THÈSE D'HABILITATION DE L'UNIVERSITÉ PIERRE ET MARIE CURIE PARIS VI

Spécialité :

INFORMATIQUE

Présentée par

Pierre SENS

Sujet de la thèse

CONTRIBUTION A L'INTEGRATION DE LA TOLERANCE AUX FAUTES DANS LES ENVIRONNEMENTS REPARTIS

Soutenue le 21 décembre 2000, devant le jury composé de :

Gérard FLORIN, Professeur au CNAM	Rapporteur
Jean-Marc GEIB, Professeur à l'Université de Lille 1	Rapporteur
André SCHIPER, Professeur à l'EPFL	Rapporteur
Claude GIRAULT, Professeur à l'Université Paris VI	Directeur
Bertil FOLLIOT, Professeur à l'Université Paris VI	Directeur
Willy ZWAENEPOEL, Professeur à l'Université de Rice	Examinateur

À mon épouse Catherine,

à mon fils, Antoine,

à ma fille, Lara.

Je tiens à remercier sincèrement :

Monsieur Claude Girault, Professeur à l'Université Paris VI, pour m'avoir accueilli dans son équipe et pour son soutien constant depuis de nombreuses années.

Monsieur Bertil Folliot, Professeur à l'Université Paris VI, pour son intérêt, son aide constante et notre collaboration fructueuse au sein du groupe de Réalisation de Systèmes dont il est responsable.

Monsieur Jean-Marc Geib, Professeur à l'Université de Lille I, Monsieur Gérard Florin, Professeur au Conservatoire National des Arts et Métiers, et Monsieur André Schiper, Professeur à l'Ecole Polytechnique Fédérale de Lausanne, pour l'intérêt qu'ils ont manifesté à mes travaux et pour l'honneur qu'ils m'ont fait d'être rapporteurs.

Monsieur Willy Zwaenepoel, Professeur à l'Université de Rice à Houston, qui me fait l'honneur d'assister à ce jury.

Je remercie tout particulièrement, les membres de l'équipe SRC du LIP6 pour la bonne humeur qu'ils y font régner. Plus particulièrement, Lionel Seinturier, pour ses relectures de mon manuscrit et sa bonne humeur communicative. Philippe Darche et Denis Poitrenaud pour leur amitié et leur soutien.

Je tiens à remercier amicalement, Luciana Arantes, Philippe Cadinot, Damien Collard, Yanal Haj-Mahmoud, Olivier Marin, dont j'ai participé à l'encadrement de thèse et sans qui naturellement le travail présenté dans le manuscrit n'aurait pas pu avoir lieu. Je remercie également les étudiants en D.E.A. que j'ai encadré avec qui j'ai travaillé sur les différents projets présenté dans cette thèse.

Enfin, je remercie l'ensemble du personnel de LIP6 et particulièrement du pôle MSI pour leur gentillesse et l'aide qu'ils ont sues me procurer.

Table des matières

Chapitre 1

Introduction

Sommaire

[17, 18, 16] La tolérance aux fautes est devenue incontournable dans la conception des supports d'exécution répartie. Elle répond à une réalité économique. Des études ont montré le coût sans cesse croissant des défaillances dans l'industrie informatique. En effet, les environnements distribués sont particulièrement sensibles aux fautes. La défaillance, même transitoire, d'un des sites entraîne souvent l'échec de l'application ou une incohérence dans les résultats produits. Ainsi avec la prolifération des architectures réparties, l'intégration efficace et transparente des techniques de tolérance aux fautes est devenue cruciale.

Cependant, peu de systèmes intègrent des mécanismes de tolérance aux fautes. Ceci s'explique par le surcoût induit par ces techniques et leur complexité de mise en œuvre. Nous synthétisons dans ce mémoire les principales stratégies pour gérer les fautes et nous proposons des solutions pour concevoir et intégrer les techniques de tolérance aux fautes tout en préservant et même améliorant le temps de réponse des applications. Nous visons les applications scientifiques qui de par leur durée d'exécution et leur dispersion sur plusieurs sites sont particulièrement sujettes aux fautes. Nous nous intéressons également aux applications multi-agents dont la répartition à une grande échelle et la dynamicité comportementale posent de nouveaux problèmes pour la gestion des fautes.

Ce mémoire présente les activités que j'ai menées au cours de ces six dernières années au sein de l'équipe "Systèmes Répartis et Coopératifs", dirigée par Claude Girault et Bertil Folliot, du laboratoire d'informatique de Paris 6 (LIP6). Mes recherches initialement centrées autour de la tolérance aux fautes en environnement réparti, se sont ensuite élargies vers l'intégration de stratégies de gestion de ressources (processeurs, mémoire) dans les supports

d'exécution répartie. J'ai abordé cette problématique selon trois axes : l'étude et la mise en œuvre des techniques de tolérance aux fautes, l'unification de la tolérance aux fautes et du placement de programme, la gestion de la ressource mémoire en réparti. Bien sûr, il s'agit d'un travail collectif qui a vu la participation de 5 thèses (1 soutenue et 4 en cours) et de 8 stages de Diplôme d'Etudes Approfondies en Systèmes Informatiques.

Dans la suite, les sections 1.1, 1.2 et 1.3 présentent respectivement les trois axes de recherche. La section 1.4 expose le plan de ce manuscrit.

1.1 Tolérance aux fautes en environnement réparti

La réplication de composants matériel et logiciel est largement utilisée pour tolérer les fautes. La plupart des constructeurs offrent dans leur gamme des machines et des systèmes à haute disponibilité où les composants sont répliqués. Par exemple Compaq propose les serveurs NonStop (successeurs de Tandem), de même Sun, HP et IBM proposent des serveurs avec des processeurs redondants. Dans le monde des réseaux de stations de travail, des boîtes à outils facilitent la conception d'applications réparties tolérant les fautes de machines. Ces plates-formes intègrent des protocoles de diffusion fiable afin de constituer des groupes de composants logiciels [42, 196, 125]. Cependant, peu de systèmes répartis intègrent une gestion complète et transparente des fautes : de la détection au recouvrement [174, 130].

Nous avons conçu une plate-forme logicielle, baptisée STAR [165, 167, 166, 168, 169], pour tolérer les fautes des applications réparties sur un réseau local de stations de travail. L'objectif de ce travail était de montrer qu'une approche purement logicielle de la gestion des fautes en environnement réparti est une alternative performante par rapport aux approches basées sur des composants matériels dédiés. STAR masque complètement au niveau applicatif la panne des machines en reprenant automatiquement les processus des sites défaillants. La plate-forme intègre plusieurs techniques de point de reprise et de journalisation de messages ainsi que de nombreuses optimisations (sauvegarde incrémentale et asynchrone). Dans les plates-formes à base de point de reprise telle que STAR, libckpt [147], Condor [189] ou Manetho [72], le support stable où sont réalisées les sauvegardes est un point clef. Dans le cadre du projet STAR, nous avons donc conçu un support stable logiciel en développant un système de fichiers répliqués (appelé SFS). SFS a été optimisé pour réduire la latence d'accès aux serveurs de fichiers. Des mesures de performance ont montré l'efficacité de la plate-forme et l'importance prédominante des optimisations. Cette étude a mis en évidence d'une part les surcoûts des techniques de tolérance aux fautes qui ne peuvent être absorbés que par une gestion globale des ressources du réseau (ce que nous avons mis en œuvre dans le projet GatoStar) et d'autre part la nécessité d'adapter la gestion des fautes en fonction du comportement de l'application [110, 170].

Le besoin d'adaptation de la tolérance aux fautes est particulièrement crucial dans les systèmes multi-agents. Les architectures multi-agents suscitent un intérêt grandissant en tant qu'outil facilitant la conception, le développement et le déploiement d'applications réparties. Or, du fait de leur répartition à une échelle importante, ces plates-formes deviennent

particulièrement sensibles aux défaillances. De plus, les applications multi-agents sont très dynamiques (le degré de "criticité" des agents peut énormément varier en cours d'exécution). Pour répondre à ces besoins, nous avons participé au développement de la plate-forme DarX [209]. DarX intègre des stratégies de tolérance aux fautes dans la plate-forme multi-agents DIMA [95] développée au LIP6 par Zahia Guessoum dans l'équipe OASIS dirigée par Jean-Pierre Briot. DarX adopte une approche réflexive en permettant d'adapter dynamiquement la gestion des fautes en fonction de l'environnement (latence, débit et taux de fautes) et du niveau de criticité des agents. Les agents sont répliqués sur un ensemble de sites. A tout moment la stratégie de réplication et ses paramètres peuvent être modifiés en cours d'exécution. Un premier prototype de DarX est opérationnel et les premières mesures sont prometteuses. Nous visons à définir des heuristiques pour choisir automatiquement la stratégie de réplication la plus adaptée au contexte courant de l'agent (cette étude fait l'objet de la thèse d'Olivier Marin qui a débuté en octobre 2000).

1.2 Unification de la tolérance aux fautes et du placement

Un traitement purement logiciel des fautes entraîne un surcoût augmentant de manière importante le temps d'exécution des applications. L'utilisation de toutes les ressources de l'environnement réparti permet de limiter ce coût. En effet, des études ont montré que dans les réseaux de stations de travail, de nombreux sites sont sous-utilisés ou inactifs à un instant donné. Ainsi, plus d'un tiers des stations peuvent être considérées comme inactives aux heures où les machines sont les plus utilisées [191, 86]. La répartition de charge consiste à faire bénéficier les applications de la puissance de tous les processeurs et particulièrement de ceux qui sont inactifs [121, 35]. Ainsi, grâce à un placement judicieux, les applications réduisent leur temps de réponse compensant une partie du surcoût engendré par la gestion des fautes. Malgré l'apport du placement sur les performances, peu d'expériences ont étudié l'intégration de la répartition de charge dans des plates-formes tolérant les fautes, on peut néanmoins citer DAWGS [58], Paralex [20] ou ICARE [113].

Nous avons unifié avec Bertil Folliot les concepts et mécanismes de tolérance aux fautes et de placement dynamique. Ce travail a donné lieu à la réalisation de GatoStar [45, 88, 87, 86, 89]. Cette plate-forme fournit aux programmeurs des algorithmes pour placer automatiquement leurs programmes et pour tolérer les fautes. GatoStar est l'unification du système de placement Gatos [84] développé par Bertil Folliot et de STAR.

Le gestionnaire de placement est responsable du placement initial des programmes et du replacement en cas de faute. Plusieurs critères sont pris en compte comme la charge et la consommation de ressources des applications. Le gestionnaire des fautes est responsable du support de stockage stable, de la reprise des processus, de la localisation globale des processus et du suivi des messages. Nous avons également intégré des algorithmes de migration en réutilisant le mécanisme de points de reprise de STAR. La migration des processus permet de réagir aux variations de charge en cours d'exécution. Des mesures de performances effectuées en environnement réel ont mis en évidence l'intérêt de cette unification et la synergie existante entre le placement et la tolérance aux fautes.

Une des difficultés des systèmes de placement est d'évaluer et mettre au point de nouvelles heuristiques d'allocation en fonction des applications et de la plate-forme d'exécution. Nous avons conçu, dans le cadre de la thèse de Yanal Haj-Mahmoud, l'outil SIGAP [98, 101, 99, 100]. SIGAP est un modèle paramètrable et configurable pour simuler le comportement d'un système réel de distribution de charge. Il prend en compte la configuration des applications, de l'environnement et des algorithmes de répartition de charge. Afin de valider notre approche nous avons calibré et raffiné notre modèle, jusqu'à obtenir des performances par simulation quasiment identiques à celles observées en environnement réel. SIGAP permet ensuite d'extrapoler les mesures de GatoStar pour diverses architectures et applications et de faciliter la conception d'algorithmes de placement. Notamment, lors de cette étude, des algorithmes à base de seuils dynamiques ont été mis au point dans SIGAP et ensuite insérés avec succès dans GatoStar.

Les algorithmes de placement multi-critères des systèmes de placement tels que MOSIX [31], Utopia [207] ou GatoStar utilisent généralement des informations assez grossières sur les applications (temps processeurs consommés par chaque processus, taux global de communication, nombre de fichiers utilisés ...). Même si ces informations sont importantes et permettent d'affiner les décisions de placement, elles restent néanmoins globales et ne permettent pas de capturer le degré réel de parallélisme entre les processus. Or, ce degré de parallélisme entre programmes est une information essentielle pour le placement. Il permet de regrouper sur une même machine deux processus se synchronisant régulièrement avec un taux de recouvrement des traitements faible. Les modèles de représentation classiques ne permettent pas de capturer une telle information. Afin de définir des stratégies de placement plus efficaces, nous avons défini un modèle de description d'applications permettant d'extraire précisément le degré réel de parallélisme entre les processus d'application. Ensuite des techniques de recherches opérationnelles permettent d'utiliser ces informations pour calculer un placement proche de l'optimum. Ce travail fait l'objet d'une collaboration avec Michel Minoux co-responsable de l'équipe ANP du LIP6 par le biais de la thèse de Damien Collard qui a démarré en 1998.

1.3 Gestion de la mémoire

La gestion de la mémoire est une composante essentielle pour concevoir des supports d'exécutions efficaces. En effet, même si les charges des processeurs sont équilibrées, une mauvaise répartition de la mémoire peut écrouler les performances d'une application en faisant intervenir des périphériques de pagination (swap). Nous abordons la gestion de la mémoire selon deux axes : au niveau du noyau du système en proposant un paginateur réparti et au niveau intergiciel (middleware) en proposant une mémoire partagée répartie adaptée à un ensemble de réseaux homogènes interconnectés.

L'augmentation des débits offerts par les réseaux vers de haut débit amène de nouvelles opportunités pour développer des paginateurs performants. En effet, ils permettent d'accéder

à des données plus rapidement lorsqu'elles sont situées dans la mémoire centrale d'une machine distante inutilisée que lorsqu'elles sont situées sur un disque local. Avec Philippe Cadinot, nous avons mis en œuvre dans FreeBSD un mécanisme de pagination en mémoire distante, Maîs [48]. Maîs a pour objectif d'adapter les stratégies de swap aux caractéristiques des réseaux. En effet, les systèmes de pagination sont optimisés pour les échanges sur disques cherchant par exemple la contiguïté sur disque pour optimiser les déplacements de tête ou limiter les interruptions des contrôleurs DMA. Ainsi pour décharger et recharger efficacement des pages en mémoires distantes, nous avons conçu des stratégies qui répartissent la charge mémoire entre les machines et introduisent plus d'asynchronisme entre le système et les périphériques de swap (i.e., les machines distantes) qui désormais peuvent jouer une rôle actif pour élaborer des stratégies de pré-chargement. Maîs peut servir de support système pour construire d'autres mécanismes comme des mémoires réparties partagées ou des systèmes Raid.

La mémoire partagée répartie (MPR) simplifie le développement d'applications parallèles. Cependant les MPR actuelles sont limitées aux ressources mémoire et processeur d'un réseau local. Pour que ces applications puissent facilement profiter d'une plus grande puissance de calcul, nous avons étendu, dans le cadre de la thèse de Luciana Arantes (soutenance prévue en décembre 2000), la MPR TreadMarks [6] (Université de Rice) à un ensemble de réseaux interconnectés [13, 12, 11, 14, 10, 15]. Le principal objectif d'une MPR sur une telle plate-forme est donc de réduire le coût des communications inter-réseau en privilégiant un partage de données au sein d'un même réseau. Nous avons donc modifié le protocole de cohérence LRC (Lazy Release Consistency) défini par TreadMarks en introduisant deux concepts : l'horloge barrière-verrou [12, 11] et le cache réseau [15]. Dans l'implantation de LRC, les horloges vectorielles permettent de contrôler la causalité des modifications des variables partagées. La propagation de ces modifications a lieu lors d'opérations sur des verrous et des barrières. Notre horloge tire partie de cette propriété. Sa taille n'est plus dépendante du nombre de sites mais du nombre de variables de synchronisations (ce nombre reste généralement faible dans les applications sur les MPR). Nous montrons que l'horloge barrière-verrou permet de caractériser la causalité des opérations de synchronisation. Le cache réseau permet, quant à lui, de réduire la latence inter-réseau qui est le principal goulot d'étranglement des protocoles de cohérence dans les plates-formes multi-réseau. Deux caches ont été introduits. Le cache implicite exploite les informations de protocole LRC présentes dans chaque réseau local pour éviter des accès distants. Le cache étendu permet, sur chaque réseau, de collecter les informations de cohérence des autres réseaux. Ce cache met en œuvre des stratégies de pré-chargement.

1.4 Plan

Le document est composé de trois parties correspondant aux trois axes de recherche précédemment décrits.

Dans la première partie, dédiée à la tolérance aux fautes, le chapitre 2 présente une synthèse des techniques de tolérance aux fautes en environnement réparti. Nous y exposons et

comparons les principaux algorithmes de points de reprise, et les techniques de réplication. Puis, nous présentons les principales plates-formes de tolérance aux fautes opérationnelles. Enfin, nous terminons ce chapitre en décrivant la conception du système STAR et sa mise en œuvre. Nous présentons les techniques de détection et de recouvrement de fautes, le support stable et une évaluation de performances en environnement réel.

Le chapitre 3 décrit la conception et la réalisation de DarX, notre plate-forme pour la tolérance aux fautes des agents. Tout d'abord, nous présentons les principaux mécanismes mis en œuvre pour tolérer les fautes dans les systèmes multi-agents. Puis nous décrivons l'architecture de DarX basée sur la réplication d'agents et permettant d'adapter au cours d'exécution la stratégie de gestion des fautes (et ses paramètres). Nous terminons par une évaluation des performances de DarX.

La partie 2 aborde la problématique de placement et son unification avec la tolérance aux fautes.

Le chapitre 4 présente le support d'exécution GatoStar, successeur de STAR, qui fournit aux applications parallèles une répartition automatique de la charge et une gestion transparente des fautes. Tout d'abord, nous synthétisons les principales techniques de répartitions de charge. Puis, nous présentons les mécanismes de placement et de migration multi-critères de GatoStar. En extension de GatoStar, nous présentons nos recherches actuelles qui visent à affiner les algorithmes de placement en prenant en compte plus d'information sur le comportement des applications. Tout au long de ce chapitre, nous mettons en avant la synergie existante entre les techniques de placement/migration et celles permettant la tolérance aux fautes.

Le chapitre 5 présente l'outil SIGAP qui permet de concevoir et évaluer des stratégies de placement. Nous décrivons, dans un premier temps, le modèle générique à base de files d'attente et son calibrage sur GatoStar. Puis nous présentons une étude de performances pour différentes configurations en essayant d'extraire les paramètres pertinents pour des heuristiques efficaces d'allocation de tâches.

La dernière partie traite de la gestion mémoire en environnement réparti.

Le chapitre 6 décrit notre paginateur en mémoire distante, Maîs. Nous présentons l'architecture du paginateur et son adaptation aux caractéristiques de l'architecture cible (le réseau haut débit HSL) qui oblige à repenser les politiques de "swap" standard des systèmes d'exploitation.

Le chapitre 7 présente notre prototype de mémoire partagée répartie (MPR) multi-réseaux. Après une courte description des caractéristiques des MPR, nous présentons notre horloge logique “barrière-verrou”. Puis nous décrivons notre extension de TreadMarks pour gérer des interconnexions de groupes de machines (ou clusters). Nous terminons ce chapitre par une étude de performances.

Enfin, le chapitre 8 conclut sur nos travaux en résumant nos contributions et décrivant nos perspectives de recherches.

Première partie

Tolérance aux fautes en environnement réparti

Chapitre 2

Tolérance aux fautes en environnement réparti

Sommaire

2.1 Introduction

La tolérance aux fautes est devenue indispensable pour l'exécution d'applications réparties. En effet, l'augmentation du nombre de sites impliqués et la durée des applications scientifiques parallèles augmentent la probabilité d'occurrences de fautes la faisant tendre vers un. Sans un traitement approprié, une application ayant de longs temps de traitement a une probabilité faible d'arriver à son terme.

La tolérance aux fautes a été largement étudiée essentiellement dans le contexte des réseaux locaux et plus récemment pour des réseaux à large échelle. Dans ce chapitre, nous synthétisons

les techniques et algorithmes principaux en nous focalisant sur les techniques à base de points de reprise pour des processus communicants.

Nous présentons également le système Star que nous avons développé dans l'équipe SRC du LIP6. Star est une plate-forme logicielle pour tolérer les fautes des applications réparties. L'objectif est de montrer qu'une approche logicielle au-dessus du système d'exploitation peut être une alternative performante aux architectures basées sur des matériels dédiés et/ou des modifications du noyau.

Dans Star, la défaillance des sites d'exécution est transparente au niveau applicatif. Notre plate-forme est paramètrable et propose plusieurs techniques de points de reprise et de journalisation de messages. Nous avons aussi développé un support stable optimisé. La plate-forme intègre également des optimisations de façon à réduire le surcoût des sauvegarde en absence de fautes.

La section 2.2 présente notre synthèse des techniques de tolérance aux fautes. La section 2.3 décrit la conception, la mise en œuvre et une évaluation de performances de la plate-forme Star.

2.2 Tolérance aux fautes dans les systèmes répartis

Nous présentons les différentes méthodes de gestion des fautes en environnement réparti. Dans un système réparti, il est possible que certains composants du système défaillent alors que d'autres restent opérationnels. Il s'agit alors d'assurer la progression correcte du système.

2.2.1 Types de fautes

Les systèmes diffèrent suivant le type de fautes à tolérer. On peut classifier les fautes affectant une ressource suivant les erreurs qu'elles génèrent. Ainsi, on distingue quatre types de fautes :

Faute franche (ou par arrêt). Cette faute provoque l'arrêt du composant,

Faute transitoire (ou par omission). Cette faute intervient ponctuellement. Elle est instantanée et ne perturbe pas le fonctionnement ultérieur du composant fautif. Elle est généralement due à des perturbations passagères du milieu physique, par exemple, le phénomène de diaphonie dans les lignes de transmissions,

Faute temporelle. Une action se manifeste trop tôt ou trop tard. Le cas le plus fréquent est celui d'une manifestation trop tardive. On retrouve, ce type de faute dans les lignes de transmission qui peuvent acheminer, dans des conditions de surcharge du réseau, un message dans un délai trop long,

Faute arbitraire (ou malicieuse). Le service fourni par le composant s'écarte d'une manière durable des spécifications pré-définies. Un exemple de faute arbitraire est la faute byzantine.

Dans les systèmes répartis, on retrouve souvent la terminologie suivante pour désigner les fautes affectant un site d'exécution :

— Soit le site défaillant s'arrête proprement en suspendant ses transmissions de messages. Dans ce cas, le site est de type **silence sur défaillance** ou *fail-silent* [151].

— soit la défaillance d'un site entraîne de sa part un comportement malicieux. Le site est alors de type **défaillance arbitraire** ou *fail-uncontrolled* [151]. Dans ce cas, le site peut : omettre d'envoyer certains messages, envoyer des messages supplémentaires, envoyer des messages avec un contenu erroné, refuser de recevoir des messages.

2.2.2 Détection des fautes

La détection des fautes est un élément essentielle des plate-forme de tolérance aux fautes qui conditionne la qualité du diagnostic et la rapidité du recouvrement de faute.

2.2.2.1 Modèles temporels

Les mécanismes de détection de faute dans un système réparti diffèrent suivant le "modèle temporel" considéré.

On distingue trois modèles utilisés dans la conception d'un système réparti qui différent quant aux hypothèses concernant les bornes minimales et maximales des délais de communication et de traitement [8] (ces délais comprennent le temps nécessaire à l'émission, la transmission et la réception d'un message) :

— le modèle DBC (Délais Bornés et Connus) : les bornes existent et leurs valeurs sont connues a priori,
— le modèle DBI (Délais Bornés mais Inconnus) : les bornes existent, mais leurs valeurs sont inconnues,
— le modèle DNB (Délais Non Bornés) : les bornes n'existent pas.

Le choix du modèle détermine les solutions à un problème. Par exemple, le problème de consensus réparti a des solutions probabilistes dans les trois modèles [82, 56] et des solutions déterministes uniquement dans les modèles DBC [117, 164] et DBI [63, 66].

On distingue aussi souvent deux approches :

— L'approche **synchrone** a les mêmes hypothèses sur les délais que le modèle DBC [97]. On suppose également que (1) chaque processus a une horloge logique avec une dérive bornée par rapport au temps réel et (2) il existe une borne inférieure et supérieure au temps nécessaire à un processus pour exécuter une instruction.
— L'approche **asynchrone** reprend les hypothèses du modèle DNB.

2.2.2.2 Détecteurs de fautes

Dans le modèle synchrone, les défaillances sont détectées classiquement à l'aide d'un délai de garde. Lorsqu'un processus p envoie un message à un processus q, p s'attend à recevoir une réponse avant un délai d. Passé ce délai, q est considéré défaillant. Selon le type de fautes considérées, il peut s'agir d'une faute franche ou d'une faute temporelle.

Dans le modèle asynchrone, il n'est pas possible de détecter les défaillances avec un délai de garde car il impossible de distinguer le cas où le processus distant est fautif de celui où la réponse est encore en transit. Pour apporter tout de même une solution satisfaisante, le concept de suspecteur (ou détecteur) de faute a été introduit pour T.D. Chandra et S. Toueg [56]. Un suspecteur de faute est chargé d'informer les processus corrects des fautes des autres processus. Les canaux sont supposés fiables (à savoir qu'avec un nombre fini de retransmissions, un

message finit toujours par arriver) et seules les fautes franches de processus sont considérées. T. D. Chandra et S. Toueg ont défini des propriétés permettant de caractériser le comportement et la qualité d'un détecteur de fautes. Les propriétés de *complétude* décrivent la capacité à suspecter tous les processus fautifs, les propriétés de *justesse* apportent des restrictions sur des suspicions erronées. Ainsi T.D. Chandra et S. Toueg ont défini les propriétés suivantes :

- *La complétude forte* définit qu'il existe un instant à partir duquel tout processus défaillant est définitivement suspecté par *tout* processus correct.
- *La complétude faible* définit qu'il existe un instant à partir duquel tout processus défaillant est définitivement suspecté par *au moins un* processus correct.
- *La justesse forte* définit qu'*aucun* processus correct n'est jamais suspecté.
- *La justesse faible* définit qu'il existe *au moins un* processus correct qui n'est jamais suspecté.
- *La justesse finalement forte* définit qu'il existe *un instant à partir* duquel *aucun* processus correct n'est jamais suspecté.
- *La justesse finalement faible* définit qu'il existe *un instant à partir* duquel *au moins un* processus correct n'est jamais suspecté.

A partir de ces propriétés, huit classes de détecteurs de fautes sont définies. Le tableau 2.1 présente ces différentes classes avec leur nom.

TABLE 2.1 – Classes de détecteurs de fautes

Complétude	Justesse			
	Forte	Faible	Finalement forte	Finalement faible
Forte	Parfait P	Fort S	Finalement parfait $\diamond P$	Finalement fort $\diamond S$
Faible	Quasiment parfait Q	Faible W	Finalement quasiment parfait $\diamond Q$	Finalement faible $\diamond W$

Deux résultats théoriques majeurs sont directement extraits de ces travaux. Dans [55], T.D. Chandra, V. Hadzilacos et S. Toueg démontrent que non seulement le problème de consensus peut être résolu en présence de faute par le détecteur $\diamond W$, mais que ce détecteur est le plus "faible" pour résoudre le consensus. Les auteurs de [54] ont également démontré qu'il est possible de construire un détecteur satisfaisant la complétude forte à partir de tout détecteur satisfaisant la complétude faible.

2.2.3 Réplication

La création de copies multiples des processus sur des processeurs différents est le seul moyen pour permettre à un système réparti de continuer à délivrer ses services en présence de nœuds défaillants. On distingue trois techniques de réplication qui ont notamment été largement étudiées et mises en œuvre dans les projets Delta-4 [150], Garf [94] et Bast [91] :

- la *réplication active* dans laquelle toutes les copies traitent les messages en entrée de façon à garder leur état interne étroitement synchronisé.
- la *réplication passive* dans laquelle seule une des copies (la copie primaire) traite les

messages d'entrée et fournit les messages de sortie. En absence de fautes, les autres copies ne traitent pas les messages d'entrée ; leur état interne est mis à jour régulièrement au moyen de points de reprise transmis par la copie primaire. Des techniques de reprise arrière sont utilisées en cas de fautes de la copie primaire.

— la *réplication semi-active* est une technique hybride entre la réplication active et la réplication passive ; les messages d'entrée sont diffusés et traités par toutes les copies mais une seule d'entre elles (le leader) fournit les messages de sortie. En absence de fautes, les autres copies (les suivants) mettent à jour leur état interne soit par traitement direct des messages soit au moyen de notifications de la copie primaire.

La réplication active permet de traiter tout type de fautes. Notamment, elle est la seule à pouvoir se prémunir des défaillances arbitraires en mettant en œuvre un vote sur les sorties des copies. L'avantage principal de la réplication active est que le recouvrement de fautes est quasi-instantané puisque toutes les copies sont maintenues dans le même état. Cependant, cette technique nécessite en permanence d'importantes ressources de traitement. Elle exige, en effet, la création de plusieurs processus sur différentes machines ainsi qu'une utilisation intensive du réseau. De plus, la réplication active n'est applicable qu'à des processus au comportement déterministe, dans le cas contraire, les copies risqueraient de diverger. En raison de sa propriété de recouvrement rapide, ce type de réplication est particulièrement adapté à un environnement exigeant des temps de réponses bornés.

Dans la réplication passive, seule la copie primaire est active à un instant donné. En cas de défaillance de celle-ci, une des copies secondaires la remplace et s'exécute à partir du dernier point de reprise transmis par la copie primaire. La réplication passive s'apparente aux techniques à base de mémoire stable [26, 155], les copies servant de support stable pour la copie primaire. Il existe de nombreuses techniques de points de reprise. Ces différentes techniques sont détaillées dans le paragraphe 2.2.4.

La réplication passive est reconnue comme étant plus performante que la réplication active en absence de fautes. En effet, les sites contenant une copie ne participent pas au traitement impliquant une surcharge de calcul plus faible. De plus, cette approche n'exige pas un comportement déterministe de l'application. Cependant, ces avantages potentiels doivent être pondérés par les facteurs de surcharge suivants :

— le surcoût dû à la sauvegarde des points de reprise,
— le surcoût et le délai de traitement des techniques de reprise arrière en cas de fautes de la copie primaire.

La réplication passive est souvent préférée dans des environnements où les fautes sont rares et lorsqu'il n'y a pas de forte contrainte temporelle. Ces environnements correspondent essentiellement aux réseaux d'ordinateurs faiblement couplés. Cette technique a été notamment adoptée dans les projets Delta-4 [179] , Mach [19] , Chorus [27] et Manetho [73] .

L'objectif de la réplication semi-active est de tirer parti des propriétés de performances de la réplication active et de la capacité potentielle de la réplication passive à gérer les processus non-déterministes. Les copies suivantes reçoivent en entrée les mêmes messages et modifient leur état interne en exécutant de manière autonome les parties de code déterministe. La copie leader est responsable des prises de décisions non-déterministes et informe ses suivantes de ses choix par des messages de notification ou mini-point de reprise. Contrairement à la technique de réplication active, il n'est pas nécessaire que les copies suivantes reçoivent les messages

dans le même ordre. En effet, c'est le leader qui impose son ordre de réception aux autres copies, en envoyant des notifications à chaque réception de messages. Ainsi, les copies suivantes reconstituent l'ordre de réception du leader et elles ne traiteront pas un message tant que le leader ne l'aura pas notifié. Combinant les avantages des deux autres types de réplication, la réplication semi-active est une approche attractive.

Nous résumons les propriétés des trois techniques de réplication dans la table 2.2 tirée de [150].

TABLE 2.2 – Comparaison des techniques de réplication

Type de réplication	Coût de recouvrement	Non-déterminisme	Type de faute
Active	Le plus faible	Interdit	Silence/Arbitraire
Passive	Le plus élevé	Permis	Silence
Semi-active	Faible	Résolu	Silence

Le choix de la technique de réplication est délicat. S'il est clair que dans les environnements temps réel la réplication passive n'est pas adaptée, dans les autres cas plusieurs critères interviennent : le surcoût en traitement, le surcoût en communication, les types de défaillances à traiter, les modèles d'exécution des applications.

