Chapitre 6 - Mise en oeuvre

«L'image d'un système unique crée, dans l'esprit de l'utilisateur l'illusion que tous les serveurs du réseau font partie d'un même système ou qu'ils se comportent comme un ordinateur unique» (Andrew Tannenbaum - Modern Operating System)

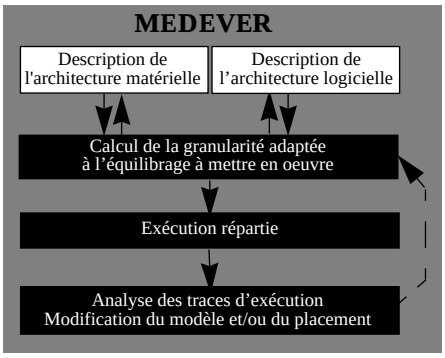
Résumé

L'environnement IDEFIX (*Integrated Development Environment with Flexible dIstributed eXecution*) a été créé dans le but de placer dynamiquement une application répartie ou parallèle sur un environnement matériel hybride. Ce chapitre est consacré à la description de l'architecture générale et des choix d'implémentation d'IDEFIX. Dans un premier temps, nous présentons la mise en oeuvre de HADEL, notre langage de description d'architecture matérielle hybride. Puis, nous décrivons, Gatosta, la plate-forme d'équilibrage de charge utilisée. Enfin, nous présentons l'architecture de notre prototype de plate-forme et nos réalisations lors de sa mise en oeuvre.

Apports scientifiques

Nous avons mis en oeuvre IDEFIX en nous appuyant sur la plate-forme de répartition de charge Gatostar développée dans notre équipe. Dans IDEFIX, les architectures logicielles décrites en VODEL-D sont stockées dans un segment de mémoire partagée et évoluent lors de l'exécution de l'application. Nous avons fait ce choix afin de pouvoir gérer des sessions multi-utilisateurs et de rendre transparent l'accès à l'architecture logicelle d'exécution. Une fois l'exécution terminée, il est alors possible de comprarer l'architecture logicielle d'exécution finale par rapport à l'architecture logicielle d'exécution. Les premiers résultats sont encourageants, même si le coût d'accès au segment de mémoire partagée n'est pas négligeable.

Plan



Mise en oeuvre d'IDEFIX

Présentation de la mise en oeuvre d'HADEL

Présentation de Gatostar : une plate-forme de répartition de charge

Présentation de la mise en oeuvre d'IDEFIX par l'intégration de VODEL-D, HADEL et Gatostar

Mots clefs

Environnement de développement, architecture matérielle, architecture logicielle, Gatostar, équilibrage de charge, mémoire partagée répartie.

L'environnement IDEFIX (*Integrated Development Environment with Flexible dIstributed eXecution*) a été créé dans le but de placer dynamiquement une application répartie ou parallèle sur un environnement matériel hybride. IDEFIX est basée sur la méthode MEDEVER et sur nos langages de description d'architecture HADEL et VODEL-D (cf. Figure 69). Au niveau de l'implémentation, nous avons utilisé la plate-forme d'équilibrage de charge, Gatostar [Folliot & al. 95b], développée dans notre équipe.

Ce chapitre est consacré à la description de l'architecture générale et des choix d'implémentation d'IDEFIX. Dans un premier temps, nous présentons la mise en oeuvre de HADEL, notre langage de description d'architecture matérielle hybride. Puis, nous décrivons, Gatostar, la plate-forme d'équilibrage de charge utilisée. Enfin, nous présentons l'architecture de notre prototype de plate-forme et nos réalisations lors de sa mise en oeuvre.

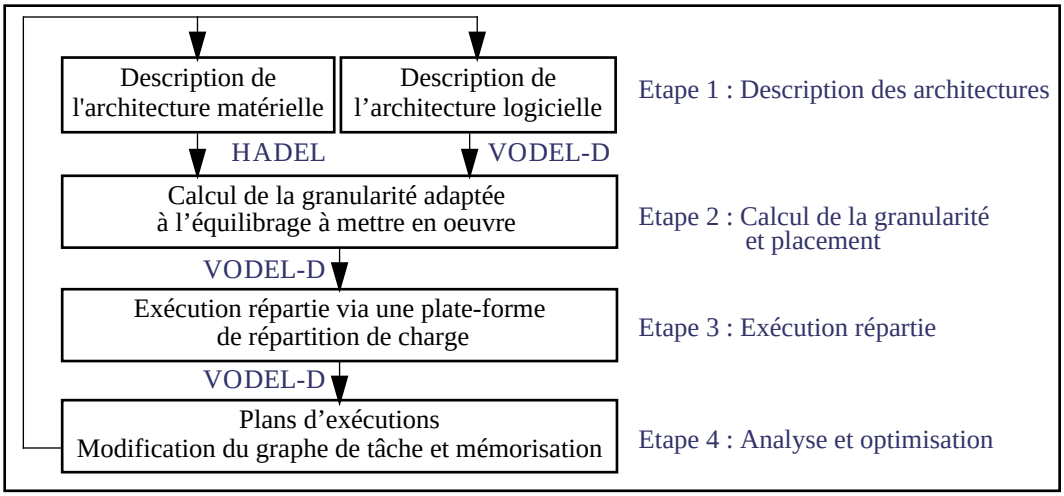


Figure 69 : La méthode MEDEVER

1. Description et gestion de l'architecture matérielle

HADEL est un langage de description d'architecture matérielle Hybride, gérant deux niveaux de description (Macro et Micro). Lors de la conception d'HADEL, nous avons tenté d'en faire un langage ouvert, susceptible d'être utilisé par d'autres applications ou d'autres environnements. Nous avons alors mis au point une architecture ouverte d'accès à HADEL et aux modèles qu'il permet de créer.

Cette architecture, présentée en Figure 70, sépare les environnements gérant nativement HADEL (comme IDEFIX), de ceux qui ne le gèrent pas, mais utilisent des descriptions d'architecture matérielles propriétaires. Dans ce dernier cas, deux solutions sont possibles pour interfacer une application et HADEL. La première consiste à simplement générer du code compatible avec le logiciel utilisé, en utilisant un moteur de réécriture. La seconde, bien plus puissante, donne un accès complet par programmation, via une bibliothèque d'accès aux fonction de gestion d'architectures matérielles.

Lors de l'implémentation de notre langage Hadel et des outils associés, nous avons particulièrement soigné l'intégration d'HADEL avec le logiciel de répartition de charge Gatostar [Folliot & al. 95a]), l'environnement de génération de code CPN/Tagada [Kordon & al. 94] et H-Tagada [Kordon & al. 95] et le langage de simulation QNAP2 [Simulog 88]).

C'est pourquoi, l'API qui a été développée (cf. Figure 71) est centrée autour d'une structure interne qui fournit trois interfaces vers :

1) l'éditeur de graphe MACAO et son langage CAMI d'importation et d'exportation de modèles ;
2) l'interface de requêtes utilisable depuis des programmes extérieurs ;
3) la base de données gérant le stockage des données.

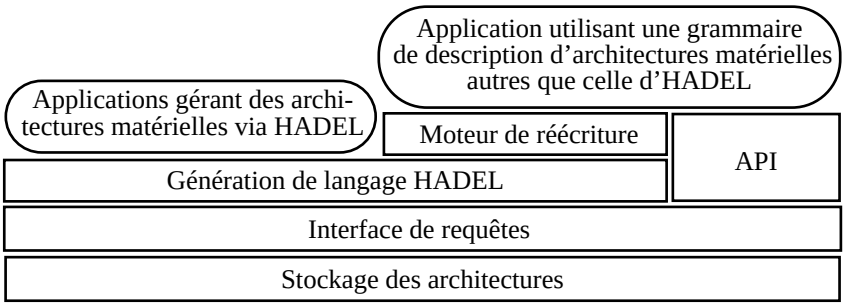


Figure 70 : Architecture de description persistante d'architectures matérielles avec HADEL

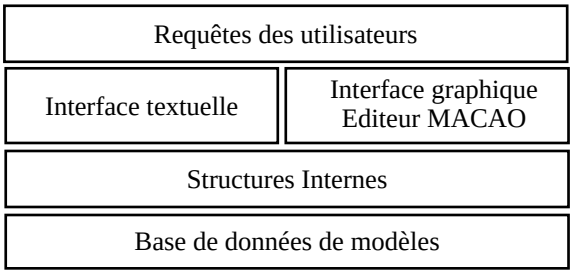


Figure 71 : L'environnement de gestion d'architectures matérielles basé sur le langage HADEL

La structure interne gère trois types d'objets :

1) *le noeud* est une structure de données intégrant les attributs concernant une H-Machine au niveau gros grain (les objets monoprocesseur, multiprocesseur) et grain fin (CPU, mémoire et port).
2) *le lien* est une structure de données intégrant les attributs d'un H-Lien.
3) *l'architecture* est composée de listes de H-Machines et de H-liens et d'informations précisant la validité de l'architecture et le sens de parcours entre les différentes granularités.

Nous étudions donc dans la suite de ce chapitre, la description d'une architecture matérielle avec Macao et son langage d'import/export CAMI. Puis, nous décrivons la mise en oeuvre du stockage persistant des descriptions créées. Enfin, nous présentons l'interface de requêtes, necessaire durant l'étape de vérification et de correspondance.

1.1. La description à l'aide de Macao

Macao est un éditeur de graphes [Macao 97], qui a été utilisé pour représenter les objets et les attributs définis dans le langage HADEL via un formalisme graphique. L'exportation du modèle graphiquement décrit vers un fichier textuel, contenant la même description mais dans le langage HADEL, se fait par l'utilisation d'un langage intermédiaire (cf Figure 72).

