*, Denver, USA, November 2001.

[Kea99] Katarzyna Keahey. PARDIS : Programmer-level abstractions for metacomputing. *Future Generation Computer Systems*, 15(5–6) :637–647, October 1999.

[KG97] Katarzyna Keahey and Dennis Gannon. PARDIS : CORBA-based architecture for application-level parallel distributed computation. In *Proceedings of Supercomputing '97*, November 1997.

[KM66] R.M. Karp and R.E. Miller. Properties of a model for parallel comuputation : Determinacy, termination and queueing. *SIAM J. Applied Math*, 14(6) :1390–1411, November 1966.

[KM77] Gilles Kahn and David B. MacQueen. Coroutines and networks of parallel processes. In B. Gilchrist, editor, *Information Processing 77*, pages 993–998. North-Holland, 1977. Proc.IFIP Congress.

[Lee03] Edward A. Lee. Overview of the Ptolemy project. Technical Memorandum No. UCB/ERL M03/25, University of California, Berkeley, CA, 94720, USA, July 2003.

[LM87] Edward A. Lee and D.G. Messerschmitt. Synchronous data flow. *IEEE*, 75(9) :1235–1245, September 1987.

[LP95] Edward A. Lee and Thomas M. Parks. Dataflow process networks. *IEEE*, 83(5) :773–801, May 1995.

[Mar85] A.J. Martin. The probe : An addition to communication primitives. In *In formation Processing Letters*, volume 20, pages 125–130, 1985.

[Mar90] A.J. Martin. *Programming in VLSI : From Communicating Processes to Delay-Insensitive Circuits*. Developments in Concurrency and Communication, C.A.R. Hoare (ed.) Addison-Wesley, 1990.

[MDGS98] Mauro Migliardi, Jack Dongarra, Al Geist, and Vaidy S. Sunderam. Dynamic reconfiguration and virtual machine management in the Harness metacomputing system. In *ISCOPE*, pages 127–134, 1998.

[MH98] V. Matena and M. Hapner. Enterprise JavaBeans, version 1.0, March 1998.

[ML02] Praveen K. Murthy and Edward A. Lee. Multidimensional synchronous dataflow. *IEEE Transactions on Signal Processing*, July 2002. (To appear).

[Obj00] Object Oriented Concepts Documentation. *ORBacus For C++ and Java*, 2000. World Wide Web document, URL : `http://www.ooc.com/ob/`.

[Obj01] Object Management Group, Inc., editor. *Common Object Request Broker Architecture (CORBA), Version 2.6*. `http://www.omg.org/technology/documents/formal/corba_iiop.htm`, December 2001.

[Ope02] Open SystemC Initiative. SystemC. `http://www.systemc.org/`, 2002.

[PA85] Keshav Pingali and Arvind. Efficient demand-driven evaluation, part 1. *ACM Transactions on Programming Languages and Systems*, 7(2) :311–333, April 1985.

[PA86] Keshav Pingali and Arvind. Efficient demand-driven evaluation, part 2. *ACM Transactions on Programming Languages and Systems*, 8(1) :109–139, January 1986.

[Par95] Thomas M. Parks. *Bounded Scheduling of Process Networks*. PhD thesis, University of California at Berkeley, 1995.

[PR98] Thierry Priol and Christophe René. Cobra : A CORBA-compliant programming environment for high-performance computing. In *Euro-Par'98*, number 1470 in Lecture Notes in Computer Science, pages 1114–1122, Southampton, UK, November 1998.

[PR03] Thomas M. Parks and David Roberts. Distributed process networks in Java. In *International Workshop on Java for Parallel and Distributed Computing*, April 2003.

[Raj00] R. Raje. UMM : Unified meta-object model for open distributed systems. In *Proceedings of 4th IEEE International Conference on Algorithms and Architecture for Parallel Processing ICA3PP'2000*, pages 454–465, 2000.

[Red97] Frank E. III Redmond. *DCOM : Microsoft Distributed Component Object Model*. Hungry Minds, Inc, 1997.

[RLS+00] O. F. Rana, M. Li, M. S. Shields, , and D. W. Walker. A wrapper generator for wrapping high performance legacy codes as Java/CORBA components. In *IEEE/ACM SC2000 Conference*, Dallas, TX, USA, November 2000. Only available on CD-ROM.