En dehors de ces trois modèles de base, d'autres types de réplication ont été définis. Par exemple, la réplication *coordinateur-cohorte* [40], est une stratégie hybride entre la réplication active et passive. Comme dans la réplication semi-active, toutes les copies reçoivent les messages. Cependant, comme dans la réplication passive seule la copie primaire traite les requêtes et renvoie les réponses aux clients.

On peut également citer la réplication *semi-passive* [62] qui diffère de la réplication passive dans le choix de la copie primaire. En effet, dans une stratégie passive, le recouvrement de la faute de la copie primaire est lent car le client doit attendre de connaître l'identité de la nouvelle copie primaire pour réémettre sa requête. Outre la lenteur de cette gestion de groupe (*membership*), la faute n'est pas totalement masquée au client. Dans l'approche semi-passive, il n'y a pas de gestion de groupe prise en charge par les clients. En cas de faute du primaire, les autres copies élisent parmi elles un nouveau primaire en utilisant un algorithme de consensus utilisant des suspecteurs de fautes (voir 2.2.2.2).

2.2.4 Points de reprise

Le recouvrement de fautes à partir de points de reprise est une des techniques couramment utilisée dans le traitement des fautes des applications réparties. Le système sauvegarde périodiquement l'état des processus sur un support fiable pour permettre de relancer les processus en cas de défaillance.

Les techniques de recouvrement à partir de points reprises sont divisées en deux classes : les points de reprise indépendants et les points de reprise coordonnés. Dans la première approche, les processus réalisent indépendamment leur points de reprise et se coordonnent lors de la phase de recouvrement. Dans la seconde approche, les processus se coordonnent au moment du point de reprise de manière à faciliter la reprise à la suite d'une faute.

2.2.4.1 Sauvegarde d'un processus

Un point de reprise est l'image du contexte d'un processus contenant son espace d'adressage (registres, segments de données, pile d'exécution) ainsi que les données du système le concernant. La taille des données à sauvegarder peut être très importante. Deux approches permettent de réduire le coût des sauvegardes :

— Les techniques incrémentales réduisent la quantité d'informations à écrire sur support stable en sauvegardant uniquement les données mises à jour depuis le dernier point de reprise [73].

— Les techniques non bloquantes visent à réduire la latence d'accès au support stable.

Dans la sauvegarde incrémentale, seules les pages de l'espace d'adressage modifiées depuis le dernier point de reprise sont écrites sur support fiable. Il faut donc pouvoir tracer les écritures de pages. La technique la plus couramment employée consiste à retirer les droits en écriture à toutes les pages lors de chaque point de reprise (via les appels système `mprotect` ou `mmap` sous Unix [148] et `mapViewOFile` sous windows NT [107]). Ainsi chaque écriture sur une nouvelle page entraîne soit un signal (sous UNIX) ou une exception (sous Windows NT). La routine de traitement associée mémorise alors le numéro de la page et restitue les droits en écriture.

D'autres techniques visent à limiter le temps de blocage en autorisant les processus à continuer leur exécution pendant l'écriture des points de reprise. Cependant, si un processus modifie une de ses pages pendant la réalisation du point de reprise, l'image sauvegardée ne représente alors plus un réel état du processus. Il y a deux solutions à ce problème.

La première solution utilise le mécanisme de *copie-sur-écriture* (copy-on-write) [83]. Au début du point de reprise, les pages à sauvegarder sont protégées en écriture ; cette protection est retirée dès que les pages sont écrites sur support fiable. Si un processus veut modifier une des pages protégées, le système copie la page dans une page mémoire nouvellement allouée et retire la protection sur la page à modifier. La nouvelle page allouée n'est pas accessible du processus ; elle est utilisée uniquement par le protocole de création du point de reprise. Cette technique a été employée par M. Elnozahy et al. dans Manetho [72] et J. Plank et al. dans la bibliothèque libckpt [147].

La seconde solution consiste à recopier préventivement localement le point de reprise [192, 108]. Avant d'effectuer la sauvegarde, les pages sont recopiées dans une zone mémoire hors de l'espace d'adressage du processus. Ainsi, elles peuvent être écrites sur support fiable sans interrompre l'exécution du processus. Celui-ci peut modifier n'importe quelle page de son espace. Si le nombre de pages modifiées est supérieur à un certain seuil, le coût de la recopie est estimé trop important et le processus reste bloqué durant le point de reprise. La méthode de pré-copie évite les traitements coûteux et complexes du copie-sur-écriture, mais elle peut devenir bloquante si trop de pages sont modifiées. M. Elnozahy et al. ont implanté dans V-system les deux techniques. Ils ont mesuré que le temps pour réaliser un point de reprise avec la méthode de pré-copie est en moyenne 40% plus grand qu'avec la copie-sur-écriture.

2.2.4.2 Maintien de la cohérence en réparti

La notion d'état global cohérent est un point clef des systèmes répartis. Elle a été traitée par [156, 153] et formalisée par K.M. Chandy et L. Lamport [57]. Nous résumons ici leurs

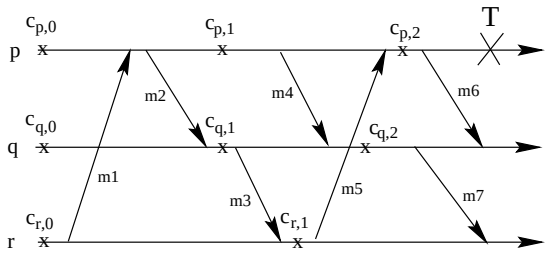

FIGURE 2.1 – Effet domino

définitions.

Dans le modèle d'exécution répartie, un événement est soit un changement d'état spontané d'un processus, soit l'envoi ou la réception d'un message. Un événement a précède un événement b si et seulement si :

— a et b sont des événements du même processus et a a lieu avant b ; ou
— a est l'envoi d'un message m par un processus et b est la réception de m par un autre processus.
— a précède c et c précède b (transitivité).

L'état local d'un processus P est défini par l'état initial de P et la suite des événements ayant eu lieu dans P. L'état global d'un système est un ensemble d'états locaux de tous les processus ainsi que l'état des canaux de communication qui représente les messages en transit. Un état global est cohérent si tous ses événements respectent la propriété de causalité où un message ne doit pas être reçu avant d'être envoyé.

Un ensemble de points de reprise représentant un état global cohérent est appelé *une ligne de recouvrement*.

A la suite d'une faute, pour obtenir un système de nouveau cohérent, il faut reprendre les processus à partir de la dernière courbe d'état cohérente. Ainsi, la reprise d'un processus peut entraîner une avalanche de reprises de processus valides. Lorsqu'un processus "défait" une émission, le message correspondant devient orphelin (il est reçu et non émis), il faut alors "défaire" la réception en reprenant le récepteur. Dans le pire des cas, tous les processus peuvent être repris à partir de leur état initial [156, 160]. La figure 2.1 illustre ce phénomène d'*effet domino*. Les Xs indiquent les points de reprise et les flèches les messages. Si le processus p défaille à l'instant T, il doit être repris à partir de $Cp, 2$. Le message $m6$ devient alors orphelin et q doit être repris à partir de $Cq, 2$. Alors $m7$ devient à son tour orphelin et r doit être repris à partir $Cr, 1$. Finalement, tous les processus vont reprendre leur exécution à partir de leur point de reprise initial.

Plusieurs algorithmes [37, 201, 199] maintiennent la cohérence en recherchant en cas de faute la ligne de recouvrement la plus récente. Ils misent souvent sur le fait que la probabilité d'avoir un effet domino important est faible. L'effet domino est dommageable, non seulement car potentiellement, la durée de reprise n'est pas contrôlable mais surtout un tel effet oblige à garder l'historique de tous les points de reprise. Les points de reprise étant souvent de taille importante, le support stable peut rapidement être saturé. Pour remédier à ce problème, les algorithmes, calculent régulièrement le lignes de recouvrement et éliminent tous les points de reprise précédant ces lignes. On parle de ramasse-miettes de points de reprise (*Checkpoint*

Garbage Collection) [200].

2.2.4.3 Interaction avec le monde extérieur

Les applications réparties sont souvent amenées à interagir avec le monde extérieur pour recevoir des données ou produire des résultats. Le monde extérieur ne peut pas s'appuyer sur des mécanismes de retour arrière en cas de faute. Par exemple, une imprimante ne peut pas revenir en arrière ou un distributeur de billet ne peut pas reprendre l'argent qu'il a donné à un client. Il est donc nécessaire de fournir au monde extérieur un comportement cohérent malgré les fautes. Ainsi avant chaque envoi vers le monde extérieur, le système doit garantir que l'état générant cet envoi pourra être reproduit en cas de faute. Chaque interaction avec l'extérieur doit donc être validée. Classiquement, une état global cohérent est sauvegardé atomiquement après chaque interaction. Cette validation avec le monde extérieur est appelée *output commit*.

2.2.4.4 Points de reprise coordonnés

Dans cette approche, les processus se synchronisent de façon à ce que le dernier ensemble des points de reprise représente à tout moment une ligne de recouvrement. A la suite d'une faute, les processus d'une même application sont repris uniquement à partir de leur dernier point de reprise limitant ainsi l'effet domino. Les principaux inconvénients de ces techniques sont le surcoût entraîné par les protocoles de synchronisation et la charge importante du support stable durant la réalisation d'un point de reprise [57, 114, 119, 109, 141]. Pour minimiser les synchronisations, plusieurs travaux se sont concentrés sur la façon de réduire le nombre de processus participant à un point de reprise [114, 119] ou de diminuer le nombre de messages nécessaires à la synchronisation [46, 37, 193].

On peut distinguer deux classes de protocoles [103] :

— les protocoles à *synchronisation explicite* où des messages de contrôle sont utilisés pour réaliser la coordination [114, 119, 109, 141],

— les protocoles à *synchronisation implicite* où les messages de l'application sont utilisés pour transporter les informations de contrôle [46, 203, 195].

2.2.4.4.1 Synchronisation explicite. L'algorithme de base consiste à geler tous les processus pendant la réalisation du point de reprise [188]. Il faut alors attendre que tous sauvegardent leur état pour continuer l'exécution. Ceci est mise en œuvre par un protocole à deux phases. A chaque processus est associé un point de reprise permanent résultant de la précédente phase de création de points de reprise. Pendant le déroulement du protocole de création, chaque processus maintient un point de reprise temporaire qui remplacera le permanent seulement si le protocole s'est terminé avec succès. Un processus a un rôle de *coordinateur* et envoie des messages de contrôle aux autres processus pour synchroniser le point de reprise. Sur réception de ce message, les processus créent un point de reprise temporaire, envoient un accusé de réception et gèlent leurs émissions. Lorsque le coordinateur a reçu tous les accusés, il envoie à chacun un message de validation. Sur réception de ce message, les processus transforment leur point de reprise temporaire en permanent et autorisent à nouveau l'émission de

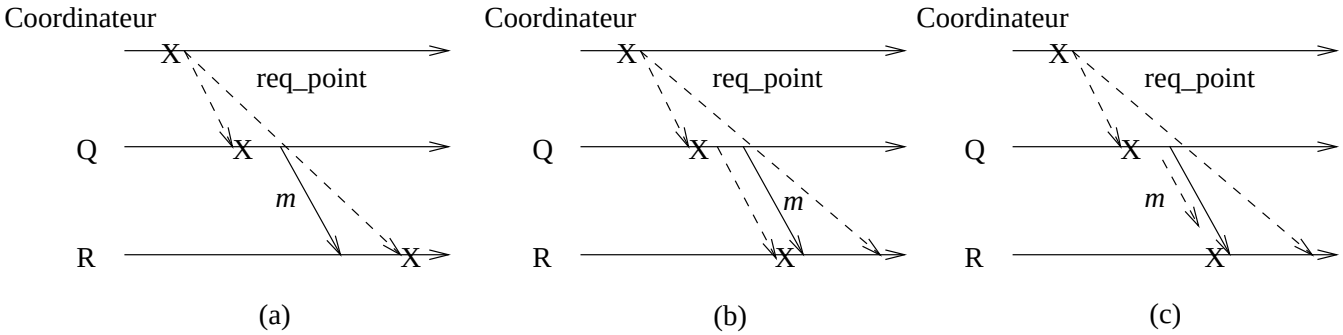

FIGURE 2.2 – Points de reprise coordonnés non bloquants (a) Points incohérents (b) Hypothèse canaux FIFO (c) Canaux non FIFO

messages. Ce protocole est simple mais il nécessite un mécanisme lourd de diffusion en deux phases et surtout il possède une forte latence.

Pour réduire la latence, les auteurs de [57, 116] ont proposé des variantes non bloquantes où les processus peuvent continuer d'échanger des messages même pendant les phases de sauvegarde. La difficulté est de garantir la cohérence des points de reprise et d'éviter des situations telles que celle présentée dans la figure 2.2(a) où le message m est potentiellement orphelin en cas de défaillance du processus Q. Ce problème est résolu simplement par l'algorithme de K.M. Chandy et L. Lamport [57] qui, en supposant des canaux FIFO, renvoie la dernière requête de point de reprise avant chaque émission de message vers un nouveau destinataire (figure 2.2(b)). Si la demande de points de reprise est contenu dans le message applicatif (*piggybacking*), l'hypothèse de canaux FIFO n'est plus nécessaire (figure 2.2(c)) [116].

R. Koo et S. Toueg [114] proposent une amélioration significative de ces algorithmes en limitant le nombre de participants aux points de reprise. Lorsqu'un processus p décide de réaliser un point de reprise, il propage sa demande uniquement aux processus susceptibles de créer une incohérence en cas de reprise, à savoir l'ensemble des processus dont il a reçu un message. Chaque processus recevant une telle demande ne réalise par forcément un point de reprise, il teste tout d'abord si il est toujours en situation incohérente par rapport au demandeur. C'est uniquement dans ce cas qu'il réalise son point de reprise. Grâce à cet algorithme, le nombre de processus participant au point de reprise est minimum.

2.2.4.4.2 Synchronisation implicite. Le problème des algorithmes à synchronisation explicite est la latence et le coût en messages du protocole à deux phases. Les algorithmes implicites exploitent la définition suivante qui indique que deux points de reprise sont cohérents si il n'existe aucun chemin causal commençant *après* un des points de reprise et se terminant *avant* l'autre point.

Briatico et al. [46] furent les premiers à proposer l'approche à synchronisation implicite (connu également sur le terme de *lazy coordination*). Dans leur algorithme, les points de reprise sont numérotés sur chaque site par un compteur incrémenté de un à chaque nouveau point de reprise. Tous les messages applicatifs sont estampillés par l'*intervalle de reprise* courant (i.e., le numéro du dernier point de reprise). Il y a alors deux types de points de reprise :

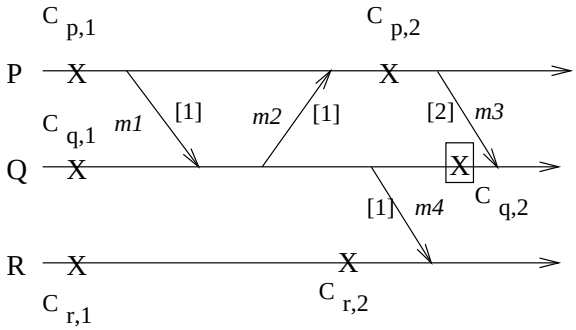

FIGURE 2.3 – Coordination implicite des points de reprise

— les points de reprise spontanés générés directement par l'application,
— les points de reprise forcés générés par la réception de messages de la manière suivante : lorsque l'estampille du message est inférieure au numéro du dernier point de reprise du récepteur, le message est donc potentiellement orphelin, et un nouveau point de reprise est créé sur le récepteur avant de délivrer le message.

Par exemple, dans la figure 2.3, à la réception du message $m3$, $m3$ est potentiellement orphelin en cas de défaillance de P. Pour limiter la propagation éventuelle des reprises, le récepteur (Q) pose un point de reprise $(Cq, 2)$ avant de traiter le message.

Cette approche est séduisante, car elle supprime les fortes latences dues aux synchronisations. Cependant, elle induit un nombre important de créations de points de reprise. Plusieurs approches ont proposé des améliorations notamment en assouplissant la condition pour créer des points de reprise forcés [203, 199]. J. Xu et R. Netzer ont notamment étendu la notion de causalité entre points de reprise en introduisant la notion de Z-chemin (*Z-Path*) .

Néanmoins, l'instant de création des points de reprise n'est plus sous le contrôle applicatif ce qui limite les possibilités d'optimiser le placement des points en fonction de la sémantique des applications.

2.2.4.5 Journalisation

Dans cette approche, chaque processus effectue les points de reprise sans concertation avec les autres. Lorsqu'une faute affecte un processus, tous les processus se coordonnent pour reprendre leur exécution à partir d'un ensemble de points de reprise cohérents.

Ces techniques reposent sur l'analyse des communications pour limiter l'effet domino. Puisque les lignes de recouvrement ne sont pas connues pendant l'exécution normale, chaque processus doit enregistrer des informations sur les canaux de communications dans un emplacement (un journal) qui puisse résister à la défaillance du processus. Ces informations sont utilisées dans les phases de recouvrement pour limiter le nombre de processus à reprendre. Il existe deux grandes classes d'algorithmes :

— les algorithmes de *journalisation optimiste* qui limitent le surcoût d'accès aux journaux en réduisant la quantité de données à écrire et en sauvegardant les informations de manière asynchrone.

— les algorithmes de *journalisation pessimiste* qui enregistrent les informations de manière synchrone pour éviter tout effet domino.

Les deux approches traitent généralement des processus déterministes.

La journalisation pessimiste a été proposée pour supprimer totalement l'effet domino [152, 184, 43, 168]. Les messages sont sauvegardés de façon synchrone. Le processus récepteur reste bloqué jusqu'à ce que le message soit écrit sur le support stable. En cas de défaillance, seuls les processus fautifs sont repris à partir de leur dernier point de reprise et ils consomment directement les messages de leur journal. Si les processus échangent beaucoup de données, le coût de ce type de journalisation peut devenir énorme. De manière générale, les protocoles à base de journalisation pessimiste entraînent un recouvrement de faute rapide pour un surcoût d'exécution important.

La sauvegarde synchrone des messages est le principal inconvénient des méthodes de journalisation pessimiste. L'approche optimiste vise à limiter le taux de dégradation en fonctionnement normal. Les messages sont sauvegardés de manière asynchrone [184, 178, 108, 73] en espérant qu'une faute ne surviendra pas entre l'instant de traitement d'un message et sa sauvegarde sur support stable. Ainsi, plusieurs messages peuvent être groupés et écrits sur support stable en une seule opération. A tout instant, des messages peuvent être en transit (i.e., non validés). En cas de défaillance, les messages en transit peuvent devenir orphelins et entraîner une incohérence impliquant la reprise de processus valides.

La sauvegarde asynchrone des messages limite le coût de la journalisation en permettant à l'émetteur de continuer son exécution sans attendre la confirmation de la sauvegarde des messages sur support stable.

La plupart des algorithmes optimistes utilisent un protocole de recouvrement en deux phases [37, 201]. Durant la première phase, des informations sur les messages échangés sont collectées, puis utilisées dans la seconde phase pour déterminer les processus à reprendre. Y.M. Wang et W.K. Fuchs [201] retardent l'écriture des messages jusqu'au prochain point de reprise. A partir des informations collectées sur les communications, ils construisent un graphe des points de reprise qui leur permet d'exclure l'ensemble des points de reprise inutiles. Ils montrent par une simulation qu'ils réduisent ainsi le surcoût de la journalisation de messages et des points de reprise.

2.2.4.6 Mise en œuvre du support stable

L'implantation du support stable est un point clef qui conditionne directement les performances des techniques de points de reprise. Dans sa définition, un support stable est capable de tolérer des fautes. Naturellement les implantations diffèrent en fonction du degré de fiabilité souhaité. On distingue quatre types de mises en œuvre :

Mémoire volatile : un processus distant sert de copie. Son état est mis à jour en RAM à chaque point de reprise. En cas de faute du processus initial, il suffit de basculer sur le processus de secours. Cette technique a été utilisée dans [43] et [108], elle est très performante mais ne permet de tolérer qu'une faute.

Disque local : une partition d'un disque local est utilisée pour sauvegarder l'état des processus s'exécutant localement. Ce type de technique est dédié à la tolérance aux

TABLE 2.3 – Comparaison des techniques de points de reprise

	Non coord.	Coordonnés		Journalisation	
		Expl.	Impl.	Pess.	Opt.
Hyp. déterministe	Non	Non	Non	Oui	Oui
Surcoût Comm.	Faible	Aucun	Faible	Le plus élevé	Elevé
Surcoût Sauvegarde	Faible	Le plus élevé	Faible	Faible	Faible
Nb. de points	Plusieurs	Un	Un	Un	Plusieurs
GC.	Complexe	Simple	Simple	Simple	Complexe
Reprise	Complexe	Simple	Simple	Simple	Complexe
Effet domino	Possible	Impossible	Impossible	Impossible	Impossible
Propagation de reprise	Illimitée	Dernier point	Dernier point	Minimum	Points précédents
"Output commit"	Impossible	Très lent	Très lent	Rapide	Lent

fautes transitoires où les processus sont relancés sur la même machine à la suite de son redémarrage [58].

Serveur distant : le disque d'un serveur distant, considéré comme plus fiable, est utilisé pour sauvegarder toutes les informations. Souvent, le site "serveur de fichiers" est choisi. Naturellement, tout repose sur la fiabilité de ce site. Pour augmenter son degré de tolérance aux fautes, une sauvegarde régulière du serveur sur un autre site peut être utilisée [72].

Système de fichiers répliqués : le support stable est implantée par des fichiers répliqués sur des machines différentes [169, 146]. Il s'agit de l'implantation tolérant le plus de fautes. Cependant cette approche est coûteuse car elle met en œuvre des algorithmes de mise à jour ayant de fortes latences (des protocoles à 2 ou 3 phases).

2.2.4.7 Etude comparative des techniques

Les protocoles de points de reprise offrent des compromis différents fonction de nombreux critères comme : la dégradation de performance de l'application, la latence des interactions vers l'extérieur, le coût de stockage, la gestion du ramasse-miettes, la simplicité du traitement des fautes, l'absence d'effet domino et la propagation des reprises. La table 2.3 illustre les différentes techniques et compare leurs caractéristiques. Cette table est en partie extraite de [71].

Les points de reprise non-coordonnés ont en général le coût le plus faible en absence de fautes mais sont sujets aux effets domino. L'absence d'effet domino par des techniques de points de reprise coordonnés ou par une journalisation à un prix en termes de surcoût d'exécution.

Les points de reprise coordonnés semblent être une approche attractive dans les environnements où les fautes sont peu fréquentes. Particulièrement, l'approche implicite est peu coûteuse en absence de fautes (pas de messages supplémentaires échangés, pas de latence du protocole de pose de point de reprise, surcoût en parasitage des messages (piggybacking) généralement faible). Cependant, ces techniques ont deux inconvénients majeurs : (i) le nombre de points de reprise engendrés par des situations d'incohérence peut être important, interdisant de fait les optimisations dans la pose des points de reprise, (ii) les interactions avec le

monde extérieur exigent une validation car ces protocoles ne garantissent pas qu'à la suite d'une faute le même scénario soit reproduit.

Les algorithmes de journalisation ont un surcoût important en absence de faute. En effet, même si la pose de points de reprise n'exige pas de synchronisation, les sauvegardes de messages peuvent ralentir de manière conséquente l'application. Cependant, ces techniques présentent les avantages suivants : (i) la pose de points de reprise est complètement libre, (ii) seuls les processus fautifs sont repris, (iii) les interactions vers le monde extérieur n'exigent pas de validation particulière car tout état peut être reproduit.

2.2.5 Groupes et diffusion

La mise en œuvre d'applications tolérant les fautes s'appuie souvent sur la notion de groupe. Un *groupe de processus* caractérise un ensemble de processus coopérant. A tout moment, un processus peut se joindre ou quitter le groupe. La *vue* d'un groupe G est la liste des processus composant G, elle est notée $v_i(G)$. A tout moment les membres d'un groupe ont la même vue. Les vues peuvent changer mais tous les processus perçoivent la même séquence de vues.

Pour implanter ces vues, un service de gestion de groupe (*group membership*) est nécessaire. Un tel service s'appuie sur un détecteur de fautes.

L'envoi d'un message m à un groupe G (appelé diffusion sélective ou *multicast*) peut revêtir différentes sémantiques :

- L'atomicité (*reliable broadcast*) garantit que le message m sera reçu par tous les membres du groupe ou aucun. Ce type de diffusion ne fournit aucune garantie sur l'ordre de réception du message. Elle est souvent utilisée lorsqu'il n'y a qu'une source de diffusion [150].
- La vue synchrone, introduite initialement par K. Birman et T. Joseph [41], garantit qui si un processus p appartenant à $v_i(G)$ a délivré m avant d'installer la vue $v_{i+1}(G)$ alors tous les processus appartenant à la vue $v_i(G)$ traiteront m avant d'installer la nouvelle vue. Les messages sont donc ordonnés en fonction des vues, en revanche au sein d'une même vue, cette approche ne donne aucune garantie sur l'ordre de traitement des messages.

Le modèle de vue synchrone (appelé également le modèle *virtuellement synchrone*) est souvent complété par une sémantique sur l'ordre de délivrance des messages :

- *l'ordre FIFO* garantit le non déséquencement des messages issus d'un même émetteur,
- *l'ordre causal* garantit que l'ordre de réception des messages par les membres du groupe reflète l'ordre causal d'émission,
- *l'ordre total* garantit que l'ordre de réception des messages est le même pour tous les membres du groupe.

2.2.6 Principaux systèmes

2.2.6.1 Plates-formes de communications fiables

Isis est une boite à outils logicielle pour la programmation d'applications réparties tolérant les fautes. Il s'agit d'un ensemble de procédures qui une fois liées aux programmes permettent

de gérer des groupes des processus et de réaliser des diffusions sélectives (ou *multicast*) avec des garanties d'atomicité et d'ordre de délivrance des messages. Isis a été la première plate-forme à utiliser le modèle virtuellement synchrone [42] qui consiste à ordonner les diffusions par rapport aux vues. Isis a introduit le modèle de *partition primaire* virtuellement synchrone où en cas de partitionnement d'un groupe seule la partition ayant une majorité de membres peut continuer à s'exécuter. Le problème de cette approche est qu'il y a des risques de blocage si il n'existe aucune partition primaire. Initialement distribué gratuitement, Isis est désormais exploité commercialement par la société Stratus Computer, Inc.

Le projet **Horus** [196] propose également une plate-forme pour développer des applications fiables. Horus a été conçu pour pallier certains points faibles d'Isis tels que la sécurité ou les contraintes temps-réel. Comme Isis, Horus fournit un support d'exécution basé sur les communications de groupe et la gestion de la réplication avec différents protocoles de cohérence. Horus fournit un ensemble de mécanismes permettant aux concepteurs d'applications de choisir la stratégie la plus adaptée à leurs exigences en matière de performance et de fiabilité. Outre ses aspects modulaire et flexible, Horus se distingue d'Isis en adoptant un modèle où des partitions minoritaires peuvent continuer à fonctionner. Ainsi il peut exister à un instant donné plusieurs vues concurrentes d'un même groupe. La difficulté de cette approche est la fusion des vues concurrentes (*partition merging*) qui peut être délicate notamment lorsque des opérations non réversibles ont été effectuées.

Deux versions d'Horus sont distribuées : une version de base codées en langage C, et une version, appelée **Ensemble** écrite en ML qui peut s'interfacer avec de nombreux autres langages.

Les plates-formes **Relacs** [21] et **Transis** [4] adoptent un schéma similaire à Horus en autorisant des vues concurrentes. Cependant, Relacs impose des restrictions pour la création de nouvelles vues concurrentes. En revanche, dans Transis aucune restriction sur les vues concurrentes n'est imposée.

La boite à outils **Phoenix** [125] permet comme Isis et Horus de construire des applications fiables en implantant des service de gestion de groupe et de diffusion. Elle utilise également le paradigme de communication virtuellement synchrone. L'originalité principale de Phoenix est de se situer dans un contexte à grande échelle dans le sens où les processus répliqués peuvent être très éloignés et leur nombre très grand. Phoenix a une stratégie intermédiaire entre les approches partition primaire d'Isis et partitions concurrentes d'Horus et Relacs. Pour éviter les blocages éventuels de l'approche d'Isis, Phoenix introduit la notion de vue intermédiaire lorsqu'il n'existe pas de partitions primaires.

2.2.6.2 Systèmes à objets répartis tolérants les fautes

Le système **Arjuna** [174] fournit des outils pour la construction d'applications fiables. Ces applications sont structurées en actions atomiques imbriquées [139] agissant sur des objets persistants. Arjuna est implanté en C++ et utilise le concept d'héritage. Ainsi, les objets des utilisateurs peuvent hériter de propriétés telles que la persistance et le recouvrement. Arjuna est basé sur un modèle client-serveur : les invocations d'objets sont toujours issues de clients à destination des serveurs qui gèrent les accès à ces objets.

Arjuna traite uniquement les pannes de sites de type silence sur défaillance. Lorsqu'un

site défaille, deux situations sont possibles : soit le client est en panne et alors le serveur avec lequel il communiquait devient orphelin, soit le serveur est défaillant et alors le client ignore la panne jusqu'à la prochaine invocation de l'objet géré par le serveur (c'est à ce moment que la panne est détectée). Les serveurs orphelins peuvent attendre indéfiniment la prochaine requête de leur client. Pour détecter une telle situation, les serveurs vérifient régulièrement si leurs clients sont toujours valides.

Garf [130] est un environnement de programmation pour faciliter le développement d'applications réparties tolérant les fautes. Garf a été réalisé en Smalltalk et se base sur Isis pour les communications de groupe. Les applications sont structurées en objets qui peuvent être répliqués de manière transparente à l'aide d'une classe d'`encapsulateur` qui réalise la gestion de groupe et d'une classe `messagers` pour la transmission des messages. Garf implante plusieurs stratégies de réplication (active, passive, semi-active et coordinateur-cohorte).

2.2.6.3 Tolérance aux fautes et CORBA

Les systèmes d'objets répartis basés sur des standards tels que CORBA de l'OMG fournissent aux applications des fonctionnalités comme la transparence de localisation, l'interopérabilité, la portabilité. Cependant, jusqu'à très récemment, la tolérance aux fautes n'était pas prise en compte dans CORBA. Depuis avril 2000, l'OMG propose une spécification de la tolérance aux fautes [143]. La notion de groupe d'objets est introduite. Les objets peuvent être répliqués au sein d'un groupe auquel est rattaché des propriétés de tolérance aux faute telles que la stratégie de réplication (quatre stratégies sont proposées de la réplication purement active à la réplication passive à base de copie primaire) ou les nombres minimum et maximum de copies. Les propriétés sont associées lors de la création d'un groupe mais il est également possible de les changer ensuite dynamiquement. Le groupe permet de rendre transparent la réplication. La stratégie et les membres d'un groupe sont masqués au client invoquant des méthodes sur le groupe d'objets répliqués. Pour des raisons d'extensibilité, l'OMG propose également la notion de domaine, chaque domaine ayant un gestionnaire de réplication spécifique.

Les spécifications finales de l'OMG étant très récentes, il n'y a pas encore d'implantation de la tolérance aux fautes suivant la norme. Cependant, plusieurs projets abordent la fiabilité dans CORBA.

On distingue trois approches d'implantation :

— L'approche par *intégration* vise à porter un ORB au dessus d'un sous-système de communications fiables. Par exemple, **Orbix+Isis** [106] fut le premier système commercial qui supportait la création d'application CORBA tolérant les fautes. **Electra** [124] suit également cette approche. Electra tolère les partitionnements de réseaux et les fautes transitoires (pour résister par exemple à un redémarrage de machines). Il repose sur la notion de groupe d'objets répartis sur plusieurs machines. Electra a une approche dynamique, le degré de réplication peut être augmenté en cours d'exécution pendant que des clients adressent des requêtes aux objets. Il existe plusieurs implantations d'Electra reposant sur les sous-systèmes de communication Horus, Ensemble et Isis. Si elle peut être performante cette méthode oblige généralement à modifier l'interface de l'ORB et est donc moins portable.