Ainsi, lors du chargement d'une description dans le logiciel Macao, nous appliquons la stratégie suivante :

- **Etape 1** : chargement de l'architecture au niveau «gros grain»,
- **Etapes 2 à N :** chargement sur demande des descriptions «grain fin» correspondant aux noeuds «multi-processeur» de la description «gros grain».

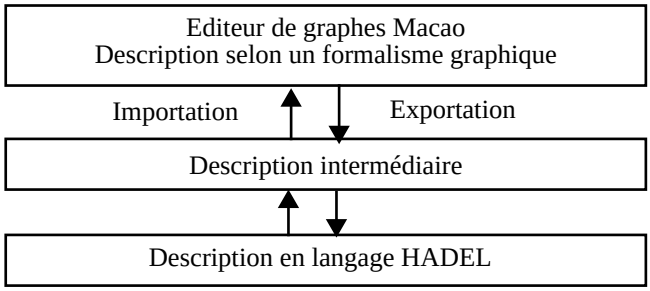


Figure 72 : Importation et exportation de description d'architectures matérielles avec HADEL

1.2. Gestion de bibliothèques d'architectures matérielles

Des informations sémantiques sont attachées à certains attributs des objets décrits dans le langage HADEL. Ces attributs sont alors utilisés de manière statique lors de l'étape de vérification sémantique du modèle ou de manière dynamique lors de l'étape de correspondance. Toutes les informations sémantiques sont consignées dans une base de données, qui est modifiable par l'utilisateur.

Les fonctionnalités offertes par l'API de gestion de bibliothèques d'architectures matérielles sont :

- *l'importation* de la description textuelle transmise par le logiciel de dessin Macao est réécriture sous une forme textuelle (langage HADEL) ;
- *les vérifications* de la description de l'architecture pour détecter les incohérences structurelles ou syntaxiques ;
- *le stockage* des entités de l'architecture décrite par l'utilisateur (CPU, machine-mono-processeur, etc.) et des bibliothèques d'architectures ;

1.2.1. Base de données de H-Machine de type monoprocesseur

Pour une machine monoprocesseur, on stocke les informations suivantes :

- la famille d'architecture à laquelle cette machine appartient ;
- les H-Liens qui peuvent-être connectés et leur nombre maximum et minimum ;
- le maximum et le minimum de taille mémoire disponible ;
- la vitesse maximale (quand la machine est peu chargée) et minimale (quand la machine est trop chargée) du monoprocesseur ;
- L'existence d'une unité de calcul des nombres flottants.

Un exemple des types de champs possibles et de leur valeur est donné dans le Tableau 42

Tableau 42: Exemples de valeurs d'attributs pour une machine monoprocesseur.

Famille	**Type**	**Liens**	**Mémoire**	**Vitesse (**MIPS**)**	**FPU**
Sparc	sparc5	ethernet [1-2]	16-64	60-90	oui
Sparc	sparc10	ethernet [1-2]	16-128	95-110	oui
Motorola	sun3	ethernet 1	4-12	5-15	optionnel
Intel	pentium	ethernet [1-3]	1-128	1-2	Intégré

1.2.2. Base de données de H-Liens

Pour un H-Lien donné, les informations suivantes sont stockées :

- le débit maximum et minimum théorique ou constaté ;
- le type de connexion ;
- le nombre maximum de machines supportées.

Le Tableau 43 donne des exemples des valeurs possibles.

Tableau 43: Exemples de valeurs d'attributs pour un H-Lien

Type	**Débit min-max**	**structure spéciales**	**max machines**
série	1-64 Kbp / s	point à point	2
ethernet	2-10 Mbp / s	bus	254
fast-ethernet	10-100 Mbp / s	bus	254
token-ring	1-4 Mbp / s	anneau	254

1.2.3. Base de données de processeur (CPU)

Les informations concernant un objet de type CPU sont :

- la famille architectural à laquelle il appartient ;
- la taille minimum et maximum du cache ;
- la vitesse minimum et maximum du processeur ;
- le nombre maximum de liens simultanés qu'il peut gérer (4 pour un Transputer par exemple) ;
- la taille maximum de mémoire RAM à laquelle il peut accéder.

Le Tableau 44 donne des exemples des valeurs possibles.

Tableau 44: Exemples de valeurs d'attributs pour un processeur

Famille	**Type**	**Taille du Cache (**Ko**)**	**Nb d'entrées sortie et vitesse (**Mbp / s**)**	**Taille mémoire max (**Mo**)**	**Vitesse (**MIPS**)**
Intel	Pentium 120	0-64	1-12	64	50-100
Motorola	68000	0-4	1-3	4000	3-12
Motorola	68020	0-16	1-20	4000	5-15

1.3. Gestion des requêtes

Le module de requête a pour but de déterminer l'existence d'une architecture correspondant à un certain nombre de critères définis par l'utilisateur. Ce module est utilisé pour :

- *l'interrogation* des bibliothèques d'architectures à des fins de validation d'une description faite par un utilisateur ;
- *l'interrogation* des bibliothèques d'architectures a des fins de correspondance avec des machines réelles du réseau;
- *la génération* automatique d'une description, même partielle, en explorant un réseau local (sous Unix uniquement).

Le mode d'utilisation de ce module est le suivant :

1) l'utilisateur élabore une requête sur une architecture donnée ;
2) la requête est exécutée ;
3) la réponse à la requête, appelée un compte rendu, est donnée sous forme de messages d'erreurs, d'avertissements ou de résultats ;
4) en cas de réponse non erronée, la construction de l'architecture répondant à la requête est faite. L'utilisateur peut alors parcourir cette architecture.

L'élaboration d'une requête de base se fait soit par sélections successives sur différents critères, soit par composition (intersection, union, différence, choisir). Pour ce type de requête, une fonction unique a été définie : *Type_de_REQUETE (Opérateur, valeur)*.

Les opérateurs pris en compte sont : supérieur, inférieur, égal, différent, supérieur ou égal, inférieur ou égal, appartient, n'appartient pas. Les types de requêtes sont liés aux noms des attributs interrogés pour une entité donnée. Ainsi, par exemple, si on cherche une H-Machine dont la mémoire est supérieure à 32 Mo, on utilisera la commande suivante : MEMORY_SIZE (>, 32). Si on cherche les trois H-Machines les plus rapides et ayant un processeur dans la famille Sparc, la requête générée sera la suivante :

CHOOSE(3, MOST_MIPS, NODE_TYPE (equal, SPARC))

Un arbre de requête est alors généré et un évaluateur tente de répondre à la question. Un compte rendu est ensuite généré.

1.4. Conclusion

Lors de l'implémentation de outils de description et de gestion de modèles décrits en HADEL, nous avons privilégié une approche par composant. Le but étant de favoriser l'interopérabilité de ces outils avec des environnements externes. Nous avons donc particulièrement soigné les bibliothèques d'interface offrant la gestion des modèles. L'utilisation de ces outils par une plate-forme de répartition de charge telle que gatostar est alors possible.

2. Présentation de Gatostar

Nous présentons dans la suite de cette section l'environnement système de Gatostar [Folliot 92 & 96]. Puis, nous étudions le modèle d'application et le langage de description qu'il propose. Nous passons enfin en revue les politiques de placement dynamique multi-critères disponibles.

2.1. Environnement système de Gatostar

Gatostar est un environnement de répartition dynamique de charge qui fonctionne sur un réseau local de stations de travail disposant de processeurs hétérogènes et de systèmes d'exploitation compatibles UNIX. Gatostar a été écrit entièrement au dessus du système d'exploitation, sans modification du noyau et sans support matériel spécifique. Gatostar est multi-utilisateurs, multi-applications et gère le calcul du vecteur de charge des stations et son transfert, la détection des fautes et le redémarrage des machines. Récemment, Gatostar s'est enrichi d'un service de migration de processus [Folliot 96]. Enfin, les modèle de fautes gérées par Gatostar est le modèle "silence sur défaillance". Notons que la faute du réseau est vue comme la faute d'une machine et que Gatostar ne prend pas en compte les cas difficiles du partitionnement du réseau.

2.2. Le modèle d'application et le langage GEL

Gatostar gère les applications dynamiques à longue duré de vie et à gros grain de parallélisme. Le modèle d'application est celui des processus communiquants reliés entre eux par un graphe de précédence. La description des applications est réalisée par le concepteur de l'application :

1) soit dans un fichier décrivant au moyen du langage GEL (*Gatos External Language*) pour la partie statique de l'application ;
2) soit en utilisant des fonctions de la bibliothèque pour la partie dynamique (création/destruction de processus, communication, synchronisation) [Folliot 92].

A l'aide du langage GEL, il est possible d'effectuer :

- la description qualitative et quantitative d'une application : processus, communication, fichiers et codes binaires hétérogènes ;
- le choix du critère de placement : charge, temps, mémoire, communication ;
- la définition de forces d'attraction et de répulsion entre processus. Cette fonctionnalité est utilisée pour contraindre l'allocation de certaines ressources tout en conservant une certaine indépendance vis-à-vis de l'architecture physique du système. Ces attractions sont spécifiées dans le graphe d'exécution au moyen des constructeurs «MEME» pour l'attraction et «DIFF» pour la répulsion.

Un exemple d'application est donné dans la Figure 73. Les programmes A et B commencent leur exécution en parallèle. Lorsque A et B sont terminés, le programme C commence. Le programme C communique avec le programme D, utilise le fichier f1 et produit des résultats dans le fichier f2. La description en GEL du programme indique :

- que le nom du programme est C (mot clef «PROG») ;
- que le programme C.sparc s'exécute sur des architectures SPARC ;
- que les algorithmes de placement doivent prendre en compte les besoins en temps processeur de C et la charge des machines (mot clef «ALGO»);
- C doit s'exécuter après la fin des processus A et B (mot clef «APRES») ;
- que C utilise le fichier f1 et produit le fichier f2 ;
- que C communique avec D, qui se trouve sur une machine distante de celle de C (mot clef «DIFF MACHINE»).