[Rom99] Ed Roman. *Mastering Enterprise JavaBeans*. Wiley Computer Publishing, 1999.

[SMDD01] Julien Soula, Philippe Marquet, Jean-Luc Dekeyser, and Alain Demeure. Compilation principle of a specification language dedicated to signal processing. In *Sixth International Conference on Parallel Computing Technologies, PaCT 2001*, pages 358–370, Novosibirsk, Russia, September 2001. Lecture Notes in Computer Science vol. 2127.

[Sou01] Julien Soula. *Principe de Compilation d'un Langage de Traitement de Signal*. Thèse de doctorat (PhD Thesis), Laboratoire d'informatique fondamentale de Lille, Université des sciences et technologies de Lille, December 2001. (In French).

[Str01] Bjarne Stroustrup. *Le Langage C++*. Pearson Education, 2001.

[Vin97] Steve Vinoski. Corba : Integrating diverse applications within distributed heterogeneous environments. *IEEE Communications Magazine*, 35(2), February 1997.

[WRW96] A. Woolrath, R. Riggs, and J. Waldo. A distributed object model for the Java system. *Computing Systems*, 9(4) :291–312, 1996.

[WWM99] Darren Webb, Andrew Wendelborn, and Kevin Maciunas. Process networks as a high-level notation for metacomputing. In *Workshop on Java for Parallel and Distributed Computing (IPPS)*, Puerto Rico, April 1999.

[Xml] Extensible markup language (XML). `http://www.w3c.org/xml`.

Environnement fonctionnel distribué et dynamique pour systèmes embarqués

RÉSUMÉ. Dans cette thèse, nous nous sommes intéressés à la conception d'un environnement d'exécution pour des applications réparties dynamiques. Nous avons défini et utilisé le modèle des réseaux de processus distribués de Kahn, comme modèle de base de notre environnement d'exécution. L'extension du modèle de Kahn pour supporter la distribution a permis de faire le lien entre les systèmes distribués et les applications des réseaux de processus de Kahn (simulation des systèmes embarqués, application de traitement de signal, traitement vidéo, ...) ouvrant ainsi la voie à la construction d'applications de simulation dans un environnement distribué.

Nos travaux couvrent essentiellement trois facettes :

1. La simulation distribuée : notre approche est basée sur l'utilisation d'une méthodologie à base de composants, la transparence des communications et l'interactivité du déploiement.
2. La dynamicité des systèmes distribués : l'intégration de plusieurs aspect de dynamicité dans le support pour faire évoluer l'application et s'adapter aux changements de l'environnement d'exécution.
3. Le traitement de signal multidimensionnel : l'adéquation du support d'exécution distribué pour des applications ARRAY-OL (langage dédié au traitement de signal), par construction d'un modèle d'exécution particulier et par sa mise en œuvre.

Mots-clés: calcul distribué, réseaux de processus de Kahn, CORBA, systèmes répartis dynamiques, traitement de signal multidimensionnel

Dynamic distributed functional simulation environment for embedded systems

ABSTRACT. In this thesis, we are interesting in the design of an environment of execution for dynamic distributed applications. We have defined and used the Distributed Kahn Process Networks model Kahn, as a basic execution model. The distribution of the model make it possible to establish the link between the distributed systems and Kahn process networks applications (embedded systems simulation, signal processing application, video and audio processing ...) thus opening the way to the construction of simulation applications in an distributed environment.

Our work covers primarily three topics :

1. Distributed simulation : our approach is component-based, with interactive deployment and communication transparency.
2. Dynamic distributed systems : several dynamicity aspects have been added to ou environement for load balancing and application evolution.
3. The multidimensional signal processing : we propose and implement a dataflow execution model for ARRAY-OL applications (signal processing language).

Keywords: distributed systems, Kahn process networks, CORBA, dynamic distributed systems, multidimensional signal processing

---

Abdelkader AMAR
LIFL (Laboratoire d'Informatique de Lille) Bâtiment M3 59655 Villeneuve d'Ascq Cédex - FRANCE
Décembre 2003