— L'approche par *interception* capture les appels système des objets client au niveau des couches de communication et associe à ces appels des invocation à un sous-système de diffusion fiable. Cette méthode est, en général, indépendante de l'ORB. **Eternal** [138] adopte cette approche en interceptant les messages IIOP pour permettre la réplication d'objets CORBA. Ethernal utilise le système de communication de groupe **Totem** [137] pour implanter la réplication.
— L'approche par *service* définit la tolérance aux fautes comme un service CORBA. Ainsi, **DOORS** [142] propose un service de tolérance aux fautes dans CORBA. Ce projet se concentre sur des aspects de flexibilité en laissant la possibilité au développeur d'application de choisir ses stratégies de détection et de recouvrement. **FTS-ORB** [171] fournit également un service de tolérance aux fautes concentré sur des techniques de points de reprise et de journalisation. Cependant la détection et la gestion des fautes est laissée au niveau applicatif. Enfin **Nix** développé au laboratoire de système de l'EPFL est la suite du projet Phoenix. Nix propose un service de gestion de groupe (OGS : Object Group Service)[78] enrichi de fonctions de tolérance aux fautes.

2.3 Le système Star

Star [167, 165, 166, 168, 169] est une plate-forme logicielle pour l'exécution d'applications réparties fiables que nous avons développé au LIP6. Star est conçue au-dessus du système d'exploitation Unix et ne nécessite aucun support matériel spécifique. Le but est de démontrer qu'une approche purement logicielle pour le traitement des fautes peut être réalisée de manière efficace. Nous utilisons une technique de points de reprise non coordonnés pour éviter les latences lors de la réalisation des points de reprise. La plate-forme implante plusieurs stratégies de journalisation. Elle est paramètrable de façon à pouvoir s'adapter facilement aux besoins des applications en termes de fiabilité et de performance.

2.3.1 Environnement et architecture

Nous considérons des applications s'exécutant dans un environnement réparti. Ces applications sont constituées de composants disséminés sur un ensemble de sites. Ces composants sont soit des entités d'exécution (processus) soit des fichiers. Les processus communiquent uniquement par envoi de messages.

Les processus sont supposés déterministes par morceaux (*piecewise determinictic*) où l'exécution est divisée en séquences d'intervalles déterministes. Chaque intervalle débute par un événement non déterministe. Dans Star, nous avons supposé que la réception d'un message était la seule source d'indéterminisme. Cette classe de processus intègre un grand nombre d'applications mais exclut les programmes dont les opérations dépendent du site où ils s'exécutent. C'est notamment le cas si un processus interroge l'heure locale.

Star a été développée au-dessus d'Unix (SunOS et FreeBSD) sur un réseau de stations de travail (Sparc et PC). Nous traitons les fautes franches de sites et les ruptures de lignes sur le réseau (que nous assimilons à des fautes de sites). Un site en panne se matérialise par son inaccessibilité via le réseau.

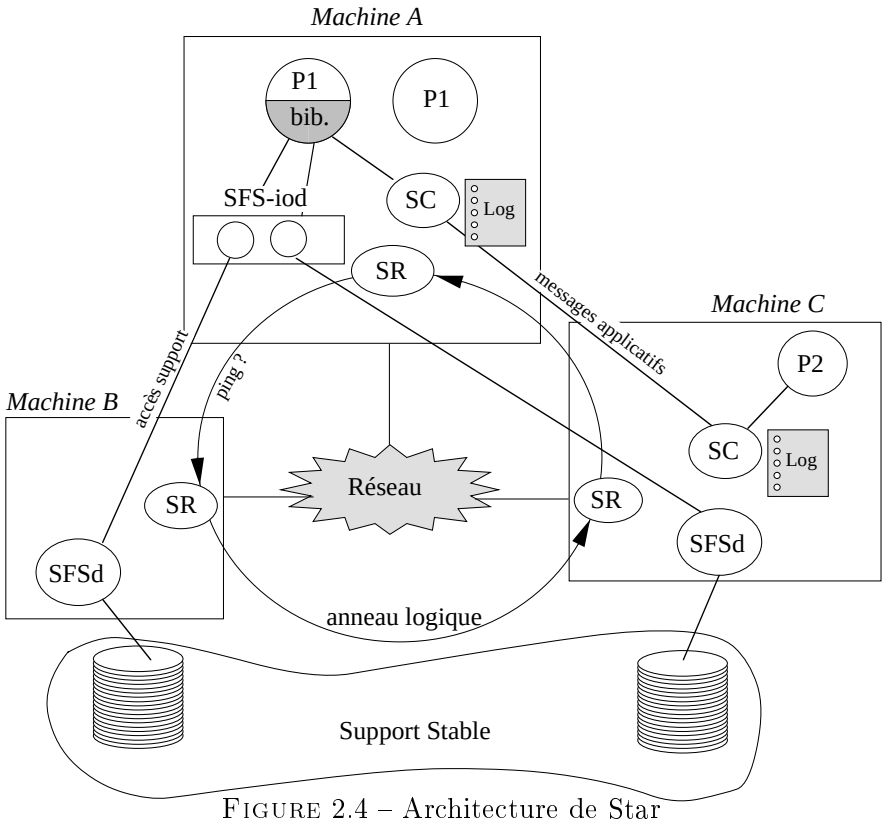

FIGURE 2.4 – Architecture de Star

Les fautes des machines suivent le modèle silence sur défaillance [150]. Nous considérons une panne comme un événement relativement rare. Clark et McMillin [58] ont estimé à 2.7 jours le taux de pannes moyen sur un réseau local. Cette hypothèse a conditionné certains de nos choix. Nous avons, en effet, visé à réduire le surcoût en fonctionnement normal par rapport au coût de la reprise sur défaillance.

Nous ne disposons d'aucun dispositif matériel de détection des fautes. La détection doit donc être logicielle : une panne est détectée si l'accès à une machine via un protocole de communication fiable échoue.

Les fonctionnalités de la plate-forme sont réalisées sur chaque site par un ensemble de serveurs coopérants et une bibliothèque (figure 2.4). Nous avons distingué quatre serveurs :

— *le serveur de communication* (SC) est responsable de l'acheminement des messages. Il implante notamment les stratégies de journalisation,
— *le serveur de reprise* (SR) détecte les défaillances de sites et gère le recouvrement de fautes,
— *le serveur de fichiers* ($SFSd$) gère l'accès aux fichiers de l'application et implante le support stable,
— *le mandataire de serveur de fichiers* (SFS-iod) gère sur les sites clients l'accès au support stable.

L'interface de Star est composée de commandes utilisateur (démarrage, arrêt, visualisation, ...) et de fonctions de programmation regroupées dans la bibliothèque Star. Chaque

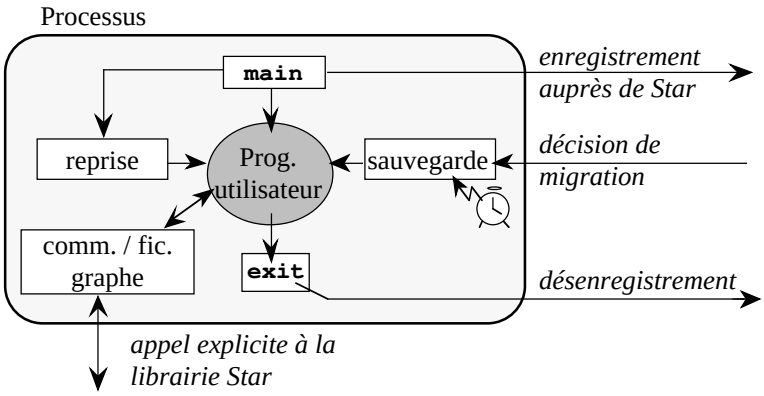

FIGURE 2.5 – Structure d'un exécutable

programme doit avoir une édition de liens avec cette bibliothèque. Les fonctions disponibles sont les suivantes (figure 2.5) :

- *main et exit* : au démarrage la fonction main s'enregistre auprès des serveurs Star. D'une manière similaire, la fonction exit avertit Star de la fin d'un processus.
- *Point de reprise et restauration* : le mécanisme de point de reprise est appelé, soit périodiquement (par défaut, avec une période de 3 minutes), soit explicitement dans le code source du programme (fonction `s_checkpoint`) ou bien par le serveur GatoStar pour initier une migration (voir 4.4). Quand un processus est repris, la fonction `restore` est automatiquement appelée à partir du programme principal (`main`).
- *Accès aux fichiers* : Star fournit une interface d'accès de type Unix aux fichiers tout en assurant la résistance aux fautes.
- *Communications* : Star fournit une interface de communication fiable de type RPC intégrant le mécanisme de journalisation.
- *Graphe d'exécution* : ces fonctions permettent de modifier dynamiquement le graphe d'exécution (ajout ou retrait de processus).

2.3.2 Détection des fautes

Pour réagir rapidement aux pannes de sites, nous adoptons une stratégie de surveillance. Le trafic normal est aussi utilisé pour recueillir des informations sur les sites.

On vise à limiter, pour chaque site, le nombre de machines distantes à surveiller. Nous définissons donc un anneau logique de détection où chaque site doit uniquement surveiller son successeur immédiat sur l'anneau.

Cette technique présente les deux avantages suivants : le trafic de contrôle supplémentaire reste relativement faible d'autant plus que le trafic normal peut être utilisé, la gestion locale est simplifiée par la surveillance d'un seul site distant.

En revanche, cette technique pose des problèmes de reconfiguration de l'anneau lors de pannes. Pour accroître la fiabilité de l'anneau, chacun de ses composants a une vision globale et connaît donc l'ensemble de ses successeurs. De cette manière, la reconfiguration de l'anneau est facilitée et celui-ci peut tolérer un nombre maximum de défaillances des sites qui le composent.

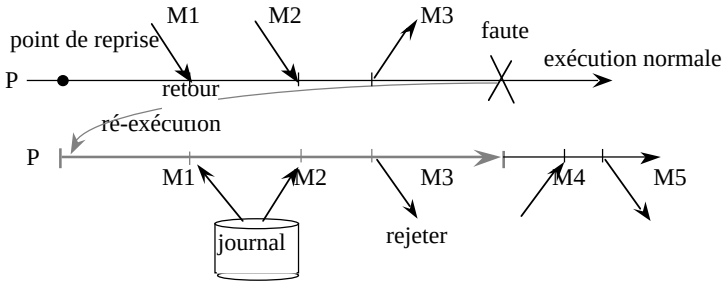

FIGURE 2.6 – Techniques de journalisation des messages

Mais ce choix impose un protocole pour maintenir la connaissance globale au niveau de chaque site. Cette connaissance est mise à jour lors du retrait (lors d'une panne) ou de l'insertion d'un site. Selon nos hypothèses ces deux événements sont relativement rares, et le coût du protocole de maintien n'est pas un point critique.

2.3.3 Recouvrement des fautes

En cas de défaillance d'une machine, les processus situés sur les sites fautifs sont relancés sur des sites valides en utilisant, dans le cadre de GatoStar, les algorithmes de placement décrits section 4.4.6. Les messages sont alors "rejoués" localement lors de la ré-exécution du processus jusqu'à l'instant de la panne. Ainsi, un processus repris se ré-exécute indépendamment des autres processus à partir des données du journal. Ce principe est mis œuvre par des techniques de journalisation de messages et de détection des ré-émissions (figure 2.6) :

— Chaque processus sauvegarde dans un journal l'ensemble des messages reçus depuis le dernier point de reprise. En cas de défaillance de son site d'exécution, le processus repris consomme directement les messages du journal.

— Un processus repris va réémettre ses messages (les processus sont supposés déterministes). Pour éviter de perturber l'exécution des processus valides, un mécanisme de détection des messages ré-émis, basé sur des estampilles, permet de les supprimer.

Ainsi dans la figure 2.6, lors de la ré-exécution du processus P, Star lui fournit à partir du journal, les messages $M1$ et $M2$ et le message $M3$ est supprimé.

Cette approche présente les avantages suivants :

— *Faible coût des points de reprise* : les processus créent leurs points de reprise sans synchronisation partielle ou globale.

— *Reprise rapide* : la reprise est limitée aux seuls processus fautifs. De plus, lors des ré-exécutions les communications sont simulées localement, diminuant ainsi le trafic sur le réseau.

— *Un seul point de reprise par processus* : c'est la conséquence directe de l'absence d'effet domino. De cette manière, l'espace de stockage associé à chaque processus est réduit.

— *Stratégie de pose de point de reprise libre* : le concepteur de l'application peut placer ses points de reprise où il le souhaite permettant de faire des optimisations en évitant de sauvegarder, par exemple, des variables intermédiaires.

L'inconvénient majeur résulte de la sauvegarde de tous les messages sur support stable

qui entraîne un surcoût important des communications et peut nécessiter des ressources de stockage importantes.

De façon à réduire le coût de ces sauvegardes, Star inclut une journalisation optimiste (voir 2.2.4.5) où les messages sont sauvegardés de manière asynchrone sur le support stable. Les messages sont gardés dans la mémoire volatile des émetteurs. Périodiquement, les sites émetteurs stockent les messages sur support stable. En cas de fautes, les messages destinés au processus repris sont retrouvés soit sur support stable soit dans la mémoire volatile des émetteurs. Si les émetteurs tombent également en panne avant d'effectuer leurs sauvegardes, l'hypothèse déterministe implique que les émetteurs repris reproduiront la ré-émission des messages manquants.

Le choix de la stratégie de journalisation (pessimiste ou optimiste) est laissé à l'appréciation de l'utilisateur lorsqu'il lance son application.

2.3.4 SFS : Support de fichiers répliqués

Nous présentons notre implémentation du support stable, SFS (Star File System).

Les données du support stable sont stockées dans des fichiers répliqués sur plusieurs sites. Le degré de réplication est spécifié par l'utilisateur au lancement de son application. Pour un degré de réplication de N, N-1 fautes simultanées sont tolérées. La cohérence, lors des mises à jour des réplicats, est garantie par un protocole de diffusion fiable à deux phases. En cas de fautes, le degré initial de réplication est maintenu dans la mesure des ressources restantes (machines et disques). La conservation du degré de réplication initiale permet théoriquement de résister à un nombre arbitraire de fautes. Le choix des sites de stockage des copies est fonction des indications contenues dans un fichier de configuration de Star.

L'implémentation de SFS doit être la plus efficace possible car les fichiers répliqués sont utilisés pour le stockage des points de reprise et des journaux de messages. Les performances de Star dépendent directement de celles des fichiers répliqués. SFS a donc été développé et optimisé en fonction des caractéristiques d'un support stable à savoir :

- Contrairement aux systèmes de fichiers standards, les mises à jours sont majoritaires par rapport aux lectures (les lectures n'intervenant que lors des phases de recouvrement de fautes),
- Les accès en écriture se font souvent par rafales, lors de l'écriture des points de reprise ou lorsque les journaux en mémoire vive sont vidés.
- Dans le contexte de Star, il ne peut y avoir de mises à jour concurrentes d'un fichier répliqué.

Pour réduire la latence d'accès aux serveurs lors des mises à jour, nous profitons du parallélisme offert par le système d'exploitation. Tous les accès à un serveur de fichiers distant sont pris en charge par un représentant local du serveur : le mandataire du serveur de fichiers. Il y a un mandataire local par serveur de fichiers distant. Les mandataires étant dans des processus différents, ils peuvent en "parallèle" accéder à leur serveur respectif. Lorsqu'un processus client veut envoyer une requête à N serveurs, il place les arguments de la requête dans un segment de mémoire partagée et réveille les N mandataires. Les mandataires lisent alors la requête en parallèle et la transmettent à leur serveur. L'accès à la mémoire partagée est géré par un protocole lecteur/écrivain équitable.

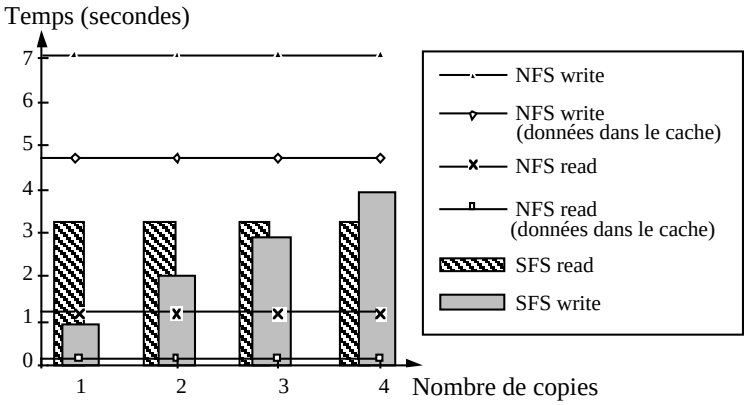

FIGURE 2.7 – Support Stable

Pour réduire les temps de transfert lors des écritures en rafale, nous avons adapté le protocole de diffusion deux phases en réduisant le nombre d'acquittements. Lors d'un envoi groupé de messages, un seul acquittement est envoyé pour tout le groupe. En cas de fautes, un mécanisme d'acquittement négatif permet de réclamer les messages manquants.

Avec ces deux optimisations, nous avons observé un gain important du temps de mise à jour des copies d'un fichier répliqué (voir la section suivante).

2.3.5 Evaluation de performances

Nous avons évalué le coût des différents mécanismes de notre plate-forme ainsi que le surcoût temporel induit pour un ensemble de d'applications réparties. Le but est d'évaluer l'impact des différents paramètres (degré de réplication, optimisation des sauvegardes, stratégies de journalisation) afin de permettre aux programmeurs d'application de choisir la configuration la mieux adaptée.

2.3.5.1 Le support stable

La figure 2.7 présente les performances de système de fichiers répliqués de Star (SFS) en comparaison avec NFS version 2. Nous présentons le temps pour lire et écrire un fichier de 1 mégaoctet en faisant varier pour SFS le nombre de copies distantes. NFS n'incluant pas la réplication, il n'y a dans ce cas qu'une seule copie. Les mesures ont été faites sur un ensemble de stations de travail de type Sun Sparc 5 et 10 possédant 32 Moctets de mémoire.

On remarque que les lectures sont plus lentes pour SFS. En fait, SFS n'est pas optimisé pour ce type d'opérations (contrairement à NFS) car le taux de lectures dans Star est beaucoup plus faible que le taux d'écritures. En revanche, les écritures sont nettement plus performantes même lorsque le nombre de copies augmente. Ces performances illustrent l'intérêt des optimisations faites dans SFS (parallélisation des requêtes et diminution du nombre d'acquittements lors des écritures en rafales).

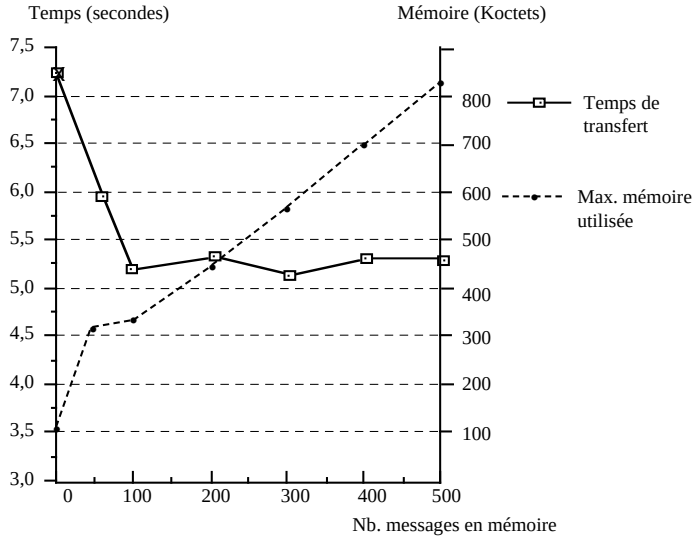

FIGURE 2.8 – Coût de la journalisation

2.3.5.2 Journalisation et points de reprise

La figure 2.8 montre le coût de la journalisation de messages en fonction du nombre de messages maintenu dans la mémoire des processus émetteurs avant une sauvegarde asynchrone sur support stable. Ainsi, en abscisse 500 signifie que 500 messages seront sauvés dans la mémoire de l'émetteur avant d'être écrits sur support stable, tandis que 0 implique une sauvegarde synchrone des messages (ce qui est équivalent à une approche pessimiste). Nous indiquons en ordonnée gauche le temps d'envoi d'un message (ce temps inclut les mécanismes de détection de retransmission, l'envoi sur le site distant et la mémorisation du message) et en ordonnée droite la quantité de mémoire requise.

Nous observons une réduction importante du temps d'émission lorsque moins de 100 messages sont mémorisés avant la sauvegarde. En dessous de ce nombre la fréquence de sauvegarde est trop élevée et la quantité de données trop faible pour obtenir des gains dus à l'asynchronisme des sauvegardes. On remarque également une stabilisation du temps d'envoi lorsque plus de 100 messages sont mémorisés. Ceci est principalement dû à l'augmentation de la charge des serveurs de communication de sites émetteur qui deviennent moins disponibles pour servir des requêtes. La courbe d'occupation mémoire illustre également cet effet. Le paramètre du nombre de messages à mémoriser est fixé lors du lancement d'une application. Ainsi, le concepteur d'une application peut ajuster ce nombre pour trouver un bon compromis entre asynchronisme et occupation mémoire.

Nous avons également mesuré le temps pour effectuer des points de reprise en fonction du degré de réplication du support stable. La figure 2.9 montre le temps moyen pour réaliser la sauvegarde d'un processus en fonction de la taille de sa pile et de ses données. Quatre configurations de support stable sont évaluées. Les mesures ont été faites sur des Sparc IPC ayant une puissance processeur modeste de 12 MIPS VAX. Malgré cette configuration, on observe une faible latence de sauvegarde due principalement aux sauvegardes asynchrones et

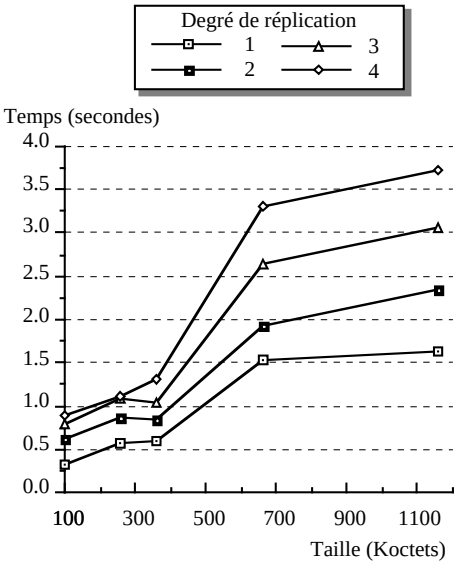

FIGURE 2.9 – Coût des sauvegardes

incrémentales.

2.3.5.3 Applications

Nous avons choisi trois applications de calcul intensif ayant des exigences différentes en mémoire et en communications. Le traitement a été réparti sur quatre processus s'exécutant sur quatre machines différentes. De plus, un processus supplémentaire ne participe par directement au traitement mais coordonne les processus de calcul. En tout, l'application est répartie sur cinq sites. Nous considérons les trois applications suivantes :

1. L'application *Gauss* réalise une élimination Gaussienne avec pivot partiel sur une matrice carrée de 1024 lignes. La matrice est distribuée entre différents processus qui se transmettent le pivot. A chaque itération de la réduction, le processus qui possède le pivot envoie la colonne du pivot à tous les autres processus.
2. L'application *matmul* multiplie deux matrices carrées de taille 1024. Le calcul est réparti sur plusieurs processus. Les seules communications nécessaires sont celles reportant la solution finale.
3. L'application *fft* calcule la Transformée de Fourier Rapide de 32768 points de données. Le problème est réparti en associant à chaque processus un intervalle de points. Comme pour l'application matmul, les seules communications nécessaires sont les synchronisations initiales et finales.

Les mesures suivantes ont été faites sur un ensemble de Sparc Station 1 possédant 24 Moctets de mémoires et connectées par un réseau Ethernet à 10Mb/s. L'environnement n'a pas été modifié (tous les démons standards de SunOs s'exécutaient).

La table 2.4 présente les besoins de chaque application en temps d'exécution, mémoire et communication. Les mesures du temps d'exécution sont faites sans gestion des fautes (i.e., sans point de reprise et journalisation des messages), elles serviront de base pour calculer le surcoût engendré par les différentes techniques de point de reprise et par la journalisation.

TABLE 2.4 – Besoins des applications

Applications	Temps (sec.)	Mémoire (Koctets)	Communication par processus(Koctets)
Gauss	344	1704	2700
matmul	723	2688	0,06
fft	1177	1200	0,06

Les applications *Gauss* et *matmul* exigent une quantité importante de données sollicitant particulièrement les mécanismes de point reprise et de restitution d'état. De plus, *Gauss* nécessite de nombreuses communications testant l'efficacité de la journalisation. Enfin, *fft* exige de plus long temps de calcul et une quantité moyenne de données.

TABLE 2.5 – Evaluation pour des applications parallèles

Appli.	Bloquant		Non-bloquant		Incrémentale	
	Temps (sec.)	%	Temps (sec.)	%	Temps (sec.)	%
Gauss	567	64.82	505	46.80	457	32.85
matmul	844	16.79	768	6.34	748	3.57
fft	1244	5.75	1228	4.36	1194	1.50

Le table 2.5, indique les surcoûts engendrés par la pose régulière de points de reprise avec une période de deux minutes. Le support stable a un degré de réplication égale à deux. Nous indiquons les temps obtenus sans optimisation des sauvegarde (colonne "Bloquant"), avec la version non bloquante, puis avec la version incrémentale. Malgré les optimisations, nous observons un surcoût important pour l'application *Gauss*. En fait, cette application n'est pas une bonne candidate aux stratégies de journalisation, principalement à cause de son fort taux de communications. Le coût lié à la sauvegarde des messages représente près de la moitié du surcoût global.

On remarque également, que la sauvegarde incrémentale réduit fortement les surcoûts des trois applications. On obtient une réduction du temps de réponse entre 24 et 63 %. Cette différence est en partie due aux taux de communication. De plus, les applications peuvent être divisées en deux groupes : celles avec une forte localité où l'espace est relativement peu modifié entre deux sauvegardes (c'est le cas de *matmul* et *fft*), et celles dont l'espace est en grande partie modifiée (c'est le cas de *Gauss*).

2.4 Conclusions

Ce chapitre a présenté une synthèse des principales techniques de tolérance aux fautes en environnement réparti. Nous avons plus particulièrement détaillé et comparé les algorithmes

à base de points de reprise. Nous avons notamment souligné qu'il est difficile de trouver un consensus autour d'une technique particulière car un grand nombre de paramètres sont à prendre en compte. En effet, les méthodes de tolérance aux fautes diffèrent quant à la nature des fautes tolérées, aux contraintes temporelles et aux exigences de fiabilité des applications.

Nous avons ensuite présenté notre plate-forme logicielle Star. Star offre un support pour développer des applications réparties. Elle masque aux processus d'applications les défaillances de sites en les reprenant automatiquement sur des site valides. Par soucis de performance, nous avons cherché à limiter le surcoût de fonctionnement des politiques de traitement des fautes. Nous avons donc adopté une stratégie à base de points de reprise indépendants, connue pour son faible surcoût. Nous avons également intégré plusieurs optimisations comme la sauvegarde asynchrone et incrémentale des processus. Une évaluation de performance a montré la grande importance de ces optimisations sur le temps de réponse des applications. Une attention particulière a également été portée sur la réalisation d'un support stable optimisé. Star offre plusieurs stratégies de journalisation de messages. La plate-forme est paramètrable : lors du lancement d'une application le programmeur peut préciser le degré de réplication de l'application, l'algorithme de journalisation et la fréquence de sauvegarde.

Cette étude a montré qu'une plate-forme purement logicielle est une alternative aux approches basées sur des composants matériels dédiés. Nous avons également mis en évidence les surcoûts que pouvaient entraîner les techniques de tolérance aux fautes qui, malgré des optimisations, ne peuvent être absorbés que par une gestion globale des ressources du réseau. De plus, la définition statique (au démarrage de l'application) des stratégies de tolérance aux fautes est un frein à une gestion performante : l'environnement étant très variable ainsi que le comportement des applications (en termes d'utilisation des ressources processeur et mémoire), une "bonne" configuration initiale peut s'avérer être, en cours d'exécution, un mauvais choix.

Chapitre 3

Tolérance aux fautes adaptative : la plate-forme multi-agent DARX

Sommaire

3.1 Introduction

Les architectures multi-agents suscitent un intérêt grandissant en tant qu'outil facilitant la conception, le développement et le déploiement d'applications réparties. Les agents sont particulièrement indiqués quand le problème est dynamique et réparti physiquement, par exemple : l'aide à la gestion de crises, le contrôle de processus, les ateliers de production

flexibles, la robotique collective ou la simulation d'éco-systèmes. Ainsi plusieurs plates-formes (ou *framework*) comme Voyager [93] ou Mole [32] offrent des moyens simple pour concevoir et exécuter des agents sur des machines distantes.

Les systèmes multi-agent commençant à être répartis sur un nombre important de sites, la tolérance aux fautes devient une réelle préoccupation de ces plates-formes. Or, si l'algorithmique répartie a produit un certain nombre de protocoles de tolérance aux fautes à base de réplication passive ou active, un premier constat est que de tels protocoles ont encore été peu appliqués aux systèmes multi-agent.

Une autre constatation est que de tels protocoles sont souvent appliqués de manière statique aux applications réparties. C'est-à-dire que c'est à la charge du concepteur de l'application de décider quels serveurs sont particulièrement critiques et donc à répliquer. Une telle assignation statique est raisonnable pour des applications stables où les rôles des serveurs sont clairement identifiés. Mais dans des applications très dynamiques (par exemple, la gestion de crises), la criticité de serveurs et des assistants logiciels peut varier énormément au cours du processus de résolution. Les protocoles de réplication étant assez coûteux (duplication de messages et synchronisation pour garantir un ordre total), il n'est pas raisonnable de les appliquer de manière systématique et statique à tous les composants logiciels.

Nous proposons donc des stratégies de réplication adaptables pour le système à agents DIMA [95]. DIMA est une plate-forme multi-agent développé par Zahia Guessoum dans le groupe "Framework" du thème OASIS du LIP6 dirigé par Jean-Pierre Briot.

L'introduction de la redondance au sein de l'application multi-agents donne alors la possibilité de :

— tolérer le dysfonctionnement d'un composant (ou éventuellement son fonctionnement byzantin) en basculant, le cas échéant, sur une copie de secours,
— maintenir le système en état de fonctionnement en cas de rupture d'une liaison point à point, entraînant l'impossibilité (momentanée ou définitive) de communiquer avec un agent.
— améliorer sensiblement les performances et la réactivité du système. En effet, dans le cadre d'une application client/serveur, par exemple, la possibilité offerte aux clients de s'adresser à plusieurs instances d'un même service permet de réduire la charge de ce dernier et, dans certains cas, réduire le trafic sur le réseau.

Motivés par ce constat, nous avons doté DIMA, de services et APIs autorisant la répartition et la réplication des agents. DarX permet non seulement de rendre fiable les agents de DIMA mais également d'adapter dynamiquement en cours d'exécution la gestion des fautes en fonction de l'environnement (latence d'accès au sites, charge des machines) et du niveau de criticité des agents.

Ce travail a initialement fait l'objet du stage de DEA de Jacob Zimmermann [209]. Ce travail se poursuit par la thèse d'Olivier Marin en collaboration avec le Laboratoire d'Informatique du Havre dirigé par Alain Cardon.

Dans la section 3.2, nous nous positionnons en présentant les principaux systèmes multi-agents gérant les fautes. La section 3.4 constitue une présentation générale de la plate-forme DarX et des choix techniques effectués. La section 3.5 en présente la structure et le fonctionnement. La section 3.6 évalue les performances de DarX.

3.2 Tolérance aux fautes dans les plates-formes multi-agents

Nous présentons trois exemples de mécanismes typiques offerts par des systèmes d'agents, permettant de supporter des fautes. Ces mécanismes concernent essentiellement le sauvegarde de contexte et la migration.