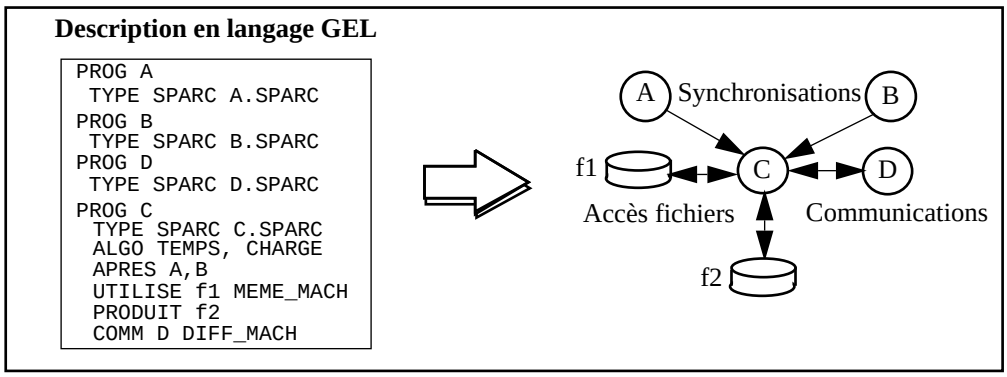


Figure 73 : Description d'une application simple avec GEL

Le lancement d'une application se fait alors avec la commande *gato*, suivie d'options d'exécutions (pour obtenir des traces d'exécution ou l'écriture de statistiques) et du nom et de localisation du fichier GEL.

2.3. Placement dynamique multi-critères

Les processus de Gatostar sont initialement alloués sur des machines inactives. Au cours de leur exécution, les processus sont déplacés vers d'autres machines en fonction des surcharges et des fautes détectées. Le but des algorithmes multicritères et de prendre en compte non seulement l'état des machines, mais également les différents critères d'allocation spécifiés (par exemple dans le fichier GEL). Dans gatostar, les critères pris en compte sont :

- le temps de réponse pour minimiser le temps d'exécution globale de toute l'application ;
- l'accès aux fichiers pour minimiser le coût d'accès aux fichiers ;
- les communications entre processus pour réduire le nombre de communications distantes et ainsi allouer sur la même machine tous les processus communiquants.
- les besoins en mémoire pour sélectionner une machine cible disposant de l'espace mémoire requis pour l'exécution.

La combinaison de ces critères assure une allocation complexe par raffinement successif de l'ensemble des machines sélectionnées pour obtenir la meilleure machine en fonction de l'ordre des critères indiqués [Boutaba & al. 92 et Folliot 92].

2.4. Synthèse

Gatostar est une plate-forme de répartition de charge et de tolérance aux fautes, dédiée aux applications à gros grain, qui communiquent peu et s'exécutent sur un réseau de station de travail hétérogènes ou non. Avec Gatostar l'utilisateur bénéficie de l'ensemble des ressources du réseau en répartissant les processus des applications sur les machines choisies. La politique de placement des processus prend en compte à la fois l'état de la charge du système et les besoins multicritères de l'application.

Dans gatostar, la surcharge moyenne pour placer dynamiquement un processus est de moins de 100 ms. L'algorithme d'allocation multi-critères prenant en compte le temps prévu d'exécution de chaque programme diminue le temps de réponse de 14 à 20% par rapport à un algorithme classique. Ces mesures de performances confirment l'intérêt de Gatostar pour l'exécution d'applications parallèles et expliquent notre choix de Gatostar comme plate-forme système centrale de notre environnement IDEFIX.

3. IDEFIX : intégration de Gatostar, HADEL et VODEL-D

L'intégration de HADEL et de VODEL-D avec la plate-forme de répartition de charge Gatostar se heurte à trois problèmes majeurs (cf. Figure 74).

Le premier est lié au calcul de la granularité des composants à placer. Gatostar pour des raisons de performance dispose de ses propres algorithmes de placement. Ces algorithmes prennent surtout en compte des contraintes liées au matériel. Or, l'équilibrage d'application utilise des algorithmes particuliers s'appliquant avant tout sur l'architecture logicielle.

Le second problème, lié au premier, concerne la transmission de la granularité calculée entre la plate-forme d'équilibrage de charge et l'application gérant le calcul de la granularité des composants logiciels à placer. Gatostar utilise GEL, alors que le langage que nous avons mis au point à cet effet est VODEL-D. Cette transmission d'information est réalisée avant l'exécution (fichier de configuration) et après l'exécution (stockage du fichier de placement optimisé suite à l'évolution de l'architecture logicielle durant l'exécution).

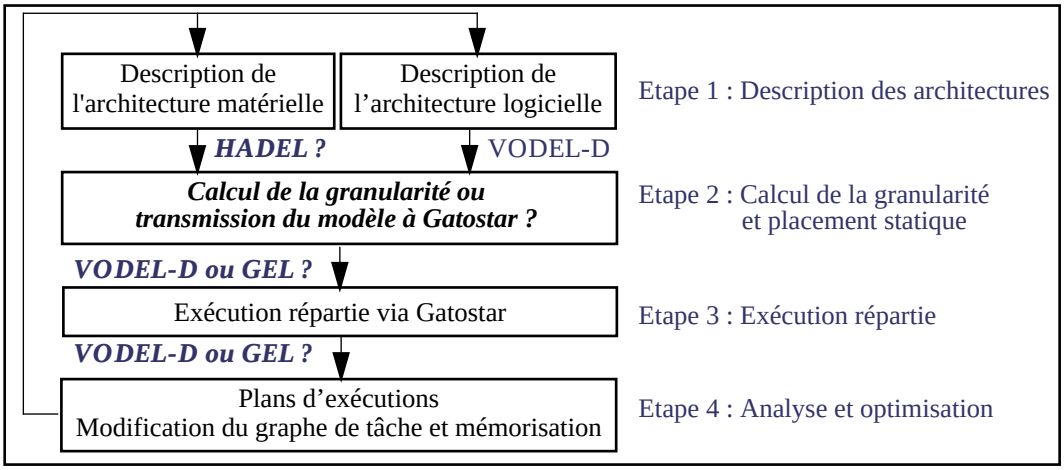


Figure 74 : Mise en oeuvre de la méthode MEDEVER dans IDEFIX

Enfin, le troisième est lié à l'utilisation du langage HADEL. Gatostar dispose de sa propre description statique des machines disponibles pour l'équilibrage de charge. L'utilisateur ne peut pas la modifier, ce qui rend l'utilisation du langage HADEL inutile. Or, nous voulons offrir à l'utilisateur la possibilité de choisir sa plate-forme matérielle.

Nous présentons dans la suite de cette section notre étude de ces problèmes en considérant d'abord une architecture logicielle statique, puis son évolution dynamique. Nous en déduisons alors trois schémas d'intégration de VODEL-D, de HADEL et de Gatostar.

3.1. Placement Statique : de VODEL-D vers GEL

GEL décrit un graphe d'exécution. Son principal défaut est lié à la nature statique de sa description et à la gestion centralisée du graphe d'exécution qu'il utilise pour réaliser le placement.

Dans IDEFIX, VODEL-D est chargé d'apporter les informations nécessaires à Gatostar pour un placement statique. Ces informations concernent la description de l'application parallèle et notamment les communications et le séquencement des processus. Nous retrouvons dans VODEL-D la description du graphe de dépendance nécessaires à Gatostar prenant en compte les communications, les fichiers, la mémoire. VODEL-D offre une compatibilité ascendante avec GEL et dispose d'un pouvoir description supérieur à celui de GEL. Ainsi, grâce aux DVSC, on spécifie le degré d'attraction ou la répulsion entre composants logiciels, le caractère migrable ou non d'un composant, l'appartenance à un groupe de processus donné, etc. Avec les DVSL et DVGL, on représente les liens de communications entre DVSC et leurs synchronisations. Si on reprend l'exemple de la Figure 73, on obtient le graphe VODEL-D de la Figure 75.

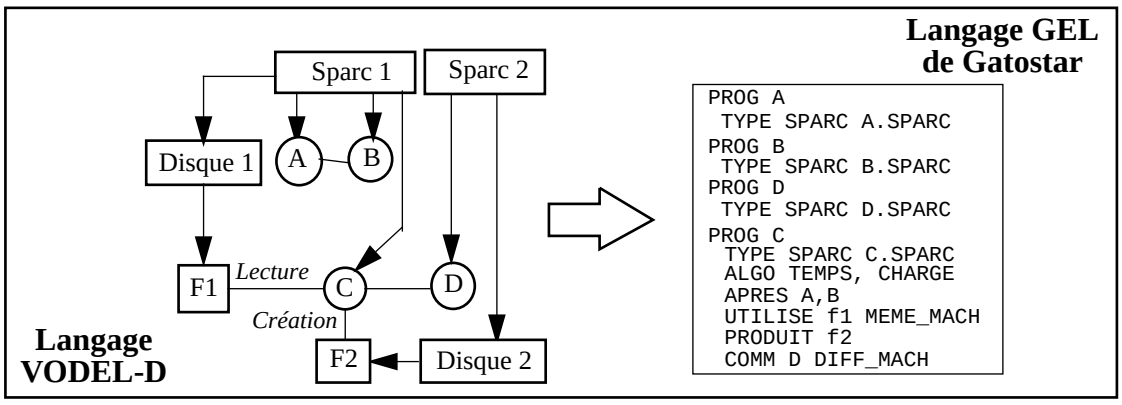


Figure 75 : De VODEL-D à GEL

Sur la Figure 75, les flèches indiquent les relations de dépendances entre entités logicielles et les traits simples les liens de communication entre entités logicielles. Sur les DVSL reliant les entités passives F1 et F2 et l'entité active C, nous avons indiqué le mode d'accès à l'entité passive (lecture et création respectivement). SPARC1 et SPARC2 sont des entités de la classe abstraite Machine matérialisant la répulsion entre les entités actives C et D.