3.2.1 Voyager

Voyager [93] est une plate-forme multi-agent distribuée par la société ObjectSpace. Elle permet le développement d'agents mobiles et autonomes et propose à cette fin des primitives adaptées.

Voyager est implantée en Java et utilise le système d'invocation distante RMI. Ses APIs apportent par contre un support de communication avancé offrant en particulier un service de diffusion et de migration.

Voyager repose sur le principe de l'agrégation dynamique. Chaque agent présente à l'environnement plusieurs interfaces (au sens de Java), appelées "facettes". L'ensemble des facettes d'un agent peut évoluer au cours de l'exécution car elles peuvent se voir ajoutées ou retirées dynamiquement.

Le service de migration de Voyager repose sur la facette IMobility, qui offre principalement la méthode moveTo(). Son invocation déclenche les opérations suivantes :

- La réception de nouveaux messages est bloquée et le système attend la terminaison du traitement des messages en cours.
- L'agent est alors sérialisé et envoyé sous cette forme sur le site de destination. La nouvelle adresse de l'agent est stockée sur le site de départ.
- L'exécution de l'agent reprend sur le nouveau site.
- Si un message à l'intention de cet agent est envoyé sur l'ancien site, ce dernier adresse une exception à l'émetteur en lui transmettant la nouvelle adresse de l'agent.

Voyager offre un service de diffusion fiable prenant en compte les critères de performances et d'extensibilité. Au lieu d'adresser séquentiellement le même message à tous les agents du groupe concerné, il utilise une inondation.

Les agents dans Voyager sont organisés en "espaces", eux-mêmes étant découpés en sous-espaces. Pour diffuser un message dans son espace, un agent commence par l'envoyer à tous les agents de son sous-espace, puis à tous les sous-espaces voisins. Lors de la réception d'un message encore non traité, chaque sous-espace le transmet à tous les agents qu'il contient, puis à tous les sous-espaces limitrophes.

Cette vague se propage donc parallèlement dans tous les sous-espaces de l'espace concerné et l'opération de diffusion engendre ainsi un coût logarithmique plutôt que linéaire par rapport à la taille de l'espace, critère décisif lorsque la taille des espaces devient importante.

En utilisant les possibilités de migration de Voyager et son service de diffusion, on peut facilement implémenter des stratégies de réplication.

3.2.2 Mole

Mole est un système d'agents mobiles répartis développé à l'université de Stuttgart [32]. Il dispose d'une infrastructure de mobilité et de communication très étendue.

Comme dans de nombreux systèmes d'agents mobiles, la notion centrale dans Mole est celle de "place". Une place est un emplacement géographique logique, entièrement situé sur un site physique unique (les places ne peuvent pas être réparties), en revanche, un même site peut héberger plusieurs places. Chaque place constitue un environnement d'exécution pour les agents du système et contient un certain nombre d'agents de service, fournissant un support pour les agents d'application.

Chaque agent peut communiquer avec les agents de la même place ou avec ceux d'une place différente. Comme dans Voyager, la communication entre places passe par l'utilisation de Java-RMI. Mole supporte les places "connectées", accessibles en permanence et les places "associées", pouvant se trouver hors-ligne (par exemple un ordinateur portable peut fonctionner en tant que place associée dans Mole).

Le système offre la méthode migrateTo(), permettant de transférer un agent d'une place à une autre (en pratique, on utilise le mécanisme de sérialisation de Java). La migration dans Mole vise plusieurs objectifs. Il peut s'agir d'installer un agent sur une place associée afin qu'il y fonctionne lorsque la communication sera coupée. Entre places connectées, la migration peut avoir pour objectif l'équilibrage de charge, mais elle représente également une approche de la réflexivité. Un agent ayant besoin d'un service non disponible là où il se trouve peut migrer sur une place offrant ce service ; il peut également migrer sur une place offrant les mêmes services, mais au moyen d'une implémentation différente.

Cette faculté, présente dans Mole, d'adapter et modifier l'environnement d'un composant au moyen de la migration a déjà été expérimentée par les concepteurs du système Apertos [205].

3.2.3 Knowbot

Knowbot [104] ne constitue pas à proprement parler une plate-forme d'agents, il s'agit d'un environnement d'exécution réparti permettant d'implémenter facilement un système d'agents. La principale orientation de Knowbot reste donc l'aspect réparti, ce qui permet au système d'offrir des services particulièrement avancés dans ce domaine. Knowbot est implémenté en Python, langage de programmation orienté objet proche de Java ; le système repose entièrement sur les facilités de sérialisation et de communication offertes par le langage.

Les deux opérations centrales dans Knowbot concernent la migration et le clonage. Le clonage est similaire à la migration, excepté le fait que l'original n'est pas détruit, mais les deux exemplaires continuent de s'exécuter en parallèle.

L'environnement d'exécution des programmes Knowbot, appelé le *Knowbot Operating System* (KOS) joue un rôle similaire à celui des places de Mole. La migration et le clonage ont toujours lieu entre KOS. Chaque KOS peut offrir aux programmes hébergés ses propres APIs, encore une fois, la migration constitue donc le moyen d'accéder à certains types de services et spécialiser ces derniers.

La communication entre programmes se fait au moyen des "connecteurs", des méta-objets encapsulant l'accès à un ORB spécifique au système, mais dont le langage de spécification d'interfaces s'inspire de l'IDL CORBA . Pour communiquer avec un serveur, un client doit donc avant tout instancier un connecteur sur ce serveur. Cela ouvre la possibilité de redéfinir ces connecteurs en vue de cacher la réplication d'un programme.

Grâce à la présence en standard des opérations de migration et de clonage dans Knowbot et l'accès à la communication sous forme d'"objets de première classe" (les connecteurs), ce système offre directement toute l'infrastructure nécessaire à la mise en place de stratégies de réplication.

3.3 Adaptation et tolérance aux fautes

La réflexivité est une approche prometteuse pour concevoir des systèmes adaptables. Elle désigne la capacité d'un programme à posséder et éventuellement modifier une représentation de son propre état. Dans le cas de la programmation par objet, cela se traduit par la notion de méta-objets. Plusieurs systèmes exploitent cette propriété pour la tolérance aux fautes : le système MAUD [1] est basé sur un support d'exécution réflexif, GARF [94] utilise une forme de réflexivité fournie par le langage SmallTalk et Friends utilise OpenC++ pour ajouter des propriétés de tolérance aux fautes et de sécurité aux applications.

L'architecture réflexive permet de modifier dynamiquement l'infrastructure de communication du système et d'utiliser des stratégies de réplication de façon transparente autorise l'adaptabilité du système [183]. On peut distinguer plusieurs niveaux d'adaptabilité :

— *paramétrage dynamique d'une architecture pré-définie* : toute l'architecture du système est fixée à la compilation, seuls des paramètres sont modifiés au cours de l'exécution. Il s'agit donc d'une stratégie hybride "compile-time" / "run-time".

— *réplication automatique* : le système est à même d'évaluer la "criticité" d'un composant et de mettre en place sa réplication le cas échéant. En pratique, la notion de criticité reste délicate à définir. Dans certains cas, il s'agit d'un paramètre du système : il appartient alors au développeur de l'application de le fixer à la compilation ou d'implémenter sa propre méthode d'évaluation de criticité. La criticité peut être également déterminée "de fait" Dans l'application de gestion d'agendas répartis implémentée sous GARF [94], chaque site possède un réplicat du gestionnaire de rendez-vous et chaque utilisateur possède un réplicat de chaque objet gérant un rendez-vous particulier. La criticité de ces deux objets est ainsi liée au nombre de leurs accointances. Enfin, la criticité d'un composant peut être analysée au cours de l'exécution, en fonction de la fréquence de défaillances : plus un objet est défaillant et plus il est considéré comme étant critique.

— *adaptation au contexte* : dans ce cas, le choix des stratégies et du degré de réplication est automatique et s'effectue en fonction du contexte et du profil du comportement des composants concernés dans le temps. A notre connaissance, de tels systèmes n'ont pas été expérimentés en pratique à ce jour.

Pour illustrer un exemple d'adaptation, nous présentons maintenant l'architecture utilisée par la plate-forme Broadway [185]. Celle-ci repose sur un concept original pour évaluer le comportement général du système et le niveau de criticité des différents composants.

Broadway assure la tolérance aux fautes par réplication des composants, avec des communications gérées par méta-objets. L'adaptabilité du système provient d'une seconde forme de réflexivité : la "réification" des gestionnaires d'exceptions.

Dans ce système, chaque faute se matérialise par la levée d'une exception par le composant

fautif. Dans une architecture traditionnelle, cette exception serait rattrapée par l'appelant, c'est à dire l'objet ayant lancé la requête. Dans Broadway, en revanche, les exceptions sont transmises à des gestionnaires indépendants, ayant chacun à charge un ou plusieurs composants. Selon la configuration du système, on pourra ainsi avoir un seul gestionnaire d'exceptions centralisé, ou répartir la gestion d'exceptions selon une structure logique à grain plus ou moins petit, selon les besoins.

Dans l'implémentation de Broadway, la défaillance d'un agent sollicité se traduit par la levée de l'exception crash. Chaque site est équipé d'un détecteur de fautes qui génère les cas échéant cette exception "crash" et la transmet au gestionnaire d'exceptions de l'agent concerné. De plus, un gestionnaire d'exceptions peut s'abonner à un détecteur de fautes, afin d'être immédiatement informé de la défaillance d'un agent même si aucune requête n'est à ce moment adressée à ce dernier.

De cette manière, le gestionnaire d'exceptions se construit une vision globale du fonctionnement des agents : par exemple, il évalue leur taux d'erreur ou constate la mort complète et définitive d'un agent. Il devient dès lors possible de prendre des mesures afin, par exemple, de ramener le taux d'échec en dessous d'un seuil fixé. Cela revient en pratique à :

— répliquer l'agent défaillant (la version actuelle effectue toujours une réplication active),
— remplacer les protocoles de communication des autres agents en fonction de cette réplication.

Broadway propose une approche intéressante et prometteuse pour construire un système adaptatif. Toutefois, elle pose le problème de la fiabilité des gestionnaires d'exceptions. Dans l'implémentation actuelle, ces derniers peuvent être répliqués, mais il s'agit alors d'une réplication statique.

3.4 Présentation générale de DarX

DarX ("Dima Agent Replication eXtension") est un ensemble d'outils, permettant la création d'applications réparties et résistantes aux fautes. Bien que sa motivation originale le destine avant tout à une utilisation en tant que sous-couche de Dima, il présente en fait un champ d'applications plus large et peut également fonctionner seul, en tant que support général pour le développement d'applications réparties.

3.4.1 Objectifs

Un but du projet DarX est de déterminer des méthodes adaptatives de gestion des fautes choisissant dynamiquement l'algorithme approprié afin de trouver à un instant donné le bon compromis entre le coût et les performances.

Une première étape consiste à identifier les caractéristiques des techniques existantes pour choisir la méthode à appliquer en fonction des contraintes de l'application et de l'environnement.

Le choix de la technique de réplication est délicat. S'il est clair que dans les environnements à forte contrainte temporelle la réplication passive n'est pas adaptée, dans les autres cas plusieurs critères interviennent tel que le surcoût de traitement, les communications, les types de fautes que l'on souhaite tolérer.

Nous visons à proposer des modèles hybrides de réplication en fonction des coûts de communication. Par exemple, lorsqu'un des serveurs répliqués a des temps de réponse trop lent il peut être temporairement exclu d'un module de réplication actif et être mis à jour ultérieurement selon le principe d'une réplication passive. De même, la fréquence de mise à jour des serveurs répliqués passivement peut être modifiée en fonction des latences de communication.

3.4.2 Conception et architecture

Nous avons conçu DarX sous forme d'un environnement pour la programmation d'applications réparties communiquant par échange de messages. Le système supporte par défaut deux politiques de réplication : la réplication active et la réplication passive avec mise à jour périodique des copies secondaires. Le programmeur dispose également d'un framework pour implémenter facilement sa propre politique. La politique de réplication est spécifiée lors de la création d'un nouveau groupe (réplication active par défaut), puis il est possible de la modifier à tout moment, en cours d'exécution.

Chaque tâche peut se voir répliquée en un nombre (théoriquement) illimité d'exemplaires, selon différentes politiques. Basé sur une architecture réflexive, DarX prend en charge la gestion des communications entre groupes de réplicats (permettant l'envoi de messages synchrones et asynchrones) aussi bien que la communication intra-groupe et assure qu'aucun message ne sera reçu plus d'une fois, de même que la réception des messages dans le même ordre par tous les réplicats d'un groupe. L'élaboration fut guidée par plusieurs principes et décisions :

— La tolérance aux fautes repose sur la notion de groupe. Un groupe DarX de réplicats constitue une entité opaque et indivisible. Une entité extérieure communique toujours avec le groupe en tant que tel, elle ne peut s'adresser individuellement aux membres du groupe.
— Chaque groupe possède un membre maître, responsable du fonctionnement correct du groupe et représente également son interface de communication avec l'extérieur. En cas le défaillance du maître, un autre membre du groupe le remplace.
— Les décisions de créer de nouveaux réplicats, ainsi que la définition de la politique de réplication et de gestion des fautes (ou la possible modification de celles-ci) proviennent toujours "de l'extérieur". Le maître du groupe est chargé d'appliquer ces ordres au sein du groupe.

DarX ne contient directement aucune politique de gestion de fautes. Le programmeur peut en revanche mettre en place un gestionnaire de fautes spécifique, mettant en oeuvre une politique de tolérance aux fautes au sein du groupe. Une panne est assimilée à l'échec de communication, c'est à dire à une exception levée par le système d'exploitation sous-jacent lors d'une tentative d'envoi de message. La définition d'une politique de réplication peut néanmoins inclure un envoi périodique de messages "ping" destinés à vérifier le bon fonctionnement de chaque membre du groupe. Dans une première approche, nous considérons l'hypothèse d'un réseau synchrone. La définition de la valeur des temporisateurs est un paramètre interne de DarX.

3.5 Architecture de DarX

3.5.1 Structure générale

DarX se compose d'une API pour la programmation d'applications réparties avec réplication et d'un environnement d'exécution prenant en charge ces applications. Comme toute application répartie, une application DarX est composée de tâches communiquant par envoi de messages. Chaque tâche, implémentée sous la forme d'un objet `DarxTask`, est encapsulée dans une "coquille" matérialisée par un objet `TaskShell`, qui constitue une interface pour DarX permettant d'accéder à la tâche et implémente certaines fonctions vitales.

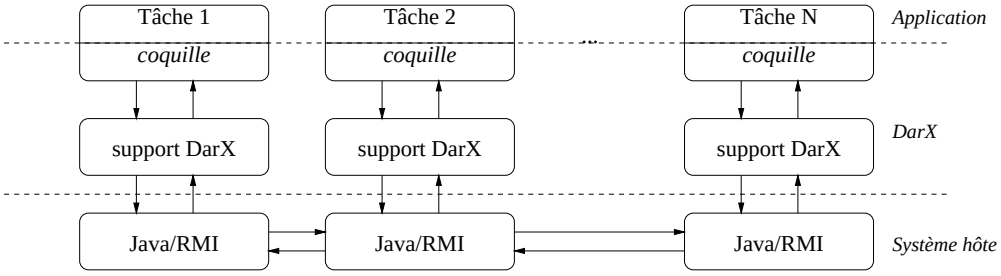

FIGURE 3.1 – Architecture de DarX

Comme le montre la figure 3.2, une tâche peut communiquer avec d'autres groupes par l'intermédiaire d'interfaces `RemoteTask`, qui représentent ses accointances. Chaque `RemoteTask` désigne en réalité une tâche particulière du groupe qu'il représente : il s'agit du maître du groupe (voir section 3.5.4). Les messages arrivant à l'intention d'une tâche sont reçus par l'objet `TaskShell` qui l'encapsule ; ce dernier les délivre alors à la tâche.

3.5.2 Implémentation

DarX est écrit entièrement en Java 1.1, de même que toutes les applications l'utilisant. Nous utilisons le système RMI [135] offert en standard par Java comme protocole de communication inter-tâches. Basé sur TCP, RMI nous garantit l'acheminement correct du message (ou informe l'application de l'échec) et un comportement FIFO sur les communications point à point, ce qui nous autorise de décharger DarX de la gestion de ces problèmes.

A partir du JDK 1.2 [134], il est possible de redéfinir dans RMI certains paramètres internes de TCP. Nous avons donc redimensionné les temporisateurs de TCP pour détecter plus rapidement les fautes.

3.5.3 Envoi et réception des messages

La transmission des messages pose deux problèmes. D'une part, il est nécessaire d'assurer un ordre d'acheminement correct : tous les membres du groupe de destination doivent recevoir les messages dans le même ordre.

Par ailleurs, si l'émetteur du message est également répliqué, les messages doivent être filtrés car chaque message ne doit être reçu qu'une fois. Nous décrivons les mécanismes utilisés

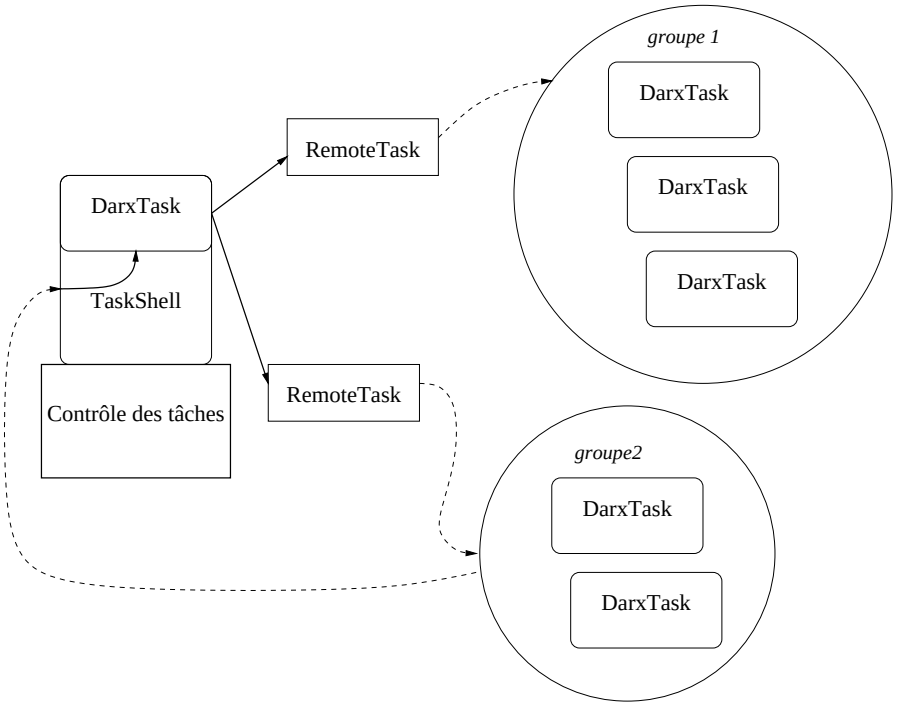

FIGURE 3.2 – Vue de l'application par une tâche

par DarX pour assurer ces services dans la section suivante.

Chaque message échangé entre groupes/tâches `DarX` est encapsulé dans un objet `DarxMessage`. Ce dernier contient non seulement le contenu du message (objet Java quelconque mais sérialisable), et de plus l'identité de l'émetteur et le numéro ordinal du message émis par cette tâche (figure 3.3). Ces deux informations seront utilisés pour le filtrage des messages.

Chaque tâche possède également deux méthodes dédiées à la réception des messages : une pour la réception de messages "synchrones" et une pour la réception des messages "asynchrones". En cas de réception d'un message synchrone, la méthode correspondante renvoie immédiatement une valeur de retour, contrairement au cas du message asynchrone. L'émetteur d'un message asynchrone continue immédiatement son exécution, tandis que l'émission d'un message synchrone est un appel bloquant jusqu'au retour de la réponse.

3.5.4 Acheminement des messages

La figure 3.4 schématise le flot des données relatif à l'acheminement d'un message dans un groupe.

Chaque `TaskShell` mémorise le numéro de série du dernier message en provenance de chaque émetteur. Si plusieurs réplicats d'une même tâche envoient le même message, le `TaskShell` éliminera les copies. Toutefois, dans le cas des messages synchrones, les valeurs de retour sont stockées dans un cache, ceci afin de renvoyer à l'émetteur du message la réponse attendue sans traiter le message une seconde fois.

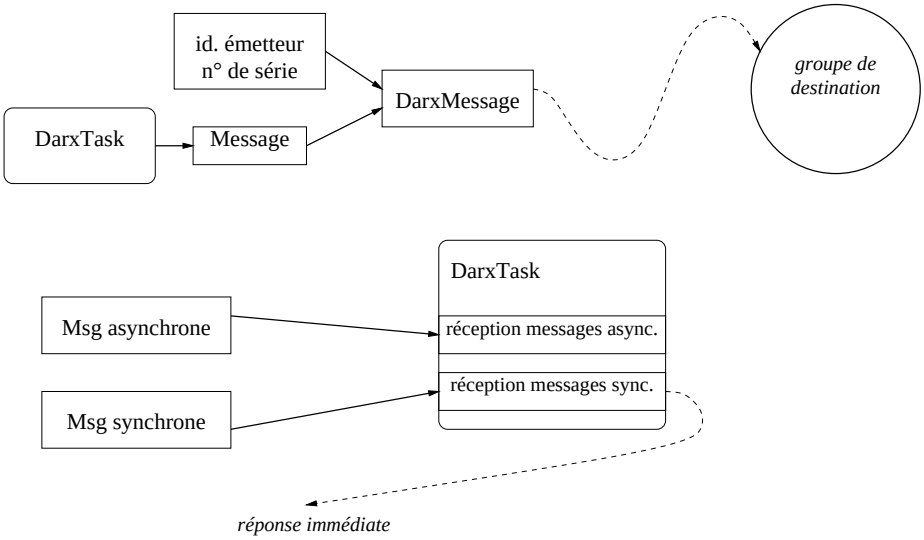

FIGURE 3.3 – Gestion des messages

Ce mécanisme peut être utilisé pour détecter des fautes byzantines. Il faudrait pour cela mémoriser également le contenu de chaque message. Deux messages provenant d'un même groupe, portant le même numéro mais de contenus différents seraient alors symptôme d'une faute byzantine.

Si le numéro du message est supérieur au dernier message reçu (c'est à dire s'il faut effectivement traiter le message), le contenu est transmis au maître puis diffusé au sein du groupe en fonction de la politique de réplication courante.

Cette structure nous permet d'éviter l'utilisation de protocoles lourds pour assurer une diffusion atomique dans le groupe. En effet, le maître du groupe est le seul récepteur des messages de l'extérieur. Par ailleurs, RMI garantit l'ordre FIFO sur les communications point-à-point ; par conséquent, les messages sont totalement sérialisés et reçus par tous les membres du groupe dans l'ordre d'émission.

3.5.5 Réplication des tâches et gestion des fautes

Chaque tâche DarX est implantée sous forme d'activité (ou thread) Java. La création d'un nouveau réplicat consiste donc à construire une image de l'état courant du thread, transférer celle-ci sur le site où le réplicat doit être créé puis y instancier un thread à partir de cette image. En Java, la sauvegarde et le rétablissement de l'état d'un thread sont assurés directement par le langage.

La politique de réplication, définie à l'échelle du groupe, est implémentée par un objet de classe `ReplicationStrategy`. Ce dernier remplit trois fonctions :

— Il est chargé d'arrêter et redémarrer le groupe. Dans une stratégie de réplication active, cela correspond à l'envoi effectif d'un ordre `suspend` ou `restart` à l'ensemble des réplicats. En revanche, dans une stratégie passive, ces opérations n'effectueront aucune action car toutes les copies secondaires sont toujours arrêtées.

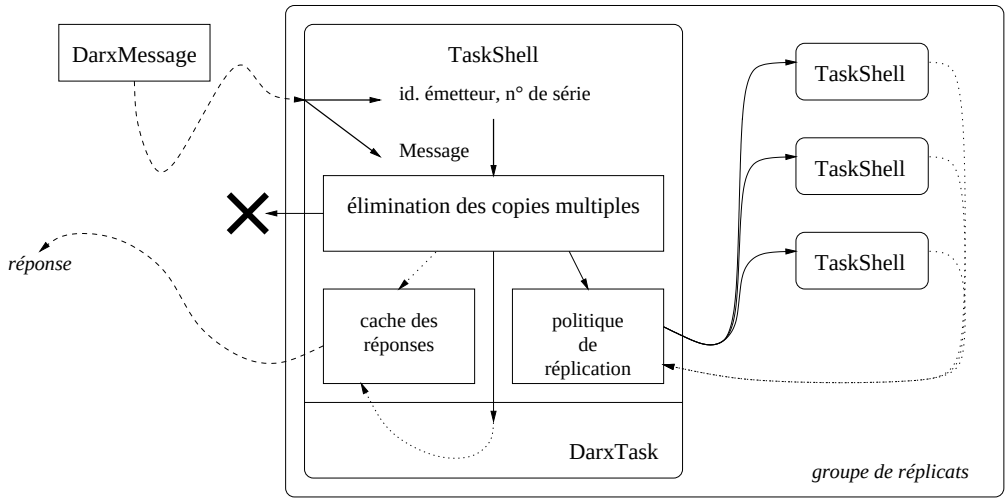

FIGURE 3.4 – Acheminement d'un message dans un groupe

— Il assure l'envoi d'un message aux réplicats. Lors de la réception d'un message, le maître du groupe invoque la `ReplicationStrategy` pour diffuser le message dans le groupe. La notion de "diffusion" est laissée à l'appréciation de l'implémenteur. Une stratégie active diffusera effectivement tous les messages, tandis qu'une stratégie passive n'effectuera que des mises à jour périodiques.
— Il exécute un code de traitement des fautes. Si, au cours d'une diffusion, la communication avec un réplicat s'avère impossible, ce dernier sera déclaré "en panne" et le gestionnaire d'erreurs sera sollicité. L'implémentation par défaut ne fait rien, aussi, le programmeur désirant mettre en place une politique de gestion des fautes devra l'ajouter par héritage Java à la `ReplicationStrategy` utilisée.

Si le maître du groupe tombe en panne, cette erreur sera constatée par une autre tâche désirant communiquer avec le groupe. Dans ce cas, un autre maître sera désigné et l'ancien maître fautif ne sera plus qu'un réplicat ordinaire. Sa panne sera alors gérée par le nouveau maître.

La stratégie de réplication peut être modifiée lors de l'exécution de l'application : il suffit pour cela d'arrêter le groupe (au sens de la stratégie en place), instancier une nouvelle `ReplicationStrategy` puis redémarrer le groupe en fonction de cette nouvelle politique. C'est donc une opération relativement coûteuse, mais néanmoins réalisable car de complexité linéaire en fonction de la taille du groupe, et on peut la supposer "peu courante".

3.5.6 Service de nommage

Afin de rendre la communication inter-groupe possible, un serveur de noms global est nécessaire. RMI n'offre pas cette fonction, c'est pourquoi elle est fournie par le serveur `NameStore` de DarX. Ce serveur joue un double rôle :

— Il permet aux tâches de trouver leurs accointances en fournissant une adresse RMI contre un nom de groupe.

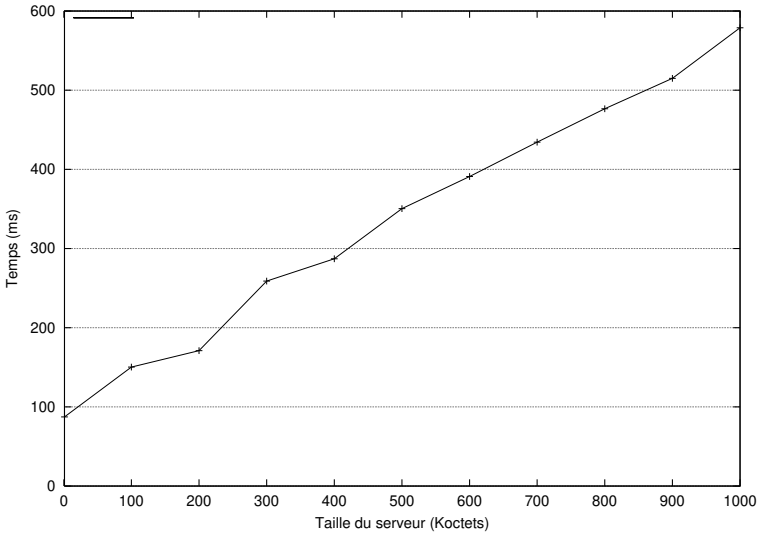

FIGURE 3.5 – Coût de réplication

— Il désigne un nouveau maître du groupe en cas de panne de celui-ci. Une tâche ayant constaté la panne du maître le signale au serveur de noms, lequel désignera un nouveau maître parmi les membres du groupe.

Le `NameStore` maintient un registre de tous les groupes et de tous les membres de chaque groupe. Chaque tâche possède un nom générique qui est de fait le nom du groupe. Ce nom est utilisé lors des communications et des requêtes.

3.6 Performances de DarX

Cette section présente une évaluation de performance des composants de base de DarX. L'étude a été faite avec le JDK 1.1.6 sur un ensemble de stations de travail Ultra-SparcII 333 MHk reliées par un Ethernet 100 MB/s.

Tout d'abord, nous avons mesuré le coût pour ajouter un nouveau réplicat en cours d'exécution. Dans ce cas, une nouvelle tâche DarX est créée sur un site distant et le maître du groupe transmet à la nouvelle tâche ses données locales. Ce mécanisme est très proche d'une migration.

La figure 3.5 indique le temps pour répliquer un serveur en fonction de sa taille. Pour un serveur avec 1 Megaoctet de données, le temps de créer et activer une copie distante est inférieur à 0.7 secondes.

Pour mettre en évidence l'efficacité de notre service de migration, nous avons comparé ses performances avec celles de Voyager [93] qui permet la migration d'agents. La figure 3.6 présente cette évaluation. Dans cette étude, un serveur est déplacé (soit par Voyager, soit par le mécanisme de réplication de DarX) entre deux PCs exécutant linux avec JDK 1.1.8. Nous remarquons, qu'en moyenne DarX est plus de deux fois plus rapide que Voyager.

Nous avons également mesuré la latence pour envoyer un message à un ensemble d'agents

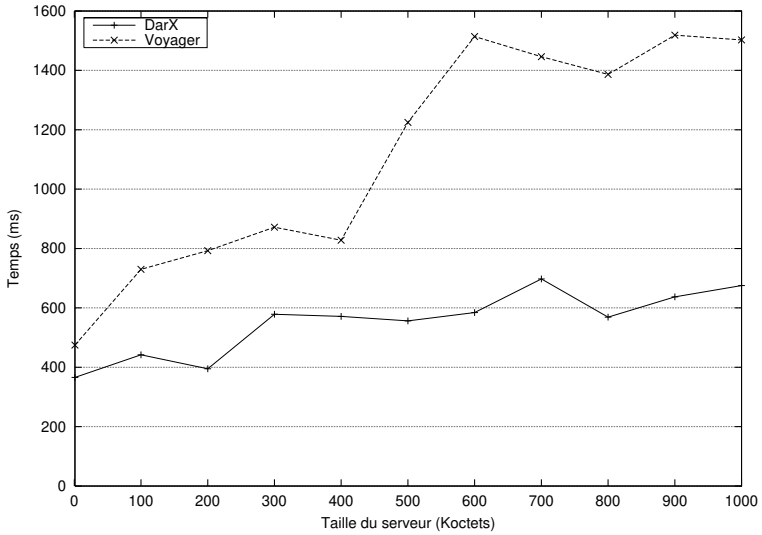

FIGURE 3.6 – Coût de la migration de serveur

répliqués. Nous avons évalué trois configurations en fonction du degré de réplication des agents. Dans la configuration DR-1, l'agent destinataire du message est local et non répliqué. Dans la configuration DR-3, l'agent a trois copies (une sur le site émetteur et deux sur deux sites distants). On remarque que le coût croît linéairement avec le taille du message, le coût pour deux réplicats n'est que 50% supérieur à celui pour un réplicat.

L'évaluation des performances des composants de base, montre que l'approche de DarX est prometteuse. L'objectif essentiel de DarX est l'adaptabilité de la gestion des fautes. La réalisation d'applications exploitant cette adaptabilité est en cours afin d'en évaluer les performances lorsque les conditions de charges ou le taux de fautes du réseau sont variables.

3.7 Conclusions et Perspectives

Nous avons présenté la plate-forme multi-agents répartis DarX. DarX permet de répliquer des agents sur un ensemble de sites. La plate-forme possède les propriétés suivantes :

— *Transparence* : la réplication d'un agent est totalement cachée aux autres agents. Un service de nommage permet de masquer les sites d'exécution des agents

— *Adaptabilité* : à tout moment un agent peut changer sa stratégie de réplication et basculer sur une des stratégies pré-définies. Il peut aussi changer le degré de réplication.

DarX est le fruit de notre expérience sur les systèmes STAR et GatoStar où malgré des mécanismes de gestion de fautes optimisés, le choix des protocoles est laissé à l'utilisateur et reste fixe pendant l'exécution. DarX est une manière plus moderne de concevoir des systèmes tolérant les fautes centrée sur la souplesse et la possibilité d'adaptation dynamique.

Nous travaillons actuellement à la définition d'une plate-forme générique afin de pouvoir nous adapter à plusieurs systèmes multi-agents (notamment MadKit [96] du laboratoire LIRMM de l'université de Montpellier). Nous visons ensuite à étendre DarX pour gérer un

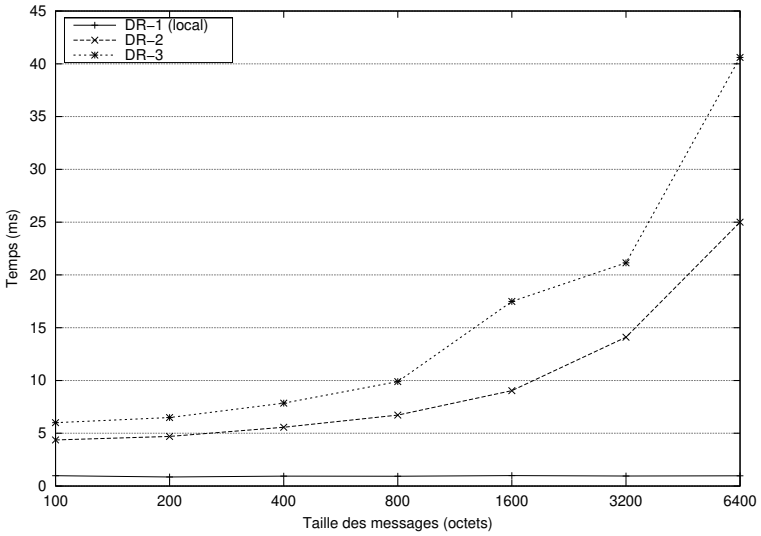

FIGURE 3.7 – Latence des communication en fonction du degré de réplication

grand nombre de machines. Ces travaux qui débutent font l'objet d'une collaboration avec le Laboratoire d'Informatique du Havre.

Deuxième partie

Placement et tolérance aux fautes

Chapitre 4

Unification de la tolérance aux fautes et de la répartition de charge

Sommaire

4.1 Introduction

La tolérance aux fautes entraîne inévitablement un surcoût de consommation de ressources (processeur, mémoire, disque) qui ne peut pas être complètement absorbé par des optimisations telles qui avons présentées dans le chapitre 2. Or, des nombreuses études ont montré que sur des réseaux locaux une grande proportion des machines est sous-utilisée qui varie selon les études entre 33% et 90 % [64, 85].

Les systèmes de placement de programmes permettent de profiter des ressources disponibles en répartissant les charges processeur et mémoire. On constate que les techniques employées par ces répartiteurs de charge ont des nombreuses similitudes avec les plates-formes de tolérance aux fautes. En effet, dans les deux cas, il faut avoir une vision globale des machines disponibles, les processus sont amenés à être placés et déplacés de manière transparente soit en cas de variation de charges ou d'occurrence de fautes.

A partir de ce constat, nous avons développé la plate-forme d'exécution GatoStar [89]. D'une part, GatoStar optimise la gestion de la ressource processeur en plaçant les programmes en fonction de la charge des machines. D'autre part, nous traitons les défaillances de machine en reprenant automatiquement les processus des sites défaillants.

GatoStar est l'unification de deux systèmes : STAR [169] qui gère les fautes et GATOS [85], développé par Bertil Folliot, qui implémente des stratégies de placement dynamique multi-critères. Nous exploitons la synergie existante entre le placement et la tolérance aux fautes : (1) les algorithmes de placement sont utilisés non seulement lors du lancement des programmes mais également lors des phases de recouvrement de fautes, (2) les points de reprise sont réutilisés pour migrer les processus en cas de variation de charge, (3) une infrastructure logicielle commune (l'anneau logique) collecte les informations de charge et détecte les défaillances.

La section 4.2 décrit les techniques de placement et de migration de programmes. La section 4.3 présente les principaux systèmes qui utilisent à la fois le placement et la tolérance aux fautes. Dans la section 4.4 nous détaillons GatoStar. Puis dans la section 4.5, nous présentons les extensions de nos algorithmes de placement sur lesquelles nous travaillons actuellement.

4.2 Répartition de charge

La répartition de la charge entre machines permet aux applications réparties de bénéficier pleinement des ressources disponibles. En effet de nombreuses études ont montré que même en heures de pointes un grand nombre de machines reste inutilisées [85, 180].

4.2.1 Partage et équilibrage

On distingue deux types d'algorithmes de répartition de charge : le partage de charge et l'équilibrage de charge :

— Le but des *algorithmes de partage* est de maximiser l'utilisation des ressources du système. Pour cela, les algorithmes de partage essayent d'éviter les *états de déséquilibre* [123] où quelques machines sont oisives pendant que des tâches attendent un service dans une autre machine. Si le transfert de tâches était instantané, alors les

états de déséquilibre pourraient être évités en transférant les tâches dés qu'une machine devient oisive. À cause de temps de collecte des informations, de décisions, et du délai de la communication, le transfert de tâche n'est jamais instantané. Généralement, les transferts se font donc des machines surchargées vers les machines sous-chargées avec l'hypothèse que les machines sous-chargées le restent pendant une certaine période de temps.

— Les *algorithmes d'équilibrage* essayent aussi d'éviter les *états de déséquilibre*, mais de manière plus systématique en tentant d'égaliser les charges de toutes les machines du système. Krueger et Livny [115] ont montré que l'équilibrage de charge peut potentiellement réduire le temps de réponse moyen des tâches par rapport au partage de charge. Cependant, l'équilibrage de charge produit un fort taux de transfert des tâches entre les machines, ce qui produit un surcoût élevé pouvant entraîner une dégradation des performances.

Dans la suite, nous utiliserons le terme général de distribution de charge.

4.2.2 Classification des algorithmes

De nombreux systèmes de distribution de charge ont été proposés qui diffèrent sur les informations utilisées lors du placement et sur l'instant où les décisions de placements sont prises. Ainsi, la répartition peut être statique ou dynamique. Dans une stratégie statique, la décision de placement est uniquement basée sur les informations concernant le comportement moyen du système [190]. Dans les approches dynamiques, les décisions sont prises en fonction de l'état courant du système [30, 89, 207]. Dans la suite, nous nous intéresserons qu'à ce dernier type d'approche.

Il y a deux grandes classes d'algorithmes de répartition dynamique [67] :

— Dans les approches *à l'initiative de l'émetteur* [68] (sender initiated), les sites fortement chargés recherchent des sites moins utilisés pour y transférer des tâches locales.

— Dans l'approche *à l'initiative du récepteur* [123] (receiver initiated), les sites inutilisés cherchent à soulager les sites chargés en important certaines de leurs tâches.

Ces deux techniques peuvent se combiner, il s'agit alors de l'approche *initiation symétrique*.

Ces algorithmes peuvent s'appuyer sur un mécanisme de *placement initial* (ou non préemptif) [207]. Dans ce cas, les tâches sont allouées à leur création sur un site et y restent jusqu'à leur terminaison. Ils peuvent également se reposer sur la *migration* (appelé également placement préemptif) [31, 122]. Dans ce cas, les décisions de placement peuvent avoir lieu à tout moment et des tâches en cours d'exécution peuvent être suspendues et déplacées sur de nouveaux sites.

La plupart des algorithmes proposés utilisent des seuils de charge pour prendre leurs décisions de placement ou de migration. Ceci permet de minimiser les échanges d'informations tout en gardant des performances proches d'algorithmes plus complexes [206].

4.2.3 Politiques

De manière plus précise, les systèmes de distribution de charge sont composés de quatre politiques : une politique de transfert, une politique de sélection, une politique de localisation,

et une politique de collecte d'information [33, 172].

La politique de transfert

Dans cette politique, un site détermine localement s'il peut participer au transfert de tâche, soit comme émetteur ou bien comme récepteur. Plusieurs politiques sont proposées dans la littérature ; la majorité de ces algorithmes est basée sur l'utilisation de deux seuils [2, 35, 31, 89]. Les seuils sont exprimés en unité de charge. A l'arrivée d'une nouvelle tâche, la politique de transfert considère le site local comme émetteur si la charge dépasse à certain seuil T1. A l'achèvement d'une tâche, elle considère le site comme récepteur si la charge descend au-dessous d'un autre seuil T2. Certaines politiques prennent la même valeur pour T1 et T2 [2, 35] ; d'autres prennent des valeurs différentes [88] ou bien variables [68, 101, 31] (par exemple, autour de la valeur moyenne de la charge du système).

La politique de sélection

Une fois le site émetteur choisi, il faut déterminer la tâche à transférer. La politique la plus simple est de choisir la tâche la plus récente dans le site, celle qui a fait rendre le site émetteur. Le transfert d'une telle tâche est peu coûteux car il s'agit d'un transfert non-préemptif (la tâche n'est pas encore exécutée). Dans des stratégies plus complexes, plusieurs facteurs sont considérés dans le choix de la tâche transférée, comme :

- *Un surcoût minimum.* Dans ce cas, le choix de petites tâches en taille mémoire est privilégié.
- *Le temps d'exécution de la tâche.* L'exécution de la tâche transférée doit être assez longue pour compenser le surcoût du transfert. Cependant, il n'existe pas toujours de relation claire entre la durée de la tâche et son temps d'exécution restant. Même si est vrai que plus une tâche s'exécute plus sa probabilité de se terminer dans un futur proche est faible.
- *L'indépendance de la tâche.* La tâche transférée doit faire peu d'appels systèmes dépendant du site émetteur, car ces appels ultérieurement produiront un surcoût supplémentaire.

D'autres critères peuvent être introduits, comme les opérations d'entrée/sortie et les opérations de communications.

La politique de localisation

Le rôle de cette politique est de trouver un partenaire de transfert, émetteur ou récepteur à la suite de la décision locale de la politique de transfert.

Les politiques distribuées [88, 31] sont souvent basées sur un sondage séquentiel des différents sites. Alternativement au sondage, la politique peut faire une diffusion d'une requête de partage.

Dans les politiques centralisées [207, 121], le site local contacte un coordinateur pour localiser un partenaire.

La politique de collecte d'informations

La politique de collecte d'informations décide *quand* il faut collecter les informations sur les états des autres sites, *où* il faut les chercher et *quelles* sont ces informations. Il y a trois

types de politique de collecte d'informations [85] :

— Les politiques *à la demande* : un site commence à collecter les informations des autres sites, seulement quand il devient un émetteur ou un récepteur.
— Les politiques *au changement d'état (ou à l'offre)* : chaque site diffuse ses informations après un changement important de son état. En centralisé, les informations seront envoyées à un site coordinateur. En distribué, elles seront envoyées à tous les sites ou à une partie d'entre eux.
— Les politiques *périodiques* : la collecte des informations est faite d'une façon périodique. Lorsque le système est trop chargé, ce genre de politiques peut alors entraîner une dégradation de performances.

4.2.4 Information de charge

La plupart des algorithmes de placement dynamique utilisent des informations résumées en une seule variable, appelée indicateur de charge, associée à chaque machine et évoluant dynamiquement avec l'état de celle-ci. Un bon indicateur de charge doit refléter la charge de travail actuelle, prédire la charge dans un futur proche, et être stable (en étant insensible aux fluctuations à court terme). Lorsque l'objectif recherché est la minimisation du temps de réponse moyen des processus (qui est l'objectif le plus courant), un indicateur couramment utilisé est la somme des longueurs de queues des processus prêts et des processus en attente d'entrées-sorties, échantillonnée toutes les quelques secondes et lissée. Dans la pratique d'autres paramètres doivent être pris en compte comme la taille mémoire nécessaire, les accès aux fichiers, la vitesse des liens de communication. De nombreux algorithmes ont été proposés pour prendre en compte un seul de ces paramètres [80, 35, 84]. Quelques systèmes combinent ces critères [30, 28, 86].

Lorsque les machines ont un caractère personnel, certains auteurs ont introduit en outre un indicateur binaire reflétant la disponibilité ou l'indisponibilité d'une machine pour l'accueil de processus extérieurs, sur la base du nombre courant de connexions ou de l'activité clavier/souris. Cependant, ces quantités ne sont pas toujours liées à une gêne réelle du propriétaire par d'éventuels processus extérieurs. Aussi, on préfère généralement utiliser un seuil sur un indicateur classique, ce seuil étant fixé empiriquement à une valeur au-delà de laquelle le propriétaire perçoit la présence de processus extérieurs.

4.2.5 Architecture des systèmes

Les différents gestionnaires du placement dynamiques sont divisés en deux groupes : ceux qui sont obligatoirement répartis sur toutes les machines, et ceux pour lesquels on a le choix entre une réalisation centralisée ou répartie. Les gestionnaires répartis sont :

— Le gestionnaire d'exécution à distance : il est chargé de recevoir les requêtes d'exécution, de lancer les processus avec les droits appropriés, de recevoir le signal de fin de processus, et de renvoyer le résultat de l'exécution au gestionnaire demandeur. Le gestionnaire d'exécution à distance est un composant généralement fourni dans les systèmes d'exploitation.
— Le gestionnaire local d'application : il est chargé de faire le lien entre l'exécution locale

des processus d'une application à travers la bibliothèque de programmation fournie par le système de placement, et les gestionnaires d'applications concernés.

— Le gestionnaire de calcul de la charge locale : il s'occupe de surveiller l'état local de la machine et de fournir ces informations au gestionnaire de collecte de l'état global. La surveillance est en général périodique, elle peut également être intégrée au noyau et être activée sur des événements systèmes pré-définis.

Les gestionnaires répartis ou centralisés sont les suivants :

— Le gestionnaire de collecte de l'état global : il récupère les informations réparties au moyen des gestionnaires de calcul de la charge locale. En général il stocke également les caractéristiques statiques de l'environnement (type et vitesse des processeurs, mémoire vive disponible, etc.).

— Le gestionnaire d'algorithmes de placement : l'algorithme de placement correspondant est appelé à la demande du gestionnaire d'applications parallèles et se base sur les informations du gestionnaire de collecte de l'état global. Ce gestionnaire peut également avoir le rôle d'ordonnanceur, en choisissant l'ordre d'exécution des processus. Les requêtes sont ensuite envoyées aux gestionnaires d'exécution distante concernés.

— Le gestionnaire d'algorithmes de migration : ce gestionnaire réalise les stratégies préemptives. En général, il est pas appelé implicitement en fonction des informations fournies par le gestionnaire de collecte de l'état global. Les requêtes de migration sont envoyées aux gestionnaires locaux d'applications de la machine destination et de la machine cible.

— Le gestionnaire d'applications parallèles : il reçoit les demandes de soumission de travaux, stocke de manière interne les informations sur l'état de l'application.

4.3 Placement et tolérance les fautes

Si l'on compare les techniques employées par les systèmes de tolérance aux fautes et de placement, on remarque de nombreux mécanismes communs comme la migration de processus, la collecte d'état global ou la gestion de "membership". Cependant, on constate que malgré la synergie entre le placement de programmes et la tolérance aux fautes, peu de plates-formes combinent ces deux aspects.

Les principaux systèmes sont REM (Remote Execution Manager) [173] , Condor [189] , DAWGS (Distributed Automated Workload sharinG System) [58] , Paralex [20] et MARS [186]. Ces systèmes sont essentiellement des gestionnaires de distribution de charge étendus pour traiter les fautes.

REM est un gestionnaire pour l'équilibrage de charge qui supporte des applications parallèles. Il fonctionne au-dessus d'Unix sur un réseau local de stations de travail. Chaque processus peut créer des fils sur des machines distantes et communiquer avec eux. Une réplication active des processus permet aux applications de résister aux défaillances de site. Cependant, la défaillance de la machine de l'utilisateur entraîne l'échec de toute l'application. De plus, le principe de la réplication active entre en conflit avec la politique de répartition de charge, en entraînant en permanence un surcoût de traitement important sur toutes les machines

exécutant une copie des programmes répliqués.

Condor et **DAWGS** s'appuient sur des principes similaires à GatoStar. Un placement initial permet d'allouer les processus sur les machines inutilisées. Ensuite les processus peuvent se déplacer en cours d'exécution en utilisant un mécanisme de point de reprise afin de répartir la charge ou traiter les défaillances de machines. Le principal inconvénient de Condor est qu'il ne gère que des processus indépendants, et donc sans possibilité de communication ou de synchronisation entre processus. Cependant, plusieurs projets comme **coCheck** de l'Université de Munich [133, 181] ou **Condor-PVM** de l'Université du Wisconsin intègrent les mécanismes de Condor dans des environnements de programmation d'applications parallèles comme PVM ou MPI . Ainsi les applications parallèles (uniquement de type maître-esclaves dans le cas de Condor-PVM) peuvent profiter des propriétés de tolérance aux fautes offertes par Condor.

DAWGS est un gestionnaire plus complet d'équilibrage de charge et de tolérance aux fautes incluant notamment la redirection des entrées/sorties. Cependant, DAWGS est basé sur des modifications du système d'exploitation, ce qui limite la portabilité du système et des applications supportées. De plus, le recouvrement de fautes dans DAWGS n'est pas configurable et un processus affecté par une faute doit attendre pour être repris, que la machine défaillante redémarre. Ainsi, il ne tire pas profit de l'existence potentielle de machines inutilisées pour la reprise des processus.

Paralex est un environnement de programmation permettant la conception d'applications parallèles. La politique de placement vise à maximiser les exécutions parallèles et minimiser les communications distantes. Paralex supporte l'occurrence de pannes durant l'exécution des programmes. Il permet aux utilisateurs de spécifier un niveau de tolérance aux fautes pour chaque application. Pour un niveau égal à k, tous les programmes seront répliqués k+1 fois. Cependant, la migration de processus pour équilibrer la charge des machines lors de l'exécution n'est possible qu'entre les sites contenant une réplique. Comme les répliques elles-mêmes ne bougent pas, cela limite les possibilités de répartition de charge et rend critique le choix initiale de l'emplacement des répliques. De plus, le degré de tolérance aux fautes initialement spécifié par le programmeur n'est pas maintenu en cas de défaillance d'une des copies.

MARS est un ordonnanceur d'applications parallèles qui répartit dynamiquement les tâches en fonction de la charge. Une des originalités de ce projet est d'adapter dynamiquement le degré de parallélisme des applications en fonction du nombre de machines disponibles. De plus, MARS intègre des techniques de points de reprise pour traiter les fautes : régulièrement l'état global de l'application est sauvegardé. Comme Condor-PVM, MARS s'appuie sur PVM pour la gestion des communications. MARS est basé sur le système de migration de threads PM^2 . Les applications considérées sont de type maître-esclaves.

4.4 Le système GatoStar

Nous décrivons dans cette section le système GatoStar que nous avons développé au LIP6. GatoStar unifie STAR à la plate-forme de placement dynamique GATOS. GatoStar

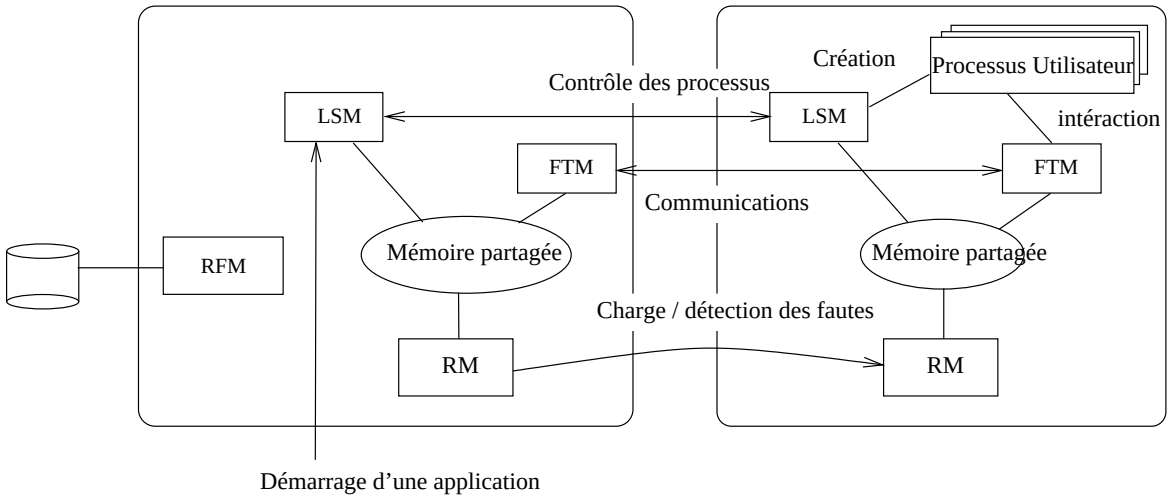

FIGURE 4.1 – Architecture de GatoStar

[45, 88, 86, 89] vise des applications à longue durée de vie et gros grain de parallélisme composées d'un ensemble de processus communicants. Ce modèle couvre un large spectre d'applications scientifiques, comme la factorisation des grands nombres, la conception VLSI ou le traitement d'images. Ces applications peuvent s'exécuter pendant des heures, des jours, voire des semaines. Dans ce contexte, il est important que le support d'exécution s'adapte dynamiquement aux changements pouvant survenir pendant l'exécution. Il faut donc pouvoir non seulement tolérer les fautes mais également réagir aux variations des charges, à l'arrivée de nouveaux utilisateurs.

4.4.1 Architecture

Le prototype actuel est composé de quatre gestionnaires s'exécutant sur toutes les machines participant à l'équilibrage de charge. La figure 4.1 présente l'architecture de GatoStar. Les quatre principaux gestionnaires sont :

- *le gestionnaire de répartition de charge* (Load Sharing Manager - LSM) est composé de deux serveurs : un serveur de calcul de la charge locale et un serveur de gestion des applications responsable du lancement et de la terminaison des programmes,
- *le gestionnaire de tolérance aux fautes* (Fault Tolerant Manager - FTM) intègre les fonctionnalités du serveur de reprise de STAR,
- *le gestionnaire d'anneau* (Ring Manager - RM) est responsable de la détection des fautes et de l'échange des informations de charge.
- *le gestionnaire de support stable* (Reliable File Manager) intègre les démons de SFS (voir section 2.3.4) pour implanter le support stable.

GatoStar assure la transparence du mécanisme de migration. Les communications et accès aux fichiers se font par l'intermédiaire d'une bibliothèque spécifique qui reprend l'interface des appels système Unix pour les fichiers et une interface de type RPC pour les communications. Un service de nommage indépendant de la localisation et un mécanisme de suivi des

communications permettent de masquer totalement la migration d'un processus au niveau applicatif.

4.4.2 Modèle d'application

Chaque application est représentée par le modèle classique de processus communicants reliés par un graphe de précédences. Ce graphe contient les informations qualitatives suivantes :

— les précédences et les communications entre programmes,
— les fichiers utilisés,
— les attirances et répulsions entre processus,
— les algorithmes d'allocation associés à chaque programme,
— les différents noms de codes exécutables de chaque programme.

Cette description peut être fournie par le concepteur d'applications, soit au moyen du GEL (Gatos Extended Language), soit en utilisant des fonctions de bibliothèque pour la partie dynamique (écrite en langage C). Le GEL est un interpréteur intégré dans le gestionnaire d'applications parallèles.

L'hétérogénéité est prise en compte dans GatoStar en associant à chaque programme un ensemble de noms de fichiers représentant des codes binaires pour différentes architectures. La sélection définitive de la version du programme est effectuée par GatoStar en fonction de la disponibilité des machines.

4.4.3 Indicateurs de charge et acquisition des informations

Dans un système homogène, la charge d'une machine est en général égale au taux d'occupation du processeur (nombre de processus actifs et en attente). Nos algorithmes sont conçus pour un système hétérogène où chaque machine peut avoir des caractéristiques différentes. Ainsi, nous définissons la charge d'une machine comme [29, 207] :

— proportionnelle au taux d'occupation du processeur : nombre de processus dans toutes les files d'attente du système (processeur et entrées/sorties) moins les processus inactifs depuis plus de T secondes,
— inversement proportionnelle à la vitesse relative du processeur (paramètre statique indiqué par l'administrateur).

Pour les besoins des algorithmes multi-critères (voir section 4.4.7), il est nécessaire de connaître, en plus de la charge, la mémoire disponible et l'activité des disques et du réseau. Nous considérons que le temps d'accès à la mémoire est homogène. Les charges disques et réseau sont calculées de manière identique à la charge processeur (proportionnelle aux taux d'occupation et inversement proportionnelle à leur vitesse relative).

Pour collecter une information globale sur l'état des machines, nous avons choisi une stratégie d'acquisition à l'offre, où les informations de charge des sites non surchargés sont envoyées périodiquement. Cette approche présente les avantages suivants :

— La décision de placement est locale et ne nécessite pas d'autres échanges de charge.
— Une information récente ne signifie pas un meilleur placement, à cause des activités imprévisibles des utilisateurs.

— L'environnement étant globalement sous-chargé, la surcharge permanente induite par l'algorithme de diffusion est en fait très faible de l'ordre de 0,05 % pour 6 machines).

Nous réutilisons l'anneau logique (voir 2.3.2) pour véhiculer les informations de charge lors de la surveillance des machines. Périodiquement (période fixée à 3 secondes), chaque machine envoie un message de détection à son successeur pour vérifier sa bonne marche. Les informations de charge sont alors ajoutées aux messages de détection. Le surcoût induit par ce parasitage est pratiquement nul.

4.4.4 Placement en fonction de la charge

Pour un programme donné, la sélection d'une machine s'effectue en deux étapes :

— Pour chaque version du programme (compilée sur une architecture différente), la machine compatible ayant la plus faible charge est sélectionnée.

— Ensuite, la machine (M) avec la plus faible charge est sélectionnée, et la version correspondante du programme est retenue. Si plusieurs machines ont la plus faible charge, la machine ayant la plus grande vitesse processeur est retenue.

Le programme est lancé sur M si la charge de celle-ci est inférieure au *seuil de surcharge*. En effet, dans le cas où toutes les machines sont surchargées, l'exécution distante serait plus coûteuse que l'exécution locale. Le programme est alors lancé sur la machine de départ. Lorsque la charge d'un site dépasse la valeur du seuil de surcharge, celui-ci se retire temporairement du mécanisme de placement. Un seuil de surcharge pour les machines ayant un utilisateur connecté permet également d'éviter que ces utilisateurs trouvent leurs machines surchargées par des programmes ne leur appartenant pas. Ce seuil de surcharge a un double emploi : ne pas saturer une machine et réduire les interférences avec les propriétaires des machines. La valeur du seuil de surcharge qui a été définie originellement en fonction des autres études [70, 3, 35, 85] (fixée à 1,0) semble la mieux adaptée à un environnement universitaire. Elle reflète l'utilisation à 100 % des capacités d'une machine. Un seuil de tolérance (fixé à 0,87) permet de supprimer les changements d'états inutiles dus à une charge oscillante autour du seuil de surcharge.

4.4.5 Variations rapides de la charge et machine "victime"

Dans un système réparti, il est difficile de déterminer localement la valeur de l'état global. Ceci est dû principalement aux délais de communication et aux fautes. Dans le cas de la répartition de charge, une connaissance exacte de la charge courante de chaque machine est impossible (et inutile) en raison :

— des délais de communication des informations sur la charge,

— des délais entre le moment où un site a pris connaissance des informations sur la charge et le moment où il place un programme,

— des activités des utilisateurs qui n'utilisent pas les services de répartition de charge.

Les variations de charge étant imprévisibles et les délais/activités précédents étant significatifs, une machine libre au moment de la prise de décision peut devenir surchargée avant le début de l'exécution du processus placé. De plus, si plusieurs allocations ont lieu en parallèle sur plusieurs machines, le même site "libre" peut être choisi pour l'exécution de plusieurs pro-

cessus créant une surcharge brutale sur ce site (problème de la victime). Cette surcharge peut être réduite après le placement par une diminution notable de la priorité des processus extérieurs ou par un mécanisme de migration (section suivante). Nous avons résolu ce problème survenant au moment du placement initial : un programme est replacé si la machine cible est devenue lourdement chargée. Cette redirection des requêtes est mise en œuvre en joignant à la requête d'exécution distante la charge estimée de la machine d'accueil. Si la charge courante de cette machine a augmenté significativement, le replacement du programme est effectué.

4.4.6 Migration de processus

Les algorithmes de migration permettent de s'adapter *en cours d'exécution* à l'évolution du comportement des applications et du système. L'évolution du système est due à des activités étrangères à GatoStar : un utilisateur se connecte ou bien envoie des processus sur des machines déjà chargées. L'évolution d'une application résulte des variations dans l'utilisation des ressources par les processus de l'application (occupation processeur, demande mémoire, communication entre processus).

La prise de décision est basée sur l'utilisation de deux seuils opposés : un *seuil de migration* et un *seuil de réception*. Sur chaque site un serveur local évalue périodiquement deux conditions :

— *Surcharge* : la machine locale est au-dessus du *seuil de migration* et au moins une machine est en dessous du *seuil de réception*. Dans ce cas, la machine locale est considérée comme émettrice et cherche à expulser des tâches.

— *Sous-charge* : une machine est en dessous du *seuil de réception*. La machine devient alors réceptrice et cherche à importer des tâches.

Ces deux seuils permettent d'une part de soulager les machines fortement chargées et d'autre part de profiter des machines qui se libèrent. Dans tous les cas, la migration n'est décidée que si l'écart de charge entre les machines source et cible est significatif.

La sélection du processus se fait par raffinement successif de l'ensemble des processus GatoStar du site source. Le choix dépend en partie de l'algorithme choisi par l'utilisateur lors du placement initial.

On cherche à déplacer le processus auquel il reste le plus long temps d'exécution. Ce choix est justifié par le fait que les processus les plus vieux ont une plus grande probabilité de s'exécuter encore plus longtemps [118]. Si on ne possède aucune statistique, le processus choisi doit avoir un temps minimum d'exécution. On évite ainsi qu'un processus passe son temps à migrer d'une machine à une autre.

4.4.7 Algorithmes multi-critères

Le but des algorithmes multi-critères est de prendre en compte non seulement l'état des machines (et le respect des utilisateurs interactifs), mais également le comportement des applications en tenant compte des différents critères d'allocation donnés dans le graphe de description (GEL). L'état des machines est caractérisé par : la vitesse des processeurs, la vitesse des disques et des liens de communication, la mémoire disponible, et le taux d'occupation des processeurs. Le comportement des applications est défini par : le temps processeur requis, la

mémoire nécessaire à l'exécution, les fichiers utilisés, et les communications entre programmes.

Vu la complexité de combiner les différents critères, le programmeur intervient dans la décision de placement en donnant des indications sur :

— le choix des algorithmes à utiliser pour chaque programme,
— des contraintes de placement au moyen de critères de localisation relative (attraction/répulsion entre programmes ou ressources),
— l'association d'un ensemble de valeurs à chaque programme (temps processeur, mémoire utilisée, volume de communication et d'accès aux fichiers).

GatoStar possède un algorithme pour chaque critère d'allocation [44] appelé en fonction des processus prêts du graphe d'exécution.