Enfin, l'arbre de précédence des entités logicielles à exécuter est créé par l'utilisation de deux plans d'exécution (cf. Figure 76). Ainsi, les DVSC A et B sont insérés dans le même plan, car ils doivent s'exécuter avant le DVSC C. Le DVSC D est situé dans le plan 2 et communique avec C.

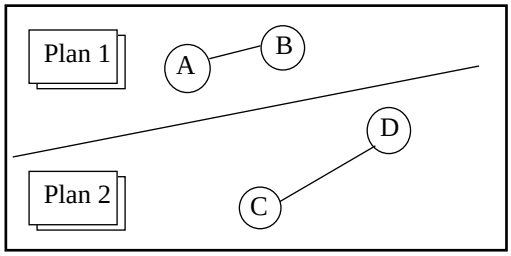


Figure 76 : Plans d'exécution du programme défini à la Figure 75

3.2. Intégration de la gestion des applications dynamiques : VODEL-D

VODEL-D rajoute de nombreuses fonctionnalités par rapport à GEL, notamment en ce qui concerne la prise en compte de la dynamicité. En effet, VODEL-D :

- est un langage dynamique de description, qui intègre la prise en compte de contraintes de migration ;
- dispose des notions d'entités passives logiques et physiques qui peuvent aider au placement ;
- décrit simplement les communications sur groupe ;
- intègre à travers la notion de plan d'exécution; des notions de suivi dynamique de l'exécution et de points de reprise facilitant la tolérance aux fautes (ce que ne fait pas GEL) ;

Outre ces différences, VODEL-D dispose de la notion d'entité d'exécution qui contrairement à l'entité de définition, a été conçue pour le placement et le suivi dynamique d'une application répartie. Ainsi, toute entité d'exécution :

- appartient à un groupe unique, a l'intérieur duquel, on spécifie des indices de liaison (modélisant les contraintes d'attraction et de répulsion) ;
- dispose d'un indice de charge qui est répercutée sur l'indice de charge du groupe ;
- s'exécute à l'intérieur d'un plan d'exécution.

Toute décision de changement, issue des gestionnaires implémentant la politique de localisation, entraîne le changement de plan d'exécution. Ce changement de plan d'exécution entraîne l'écriture des changements dans un fichier de traces, la migration éventuelle des processus entre groupes et la réévaluation des indices de charge et de liaison des entités d'exécution.

Les changements de plan d'exécution peuvent être soumis à un contrat d'exécution, défini par l'utilisateur et indiquant les préconditions à remplir pour que le changement de plan soit valide. En cas de non respect du contrat, la tentative de répartition de charge initiée par Gatostar peut alors être suspendue au nom de contraintes applicatives. Ce cas se présente lorsqu'on veut tester l'extensibilité d'une application sans modifier l'architecture matérielle sur laquelle elle s'exécute.

3.3. Intégration de VODEL-D et de Gatostar

VODEL-D et Gatostar peuvent être intégrés de trois manières différentes (cf. Figure 77) :

1) en gardant l'existant et en rajoutant une surcouche au dessus de la plate-forme Gatostar actuelle. On se contente donc de retranscrire les descriptions faites en langage VODEL-D vers des descriptions faites en langage GEL ;
2) en pilotant directement Gatostar par l'intermédiaire des API qui sont disponibles. On utilise alors les fonctions implicites (redirection des I/O standards, enregistrement, point de reprise) et les fonctions explicites (point de reprise, communications, modification du graphe d'exécution). On pourrait même intégrer un panneau de commande pour que l'utilisateur tape une des commandes interactives (démarrage, arrêt, visualisation de l'exécution de l'application) ;
3) Modifier Gatostar et intégrer VODEL-D directement dans son code source.

Nous approfondissons les trois solutions d'intégration de VODEL-D dans Gatostar dans la suite de cette section.

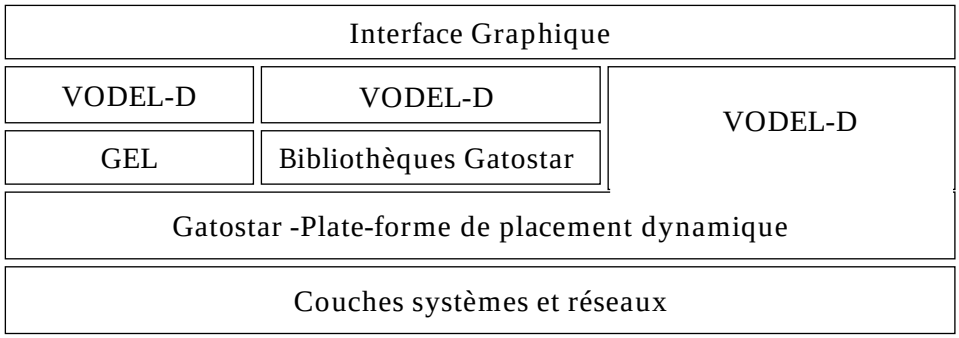


Figure 77 : Trois modes d'intégration de VODEL-D dans Gatostar

3.3.1. Fournir un fichier GEL

Notre réalisation à nous a permis de tester la génération de code GEL à partir du langage VODEL-D. Nous nous sommes alors rendus compte que beaucoup d'informations sont perdues en transcrivant VODEL-D en GEL et notamment :

- la localisation des différents objets VODEL-D qui n'existe pas en GEL ;
- la possibilité de placer une application en fonction d'un critère purement applicatif et non pas lié à la charge du réseau ou du système ;
- la notion de classe abstraite est perdue ;
- la modélisation de la granularité des canaux disparaît ;
- la modélisation des entités logicielles disparaît ;
- la dynamicité de la représentation n'est pas possible.

Néanmoins, le principal avantage de cette réalisation réside dans le fait qu'il est inutile de modifier les fichiers sources de Gatostar. Le programme ainsi créé accède à toutes les informations et structures en mémoire locale et les retranscrit dans un fichier GEL dont le nom est passé en paramètre. Ce fichier sert ensuite au paramétrage du placement dynamique de l'application par Gatostar.

3.3.2. Piloter Gatostar

Dans le deuxième cas de figure, Gatostar est piloté par un gestionnaire d'exécution. Ce dernier réalise tous les calculs de placement nécessaires et interagit avec Gatostar en utilisant la fonction *Gatoexec*. Celle ci est interceptée par Gatostar qui réalise, dans une

seconde étape, son placement en fonction de la charge courante des différentes machines et la demande de l'utilisateur.

Comme les fonctions de gestion de Gatostar sont accessibles par ce gestionnaire d'exécution, l'état du système est consultable. Les processus sont alors éventuellement placés sur des critères autres que ceux de Gatostar. Les fonctions de Gatostar pour le calcul du placement sont donc réutilisées telles quelles, court-circuitées ou affinées. Certaines informations importantes après exécution de l'application ne sont par contre pas accessibles (il aurait été intéressant par exemple de connaître les coûts de communications entre les processus durant l'exécution). Ces données servent au gestionnaire de programme pour effectuer le calcul de la granularité de l'application parallèle à posteriori et à modifier ce placement en cas de réexécution de l'application.

3.3.3. Modifier Gatostar

Dans ce dernier cas, nous supprimons l'interpréteur de commande GEL de Gatostar pour qu'il puisse directement prendre en compte les graphes de dépendance de VODEL-D. Les informations, anciennement perdues par la transcription en fichier GEL, peuvent ainsi être accessibles à Gatostar. Celui-ci disposant alors d'informations supplémentaires est susceptible d'intégrer de nouveaux algorithmes de placement dynamique. La fonction *Gato* qui permet de lancer l'exécution d'une application via un fichier GEL est donc conservée. Par contre, le nom de l'application passé en paramètre correspond à un identifiant d'un plan d'exécution défini dans VODEL-D (et non plus à un fichier GEL). La fin du lancement des processus de l'application parallèle sera validé quand tous les plans d'exécution courants n'auront plus de successeur. Le problème majeur de cette option et qu'il est nécessaire de modifier et donc de réécrire le code de Gatostar.

3.4. Synthèse

L'intégration de VODEL-D est de Gatostar est donc réalisable de manière plus ou moins fine. La prise en compte des spécificités de VODEL-D et l'abandon de GEL entraîneraient une redéveloppement important de Gatostar. Notre mise en oeuvre d'IDEFIX, a donc consisté dans un premier temps à générer un fichier GEL (les autres solutions, trop longues à mettre en oeuvre, ont été écartées). Cette mise en oeuvre, ainsi que les premiers résultats de nos expérimentations, sont présentés dans le paragraphe suivant.

4. Mise en oeuvre d'IDEFIX

Ce chapitre est consacré à la description de l'architecture générale et des choix d'implémentation d'IDEFIX. Dans IDEFIX, les informations nécessaires à la réalisation du placement dynamique sont constituées de la description des objets de VODEL-D et de leurs relations. Ces relations définissent des graphes de dépendance qui représentent l'application. L'évolution de l'application entraîne la mise à jour, selon une fréquence variable, de ce graphe. Si le graphe se trouve sur un serveur, les mises à jour risquent d'entraîner des coûts de communications importants, des protocoles de mises à jour complexes et des interfaces de programmation compliquées. Le seul moyen d'offrir la transparence d'accès et de simplifier les interfaces d'accès à la description dynamique des applications en cours d'exécution a été de passer par un segment de mémoire partagée réparti.

L'architecture d'IDEFIX, présentée en Figure 78, prend en compte cette gestion de la mémoire partagée et est composée de 8 couches distinctes.