— *Temps de réponse* : Cet algorithme vise à minimiser le temps d'exécution processeur global de toute l'application. Dans ce cas, l'utilisateur doit fournir pour chaque programme une estimation du temps processeur nécessaire. Les machines sont alors allouées aux programmes dans l'ordre décroissant de leur temps d'exécution. Ainsi, les programmes les plus long sont placés sur les machines les plus rapides et les moins chargées.
— *Accès aux fichiers* : Ce critère consiste à minimiser le coût d'accès aux fichiers. L'algorithme est basé sur plusieurs paramètres, comme le nombre d'accès estimé, la taille d'un accès, la taille du fichier, la vitesse des disques et du réseau. Deux techniques sont employées qui consistent soit à placer les programmes près de leurs fichiers ou à importer les fichiers vers les sites d'exécution.
— *Communications entre processus* : Cet algorithme a pour objet de réduire le coût des communications entre les processus. Ceci est équivalent en pratique à réduire le nombre de communications distantes et ainsi à allouer sur une même machine tous les processus communicants.
— *Besoins en mémoire* : Ce critère consiste à allouer une machine ayant l'espace mémoire requis suffisant pour l'exécution. Ceci est réalisé en déterminant les besoins en mémoire du programme et la mémoire disponible de toutes les machines.

Ces critères peuvent être combinés successivement pour assurer une allocation en fonction de plusieurs algorithmes.

4.4.8 Evaluation de performances

Nous présentons deux séries de mesures pour mettre en évidence l'intérêt de combiner la placement dynamique et la migration.

La figure 4.2 présente le temps de réponse d'une application parallèle de six tâches s'exécutant sur six machines. Les mesures ont été faites sur un réseau étudiant pendant les heures de travail. Nous comparons la stratégie de placement non-préemptive avec la migration. Chaque programme exécute un nombre d'itérations variant de 50 à 500 millions. Il n'y a pas de communication entre les programmes. L'exécution locale (sans distribution) de l'application varie de 1080 à 11.160 secondes.

Lorsque le nombre d'itérations est faible, le temps d'exécution de l'application est trop court pour que la migration améliore de manière significative les performances. Lorsque le nombre d'itérations augmente, nous observons une différence moyenne de 28 % du temps de

réponse. Ce gain est dû au correction du placement initial. En effet, initialement les programmes sont lancés sur des machines éventuellement inutilisées. Lorsque le temps de l'application augmente, il y a une forte probabilité que des utilisateurs se connectent sur les machines de l'application. Dans ce cas la migration permet d'expulser les tâches locales sur d'autres machines peu utilisées.

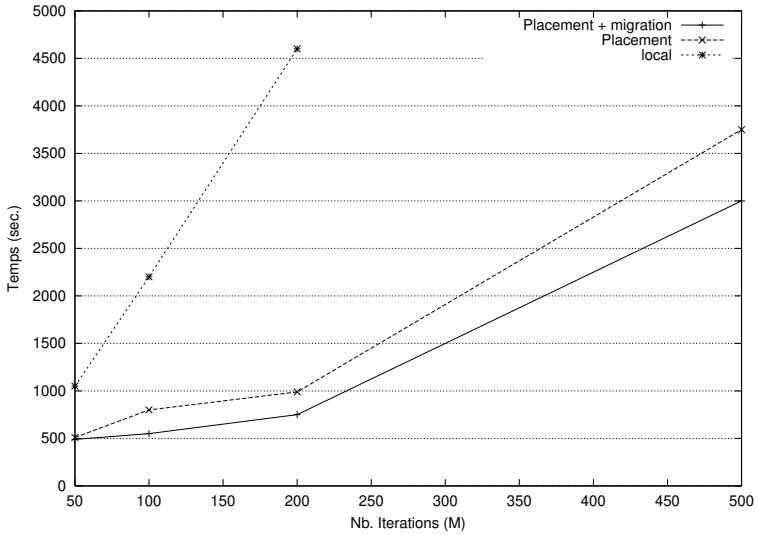

FIGURE 4.2 – Temps d'exécution en fonction du nombre d'itérations.

La migration peut apporter aussi un gain même dans un environnement non chargé. La figure 4.3 présente le gain observé pour une application composée de six programmes parallèles de 1800 secondes chacun. Nous avons fait varier le nombre de sites de un à cinq. On remarque que la migration augmente les performances en adaptant dynamiquement le nombre de programmes sur chaque machine essentiellement lorsque des programmes se terminent. Par exemple, dans une configuration à quatre sites, quatre programmes sont placés initialement sur les deux première machines (deux programmes par machine) et deux programmes sont placés sur les deux dernières machines (un programme par machine). Les deux dernières machines sont donc libérées rapidement et peuvent décharger les deux programmes de deux programmes.

4.5 Vers un placement multi-critère

Les algorithmes de placement multi-critères [31, 207] utilisent généralement des informations assez grossières sur les applications (temps processeurs consommés par chaque processus, taux global de communication, nombre de fichiers utilisés ...). Même si ces informations sont importantes et permettent d'affiner les décisions de placement, elles restent néanmoins globales et ne permettent pas de capturer le réel degré de parallélisme entre les processus. Or, ce degré de parallélisme entre programmes est une information importante pour le placement. Il permet de regrouper sur une même machine deux processus se synchronisant régulièrement

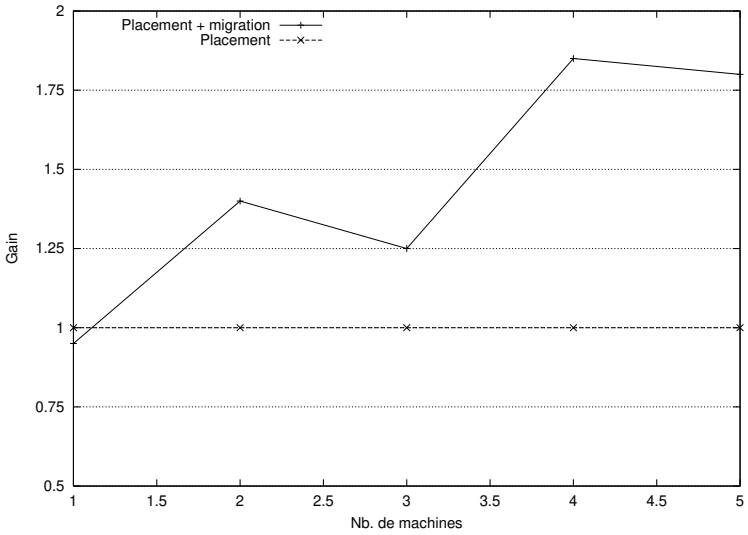

FIGURE 4.3 – Gain en fonction du nombre de machines.

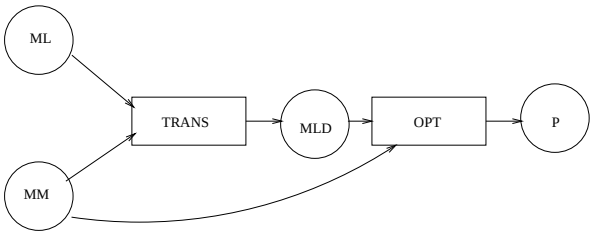

FIGURE 4.4 – Schéma fonctionnel du placement.

avec un taux de recouvrement des traitements faible. Les modèles de représentation classiques ne permettent pas de capturer une telle information.

Afin de définir des stratégies de placement plus efficaces, nous avons défini un modèle de description d'applications permettant d'extraire précisément le réel degré de parallélisme entre les processus d'application. Ce modèle est ensuite transposé et évalué sur une architecture répartie.

La figure 4.4 résume notre méthodologie. Le placement consiste à plonger un graphe logiciel dans un graphe matériel en utilisant des règles portant sur les attributs des processus, threads et tâches. Le schéma fonctionnel du traitement des graphes et du placement est représenté figure 4.4. Le Modèle Matériel (MM) est un graphe décrivant le matériel (machines et réseau) ; le Modèle Logiciel (ML) est un graphe décrivant l'application. Dans un premier temps, le module TRANS transforme le ML, indépendant du matériel, en Modèle Logiciel Dépendant (MLD). Le MLD est ensuite fourni à l'optimiseur (OPT). C'est OPT qui va réaliser le placement grâce à des heuristiques multi-critères utilisant les règles sur les attributs.

Les deux paragraphes suivants décrivent le modèles de description de l'application et de l'infrastructure. La section suivante illustre des heuristiques de placement dérivées de ces

modèles. Ces travaux font l'objet de la thèse de Damien Collard dirigée par Michel Minoux. Enfin, la section 4.5.4 présente un outil générique d'observation permettant de caractériser les applications. Cet outil a été développé dans le cadre de la thèse de Céline Boutrous.

4.5.1 Modèle de description des applications

Pour modéliser de façon fine la structure et le comportement des applications, nous les décrivons par un graphe à trois niveaux : le niveau *processus*, le niveau *thread* et le niveau *tâche*. Les processus sont divisés en threads qui sont eux mêmes découpés en tâches. Contrairement aux processus et aux threads, les *tâches* ne sont pas des entités manipulables par le système d'exploitation : cette subdivision est un artifice pour modéliser la structure des threads et permettre un placement plus fin de ceux-ci.

Nous distinguons trois types de tâches :

— Les tâches de calcul (C) sont délimitées par les communications et ne contiennent pas de communication.

— Les communications sont elles représentées par des tâches de transfert (T), qui ne contiennent pas de calcul, et peuvent modéliser aussi bien une communication réseau qu'une communication avec une mémoire ou un disque.

— Les tâches mémoire (M) ne servent qu'à représenter le volume mémoire utilisé par une tâche de calcul. Ce volume représente l'agrégation des accès mémoire effectués par la tâche C.

Les processus, threads et tâches sont caractérisés par des *attributs* sur lesquels porteront les règles de placement : architecture matérielle, système d'exploitation, contraintes temporelles (temps réel), priorité, volume mémoire manipulé, communications, etc.

4.5.2 Modèle de description de l'infrastructure

Le réseau de machines est représenté par un graphe orienté dont les nœuds sont les processeurs et les arcs les liens de communication entre ces processeurs. Ce graphe possède deux niveaux pour permettre une modélisation fine de la topologie et des caractéristiques du matériel : le niveau *machine* et le niveau *processeur*.

Le niveau *machine* permet de représenter correctement le couplage fort entre processeurs d'une même machine par opposition au couplage faible de processeurs appartenant à des machines distinctes. Ce niveau permet d'introduire une contrainte sur le placement relatif de threads appartenant à un même processus : ils doivent être placés sur une même machine.

Par *processeur*, nous entendons en fait les processeurs au sens classique (unités centrales), mais aussi les composants réseau et les mémoires. Les différentes classes de processeurs sont C (calcul) pour les processeurs, T (transfert) pour les réseaux, et M (mémoire) pour les mémoires (mémoires centrales et mémoires de masse). Les caractéristiques de ces processeurs sont exprimées par des *attributs*.

4.5.3 Heuristiques de placement

Le problème du placement est NP-complet et nécessite donc que l'optimiseur (OPT) résolve les contraintes imposées sur le placement des tâches par l'utilisation d'heuristiques ap-

pliquées à la résolution de systèmes linéaires [157]. Une première phase consiste à exprimer des règles simples (des contraintes) permettant de réduire l'espace des solutions exploré.

Soient T et P les ensembles des tâches et des processeurs. On a $T = T_C \cup T_T \cup T_M$ avec T_C, T_T et T_M les ensembles de tâches de calcul, de transfert, et de communication, respectivement. De la même façon, $P = P_C \cup P_T \cup P_M$.

On définit les variables $e_{t,p}$ qui indiquent le placement d'une tâche t sur un processeur p : $e_{t,p} = 1$ si t placée sur p, 0 sinon, avec pour contrainte $\forall t \in T \sum_{p \in P} e_{t,p} = 1$ (une tâche ne peut être placée que sur un processeur).

On peut écrire alors la contrainte mémoire : $\forall p \in P_m \sum_{t \in T_m} m(t) \times e_{t,p} \leq \mu_\lambda(p)$ en notant $m(t)$ la mémoire requise par la tâche t et $\mu_\lambda(p)$ la mémoire virtuelle disponible sur p.

En notant $\theta_f(t)$ et $\theta_d(t)$ les dates de début et de fin de la tâche de transfert t, v_t le volume de données transférées par t, et V_p et l_p les vitesse et latence du processeur de transfert p, on peut écrire : $\forall t \in T_T, \forall p \in P_T \theta_f(t) - \theta_d(t) \geq (\frac{v(t)}{V_p} + l_p)) \times e_{t,p}$. Ceci exprime le temps minimal d'une communication en fonction des attributs de la tâche de transfert et du processeur sur lequel elle est placée.

De la même façon pour le calcul : $\theta_f(t) - \theta_d(t) \geq \frac{n_i(t)}{v(p)}$ avec $n_i(t)$ le nombre d'instructions de la tâche t. Ceci exprime le temps minimal d'une tâche de calcul.

Il est possible, grâce au découpage des threads en tâches, d'avoir des informations qualitatives sur le comportement des threads, en particulier le schéma de communication, et déterminer ainsi le degré de parallélisme entre deux threads. Le placement a donc avoir comme critère de minimiser la perte de parallélisme due au placement relatif des threads communicants.

4.5.4 Observation d'application

Pour affiner les algorithmes de placement, il est nécessaire de fournir des informations précises sur les applications. Or, l'observation fine d'applications parallèles est délicate. Il faut pouvoir récolter suffisamment d'informations sans trop perturber l'application. Nous proposons donc une approche à granularité variable qui permet de réduire le coût de manière significative en diminuant le volume de communications. Une telle approche réduit les perturbations tout en fournissant les informations nécessaires. Ainsi les algorithmes de placement disposent des informations au moment de la décision, en plus des paramètres classiques, des paramètres plus pertinents tels que le sens de communications, la mémoire réellement utilisée ainsi que le temps de réponse.

Pour les communications, nous définissons deux catégories d'observation. L'observation quantitative consiste à observer le volume de communications dans son ensemble afin d'avoir une vision globale. L'observation qualitative comprend le sens des communications entre le processus ainsi que leur répartition au cours du temps.

Au niveau de la mémoire, l'observation porte essentiellement sur la quantité de mémoire allouée. Alors que la plupart du temps, la mémoire réellement utilisée par un programme est nettement inférieure à celle allouée. A cet effet, nous introduisons une nouvelle information portant sur la quantité maximale de mémoire utilisée à un moment donné. Celle ci est utilisée lors du placement pour une meilleure répartition. Connaissant le maximum de mémoire utilisée par un processus à un instant donné, un des critères de choix de la station cible est la quantité de mémoire physique disponible sur cette station.

Au cours de sa thèse, Céline Boutrous a défini une architecture d'observation répartie sur un réseau local avec un serveur de contrôle et un ensemble d'agents d'observations sur chaque site. Le serveur s'abonne auprès des agents à un ensemble d'informations. Seules ces informations lui seront transmises. Ces agents conservent localement toutes les informations récoltées pour des analyses postexécutions plus fines. L'implantation repose sur la couche de communication NMS déjà développée par Karim Foughali au LIP6 [90].

La granularité de l'observation (la période de collecte et la quantité de données) est adaptée dynamiquement en fonction des besoins. En effet, pendant l'exécution, le système oscille entre l'état stable et instable. Un système est considéré instable s'il est en train de placer un processus ou bien s'il y a une intense activité des utilisateurs, sinon il est stable. Une observation fine n'est utile que lors de la prise de décision de placement. Ainsi, la granularité de l'observation doit être fine quand le système est instable, puis elle redevient à gros grain quand il est stable. Ainsi seules les informations nécessaires sont transmises ce qui diminue le surcoût lié aux communications engendrées par l'observation. Le serveur contrôle à distance l'activité de ses agents en fonction de la stabilité du système.

4.6 Conclusions

Ce chapitre a présenté GatoStar qui est l'unification d'un système de répartition de charge, Gatos, et de la plate-forme STAR. Les techniques de placement ont été largement étudiées. L'originalité de notre approche est de combiner les deux techniques (répartition de charge et tolérance aux fautes) en éliminant des fonctions redondantes et définissant une infrastructure commune pour collecter les informations d'état global.

La répartition de la charge est basée sur un placement initial en fonction de l'état du réseau et la migration pour réagir aux variations de charge en cours d'exécution.

De cette expérience, nous avons abouti aux conclusions suivantes :

— La migration peut apporter un réel gain. L'association de cette technique avec une stratégie de placement initial s'est avérée particulièrement performante,

— Plusieurs paramètres ont une grande influence sur l'efficacité du système de placement (valeur des seuils, période de collecte de l'information). Il est délicat de fixer les valeurs.

— Il est clair que plusieurs critères sont à prendre en compte lors des décisions de placement (état global du système, estimation des ressources nécessaires à l'application). Vue la complexité de combiner ses critères (quel poids donner à chacun ?), nous avons adopté une approche statique où l'utilisateur fixe l'ordre des critères.

Le travail sur GatoStar s'est poursuivi selon deux axes :

— la définition de nouvelles heuristiques de placement multi-critère en fonction d'information fine sur le comportement de l'application. Il s'agit de la thèse de Damien Collard dirigée par Michel Minoux de l'équipe "Algorithmes Numériques et Parallélisme" du LIP6.

— la mise en œuvre d'une méthodologie pour évaluer et comparer les stratégies de placement en fonction de la configuration des applications et de l'environnement. Ce travail qui a fait l'objet de la thèse de Yanal Haj-Mahmoud [98] est détaillé dans le chapitre suivant.

Chapitre 5

L'outil de placement SIGAP

Sommaire

5.1 Introduction

L'évaluation des performances d'un système de distribution de charge est un problème difficile compte tenu d'une part du nombre important de paramètres à prendre en compte (concernant le type d'applications et d'environnements répartis) et d'autre part de la difficulté d'effectuer des mesures en environnement réel. Des algorithmes et des paramètres inadaptés à un environnement d'exécution donné et à une application donnée peuvent fortement dégrader les performances, voire conduire à l'écroulement du système.

Les performances des algorithmes de distribution de charge dépendent essentiellement de trois niveaux de configuration : la configuration de la charge à exécuter, celle du système de distribution de charge et celle de l'environnement d'exécution. Dans ce chapitre, nous proposons un modèle paramétrable et configurable qui simule le comportement d'un système réel de distribution de charge en prenant en compte ces trois niveaux de configuration.

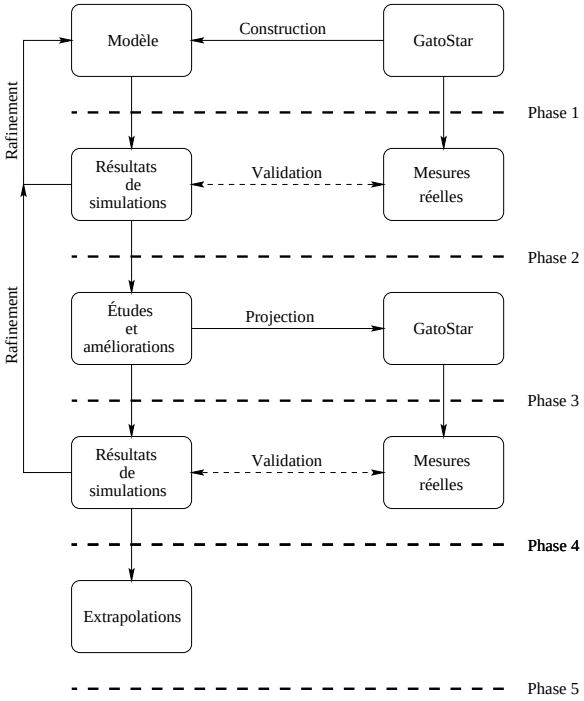

FIGURE 5.1 – Processus de développement du modèle

Notre approche consiste à construire le modèle à partir d'un système de distribution de charge GatoStar jusqu'à obtenir des performances par simulation quasiment identiques à celles observées en environnement réel. Le travail présenté a nécessité des raffinements successifs des paramètres des différents composants du modèle. Une fois tous les paramètres obtenus, il est aisé d'en modifier certains de manière à extrapoler les résultats pour diverses applications et configurations matérielles. Les simulations permettent ensuite d'analyser l'influence des paramètres internes du système de distribution sur les performances afin d'élaborer des nouvelles stratégies de distribution.

La figure 5.1 résume la méthodologie utilisée. La première phase est la proposition du modèle de base. La deuxième est le raffinement successif de ce modèle en comparant les mesures de simulations avec les mesures réelles. La troisième phase est la modification du modèle pour améliorer les performances du système. Chaque fois qu'une modification pertinente est découverte, elle est introduite dans le système réel et suivi par une nouvelle phase de validation et de raffinement des paramètres du modèle. La phase finale est l'extrapolation des paramètres du modèle pour prévoir le comportement du système réel dans de nouveaux environnements. Le travail présenté dans ce chapitre a fait l'objet de la thèse de Yanal Haj-Mahmoud [98].

Le section 5.2 présente les principales approches de simulation et positionne nos travaux. La section 5.3 décrit le modèle implanté en utilisant QNAP2. La section 5.4 montre le calibrage et la validation du modèle. Enfin, dans la section 5.5, nous analysons les performances de notre modèle en étudiant l'impact de plusieurs paramètres.

5.2 Principaux modèles

Les modèles de distribution de charge proposés dans la littérature n'intègrent généralement pas un niveau de détails très élevé. Des approximations importantes sur les lois d'arrivée, le temps de transfert des tâches et leur distribution ont tendance à éloigner les performances simulées du comportement réel.

L'exemple le plus connu est celui du modèle d'Eager [69] qui concluait en 1989 que la migration ne pouvait apporter aucun gain significatif sur un réseau de stations de travail. Des expérimentations ont montré depuis que cette conclusion est fausse. Harchol-Balter et Downey montrent dans [102] que ceci est en partie dû à l'hypothèse d'une loi d'arrivée de tâche exponentielle.

Lin et Raghavendra [120] utilise une version modifiée du modèle $M/M/n$ pour étudier l'efficacité d'un modèle centralisé de distribution de charge. Ils supposent que l'arrivée des processus est poissonnienne de taux λ. Le temps de service de chaque processus est de distribution exponentielle de moyenne $\frac{1}{\mu}$; il est donc indépendant de la taille et la durée de la tâche ce qui n'est toujours le cas dans la réalité. Ils supposent également que le délai d'échange des informations entre les sites est négligeable.

Evans et Butt [75] ont développé un réseau de files d'attente pour étudier les performances d'un modèle réparti de distribution de charge. Le modèle est très simplifié car chaque site du réseau est composé uniquement de deux files d'attente. La première représente les algorithmes d'équilibrage de charge et la deuxième représente l'unité d'exécution. Le temps de service de la première station est supposé négligeable et celui de la deuxième est de distribution exponentielle. De plus, la migration n'est pas prise en compte.

Harchol-Balter et Downey [102] ont particulièrement étudié la distribution des tâches en mesurant les durées de vie d'un grand nombre de processus (plus d'un million) dans les milieux académiques. Ils ont trouvé que la distribution de la durée de vie des processus n'est pas exponentielle ou hyper-exponentielle mais qu'elle coïncide avec une courbe de forme T^k, avec $-1.3 < k < -0.8$. Ils ont également constaté que 50 % de la charge des machines était due à moins de 1 % des processus. Ces processus correspondent à ceux ayant une vie de plus de 4 secondes. Leland et Ott [118] ont mené une étude similaire sur le comportement des tâches séquentielles dans les domaines commerciaux.

5.3 Modèle

Le modèle développé est un réseau de files d'attente. L'outil de modélisation utilisé est QNAP2 [175]. QNAP2 est un langage de description et d'analyse de systèmes de files d'attente. Son principal intérêt est d'offrir non seulement un moyen pour décrire un système, mais c'est également un outil d'analyse pouvant donner des résultats numériques, caractérisant les performances du système.

Nous avons donc développé un modèle générique [101, 99, 100] d'un système de distribution en intégrant plusieurs algorithmes de distribution et plusieurs politiques d'échange d'informations entre les sites. Le passage d'un système à un autre se fait dans une phase de configuration du modèle en choisissant les algorithmes désirés et les valeurs de paramètres.

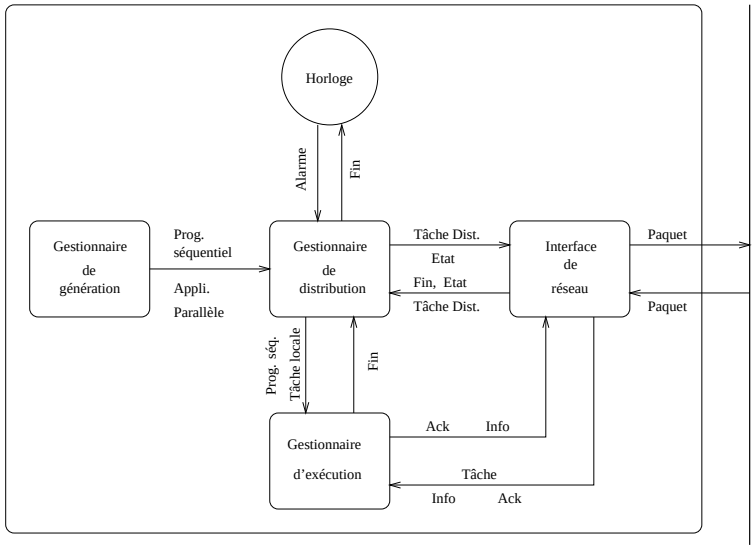

FIGURE 5.2 – Modèle d'un site

5.3.1 Composants du modèle

Le modèle est composé de cinq modules par site : l'horloge du site, le générateur de programmes, le gestionnaire de distribution, le gestionnaire d'exécution et l'interface de réseau. La figure 5.2 montre les transitions des clients entre les différentes composantes d'un site. Dans la suite de ce chapitre nous utiliserons les termes client et serveur des réseaux de files d'attente.

Les entités dynamiques (processus, messages, etc.) sont modélisées par des clients qui circulent dans les files d'attente du modèle. A chaque client est associé deux attributs principaux : sa priorité et sa classe (son type). La classe d'un client détermine l'algorithme à exécuter au niveau du serveur où se trouve ce client. Sa priorité détermine une priorité d'accès au serveur par rapport aux autres clients qui sont présents dans la même file.

Horloge

Ce module simule une horloge dans le site. Il est constitué d'une source qui crée périodiquement un client et l'envoie au gestionnaire de distribution. Ce client est considéré comme une interruption de l'horloge ; il a donc une priorité élevée par rapport aux autres.

Gestionnaire de génération

Le gestionnaire de génération est composé de deux sources de clients : les programmes séquentiels et les programmes parallèles. Un programme séquentiel est composé d'une seule tâche qui s'exécute sur le site local. Il modélise les activités locales de la machine (la charge imposée par le propriétaire de la machine ou par les démons locaux). Un programme parallèle est composé d'un ensemble de tâches reliées entre elles par des contraintes de précédence et de communication. Ces contraintes sont représentées par deux graphes : le *graphe de précédence*

et le *graphe de communication*.

L'exécution d'un programme parallèle consiste à parcourir le graphe de précédence en largeur afin d'extraire les tâches qui peuvent être exécutées en parallèle, puis de les distribuer sur les sites disponibles.

Une tâche est un fragment de code exécuté séquentiellement. Ce fragment est divisé en trois composants : le premier (la partie calcul) est exécuté sur le processeur, le second représente les opérations d'entrée/sortie, le dernier définit les communications. Nous supposons que les opérations d'entrée/sortie et de communication sont dispersées dans la partie calcul. Pour les programmes séquentiels, la tâche contient uniquement les deux premiers composants. Un client sortant du gestionnaire de génération est envoyé au gestionnaire de distribution qui gère son placement.

Gestionnaire de distribution

Le rôle du gestionnaire de distribution est d'une part de gérer la distribution des nouvelles tâches dans le réseau en utilisant un des algorithmes de placement et d'autre part de traiter les brusques variations de charge en appliquant un des algorithmes de migration. Ce gestionnaire est composé de six serveurs : le *serveur de graphe*, le *serveur de placement*, le *serveur de mise à jour*, le *serveur de migration*, le *serveur de diffusion* et le *serveur de réception* (figure 5.3).

1. Le **serveur de graphe** extrait les tâches exécutables du graphe d'exécution et les envoie au serveur de placement. Il reçoit deux types de clients. Le premier type représente les programmes parallèles avec leurs graphes envoyés par le gestionnaire de génération. Le deuxième type représente les messages générés par l'unité d'exécution qui indiquent l'achèvement et le résultat de l'exécution d'une tâche.
2. Le **serveur de placement** simule les algorithmes de placement du système de distribution de charge. Il est chargé de choisir et d'exécuter le "meilleur" placement, sur le réseau, des tâches reçues du serveur précédent, en tenant compte de l'état global du système et des critères de placement. Il envoie les tâches qui seront exécutées à distance vers le gestionnaire de communication et les tâches qui seront exécutées localement au serveur de mise à jour. Ce serveur est constitué d'une seule file d'attente.
3. Sur chaque site, une variable locale *SystemState* représente l'état global du système vu par le site. Le rôle du **serveur de mise à jour** est de mettre à jour cette variable, après la réception d'une tâche exécutée localement ou d'une tâche provenant d'un autre site. Après sa mise à jour, il envoie la tâche reçue vers l'unité d'exécution. Ce serveur est constitué d'une seule file d'attente.
4. Le **serveur de migration** gère la décision et l'exécution de la migration des tâches entre les sites pour équilibrer la charge du réseau. Il reçoit périodiquement un client en provenance de l'horloge qui déclenche l'exécution des algorithmes de migration.
5. Le **serveur de diffusion** reçoit périodiquement un client de l'horloge, il diffuse alors l'état du site à tous les sites du réseau (le contenu de la variable *SystemState*). Le transfert se fait en utilisant soit la diffusion classique à tous les sites, soit une diffusion circulaire sur un anneau logique qui relie les différents sites du réseau. Le choix de

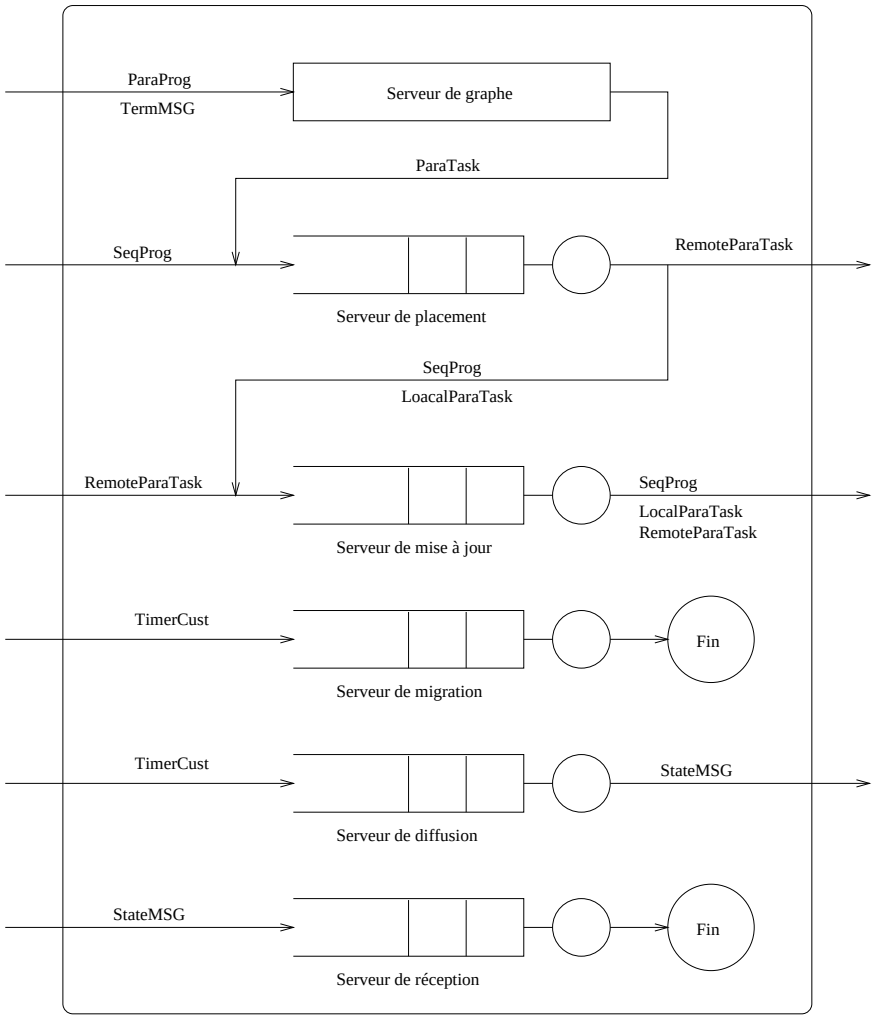

FIGURE 5.3 – Le gestionnaire de distribution

la méthode de diffusion est précisé pendant la phase de configuration. Ce serveur est constitué d'une seule file d'attente.

6. Le **serveur de réception** reçoit les messages d'états envoyés par les autres sites. Il met à jour sa vision de l'état des autres sites selon le contenu des messages reçus. Il est constitué d'une seule file d'attente.

Gestionnaire d'exécution

Le gestionnaire d'exécution est chargé d'exécuter les tâches arrivant sur le site. Il est composé de quatre sous-modules : le processeur, le disque, le serveur de messages et le serveur de fin d'exécution. Une tâche durant son exécution, est soit dans le processeur exécutant des opérations de calcul, soit bloquée dans une opération d'entrée/sortie (elle est dans la file du serveur disque) ou dans une opération de communication (elle est dans la file du serveur de messages). La tâche termine son exécution en se dirigeant vers le serveur fin d'exécution.

5.3.2 Réseau

Nous considérons le réseau comme un support logique qui réalise, d'une part, l'échange de messages entre les différentes tâches et, d'autre part, le transfert de tâches entre les différents sites. Nous distinguons trois politiques pour l'accès au réseau. Dans la première, un seul site a l'accès au réseau à un instant donné. Dans la deuxième, plusieurs sites ont le droit d'accès simultanément. Enfin dans la troisième, tous les sites ont simultanément le droit d'accès. Pour implanter ces trois politiques, nous avons utilisé une file d'attente équipée de plusieurs serveurs. Le temps de service de chaque serveur modélise le temps nécessaire pour envoyer un paquet. La longueur de la file d'attente représente la charge du réseau. Lors de la configuration du simulateur, la politique d'accès est précisée, ainsi que la distribution de temps de service et sa valeur moyenne.

5.3.3 Surcoût des algorithmes et modélisation de la charge

Il y a deux sources principales de surcoût dans un système de distribution de charge : l'exécution des algorithmes de distribution et l'échange des informations sur l'état du système.

Le surcoût introduit par l'échange des informations est pris en compte implicitement par le simulateur, car cet échange est réalisé par des transmissions et des réceptions de messages contenant les états du système. Ces messages subissent alors les différents délais des serveurs.

Le surcoût introduit par l'exécution des algorithmes de distribution est plus difficile à être simulé. La solution adoptée est basée sur l'idée que l'exécution de ces algorithmes prend une certaine capacité du processeur. Ainsi, à chaque activation d'un des serveurs du module de distribution, une tâche de priorité élevée est générée. Cette tâche, appelée *tâche système*, est toujours exécutée sur le processeur local. La distribution des durées d'exécution de chacune de ces tâches et la valeur moyenne sont précisées pendant la configuration.

Pour estimer la charge au niveau de chaque site dans le modèle, nous avons utilisé la formule du système Unix 4.4 BSD [131]. En appliquant cette formule, nous avons une estimation de la charge identique à l'estimation utilisée dans le Unix 4.4 BSD.

Application	Nombre de tâches	Durée d'une tâche
X	20	100 sec.
Y	12	1200 sec.

TABLE 5.1 – Caractéristiques des applications X et Y

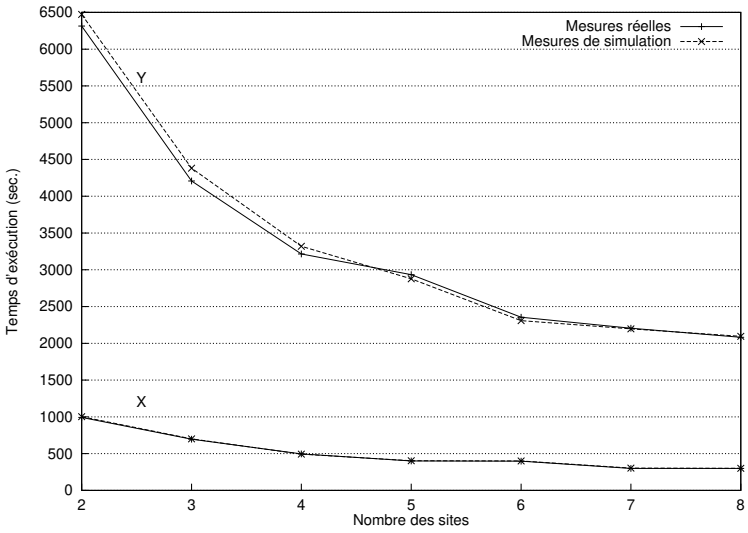

FIGURE 5.4 – Temps d'exécution de X et Y

5.4 Validation et configuration

La validation consiste à trouver les meilleures valeurs des différents paramètres du modèle pour lesquelles les résultats de simulations sont très proches des résultats du système réel. Cette phase permet de calibrer le simulateur.

La table 5.1 indique les caractéristiques de deux applications X et Y que nous avons exécutées sur le simulateur et sur le système réel. La figure 5.4 montre les résultats obtenus en fonction du nombre de sites. Nous avons observé des résultats similaires sur de nombreuses autres applications.

Nous avons également calibré le simulateur en fonction des caractéristiques physiques de l'environnement d'exécution de GatoStar à savoir un réseau Ethernet à 100 Mbit/s et un temps moyen d'accès aux disques locaux de 10 ms. De même, nous avons introduit les paramètres de surcoûts des algorithmes qui correspondent à ceux observés sur GatoStar. La différence constatée entre les mesures réelles et les mesures de simulations varie entre 5 et 10 %.

De nombreuses comparaisons ont été faites afin de vérifier le simulateur. Ces études ont confirmé le bon comportement du modèle en particulier pour les communications et les accès au disque [99]. A partir de ces résultats, nous avons estimé que ce modèle constitue une image proche du système réel et toutes les études présentées dans la section suivante ont été réalisées avec ce modèle.

Application	N	T	durée totale
A	10	100	1000 sec.
B	20	250	5000 sec.
C	30	500	15000 sec.

TABLE 5.2 – Description des applications testées

5.5 Evaluation de performances

Dans cette section nous présentons, tout d'abord, une évaluation de performances des stratégies d'allocation. Puis, nous analysons l'impact de l'environnement d'exécution en termes du nombre de machines et d'hétérogénéité sur les temps de réponse des applications.

L'objectif est non seulement d'illustrer les gains de GatoStar mais également d'extrapoler l'environnement et les applications pour d'une part définir des heuristiques de placement plus efficaces et d'autre part mettre en avant les limites en terme d'extensibilité d'un système tel que GatoStar.

5.5.1 Placement initial

Nous présentons les performances des algorithmes de placement du système de répartition en changeant les différents composants de la charge. La performance sera exprimée en gain par rapport à une exécution locale sur une machine. Le réseau simulé reproduit, en termes d'hétérogénéité de puissance, la configuration du réseau de notre laboratoire (cinq Sparcs 5 cadencées à 85 MHz, une Sparc 5 à 70 MHz, une Sparc 10 à 41 MHz et une Sparc 10 à 51 MHz).

Dans un premier temps, nous supposons que toutes les machines sont oisives et qu'aucun utilisateur n'est connecté à sa machine. Nous supposons aussi qu'une seule application de calcul intensif est lancée dans le système pour éviter l'interférence entre les applications. Cette application est composée de N tâches homogènes, chacune de durée T secondes. Il n'y a pas de contraintes de précédence entre les tâches, elles peuvent donc toutes s'exécuter en parallèle. Le tableau 5.2 montre les valeurs de N et de T pour les trois applications testées.

La figure 5.5 montre le gain pour les trois applications en fonction du nombre de sites. On remarque que le gain maximum pour l'application A est autour de 10 sites, il est atteint à partir de 15 sites car l'algorithme de placement choisit les machines les plus rapides pour placer les 10 tâches de l'application. Ce résultat est dépendant de l'hétérogénéité du réseau simulé : dans notre configuration au-dessus de 10 machines le nombre de machines puissantes reste faible. Une étude plus détaillée sur l'impact de l'hétérogénéité est présentée en 5.5.5. Le gain maximum pour l'application B est autour de 19, il est atteint pour 30 sites et celui de l'application C est autour de 21, il est atteint à partir de 30 sites.

On remarque aussi une diminution des performances lorsque le nombre de sites dépasse un certain nombre ; ceci est principalement dû au surcoût des algorithmes de distribution. Ainsi, les algorithmes de placement ont une efficacité optimum si le nombre de machines du réseau est inférieur à 25 pour l'application A, à 30 pour B et à 35 pour C. Nous en déduisons que

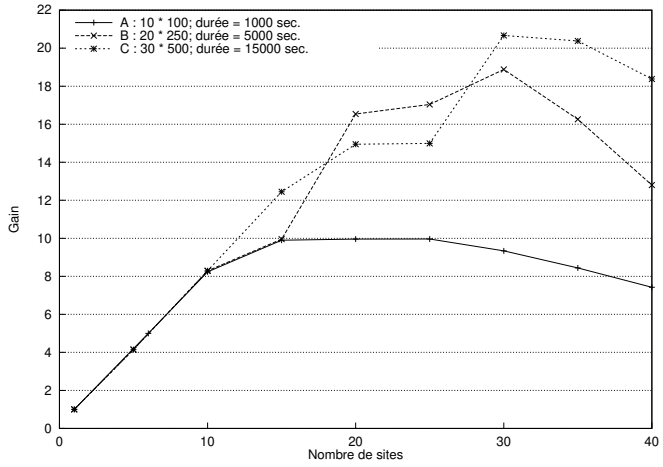

FIGURE 5.5 – Gain de l'algorithme de placement

la limite d'efficacité de ces algorithmes dépend de la configuration de charge à exécuter et de la configuration de l'environnement. Ainsi, des informations sur le comportement des applications (nombre de tâches, durées, ...) peuvent améliorer les performances des algorithmes en permettant d'ajuster dynamiquement leurs paramètres.

5.5.2 Influence de la charge locale

Maintenant, nous réalisons une nouvelle série de mesures en ajoutant une charge locale à chaque machine. Cette charge locale simule la charge induite par l'utilisateur local de la machine.

Pour simplifier la configuration de la charge locale de chaque machine nous supposons que l'arrivée des programmes est poissonnienne de taux λ et le temps d'exécution d'un programme est de distribution exponentielle de moyenne μ. Nous avons estimé que le paramètre le plus significatif est la charge locale moyenne et non la loi d'arrivée. λ et μ sont des paramètres. Nous étudions les performances en fonction du niveau de la charge locale. Donc, pour des valeurs faibles de $\rho = \frac{\lambda}{\mu}$ les machines sont peu chargées par leurs utilisateurs ; pour des valeurs de ρ proches de l'unité, le système devient très chargé. Nous imposons que les tâches générées par les utilisateurs locaux soient prioritaires par rapport aux tâches appartenant aux programmes parallèles, ainsi l'utilisateur sera peu gêné par les tâches étrangères. Le tableau 5.3 détaille les trois applications testées.

La figure 5.6 montre le gain de performances des trois applications parallèles exécutées sur un réseau de 4 machines en fonction de niveau de la charge locale. On remarque que le placement des tâches parallèles reste rentable jusqu'à une certaine valeur de la charge locale que nous appelons *niveau local maximum* (1,2 pour l'application $A0$, 1,5 pour l'application $A1$ et 1,7 pour l'application $A2$). On remarque que plus le temps d'exécution est grand, plus le niveau maximum est grand.

Nous avons également étudié l'impact du nombre de machines et remarqué que plus le

Application	N	$T(sec.)$
$A0$	20	10
$A1$	20	50
$A2$	20	250

TABLE 5.3 – Valeurs de N et de T des trois applications testées

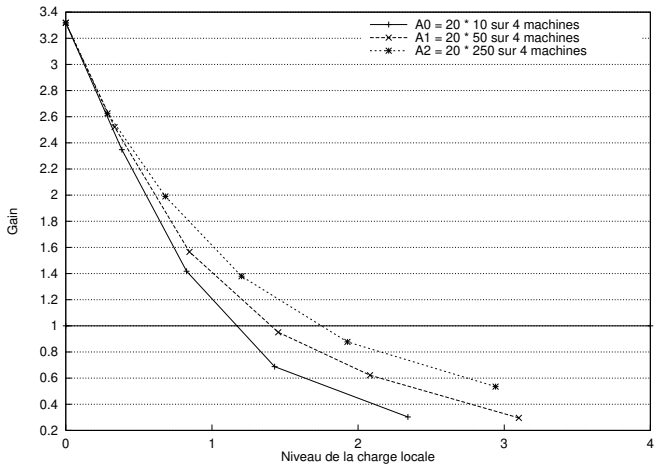

FIGURE 5.6 – Dépendance de la charge locale

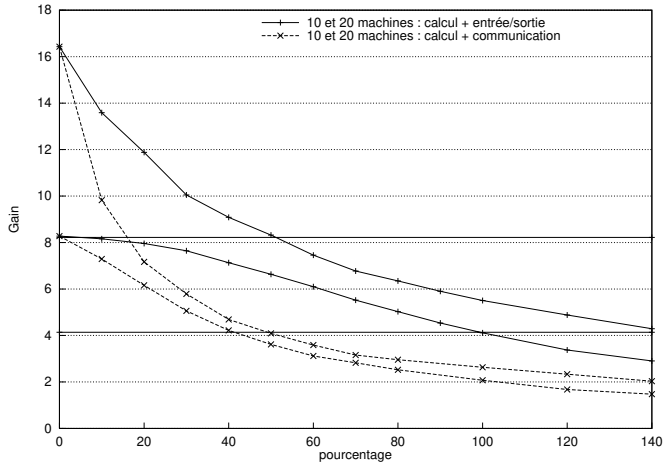

FIGURE 5.7 – Impact des communications et entrées/sorties

nombre de machines est important, plus le placement reste efficace dans des conditions de charge élevées.

Nous déduisons de ces deux études que le niveau maximum de la charge locale dépend à la fois de l'application et de l'environnement. Savoir caractériser le niveau maximum pour une application donnée et pour un réseau donné est très utile pour les algorithmes de placement, car au-delà de ce niveau, on sait que les placements distants sont inutiles et qu'il est préférable d'exécuter localement les tâches parallèles ou de les retarder.

5.5.3 Influence des communications et entrées/sorties

Pour comparer à la fois l'influence des opérations de communications et des opérations d'entrée/sortie, nous avons choisi deux applications chacune constituée de 20 tâches. Chaque tâche de la première application est composée d'une partie calcul et d'une partie entrée/sortie ; chaque tâche de la deuxième application est composée d'une partie calcul et d'une partie communication. Les deux applications ont été exécutées sur un réseau de 10 et 20 machines.

La figure 5.7 montre le gain de performances en fonction du pourcentage d'entrée/sortie pour la première application et du pourcentage de communication pour la seconde. On remarque que les gains de la première application sont plus élevés que ceux de la seconde ce qui indique que les communications ont une influence plus forte sur les performances que les entrées/sorties. Par exemple, le gain est divisé par 2 à partir de 50 % d'E/S, et à partir seulement de 20 % de communications. Nous en déduisons pour les algorithmes de placement que le critère de communication est prioritaire par rapport au critère d'entrée/sortie.

5.5.4 Migration

Nous examinons maintenant le gain supplémentaire apporté par la migration par rapport au placement. Nous avons considéré les trois applications homogènes A, B et C présentées dans la table 5.2.

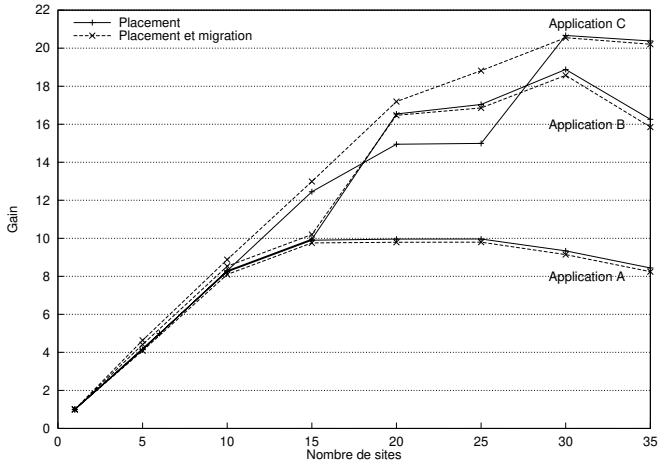

FIGURE 5.8 – Gain de migration pour les applications homogènes

La figure 5.8 montre la comparaison des gains entre une stratégie de placement seul et une autre combinant placement et migration. On remarque que les performances sont quasi-identiques pour A et B (en faveur du placement seul à cause du surcoût supplémentaire de la migration). Cela s'explique car, dans ce cas, le placement effectué est proche du placement optimal. A cause de l'homogénéité des tâches et des différences mineures de puissance des machines, le transfert de tâches vers les machines rapides libérées ne suffit pas pour compenser le surcoût de la migration.

L'application C, qui est composée de 30 tâches, se comporte différemment entre 15 et 30 machines. Par exemple, pour 20 machines, le placement initial est le suivant : deux tâches sur les dix machines les plus rapides et une tâche sur les dix autres. Après un certain temps, les 10 dernières tâches terminent leur exécution, à ce moment la migration d'une des deux tâches des machines rapides vers une machine plus lente mais inutilisée améliore fortement les performances. Le même raisonnement est applicable dans le cas de 25 machines. Le gain supplémentaire pour 20 machines est de l'ordre de 15 %, et de 26 % pour 25 machines. Le gain supplémentaire dans le cas de 25 machines est justifié car les 5 machines recevant les tâches transférées sont les 5 machines les plus rapides.

D'autres études ont été faites pour des applications irrégulières avec des tâches de durée variable. On constate là aussi des gains importants dans certaines configurations dus au même phénomène observé pour l'application C. Des résultats précédents, nous caractérisons les deux cas pour lesquels la migration peut ajouter un gain supplémentaire conséquent :

— Les tâches sont homogènes mais la structure de l'application et le nombre des sites disponibles amènent forcement à un état de déséquilibre (par exemple, l'application C avec 20 et 25 sites).

— il existe un degré élevé d'hétérogénéité de temps d'exécution des tâches pouvant générer des états de déséquilibre dans le système.

Dans les deux cas précédents, et selon tous les tests effectués sur la migration nous n'avons pas pu observer un gain supplémentaire dépassant 35 % (de nombreuses autres distributions

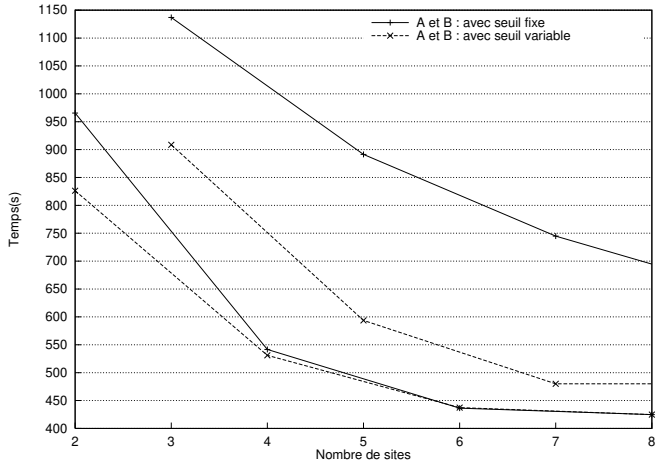

FIGURE 5.9 – Simulation de temps de réponse selon la politique de seuil

mathématiques ont été testées avec différents degrés d'hétérogénéité). Nous avons également mesuré les temps de réponses en appliquant une stratégie de migration seule (sans placement initial) comme cela a été fait dans le système MOSIX [31]. Dans tous les cas, les gains de performances observés étaient inférieurs aux deux stratégies présentées.

5.5.4.1 Adaptabilité des algorithmes de distribution

Nous montrons dans cette partie, la nécessité d'adaptation des algorithmes de distribution de charge. Notre étude porte sur la valeur des seuils qui sont à la base des algorithmes de prise de décision.

Nous avons simulé deux stratégies de seuil. Dans la première stratégie, implémentée dans GatoStar, l'allocation et la migration sont basées sur des seuils fixes. Dans la deuxième, les seuils sont adaptatifs et dépendent de la charge globale du système. Les différents seuils utilisés sont définis de la manière suivante :

$$Seuil\ de\ charge = charge\ moyenne$$
$$Seuil\ de\ reception = (1 - FacteurCharge) * charge\ moyenne$$
$$Seuil\ de\ migration = (1 + FacteurCharge) * charge\ moyenne + DELTA - CHARGE$$

où $FacteurCharge$ est expérimentalement fixé à 0.4. $DELTA-CHARGE$ est une constante qui garantit un ecart minimum entre les seuils. La charge moyenne globale est calculée périodiquement avec la formule suivante :

$$charge\ moyenne = \frac{\sum_{i=1}^{N_p} \frac{LoadAvg_i}{vitesse_i}}{Nombre\ de\ machines}$$

où $LoadAvg_i$ est la charge moyenne de la machine i. Dans cette politique, les valeurs des seuils suivent toujours la charge moyenne du système.

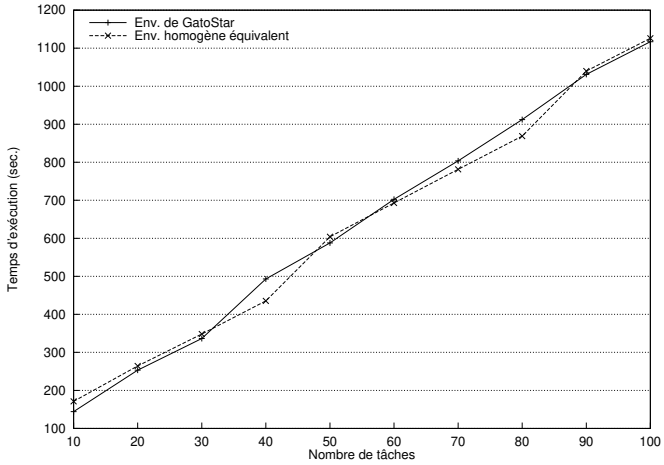

FIGURE 5.10 – GatoStar sur un environnement homogène

La figure 5.9 présente une comparaison de ces deux stratégies pour deux applications parallèles : l'application A est composée de 40 tâches identiques, l'application B est composée de 48 tâches de durées différentes (24 tâches longues et 24 tâches courtes). On remarque que la stratégie adaptative à seuils variables améliore le temps de réponse du système particulièrement pour les applications hétérogènes (jusqu'à 30 % pour l'application B).

A partir de ces résultats, la stratégie des seuils adaptatifs a été intégrée dans le système GatoStar. Des améliorations importantes des performances ont été effectivement observées sur des applications réelles.

5.5.5 Hétérogénéité du réseau

Pour étudier l'influence de l'hétérogénéité sur les performances, nous avons introduit la notion de vecteur de puissance. Le vecteur de puissance d'un réseau de M machines est un vecteur de M composantes où la composante i représente la vitesse de la machine numéro i. Par exemple, le vecteur de puissance pour l'environnement du laboratoire est :

$$G = (1.205, 1.205, 1.205, 1.000, 1.205, 1.024, 1.205, 1.265)$$

Nous exprimons la puissance d'un réseau en définissant la norme donnée par :

$$\|P\| = \sum_{i=1}^{M} p_i$$

où p_i est la vitesse de la machine i. On dit que deux réseaux sont équivalents si leurs vecteurs de puissance ont la même norme.

Nous avons comparé les performances du système GatoStar dans deux environnements : l'environnement du laboratoire et l'environnement homogène équivalent. Nous considérons une application homogène de N tâches de durée T secondes. La figure 5.10 montre le temps

d'exécution en fonction de N. On remarque que la différence entre les performances de l'environnement du laboratoire et l'environnement homogène équivalent est négligeable. En réalisant des tests sur d'autres configurations [98], nous avons observé la même tendance. Nous en avons déduit que la norme définie ci-dessus peut être utilisée pour mesurer la puissance du réseau.

5.6 Conclusions

Nous avons présenté un modèle paramétrable et configurable de système de distribution de charge dans un réseau de stations de travail. Le modèle a été construit à partir du système de placement GatoStar. L'avantage principal de notre approche, par rapport à d'autres modélisations, est de se rapprocher le plus finement possible des mesures observées dans un système réel de distribution de charge. Le travail présenté a nécessité des raffinements successifs des paramètres du modèle et du modèle lui-même jusqu'à obtenir des performances par simulation quasiment identiques à celles obtenues par le système GatoStar.

Nous avons mis en évidence l'efficacité des algorithmes de placement notamment pour les applications homogènes. Nous avons aussi étudié les performances des algorithmes de placement sur un réseau chargé par des utilisateurs locaux et analysés l'influence des opérations d'entrées/sorties et des communications sur les performances. Nous avons ainsi trouvé qu'un critère basé sur la communication est plus prioritaire qu'un critère basé sur les entrées/sorties.

Nous avons également mis en évidence le surcoût et le gain apporté par de la migration par rapport aux algorithmes de placement. Deux points essentiels concernant ces algorithmes. Le premier est de savoir caractériser les applications où la migration peut être rentable. Le deuxième est de pouvoir détecter, pendant l'exécution de ces applications, le moment où il faut déclencher la migration.

Enfin, nous avons montré que l'efficacité d'un algorithme de distribution dépend en grande partie de sa capacité à s'adapter en fonction de la charge du système et de l'environnement d'exécution.

Cette étude a permis d'améliorer les performances du système GatoStar en y intégrant avec succès la stratégie à seuils variables, conçue avec SIGAP. Le simulateur nous a également permis de tester GatoStar dans des conditions plus extrêmes (forte charge, nombre de machines élevé). Il a permis ainsi de montrer la limite du système (au delà de 35 sites). Mais SIGAP n'est pas uniquement un outil de mise au point de GatoStar, c'est un outil générique qui permet de tester et améliorer les systèmes de distribution. Une perspective directe de ce travail et de simuler d'autres systèmes de répartition comme MOSIX [31], LSF [149] ou UTOPIA [207].

Troisième partie

Gestion de la mémoire : Optimisation et Support d'exécution extensible

Chapitre 6

Pagination en mémoire distante : Maîs

Sommaire

6.1 Introduction

La mémoire virtuelle utilise traditionnellement les disques comme support de pagination, mais ceux-ci sont lents. Un défaut de page demande de l'ordre du million de cycles processeur, et cela va en empirant : la vitesse des processeurs augmente très rapidement, alors que les disques stagnent. D'autre part, la hiérarchie des temps d'accès a changé : les réseaux sont maintenant bien plus rapides que les disques. De plus, les logiciels de gestion des échanges sur les réseaux ont atteint une maturité comparable aux systèmes de gestion de fichiers. L'idée de la pagination en mémoire distante (PMD) est donc de substituer la mémoire physique des

machines du réseau aux disques durs en tant qu'espace de *swap*. Ainsi la latence est plus faible et le débit plus élevé [59].

Des mesures sur l'utilisation des stations de travail ont montré que l'utilisation de la mémoire physique à des fins de pagination dans les réseaux de stations était possible sans gêner les utilisateurs. La mémoire est en effet globalement sous-utilisée : plus de 60 % des stations sont disponibles 100 % du temps [7] et rarement plus de 50 % de la mémoire totale est utilisée [128], même aux heures de travail les plus intenses.

Ainsi la pagination distante est une alternative séduisante pour augmenter les performances des systèmes. Des expériences menées notamment à l'Université de Washington [76] montrent que même sur un réseau Ethernet à 10Mbit/s la pagination en mémoire distante est plus performante que celles sur disque.

A partir de ces constats, nous avons conçu un nouveau système de pagination distante, Maîs. Maîs a pour objectif d'adapter les stratégies de swap aux caractéristiques des réseaux. En effet, les systèmes de pagination sont optimisés pour les échanges sur disques cherchant par exemple la contiguïté sur disque pour optimiser les déplacements de tête ou limiter les interruptions des contrôleurs DMA. Pour décharger et recharger efficacement des pages en mémoires distantes, il faut reconcevoir ces stratégies en cherchant à répartir la charge mémoire entre les machines ou en introduisant plus d'asynchronisme entre le système et les périphériques de swap (i.e., les machines distantes) qui désormais peuvent jouer un rôle actif pour élaborer des stratégies de pré-chargement.

Dans ce sens, Maîs se distingue des autres projets de recherches comme GMS ou le système développé par Markatos et Draminitinos [127] en faisant intervenir activement non seulement les machines "clientes" qui exécutent les applications mais également les serveurs de pages qui cherchent à anticiper les futures demandes de pages.

Maîs est un projet qui a fait l'objet de plusieurs stages de DEA et de la thèse de Philippe Cadinot. Dans une approche incrémentale, trois versions ont été réalisées dans le système FreeBSD :

— Maîs-1 intègre dans le noyau un nouveau paginateur permettant de "swapper" automatiquement dans la mémoire des serveurs distants mais sans changer les heuristiques de swap de BSD.
— Maîs-2 modifie les algorithmes de déchargement et chargement de page pour tirer profit du réseau en introduisant plus d'asynchronisme pour réduire la latence des défauts de page.
— Maîs-3 intègre une pagination de second niveau et conçoit des stratégies de pré-chargement pour réduire le nombre de défauts de pages.

Les différentes versions de Maîs ont été conçues dans le cadre du projet Multi-PC du département architecture du LIP6. Dans cette architecture, des cartes PC standards sont reliées par un réseau haut-débit. Ce type de support est particulièrement favorable à l'implantation de stratégies de swap réparties.

La section 6.2 présente le projet Multi-PC. Les sections 6.3, 6.4 et 6.5 présentent les trois évolutions de Maîs. La section 6.6 présente les performances obtenues et la section 6.7 compare nos travaux aux autres projets.

6.2 L'architecture Multi-PC

La plate-forme cible est la machine Multi-PC (MPC) construite au laboratoire LIP6 de l'Université Pierre & Marie Curie.

Le projet MPC a pour but de construire une machine parallèle performante de type grappe de stations de travail à faible coût en utilisant les composants standards du monde PC : cartes mères, processeurs Pentium, bus PCI, etc. Le système d'exploitation est FreeBSD, un UNIX gratuit descendant de 4.4BSD [132].

La seule composante spécifique de cette architecture est le réseau à haut débit HSL (*High Speed Link*). HSL a une faible latence (5 μs) et utilise des liaisons séries point-à-point bi-directionnelles à 1 Gbit/s. Sur une carte HSL est placé un routeur RCube 8×8 à routage *wormhole* [158].

Le réseau HSL fournit une primitive de communication optimisée : l'écriture distante. Le contrôleur de réseau se comporte comme un contrôleur DMA où la lecture des données est effectuée par le contrôleur de réseau du nœud local, et où l'écriture des données est effectuée par le contrôleur de réseau du nœud distant. Les données sont découpées en paquets par le matériel et celui-ci effectue un comptage en réception pour signaler la fin de la transmission par une interruption si tous les paquets ont été correctement reçus. Du point de vue logiciel, l'écriture distante est fournie par la couche PUT, qui est l'API de plus bas niveau. RCube (Rapid Reconfigurable Router) est un router à très faible latence (150 nano-seconde sans contention). Il inclut 8 liens bidirectionnels ayant chacun un débit de 1Gigabit/s soit un débit global de 640 Megaoctets/s. Les tables de routage sont programmables. Son architecture à base de "crossbar" 8x8, permet à RCube de faire transiter les paquets depuis n'importe lien d'entrée vers un lien de sortie quelconque.