La couche système, qui est la couche la plus basse, fournit l'environnement d'exécution réparti. Elle est basée sur l'outil de répartition de charge GatoStar au dessus d'Unix. La couche de gestion de la mémoire partagée manipule les structures de données dynamiques et réparties issues du langage VODEL-D. Une bibliothèque de création et de gestion des graphes VODEL-D, via un segment de mémoire partagée a été implémentée.

Tous les processus placés sur une même machine peuvent ainsi accéder et modifier les informations stockées dans le segment sans aucun échange de message. Un gestionnaire de cohérence est placé sur toutes les machines composant l'environnement d'exécution des applications réparties. Il accède au segment de mémoire partagée, local à la machine et collabore avec les autres gestionnaires pour garantir l'intégrité des descriptions VODEL-D. Un mécanisme a été développé au dessus de ce gestionnaire afin de permettre une utilisation multi-utilisateurs et multi-application de ce segment. Afin de favoriser la modularité du système, quatre compilateurs ont été implémentés pour la modification ou l'insertion de structures de données dans le segment de mémoire partagée. Enfin, un outil graphique facilite l'utilisation des objets de VODEL-D et permet une meilleure visualisation des liens de communication et des dépendances entre classes.

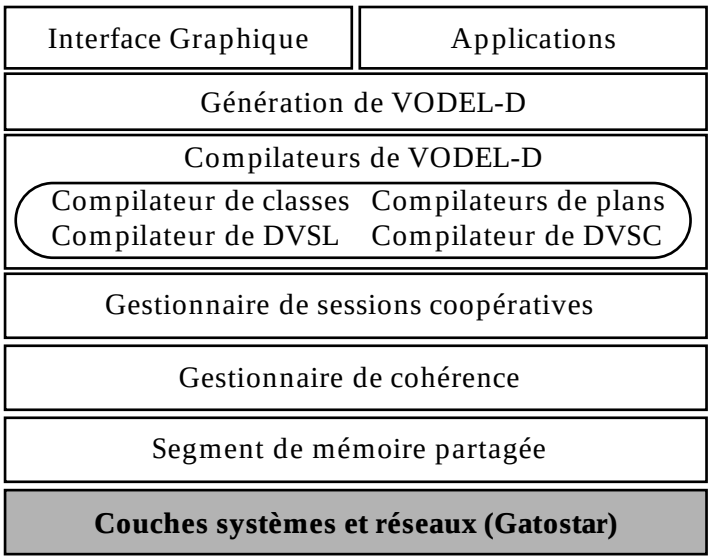


Figure 78 : Architecture d'IDEFIX

Le gestionnaire de segment de mémoire partagée est présenté dans la section 1 et le gestionnaire de cohérence du segment dans la section 2. La section 3 introduit la notion de sessions coopératives. Ces sessions ont pour but de contrôler les opérations effectuées sur les objets gérées en mémoire par les applications. La section 4 passe en revue les quatre compilateurs de VODEL-D (compilateur de DVSC, de DVSL, de classes et de plans). Ces compilateurs retranscrivent du code VODEL-D à l'intérieur du segment de mémoire partagée. La section 5 présente l'interface graphique d'IDEFIX et un outil de visualisation de graphes VODEL-D. Enfin, la dernière section présente les premières mesures de performance effectuées.

4.1. Le segment de mémoire partagée

Les informations à stocker sont les objets VODEL-D et les liens existant entre eux. Une fois compilé les descriptions VODEL-D donnent naissance à des graphes de dépendance représentant l'application parallèle et/ou répartie. Toute application utilisatrice des graphes de dépendance de VODEL-D, comme Gatostar, doit pouvoir les récupérer d'une manière simple et efficace. Pour se faire, nous avons dénombré quatre possibilités :

- **un serveur de graphes VODEL-D unique**

  Le stockage est dans ce cas centralisé sur un serveur. Celui ci est chargé de la compilation des différents fichiers et réalise un graphe des objets. Aucune information n'est à dupliquer et la cohérence des graphes est totale. L'inconvénient majeur de cette solution est la surcharge importante induite sur la machine accueillant le serveur et sur le réseau en terme de communications.

- **des serveur de graphes VODEL-D coopérants**

  Dans cette solution plusieurs serveurs existent et permettent dès lors de répartir le travail de maintien de l'état des graphes de dépendance. Cette solution règle le problème du goulot d'étranglement des accès au serveur, mais pas celui concernant la récupération d'informations. En effet, l'application désirant utiliser les fonctionnalités de VODEL-D ne peut récupérer les informations stockées qu'au moyen d'un dialogue avec le(s) serveur(s) par échanges de messages. Or, ce dialogue entraîne un temps de latence trop important pour certaines applications critiques telles que la répartition de charge. De plus, l'information étant dupliquée des coûts supplémentaires sont à prévoir pour assurer le maintient de la cohérence entre ces différents répliques. Or ces coûts sont suffisamment importants pour réduire à néant les gains du placement.

- **un segment de mémoire partagée local**

  Pour éviter certains de ces coûts de communication, on décide de stocker les informations concernant l'état des applications s'exécutant sur une même machine dans un segment de mémoire partagée local. Toute application est dès lors à même de s'attacher au segment local et de récupérer les informations en minimisant les délais d'attente. Les gestionnaires locaux coopèrent alors entre eux pour assurer le maintien de l'état global de l'application répartie et/ou parallèle. Dans IDEFIX, ce sont principalement des graphes de dépendance issues des compilateurs VODEL-D qui sont échangés entre gestionnaire. Or les graphes sont construits à l'aide de pointeurs et l'échange de pointeurs dans un segment de mémoire partagée n'est pas possible.

- **un gestionnaire de mémoire partagée globale et réparti**

  L'avantage d'un tel gestionnaire est qu'il nous décharge complètement de toute la gestion de bas niveau (communication, maintien de la cohérence des segments mémoire, etc.), tout en fournissant un moyen simple de programmation.

Nous avons d'abord choisi d'implémenter un gestionnaire de segment de mémoire partagée local, pour valider notre approche. Une fois l'approche validée, nous pensons utiliser le gestionnaire de mémoire partagée répartie Treadmarks [Amza & al. 96] pour sa stabilité et ses performances.

Choix d'implémentation du gestionnaire de mémoire partagée local

Dans un segment de mémoire partagée, il n'est pas possible d'échanger des pointeurs, car l'espace mémoire géré est virtuel. Un pointeur est donc une adresse virtuelle dans l'espace mémoire du processus. Chaque adresse virtuelle est propre aux processus possédant le même espace d'adressage. Mais deux processus A et B attaché à un segment de mémoire partagée possèdent deux adresses virtuelles d'attachement différentes. Cette adresse est choisie par le système en fonction de l'espace d'adressage courant du processus appelant. Dans ce cas, un pointeur du processus A contiendra un adresse virtuelle sur l'espace d'adressage de ce même processus. L'adresse du pointeur A dans l'espace du processus B ne pointera bien évidemment pas sur la même zone de donnée. Pour résoudre ce problème nous avions trois solutions :

- **Attacher le segment aux mêmes adresses virtuelles**

  Si tous les processus localise le segment à la même adresse virtuelle, tout pointeur peut être échangé, en supposant qu'il pointe bien sur une zone mémoire du segment de mémoire partagée. Cette solution a l'avantage d'être simple conceptuellement, mais elle est difficilement applicable réellement, car le choix de l'adresse commune n'est pas trivial. Cette solution a donc été abandonnée.

- **Faire une simple transcription d'adresses en mémoire**

  Une autre solution consiste à faire une simple transcription d'adresse. Par une

bibliothèque minimale d'accès, les adresses sont transformées en une adresse pivot calculée en soustrayant l'adresse d'attachement de segment avec l'adresse de la zone pointée. Si cette méthode résoud le problème de l'échange de pointeur, elle laisse celui de l'allocation mémoire entier. Effectivement, si les pointeurs sont utilisés, c'est en parti pour leur caractère dynamique. Il faut donc que la bibliothèque possède des fonctions d'allocation et de désallocation mémoire. Cette solution trop complexe à mettre en oeuvre a aussi été abandonnée.

- **Implémenter une bibliothèque de gestion**

  La dernière solution consiste à créer une bibliothèque autonome et portable de gestion de la mémoire partagée. Nous avons opté pour cette solution, en nous inspirant fortement des travaux réalisés sur la gestion mémoire dans Orca [Bal & al. 96]. Grâce à ce segment de mémoire partagée, toute application utilisant VODEL-D peut consulter les différents graphes construit en mémoire.

L'accès à la mémoire partagée locale étant possible grâce à une bibliothèque aux interfaces clairement définies, il nous restait à assurer le maintien de la cohérence des informations manipulées dans le segment de mémoire partagée.

4.2. Le gestionnaire de cohérence

Le problème de la cohérence se pose dés lors que les différents sites modifient une même partie de l'espace partagé. En effet, comment assurer que ces modifications sont visibles par tous les sites ? Nous présentons dans la suite un rapide état de l'art des modèles et des protocoles de cohérence (fortement inspiré du mémoire de thèse de C. Dumoulin [Dumoulin 97]). Puis nous présentons l'algorithme que nous avons mis en oeuvre.

4.2.1. Modèles et protocoles de cohérence

Le modèle de cohérence le plus intuitif est le modèle de cohérence stricte [Censier & al. 78].

Définition 20 : Modèle de cohérence stricte

Un système est strictement cohérent si une opération de lecture sur une donnée à un emplacement mémoire précis retourne la valeur obtenue par la dernière opération d'écriture sur cette donnée.

Cette propriété assurée directement par l'ordre des programmes en univers séquentiel nécessite d'être revue dans un univers parallèle. L'objectif des premiers modèles de cohérence inventés, comme la cohérence séquentielle, était de fournir une mémoire dont le comportement était proche d'une mémoire centralisée (toute écriture entraînant une mise à jour immédiate). Les modèles de cohérence plus récents, tels que la cohérence relâchée, ont pour objectif de limiter la quantité d'informations échangées. Pour cela, ils relâchent les contraintes pesant sur le comportement de la mémoire et introduisent une opération de synchronisation dans cette mémoire. Cette opération permet au protocole mieux exploiter les types d'accès à la mémoire. On distingue alors le modèle de cohérence (qui est la manière dont réagit la mémoire quand on l'utilise), du protocole pour maintenir cette cohérence (qui est la manière dont est implémentée la mémoire partagée).

On distingue alors les modèles sans opération de synchronisation des modèles avec opération de synchronisation.

Modèles sans opération de synchronisation

Le modèle de cohérence séquentielle a été proposé par Lamport [Lamport 79] et mis en oeuvre dans un système réparti nommé Ivy [Li & al. 89].

Définition 21 : Modèle de cohérence séquentielle (sequential consistency)

Le résultat de toute exécution est le même que celui où les opérations mémoire de tous les processus seraient exécutées dans un ordre séquentiel donné. Dans cet ordre séquentiel, les opérations de chaque processus individuel apparaissent dans l'ordre spécifié dans leur programme.

La cohérence causale [Hutto & al. 90] a été proposée comme une alternative à la cohérence séquentielle. Les auteurs de cet article ont alors découvert que seuls les opérations étant en relation causale doivent apparaître dans le même ordre pour tous les processus. Ainsi, ils ont pu relâcher les contraintes imposées sur les opérations d'écriture.

Définition 22 : Modèle de cohérence causale (causal consistency)

La cohérence causale assure que chaque site observe dans sa séquence l'opération de lecture avant celle d'écriture. Les écritures potentiellement en relation causale doivent être vues par tous les processus dans le même ordre. Les écritures concurrentes peuvent être vues dans des ordres différents sur les différents sites.

Modèles avec opération de synchronisation

Il est possible d'aller plus loin en ce qui concerne l'allègement des contraintes pesant sur le maintien de la cohérence. Ces nouveaux modèles de cohérence distingue deux types d'accès : les accès aux variables partagées et les accès aux variables de synchronisation. Ces derniers permettent de synchroniser la mémoire après un certain nombre d'opérations effectuées sur les variables partagées. Nous allons décrire la cohérence faible, puis la cohérence relâchée et la cohérence par entrée.

Le modèle de cohérence faible [Dubois & al. 86] introduit une variable de synchronisation qui est appelée explicitement par le programmeur, avant et après une opération dans la mémoire partagée.

Définition 23 : Modèle de cohérence faible (Weak Consistency)

Il est basé sur trois propriétés :

les accès aux variables de synchronisation présentent une cohérence séquentielle ;

l'accès à une variable de synchronisation ne peut pas se terminer tant que les opérations d'écritures et de lecture ne sont pas terminées sur les sites ;

il ne peut pas y avoir d'accès en lecture ou en écriture dans la mémoire partagée tant que tous les accès aux variables de synchronisation ne sont pas terminés.

Ce modèle de cohérence permet de regrouper les modifications intervenues dans la mémoire partagée et de les diffuser en un seul message lors de l'opération de synchronisation. Ce modèle ne fait aucune distinction entre le commencement d'un accès et la fin de cet accès. Le protocole de maintien de la cohérence doit donc s'assurer que toutes les écritures sont terminées et que toutes les modifications ont été prises en compte.

C'est pourquoi le modèle de cohérence relâchée [Garachorloo & al. 90] utilise deux variables de synchronisation (*acquire* et *release*). Les accès aux variables de synchronisation sont appelées accès spéciaux, en opposition aux accès ordinaire.

Définition 24 : Modèle de cohérence relâchée (Release Consistency)

Une mémoire présente une cohérence relâchée si elle remplit ces conditions :

les accès spéciaux présente une cohérence séquentielle ;

avant qu'un accès ordinaire ne puisse être accompli en respect avec les autres processus, toutes les acquisitions (acquire) précédentes doivent être terminées ;

avant qu'une relâchement (release) ne puisse être accompli en respect avec les autres processus, tous les accès ordinaires (lectures et écritures) précédents doivent être terminés.

Il existe deux variantes du modèle de cohérence relâchée :

- le modèle de cohérence relâchée impatiente (*eager release consistency*) a été implémenté dans Munin [Carter & al. 91]. Il consiste à propager les modifications intervenues entre un *acquire* et un *release* au moment du *release*. Toutes les modifications intervenues entre ces deux évènements sont envoyées aux copies en un seul message et peuvent être reçues par des processus qui ne s'en serviront pas.
- le modèle de cohérence relâchée paresseuse (*lazy release consistency*) a été implémenté dans TreadMarks [Amza& al. 96]. Il consiste à propager les modifications intervenues sur un site au moment du *acquire*. Le processus acquéreur demande alors les dernières modifications intervenues. Les modifications ne sont envoyées qu'aux processus utilisant la mémoire partagée et au moment où ils en ont besoin.

Enfin, le modèle de cohérence par entrée proposée dans Midway [Bershad & al. 93] propose d'associer une variable de synchronisation à une ou plusieurs variables partagées. Ainsi, lors des opérations de synchronisation seules les variables partagées associées sont à prendre en compte. Ce modèle de cohérence autorise l'existence de plusieurs sections critiques, parcourues parallèlement par plusieurs processus et mettant en jeu des variables distinctes. L'inconvénient est que c'est le programmeur qui doit explicitement associer les variables partagées aux variables de synchronisation.

Synthèse

L'utilisation d'un modèle de cohérence garantit au programmeur un état précis de la mémoire. Toutes les mémoires partagées réparties ont le même inconvénient : le temps d'accès à une information partagée est plus long qu'à une information ordinaire.

Un modèle de cohérence forte entraîne un comportement proche d'une mémoire centralisée, mais dégrade les performances de l'application. L'utilisation de modèles relâchés augmentent les performances, en forçant la cohérence sur un groupe réduit d'opérations par synchronisation. Mais les modèles relâchés demandent un plus grand travail du programmeur, qui doit explicitement indiquer les acquisitions et les relâchements des variables de synchronisation.

4.2.2. Le modèle mis en oeuvre

L'algorithme développé pour la gestion de la cohérence des différents graphes dans un univers réparti est un dérivé de l'algorithme de Li et Hudack [Li & al. 89]. L'environnement d'exécution de VODEL-D est composé de plusieurs stations de travail connectées entre elles via un réseau Ethernet. Ces stations sont principalement des Sun Sparc et HP. Unix est le système d'exploitation pilotant ces machines. Comme VODEL-D est implémenté dans ce contexte très précis, l'algorithme de Li et Hudack peut alors être optimisé afin d'obtenir de bonnes performances. L'un des points d'optimisation concerne l'invalidation des pages. Nous présentons dans la suite l'algorithme de Li et Hudak et les optimisations que nous avons implémenté.

L'algorithme de cohérence de Li et Hudack

La mémoire virtuelle de chaque machine est découpée en pages. La mémoire virtuelle de l'ensemble du réseau est composée de toutes les pages mémoires des machines faisant partie du réseau. Chaque page possède un identifiant unique. Les accès en lecture n'entraînent qu'un verrouillage local de la page, ce qui permet pour une même page d'être accédée simultanément sur des machines différentes. La page accédée en écriture, qui ne peut être partagée, est verrouillée localement ainsi que tous les répliquas de cette page. Une fois que ce verrouillage est effectué, seul un unique écrivain accède à la page. Les répliquas de cette page sont alors invalidés. Pour retrouver les répliquas des pages un arbre réparti est construit au moyen d'un localisateur appelé propriétaire probable. Une fois les modifications effectuées, la page est alors déverrouillée. Tout accès ultérieur entraînera un défaut de page et la page sera alors rapatriée depuis le dernier propriétaire probable.

Les principales propriétés de l'algorithme sont les suivantes:

- plusieurs lecteurs pour une même page sont permis ;
- un seul écrivain par page ;
- la cohérence séquentielle est assurée ;
- la duplication des pages est passive car la page n'est rapatriée qu'au moment de son accès.
- l'algorithme est entièrement réparti.

Adaptation de l'algorithme

Voici trois points que nous avons modifiés dans l'algorithme de base :

1) *Optimisation des échanges de messages*. Notre environnement d'exécution est constitué de stations de travail sous Unix et les échanges de messages se font via le protocole TCP/IP. Or dans ce dernier, la trame d'échange possède une taille minimale. Une invalidation n'est composée que d'un seul champ correspondant à l'identifiant de la page. Si dans un cas général, cet échange de messages parait optimisé pour de faible taille, nous nous apercevons que dans notre contexte de travail, celui ci entraîne l'émission d'une trame dont 99% des ses octets sont inutilisés. Notons que dans le cas des machines massivement parallèles le problème subsiste. Dans ce cas, ce n'est plus une perte de place dans les trames qui est mise en cause, mais un temps de latence trop long (en effet, ce qui coûte le plus cher dans l'échange d'informations dans les machines parallèles est l'émission du premier octet). Ce problème est relevé dans plusieurs travaux comme dans [Aho & al. 89]. Pour palier à cette latence les compilateurs comme Fortran D regroupe les données à véhiculer. La solution apportée pour optimiser cette communication indispensable, est de forcer la réplication des pages. Celles ci deviennent alors actives. La page modifiée est insérée dans le message d'invalidation et est disponible sur la machine, lors d'un accès ultérieur. On évite alors la communication engendrée pour la récupération de la page souhaitée. L'invalidation se fait alors après la modification de la page, celle ci étant insérée dans le message, et transmise à tous les sites qui possédaient l'ancienne page au moyen de la technique des propriétaires probables.
2) *Verrouillage des pages locales*. La lecture simultanée de page n'est permise que dans le cas précis ou les lecteurs sont sur des machines distinctes. En effet puisque les pages sont verrouillées localement, une seule lecture locale est autorisée. Or il parait logique que cet accès simultané puisse se faire pour des processus d'une même machine. La solution apportée à ce problème consiste à utiliser un moniteur pour verrouiller les pages plutôt que de simples sémaphores.

L'algorithme ainsi modifié réduit le nombre de communications, car seules les communications pour les invalidations subsistent. De plus, l'utilisation des techniques de diffusion sur Ethernet (*Broadcast*) permettent d'échanger une seule trame, donc un seul message. Nous sommes néanmoins conscient que ce choix nuit à la portabilité et à la fiabilité de notre implémentation. Néanmoins, les principales modifications on été réalisées en vue d'obtenir les meilleures performances possibles pour l'environnement correspondant à un ensemble de stations connectées entre elles via TCP/IP sur un réseau Ethernet.

Le paragraphe suivant présente la notion de session coopérative, développées pour que l'utilisation du segment de mémoire partagée soit réalisable par plusieurs utilisateurs simultanément et en garantissant un niveau de sécurité et d'efficacité suffisant.

4.3. Sessions coopératives

Le travail coopératif est un élément important d'IDEFIX. L'idée est d'offrir à plusieurs applications la possibilité de travailler sur un même objet VODEL-D. Si les graphes sont accessibles de façon concurrente et par différents utilisateurs, des mécanismes de protection doivent être mis en place. Nous n'avons pas envisagé de créer une application d'authentification. Cette composante de contrôle de sécurité d'accès sera donc gérée par le système Unix, même s'il est à noter que ce dernier n'est pas adapté pour la création de sessions coopératives.

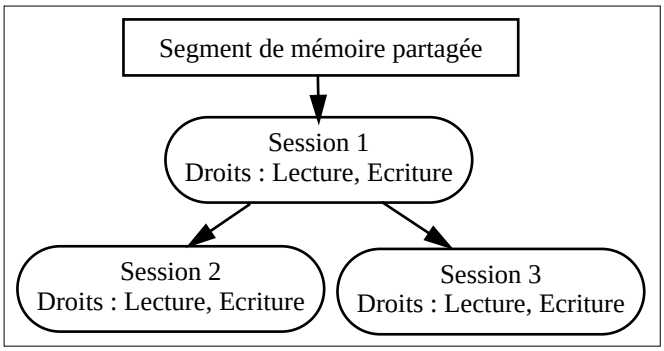


Figure 79 : Exemple de trois sessions coopératives

Le principe du travail coopératif dans VODEL-D est basé sur les sessions coopératives. Une première session, appelée root, possède tous les droits sur tous les fichiers contenant des descriptions VODEL-D (DVSL, DVSC, classes et plans). Les utilisateurs et les applications sont vues de manière identique. Chaque application peut alors ouvrir une session qui va s'abonner à la session root.

Ainsi par exemple, dans la Figure 79, la session 3 s'est abonnée à la session 1 (root). Une session applicative possède des droits en lecture et écriture sur les fichiers VODEL-D. Quant un fichier est créé, il appartient à la session créatrice, si bien qu'une session est en faite une liste de fichier accessibles en lecture et/ou en écriture. La liste des applications pouvant accéder à une session est gérée par un utilisateur particulier appelé root. Les sessions sont construites de manière hiérarchique et toute sous session d'une session possède des droits inférieurs (liste des fichiers accessibles plus restreinte) à la session mère. Par cette technique, on abstrait le problème des droits des applications en le restreignant au problème des sessions.

Les applications désireuses d'utiliser IDEFIX et produire des graphes VODEL-D, s'abonnent à une session coopérative. Les premières applications développées qui utilisent les sessions coopératives sont les compilateurs de VODEL-D.

4.4. Les compilateurs de VODEL-D

Les compilateurs ont pour objectif d'insérer les objets VODEL-D créés par IDEFIX et les relations inter-objets dans le segment de mémoire partagée. Ils sont au nombre de quatre : un compilateur de DVSC, de DVSL, de classes et de plans. Posséder quatre compilateurs permet de mieux découper l'application et évite de tout recompiler à chaque fois.

4.4.1. Le compilateur de classes

Le compilateur de classes à pour but de vérifier la syntaxe du fichier de description des classes d'objets et d'insérer ceux ci dans le segment de mémoire partagée. L'analyse syntaxique construit les différentes entités et définit les liens de dépendance entre les objets représentant les entités. Les erreurs syntaxiques sont détectées et signalées. Si le fichier est valide, les objets décrits sont insérés dans le segment de mémoire partagée via la bibliothèque de gestion du segment.

Néanmoins, il n'est pas souhaitable de définir tous les types d'entités logiques possibles, ainsi que les propriétés qui leurs sont associées et ceci pour plusieurs raisons :

- la classification de toutes les entités logiques existantes dans un système est hors de notre propos ;
- la liste des propriétés des entités logiques est très liée au système sous-jacent et dépend du type d'applications ;
- l'adjonction de toutes les entités logiques entraînerait une lourdeur du langage et donc un surcoût important lors de la compilation.

Pour résoudre ce problème, nous avons créé les objets noirs.

Définition 25 : Objet noir

*Un objet noir est une entité passive logique contenant un ensemble **de données** de n'importe quel type.*

Le langage VODEL n'est donc pas dépendant d'une application particulière puisque tout type d'objet peut être défini. La définition et l'intégration de nouvelles classes dans le squelette d'implémentation se fait au moyen d'un compilateur spécifique appelé DNC (*Define New Class*). Néanmoins, la modification dynamique des classes du langage entraîne des problèmes de cohérence entre différentes versions de VODEL. Pour pallier ce problème, les objets redéfinis possèdent quelques restrictions (Définition 26) afin que le nouveau langage soit cohérent et ne perturbe pas les applications travaillant avec un VODEL non augmenté.

Définition 26 : Règles de cohérence en cas de rajout de classes

Si l'entité modélisée est passive logique elle doit dépendre d'au moins une entité passive physique. Ce champ correspond à la localisation de l'entité logique.

Si l'entité modélisée est passive physique elle doit dépendre d'au moins une entité passive physique. Ce champ correspond à la localisation de l'entité physique.

Toutes classes modélisant une entité ne répondant pas aux définitions précédentes doivent être déclarées avec un type abstrait.

L'utilitaire DNC a pour but de modifier la grammaire de VODEL-D afin de créer des classes personnalisées. Dans VODEL-D, tout objet possède trois propriétés de base qui sont le caractère de sa classe (abstraite / concrète), le type de sa classe et ses dépendances. Cet objet minimal est appelé un noeud. Le corps de l'objet peut dès lors être défini

suivant l'utilisation du langage VODEL-D. Le programmeur défini dans un premier temps la structure de l'objet en langage C dans un fichier dnc.h. Puis, la grammaire permettant d'analyser syntaxiquement l'objet est écrite dans le fichier dnc.y. Cette grammaire suit la syntaxe du langage Yacc. Les *tokens* (correspondant aux identifiants de la grammaire) sont décrits dans le fichier dnc.tok. L'analyseur lexical est écrit dans le fichier dnc.l. Ces quatre fichiers définis, le programmeur lance le programme DNC qui automatiquement réécrit le fichier yacc et lex de VODEL-D et recompile le programme chargé de reconnaître les objets et de construire le graphe associé. Grâce à DNC, le langage VODEL-D devient extensible.

4.4.2. Le compilateur de DVSL

Le compilateur de DVSL se charge d'insérer les objets DVSL dans le segment de mémoire partagée. Comme son homologue, il vérifie la syntaxe du fichier de description des DVSL, mais aussi la validité des références faites sur les objets de type entité. Si l'objet entité n'est pas présent dans le segment, une erreur à la compilation est alors générée. Cette erreur fournit le nom de référence de(s) objet(s) inconnu(s). Le déverminage est ainsi grandement simplifié.

4.4.3. Le compilateur d'entités logicielles

Le compilateur des entités logicielles est chargé de vérifier et d'insérer dans le segment de mémoire partagée les entités logicielles. Son fonctionnement est identique au compilateur de DVSL. Un contrôle sur les références de DVSL est effectué ainsi qu'un contrôle syntaxique.

4.4.4. Le compilateur de plan

Le compilateur de plan offre les mêmes fonctionnalités que le compilateur d'entité logicielle. Celui-ci a pour but de vérifier et d'insérer les objets plans. Un contrôle syntaxique du fichier est effectué ainsi qu'un contrôle sur les références d'entité logicielle.

Une interface graphique et un module d'espionnage améliorant la convivialité d'utilisation d'IDEFIX ont été conçus et sont présentés au paragraphe suivant.

4.5. Deux utilitaires développés dans IDEFIX

Une interface graphique de description d'application parallèle a été réalisée et a été développée avec les librairies Motif [Gregory 92]. Elle permet de manipuler les différents graphes. Il ne faut pas la voir comme une surcouche au langage, car elle fait partie intégrante de la méthodologie de développement présentée. En effet, le travail coopératif est dynamique et une vision des différents graphes est nécessaire sous peine d'avoir une conception difficile d'application multi-utilisateur.

4.5.1. L'interface graphique

Cet utilitaire a pour but de faciliter la représentation des applications parallèles. Les objets graphiques représentent les objets entités. Grâce à cet utilitaire la description de l'héritage et la visualisation des différents graphes sont très fortement simplifiées. Du graphe de description, il est possible de générer un fichier VODEL-D ou d'insérer automatiquement les objets dans le segment. Dans ce cas précis, un appel aux compilateurs est réalisé directement. Comme l'interface graphique n'accède jamais directement au segment de mémoire partagée, elle peut être exécutée sur n'importe quelle machine y compris celles n'appartenant à la plate-forme IDEFIX. Cette interface reprend toutes les spécificités du langage. Son principal atout demeure dans sa capacité à définir graphiquement les dépendances multiples entre classes. Le système de description hiérarchique et graphique des communications est en cours d'implantation.

4.5.2. L'outil de visualisation du graphe

Nous avons implémenté un utilitaire de visualisation des graphes dans le segment de mémoire partagée. La visualisation est donnée sous forme de texte, si bien qu'il est possible, lors du suivi dynamique d'une application, d'accéder aux fonctions de visualisation et d'enregistrer ces informations dans un fichier texte consultable après l'exécution de l'application. Les différentes fonctions décrivent respectivement : le graphe d'héritage des objets, le graphe des DVSL, le graphe des entité logicielles et le graphe des plans. Une simple édition de liens est nécessaire pour utiliser ces fonctions.

4.6. Performances

Nous avons implémenté la plate-forme IDEFIX en utilisant la plate-forme de répartition de charge Gatostar [Folliot & al. 96].

Nos premiers tests ont montré que le stockage des informations dans un segment de mémoire partagée centralisé est pertinent. L'utilisation du segment de mémoire partagée produit, certes, un surcoût de 40% par rapport à l'utilisation des échanges de messages par IPC System V. Le surcoût provoqué par l'utilisation du segment est déduit des mesures effectuées sur une application de ping-pong de 500 messages entre deux processus placés sur une même machine (cf. Figure 80). Notre solution a, par contre, l'avantage d'être peu sensible à la taille du message envoyé (de 100 à 1000 octets).

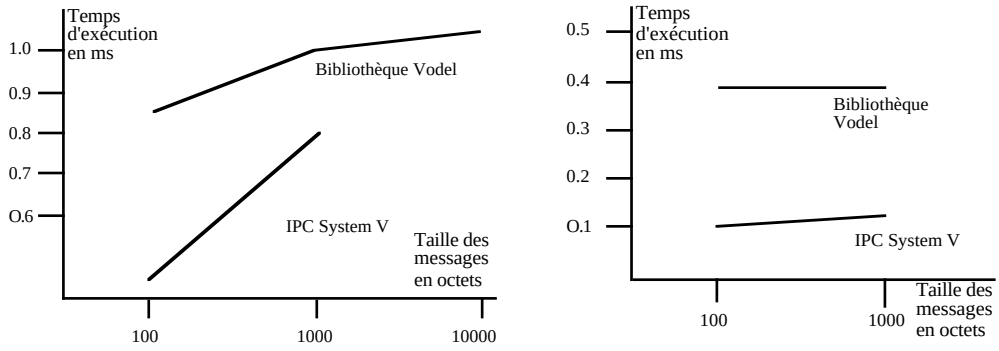


Figure 80 : Temps d'exécution du ping-pong en mode système et en mode utilisateur

Enfin, l'étude de l'exécution du programme de ping-pong entre deux processus, nous a permis de démontrer que près de 60% des fonctions appelées pour l'accès aux données sont des appels systèmes. L'intégration de notre gestionnaire de mémoire partagée à l'intérieur du noyau du système d'exploitation augmenterait de façon significative ses performances. L'inconvénient majeur de cette solution étant de nous rendre captifs d'un système d'exploitation donné.

5. Perspectives

Suite à ces résultats, nous pensons utiliser pour l'implémentation répartie du gestionnaire de cohérence, le gestionnaire de mémoire partagée répartie TreadMarks [Amza & al. 96], commercialisé par la société ParallelTools. Cette plate-forme se présente comme une bibliothèque utilisateur et ne nécessite aucune modification du noyau Unix, ce qui la rend portable. L'unité de partage est la page. Treadmarks met en oeuvre un protocole à invalidation et un modèle de cohérence faible. En particulier, un protocole de cohérence à écrivains multiples, autorise plusieurs processus à écrire simultanément dans la même page à des emplacements différents [Keheler 94]. Une liste de modification est ensuite construite à partir de la comparaison des pages modifiées et transmises aux différents

sites. Nos travaux s'attachent à l'heure actuelle à l'intégration de Treadmarks dans IDEFIX.

Enfin, l'évolution à court terme d'IDEFIX concerne les points suivants:

- création d'une interface graphique de programmation multi-plateforme ;
- création d'une interface conviviale pour l'utilitaire DNC ;
- mise en oeuvre de l'équilibrage d'application, en intégrant les outils de génération de code Tagada dans IDEFIX.
- spécifier et concevoir un gestionnaire de programmes basé sur les entités logicielles de définition et d'exécution.
- améliorer et mettre en oeuvre la notion de sessions coopératives.
- développer une plateforme de conception et d'exécution d'application parallèle reprenant les mécanismes de conception de MEDEVER, en y intégrant des bibliothèques d'algorithmique parallèles.

6. Références bibliographiques

[Aho & al. 89] A.V. Aho, R. Sethi & J.D. Ullman, «Compilers: Principles, Techniques and Tools», Addison Wesley Eds, 1989.

[Amza &l al. 96] C. Amza, A.L. Cox, S. Dwarkadas & al., «Treadmarks: Shared Memory Computing on Network of Workstations», Computer, pp. 18-28, February 1996.

[Bal & al. 96] H.E. Bal, R. Bhoedjang, R. Hofman & al., «Orca: a Portable User-Level Shared Object System», Technical Report IR-408, Vrije Universiteit, Amsterdam, June 1996.

[Bershad & al. 93] B.N. Bershad, M.J. Zekauskas & W.A. Sawdon, «The Midway Distributed Shared Memory System», Proceedings of the IEEE COMPCON Conference, San Fransisco, California, pp. 528-537, February 1993.

[Boutaba & al. 92] R. Boutaba & B. Folliot, «Programs and File Allocation Algorithm for Large Scale Distributed Systems», Applications in Parallel and Distributed Computing, IFIP W.G. 10.3C. Girault Eds., North Holland, pp. 165-174, 1994.

[Carter & al. 91] J.B. Carter, J.K. Bennet & Z.W. Zwaenepoel, «Implementation and Performance of Munin», Proceedings of the 13th ACM Symposium on Operating Systems Principles, pp. 152-164, October 1991.

[Censier & al. 78] L.M. Censier & P. Feautrier, «A new Solution to Coherence Problems in Multicache Systems», IEEE Transaction on Computer Systems, Vol 27(12), pp. 1112-1118, December 1978.

[Dubois & al. 86] M. Dubois, C. Scheurich & F.A. Briggs, «Memory Access Buffering in Multiprocessors», Proceedings of the 13th Annual International Symposium on Computer Architecture, pp. 434-442, 1986.

[Dumoulin 97] C. Dumoulin, «Dream : une mémoire partagée répartie à cohérence programmable», Thèse de l'Université des Sciences et Technologies de lilles, UFR d'IEEA, Bat M3, 59655 Villeneuve d'Ascq Cedex, Janvier 1997.

[Folliot 92] B. Folliot, «Méthodes et outils de partage de Charge pour la conception et la mise en oeuvre d'Applications dans les systèmes répartis hétérogènes», Thèse de l'Université Pierre et Marie Curie, 4 place Jussieu, 75252 Paris cedex 05, Rapport de Recherche 93-27, Décembre 1992.

[Folliot & al. 95a] B. Folliot, K. Foughali & P. Sens, «Placement d'applications parallèles dans les réseaux de station de travail : l'expérience GATOSTAR», Journées de recherche sur le placement dynamique et la répartition de charge: Application aux systèmes répartis et parallèles, Université Pierre et Marie Curie, Paris, Mai 1995, pp 75-78.

[Folliot & al. 95b] B. Folliot, P. sens & P.-G. Raverdy, «Plate-forme de Répartition de Charge et de Tolérance aux Fautes pour Applications Parallèles en Environnement Réparti», Calculateurs Parallèles, Vol. 7 (4), pp. 345-366, 1995.

[Folliot 96] B. Folliot, «Contribution à une approche système du placement dynamique dans les systèmes répartis hétérogènes», Thèse d'habilitation à diriger les recherches de l'Université Pierre et Marie Curie, Décembre 1996.

[Garachorloo &al. 90] K. Garachorloo, D. Lenoski, J. laudon, P. Gibbons, A. Gupta & J. Hennessy, «Memory Consistency and Event Ordering in Scalable Shared-Memory Multiprocessors», Proceedings of the 17th Annual International Symposium on System Computer Architecture, CS Press Ed., pp. 15-26, 1990.

[Gregory 92] K. Gregory, «Programming with Motif», Springer Verlag Eds., 1992.

[Hutto & al. 90] P.W. Hutto & M. Ahamad, «Slow Memory: Weakening Consistency to Enhance Concurrency in Distributed Shared Memories», Proceedings of the 10th International Conference on Distributed Computing Systems, pp. 302-311, May 1990.

[Keheler 94] P. keheler, «Distributed Shared Memory Using Lazy Release Consistency», Thèse de doctorat, Rice University, 1994.

[Kordon &al. 94] F. Kordon & W. El Kaim, «CPN / Tagada User Guide», in the CPN / AMI environment version 1.3, Technical Report, laboratoire MASI, Université Pierre et Marie Curie, Paris, 1994.

[Kordon &al. 95] F. Kordon & W. El Kaim, «H-COSTAM: A Hierarchical Communicating State Machine Model For Generic Prototyping», Proceedings of the 6th International IEEE Workshop on Rapid System Prototyping, June 1995, pp 131-138.

[Lamport 79] L. Lamport, «How to Make a Multiprocessor Computer that Correctly Executes Multiprocessor Programs ?», IEEE Transactions on Computers, pp. 690-691, September 1979.

[Li & al. 89] K. LI & P. Hudak, «Memory Coherence in Shared Virtual Memory Systems», ACM Transactions on Computer Systems, Vol. 7(4), pp. 321-359, November 1989.

[Macao 97] <http: / / www-src.lip6.fr / logiciels / macao>.

[Simulog 88] Simulog (Eds.), «Manuel de référence de QNAP2», 1988.