Le faible rapport entre performance et coût rend très attractif ce type d'architecture qui présente une réelle alternative aux super-calculateurs. De nombreux projets proposent de relier des cartes standards par un réseau à haut-débit tel que Ethenet 1Gbit, ATM, SCI ou Myrinet. Ainsi, l'équipe "Réseaux à Haut Débit" de l'Université de Lyon 1 a développé une plate-forme de PC interconnectés par Myrinet et un protocole optimisé, BIP. [154]. Ce protocole permet aux applications d'échanger des données avec un débit d'un Gigabit/s. Les "Actives Messages" (AM)[198] proposés initialement par l'Université de Berkeley fournissent des briques de bases pour développer des protocoles réseaux optimisés de haut niveau. Les AM ont été réalisés sur des plates-formes Myrinet et un réseau ATM . Les "Fast Messages" (FM) [144] de l'Université d'Illinois Urbana Champaign proposent une approche similaire aux "actives messages". Le projet U-net [202]de l'Université de Cornell a une approche originale en permettant de développer des protocoles de communication dans l'espace utilisateur évitant ainsi aux messages la traversée des couches systèmes.

La pagination en mémoire distante, qui a déjà sur Ethernet 10 Mbits/s des performances supérieures à la pagination sur disque [59, 79, 163, 34], bénéficie grandement des caractéristiques de ce type de réseau. Le haut débit permet un transfert rapide des pages, et la faible latence permet le transfert efficace de petits messages de contrôle.

6.3 Intégration d'un paginateur réparti : Mais-1

Maîs-1 est un paginateur réparti développé par Philppe Cadinot et Neilze Dorta pour la machine MPC [49, 50].

6.3.1 Architecture

Maîs utilise la mémoire physique des stations inactives comme espace de *swap* via un paginateur spécial, grâce à la possibilité dans FreeBSD de définir (relativement) facilement de nouveaux paginateurs dans le noyau.

Un processus voulant profiter de la pagination distante doit s'enregistrer explicitement auprès de Maîs et la pagination à distance est ensuite totalement transparente. L'enregistrement explicite permet d'éviter que les processus systèmes ne l'utilisent, ce qui rendrait le noyau sensible aux pannes des serveurs.

Maîs-1 est constitué des composants suivants (Figure 6.1) :

— Un *driver* de périphérique, `/dev/mais`, chargeable dynamiquement et contenant le code du paginateur Maîs . Ce *driver* communique avec le *driver* HSL pour transférer les données entre machines.

— Un démon qui exécute le code du serveur. Il traite les opérations d'éviction et de chargement de pages.

Les serveurs réservent pour chacun de leurs clients un *pool* de pages physiques dans lesquelles le démon de pagination (le "pagedaemon" de FreeBSD) du nœud client évince les pages locales. Les pages de *pool* sont renouvelées au fur et à mesure de leur consommation par le client.

6.3.2 Performances

Un défaut de page traité par Maîs-1 ne coûte que de 0,463 ms (pour 1 page) à 75 ms (pour 10×16 pages) alors que la pagination classique sur disque de FreeBSD coûte normalement de 12,2 ms (1 page) à 117,5 ms (10×16 pages), soit une accélération allant de 26,35 à 1,57. De même, une éviction par Maîs-1 prend de 5,8 ms (pour 10×1 pages) à 58,7 ms (pour 10×16 pages), à comparer avec 13,4 ms et 135,4 ms respectivement, dans le cas de la pagination classique, soit une accélération de 2,3 [49].

Maîs-1 obtient donc de très bonnes performances mais son implémentation n'exploite pas pleinement les possibilités offertes par HSL :

— La moitié de la latence système est due à la traversée des couches de communication au-dessus de laquelle est implémentée Maîs-1 [49].

— Maîs-1 est tributaire d'algorithmes d'éviction et de chargement des pages qui ont été pensés pour la pagination sur disque : le traitement des défauts de page est synchrone.

Les objectifs de Maîs-2 ont donc été définis afin de lever ces limitations :

— portage sur la couche basse de communication PUT de MPC pour augmenter significativement les performances en diminuant la latence système,

— défaut de page asynchrone, pour utiliser pleinement les possibilités de la pagination à travers le réseau,

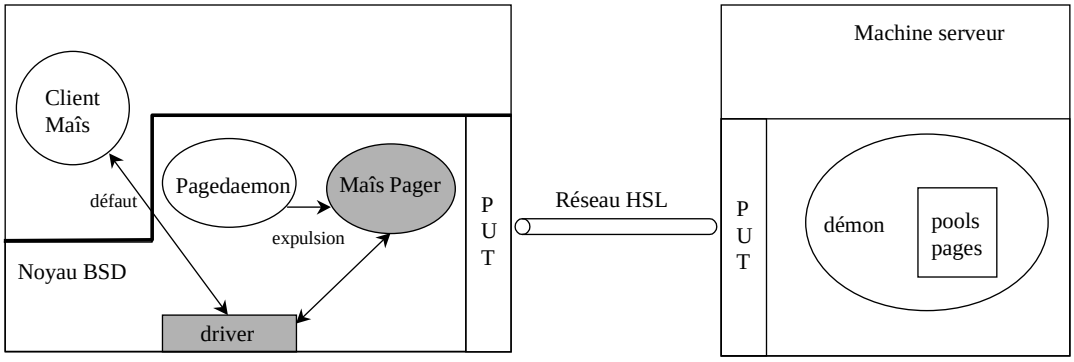

FIGURE 6.1 – Architecture de Maîs.

6.4 Adaptation à l'infrastructure réseau : Maîs-2

Le défaut de page classique de la mémoire virtuelle a été pensé pour la pagination sur disque, qui sérialise les accès et cherche à optimiser ceux-ci en ne chargeant que les pages contiguës sur le disque. Le défaut de page est donc synchrone : l'application suspend son exécution lors du défaut de page et ne la reprend que quand toutes les pages ont été lues, y compris les pages pré-chargées. Ce synchronisme n'exploite pas les possibilités de recouvrement calcul / communication offertes par le réseau HSL : l'écriture des pages en mémoire physique est prise en charge par la carte réseau et l'application peut ainsi reprendre son exécution dès réception de la page fautive, les pages pré-chargées étant écrites en mémoire parallèlement, sans interruption de l'application. L'objectif de Maîs-2 est donc d'exploiter cette caractéristique. Maîs-2 a été développé par Damien Collard et Philippe Cadinot [47].

Afin de bien comprendre le déroulement d'un défaut de page dans Maîs-2, la section 6.4.1 présente le mécanisme et la politique d'éviction de Maîs-2. La section 6.4.2 décrit le défaut de page asynchrone de Maîs-2.

6.4.1 Eviction

Dans FreeBSD, l'éviction se fait par groupes de 8 pages maximum (ou *clusters*). La sélection des sites de réception de ces pages est faite par le paginateur Maîs par tourniquet (*round robin*). Un bloc d'éviction est donc généralement réparti sur plusieurs serveurs. Le tourniquet est préférable au remplissage successif des mémoires des serveurs car il permet :

— un équilibrage des charges mémoires en répartissant équitablement les pages,

— un certain parallélisme (limité par l'émission des messages en série sur le lien de sortie HSL) lors de l'éviction de groupes (ou *clusters*) de pages,
— l'implémentation de la tolérance aux fautes, voir la section 6.5.3.

L'éviction se termine quand les acquittements de toutes les pages évincées ont été reçus. Par rapport à la pagination sur disque, l'éviction d'un *cluster* de pages est plus rapide car, les pages étant évincées sur différents serveurs, les évictions sont quasi-simultanées, et les acquittements sont reçus au bout d'un temps proche du temps nécessaire à l'éviction d'une seule page (dans le meilleur cas, *i.e.* quand il y a au moins autant de serveurs que de pages évincées).

La répartition des pages évincées est aussi avantageuse au moment du pré-chargement, car les pages sont pré-chargées quasiment en parallèle. Le temps de pré-chargement est donc sensiblement égal au temps de réception d'une seule page.

6.4.2 Défaut de page asynchrone

Le nombre de pages pré-chargées par FreeBSD 3.0 est fixé à 8 dans le noyau (y compris la page fautive). FreeBSD permet d'adapter la fenêtre de pré-chargement en fonction du type d'accès réalisé sur la page :

— Normal : 3 pages sont pré-chargées "en arrière" de la page fautive et 4 "en avant"
— Séquentiel : zéro en arrière, 7 en avant
— Aléatoire : aucune page pré-chargée.

Ces chiffres sont des maxima : s'il n'est pas possible d'allouer suffisamment de cadres de pages ou si les pages à pré-charger n'existent pas ou n'appartiennent pas au même objet que la page fautive, elles ne sont pas pré-chargées. De plus, l'adaptation de la fenêtre n'est pas dynamique : le type de fenêtre est déterminé à la création de l'objet.

Afin de régler dynamiquement ces paramètres nous avons court-circuité le mécanisme de pré-chargement de FreeBSD. Désormais, le paginateur contrôle le pré-chargement.

Pour récupérer les pages pré-chargées, Maîs-2 maintient un *pool* de *buffers* de pré-chargement. Lorsqu'un processus P subit un défaut de page sur la page p, seule p est demandée au paginateur (Figure 6.2).

Soit d_1 ce défaut de page. Le paginateur recherche la page (1) dans les *buffers*, puis (2) sur les serveurs si elle n'a pas été trouvée en (1). Pendant que l'application traite la page p, le paginateur demande aux serveurs le pré-chargement de pages dans ses buffers. Un accès subséquent à une page pré-chargée provoque un nouveau défaut de page d_2. Toutes les pages qui sont du "même côté" que la page fautive par rapport à la page qui avait provoqué le défaut de page d_1 sont alors "mappées" dans l'espace d'adressage du processus. Il faut ensuite allouer de nouveaux *buffers*, une partie du *pool* étant maintenant "mappée" dans l'espace d'adressage du processus fautif.

Cette approche induit un nombre supérieur de défauts de page : 2 au lieu de 1 pour un défaut de page standard. Une autre solution était de mapper les pages pré-chargées dans l'espace d'adressage du processus dès leur réception, ce qui n'augmente pas le nombre de défauts de page. Le but du mapping lors du second défaut de page était d'éviter que l'on ne mappe des pages pour rien, qui deviendraient alors des pages actives, et provoqueraient par la suite l'éviction de pages plus utilisées.

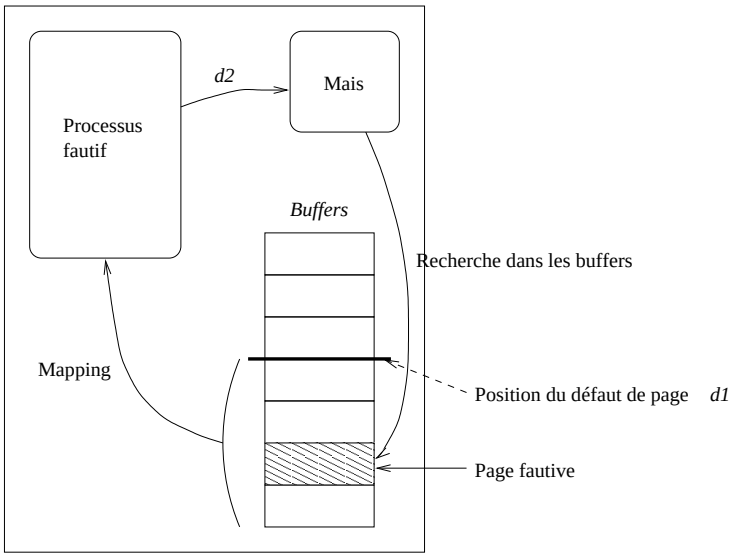

FIGURE 6.2 – Défaut de page – page trouvée dans les *buffers*.

Le pré-chargement est accompli de manière totalement asynchrone par le démon Maîs.

Bien que Maîs-2 puisse utiliser toute la mémoire physique présente dans le réseau, cela est insuffisant pour certaines très grosses applications qui ont besoin de beaucoup plus de mémoire. Les objectifs de Maîs-3 ont donc été définis afin de lever ces limitations en offrant un second niveau de *swap* (sur disque), pour exécuter des applications de très grande taille.

6.5 Extension du paginateur : Maîs-3

L'utilisation de la mémoire distante permet d'exécuter des programmes qui nécessitent plus de mémoire que n'en possède physiquement la machine sur laquelle ils s'exécutent, mais dans le cas de très grosses applications, cela peut s'avérer insuffisant. Par exemple, dans le thème Calcul Formel du LIP6, J.C. Faugère a développé une application qui peut résoudre des problèmes qui nécessiterait plus de 1Go de RAM.

L'idée de Maîs-3 est donc d'utiliser non seulement la mémoire distante mais aussi les disques distants [59, 79, 163, 105]. La mémoire distante peut être vue comme un cache s'interposant entre la mémoire physique du client et les disques des serveurs : lorsqu'un serveur de mémoire n'a plus de place en mémoire physique, il évince ses plus vieilles pages sur son disque.

Maîs-3 cherche également à introduire plus de souplesse en autorisant les machines à basculer dynamiquement d'une fonction de serveur à celle de client. Ceci permet d'adapter le nombre de serveurs dédiés aux besoins applicatifs. Nous cherchons également à introduire les techniques de tolérance aux fautes en répliquant les pages entre serveur. Maîs-3 est en cours de développement.

6.5.1 Eviction

Un serveur évince sur son disque local un certain nombre de ses plus vieilles pages lorsqu'il va être à court de mémoire. On ne cherche pas à faire du remplacement LRU global à tous les serveurs comme dans GMS [76]. Cette éviction peut être faite de manière transparente (pour Maîs) ou de façon explicite par le serveur Maîs. Dans le cas de l'éviction implicite, les pages d'un processus serveur Maîs sont évincées par le mécanisme de mémoire virtuelle de FreeBSD. L'éviction est dans ce cas incontrôlable par Maîs ce qui peut poser des problèmes d'efficacité.

En effet, pour effectuer le pré-chargement de pages à partir du disque, le processus serveur doit faire une lecture dans chacune de ces pages. Ainsi, ces pages seront rechargées en mémoire physique par la mémoire virtuelle. Un inconvénient est que la mémoire virtuelle risque de recharger d'autres pages autour de chacune des pages fautives (au maximum 3 en arrière et 4 en avant), or ces pages sont pré-chargées d'après le schéma d'accès du processus serveur (local), pas du processus client (distant). Mais d'une part, ces pages sont susceptibles d'être celles désirées et d'autre part, la mémoire de la machine serveur étant très sollicitée il est probable que la mémoire virtuelle ne trouve pas suffisamment de pages libres et ne charge donc aucune page autour de la page fautive (elle réveillera quand même le démon *pageout*).

Dans le cas de l'éviction explicite, les pages du processus serveur sont verrouillées pour ne pas être évincées par la mémoire virtuelle. Le serveur doit gérer lui-même l'éviction de ses pages. Le serveur est donc plus complexe que dans le cas de l'éviction implicite, et une limitation évidente est qu'un processus dans BSD ne peut verrouiller plus d'un tiers des pages physiques [132], d'où une sous-utilisation de la mémoire.

Compte-tenu de la complexité supérieure de l'éviction explicite et des limitations imposées par BSD, la solution choisie a été de construire un serveur à éviction implicite. Cela a des conséquences sur les performances mais constitue une première étape. Dans un deuxième temps, s'il est possible de modifier la limite des 1/3 de mémoire verrouillable, on pourra implémenter l'éviction explicite.

6.5.2 Pré-chargement adaptatif

Le fait qu'un serveur évince ses pages sur son propre disque fait qu'un défaut du client peut demander un temps de traitement plus long que s'il n'utilisait pas les services de Maîs : si la page fautive est sur le disque d'un serveur, le temps de rechargement est la somme des temps réseau et temps disque. Cependant, des techniques de pré-chargement sur le serveur devraient permettre de réduire significativement la probabilité d'apparitions de tels cas.

Le serveur ne maintient pas de *buffer* de pré-chargement comme le client. Il évince ses plus vieilles pages pour les remplacer par les pages pré-chargées pour le client, qui deviennent des pages jeunes.

Sur un client, on ne doit pas pré-charger trop de pages car cela consomme de la mémoire et on risque de pré-charger les mauvaises pages. Sur un serveur, en revanche, on peut se permettre une politique plus agressive et pré-charger plus de pages, le coût du pré-chargement étant nul pour le client, puisqu'il n'est pas impliqué dans l'opération. La fenêtre de pré-chargement du serveur est donc plus large que celle du client, ce qui permet d'augmenter la probabilité que le prochain défaut de page tombe dans la fenêtre pré-chargée. On peut soit opter pour

une politique statique du type de celle de la mémoire virtuelle de FreeBSD soit une politique dynamique avec fenêtre adaptative.

— *Politique statique.* Comme dans FreeBSD, on pré-charge un certain nombre de pages en arrière de la page fautive et un certain nombre en avant. La position de la page fautive ainsi que la taille de la fenêtre sont fixes. C'est une politique simple mais qui peut s'avérer insuffisante dans le cas d'accès non "normaux" aux pages

— *Politique dynamique.* Deux politiques dynamiques (combinables) sont envisageables. La première est d'adapter la taille de la fenêtre, la deuxième d'adapter la position de la page fautive dans la fenêtre de pré-chargement (figure 6.3) .

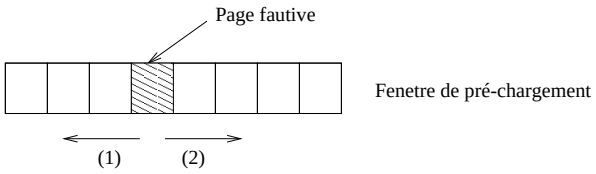

FIGURE 6.3 – Fenêtre de pré-chargement adaptative. Si les défauts de page sont séquentiels croissants, la page fautive est déplacée en arrière (1). S'ils sont séquentiels décroissants, la page fautive est déplacée en avant (2).

Dans un premier temps, nous travaillons sur une approche statique en faisant varier différents paramètres afin de comprendre le comportement des applications. Basés sur cette étude, notre objectif est ensuite de développer une approche dynamique.

6.5.3 Tolérance aux fautes

Les fautes sont un point sensible dans les PMD : la probabilité de défaillance d'une application cliente de Maîs est proportionnelle au nombre de serveurs sur lesquels elle évince ses pages. Afin de réduire cette probabilité nous envisageons d'appliquer les deux techniques suivantes :

— *la réplication des pages* : cette technique consomme d'importantes ressources mémoires et peut limiter fortement les capacité des serveurs, elle permet cependant de tolérer une ou plusieurs fautes.

— *le calcul des pages de parité par* clusters *de pages* : cette technique est peu coûteuse et permet de tolérer la panne d'un serveur

6.5.4 Insertion et retrait dynamique de serveurs

Dans Maîs-2 les serveurs sont déterminés statiquement. Dans Maîs-3, pour prévoyons de faire passer un nœud du réseau du statut client ou du statut “hors-jeu” au statut de serveur actif et vice-versa, en cas de défaillance ou de disponibilité d'une station précédemment occupée à d'autres tâches. Le retour d'un utilisateur sur sa station doit aussi provoquer la migration des pages hébergées par celle-ci vers d'autres serveurs, et ce, de la façon la plus transparente possible pour l'utilisateur.

TABLE 6.1 – Temps d'exécution de la fonction *getpages*.

Nb. de pages	Défaut de page Std.(ms)	MAÎS(ms) : HSL+Client+Serveur		Gain Maîs	Max.
1 (Maîs V2)	12,20	0,349	(0,113+0,094+0,142)	34,96	107,96
1	12,20	0,49	(0,113+0,22+0,16)	24,80	107,96
10x1	24,30	4,92	(1,13+2,20+1,59)	4,94	21,50
10x2	33,34	9,26	(2,26+4,00+3,00)	3,60	14,75
10x4	43,37	17,03	(4,52+6,05+6,46)	2,55	9,59
10x8	78,78	32,42	(9,04+10,48+12,90)	2,43	8,71
10x16	117,49	62,49	(18,08+19,19+25,22)	1,88	6,50

6.6 Performances

6.6.1 Plate-forme d'expérimentation

Maîs a été développé sur une plate-forme de test composée de PC Pentium 150 Mhz interconnectés par un concentrateur Ethernet 100 Base T (100 Mbit/s). Nous y avons ajouté un émulateur du protocole d'écriture à distance de la machine MPC.

Nous avons remplacé le temps de transfert réseau sur le réseau 100 Mbit/s (qui n'est pas significatif étant donné la présence de l'émulateur) par les résultats obtenus lors de la simulation du réseau HSL.

6.6.2 Mesures

Le tableau 6.1 donne les durées d'exécution de la fonction *getpages* qui réalise le défaut de page. Ces mesures ont été faites pour des lectures aléatoires d'une page et des transferts contigus (les pages sont toujours contiguës sur disque) de 10 clusters de 1 à 16 pages.

La première ligne donne les temps pour la version 2 de Maîs, les suivantes concernent la version antérieure. La première colonne donne la latence du défaut de page sur disque, la seconde indique la latence d'un défaut de page dans Maîs. Elle est constituée de 3 latences : latence réseau (obtenu lors de la simulation du réseau), latence chez le client et latence du traitement chez le serveur. Enfin, nous donnons l'accélération obtenue avec Maîs ainsi que sa valeur maximale (obtenue avec une latence système nulle).

Nous observons une accélération de 34,96 lors d'accès aléatoire (1 page). Ce gain qui représente le pire cas pour le swap (accès aléatoires) est très encourageant. De plus, dans Maîs-2, les défauts de pages étant asynchrones la première page d'un cluster arrive toujours dans le même temps permettant à l'application de continuer plus tôt son exécution. Les autres chiffres sont comparés à des accès contigus sur disques ce qui explique que l'accélération diminue d'un ordre de grandeur. En outre, les temps obtenus pour des clusters de taille supérieure à 2 ne sont là qu'à titre de comparaison : dans FreeBSD, le pré-chargement ne se fait que sur des clusters de deux pages. Cette première amélioration entre les deux versions du système est principalement dû au changement du niveau d'accès du client au protocole réseau.

6.7 Positionnement

La pagination en mémoire distante (PMD) est un domaine relativement récent (relativement à la mémoire partagée répartie par exemple). Comer et Griffioen [59] sont les premiers à mentionner l'utilisation de la mémoire physique distante comme espace de *swap*. Leur modèle est asymétrique statique : les serveurs sont des machines dédiées avec une vaste mémoire physique et ne peuvent être clients, tout comme les clients ne peuvent être serveurs. C'est un inconvénient évident car certaines machines sont dédiées et ne peuvent servir pour d'autres tâches et la mémoire de machines clientes inactives ne peut être exploitée. Ils définissent un protocole de communication (XPP) leur permettant de supporter des grappes hétérogènes, en particulier plusieurs tailles de pages. Maîs ne traite pas ce problème, la machine MPC étant supposée constituée de nœuds homogènes (des Pentium). Le réseau utilisé est un simple Ethernet 10 Mbit/s et les avantages sur la pagination sur disque sont déjà évidents.

Felten et Zahorjan [79] étudient les problèmes posés par une PMD de type client/serveur symétrique dynamique. Le serveur peut devenir client, ce qui nécessite de migrer ses pages vers d'autres serveurs. Plusieurs politiques sont étudiées, aussi bien pour le choix des serveurs par le client que pour la migration des pages, soit vers d'autres serveurs, soit vers le disque du serveur lui-même, soit vers le disque du client. L'insertion et le retrait dynamique de serveurs sont gérés par un serveur de noms, auprès duquel les serveurs s'enregistrent lorsqu'ils deviennent actifs et se désistent lorsqu'ils redeviennent "dormants". Leurs expérimentations sont réalisées sur un réseau Ethernet 10 Mbit/s et les résultats sont extrapolés pour un réseau FDDI 100 Mbits/s. Ils modélisent leur systèmes par des files d'attente. Le défaut de page rapporté est 2.5 fois plus rapide que sur disque.

Bernard et Hamma [34] présentent des résultats expérimentaux de PMD dans le meilleur cas (stations inactives) et étudient en particulier l'impact qu'ont la PMD et l'activité locale de la station l'une sur l'autre. Ils proposent un algorithme de décision d'activation / d'abandon de la PMD en fonction des taux de défauts de pages locaux (propre au serveur) et distants (pour le compte d'un client).

Iftode *et al.* [105, 145] étudient la PMD dans le cas de multi-ordinateurs (Intel iPSC/860, Delta, Paragon, TMC CM-5, J-Machine, Mosaic). Seuls certains nœuds possèdent des périphériques d'E/S (disques), et le rapport nœuds de calcul / nœuds d'E/S est supposé augmenter lorsque la taille du multi-ordinateur augmente, ce qui renforce l'intérêt pour la PMD, les disques devenant en effet dans ce cas un important goulot d'étranglement, alors que la mémoire physique croît en proportion du nombre de nœuds. L'introduction de la pagination distante améliore les performances globales des applications test sur iPSC/860 (dont la bande passante est de 22 Mbits/s) d'un facteur 3. Le gain est particulièrement net pour les applications traitant de larges structures de données et donc la localité des références est faible.

Markatos et Dramitinos [65, 128] implémentent une PMD sur DEC/OSF1 sur un réseau Ethernet 10 Mbits/s et rapportent des temps de pagination deux fois plus faibles que sur disque. Ils étudient les problèmes posés par la tolérance aux fautes et détaillent plusieurs mécanismes de parité (*delayed parity*, *parity logging*...) [126]. Leur système permet de tolérer la panne d'un site et également de réagir à des variations de charge.

Schilit et Duchamp [163] décrivent une PMD dans le cadre de l'informatique mobile. Ils considèrent des ordinateurs portables, qui ont typiquement peu de mémoire et se déplacent, ce

qui nécessite une association client/serveur dynamique et variant dans le temps ainsi qu'une migration des pages. Leur système est implémenté sur Mach 2.5 et utilise son interface de *pager* externe. Leurs résultats sont relativement décevant, il faut 45 ms pour évincer une page de 4Ko. En fait, il montre qu'une partie importante du temps (16 ms) est passée dans les couches de protocole du système (Mach IPC et TCP).

GMS (Global Memory System) [76] est une PMD plus élaborée que celles vues précédemment. GMS considère deux types de pages : les pages *locales* (appartenant à des processus de la machine locale) et les pages *globales* (appartenant à des processus de machines distantes et hébergées par la machine locale). La mémoire des machines est ainsi partitionnée en mémoire locale et mémoire globale, cette dernière étant une partie de l'espace de *swap* total.

GMS est plus qu'une simple PMD dans le sens où il intègre des mécanismes de gestion de pages partagées mais il nécessite aussi une modification beaucoup plus profonde de la mémoire virtuelle. Les pages globales ne sont pas partagées, seules les pages locales sont répliquées. Cela permet d'envisager l'utilisation de GMS par des systèmes tels que NFS ou des systèmes de mémoire partagée répartie, auquel cas la politique de cohérence est laissée aux soins de ces systèmes.

Le remplacement de page dans GMS est aussi plus complexe que dans les autres PMD. GMS maintient en effet une information globale sur l'âge des pages afin d'évincer les pages les "moins valables" (*i.e.* les pages globalement moins utilisées). A cette fin, GMS divise le temps en *époques*. A chaque changement d'époque, un site est élu pour centraliser les informations d'âge des pages de toutes les machines et ensuite les redistribuer à toutes les machines.

L'existence de différents types de pages (locale unique, locale répliquée, globale) complique le choix de la victime. L'ordre d'éviction retenu est d'abord les pages globales, suivies des pages locales répliquées puis des pages locales uniques.

6.8 Conclusions et Perspectives

Maîs est un système de pagination en mémoire distante à deux niveaux de *swap* pour la machine MPC. Sa particularité est d'utiliser l'asynchronisme permis par le protocole d'écriture directe en mémoire distante du réseau HSL de la MPC, offrant ainsi un important recouvrement calcul/communication. Les performances d'une PMD dépendent directement des performances du réseau sous-jacent. Le haut débit et la faible latence de HSL sont donc essentiels dans les performances de Maîs.

Les perspectives sont nombreuses. Nous envisageons l'adjonction de mécanismes de contrôle de concurrence pour gérer des pages partagées dans le *swap* afin de supporter une mémoire partagée répartie, en profitant des spécificités de HSL.

Nous envisageons également d'implanter un mécanisme de migration de processus optimisé. Dans ce cas l'espace mémoire d'un processus serait réparti sur plusieurs machines, la migration consisterait alors à déplacer le mot d'état du processus, Maîs se chargeant de rapatrier les pages mémoire sur les sites d'exécution. On peut ainsi espérer des temps très faibles de migration et concevoir des stratégies d'équilibrage de charge.

Chapitre 7

Mémoire Partagée Répartie Extensible pour les Multi-Réseaux

Sommaire

7.1 Introduction

Les mémoires partagées réparties (MPR) fournissent l'abstraction d'une mémoire partagée sur des systèmes ayant des mémoires physiquement réparties. C'est un support très attractif qui simplifie le développement d'applications parallèles. Cependant, la plupart des MPRs ont été implantées sur des réseaux de stations de travail ou sur des multi-processeurs symétriques (SMPs) et sont généralement adaptées qu'à des réseaux locaux.

Le nombre de machines d'un réseau local étant relativement faible, l'idée d'étendre une MPR à un ensemble de réseaux inter-connectés (des multi-réseaux) est séduisante pour fournir une plus grande capacité de traitement et un espace de mémoire physique plus important. Cependant, les plates-formes multi-réseaux ont des caractéristiques physiques qui peuvent limiter fortement les performances des applications. La latence entre des machines des réseaux distants peut être beaucoup plus élevée que celles entre des machines d'un même réseau, et le débit est souvent plus faible. Il faut donc limiter au maximum l'échange de données entre les réseaux. Cela signifie qu'une MPR multi-réseaux n'est pas adaptée à tout type d'applications. Elle vise particulièrement des applications parallèles à gros grain présentant une forte localité dans les références aux données avec des rares opérations de synchronisations ou de mises à jour globales. Des travaux récents [24] ont montré qu'avec des optimisations appropriées, prenant en compte la latence et le débit, des applications de granularité moyenne peuvent s'exécuter de façon performante sur des plates-formes multi-réseaux.

Motivés par ces études et l'augmentation substantielle de puissance des multi-réseaux, nous avons étendu la MPR TreadMarks pour l'adapter aux multi-réseaux. TreadMarks est une MPR connue pour ses très bonnes performances. Pour que cette MPR garde un haut niveau de performance dans le cadre des multi-réseaux, nous avons orienté nos recherches selon les deux axes complémentaires suivants :

— *l'extensibilité du protocole de cohérence pour gérer un grand nombre de machines*,

— *la réduction du trafic inter-réseau*,

Le modèle de cohérence relâchée paresseuse (LRC : Lazy Release Consistency), qui a été initialement défini dans TreadMarks [6], est reconnu comme l'un des plus efficace pour des réseaux locaux. En relâchant le modèle de cohérence mémoire, LRC réduit le nombre de messages et de données transférées entre les processeurs. Les opérations de synchronisation des programmes établissant un ordre d'accès à la mémoire sont utilisées pour propager les informations sur les données partagées.

La plupart des implémentations du protocole LRC sont peu extensibles [9]. Ceci est en partie lié à l'utilisation des horloges vectorielles, définies par **Mattern**[129] et **Fidge**[81], pour contrôler l'ordre partiel des mises à jour et la causalité des opérations d'acquisition et de libération des verrous. Ces horloges ayant une entrée pour chaque nœud du système, elles sont de fait pas extensibles. De plus, pour contrôler la propagation des modifications, les horloges vectorielles sont stockées dans la mémoire des processeurs et incluses dans de nombreux messages. Donc, si le nombre de processeurs est important, comme c'est le cas dans les multi-réseaux, la grande taille du vecteur aura une influence directe sur les performances de la MPR. Nous avons donc défini une horloge logique dont la taille est indépendante du nombre de nœuds du système [13, 12, 11]. Nous l'appelons *l'horloge barrière-verrou*. Sa taille est proportionnelle au nombre de variables de synchronisation utilisées par l'application (très

souvent ce nombre reste petit). Nous montrons par la suite que le protocole LRC peut être implémenté en utilisant les horloges barrière-verrous pour contrôler le *chemin de causalité* des opérations d'acquisition et de libération des verrous.

Pour réduire la latence et le trafic entre les réseaux, nous avons également défini la notion de *cache réseau* [10, 15], dans lequel les processus d'un même réseau échangent leurs informations pour réduire le nombre de requêtes à des réseaux distants. Chaque réseau local se comporte comme une machine SMP à mémoire partagée où les données stockées dans la mémoire physique partagée sont automatiquement disponibles pour tous les processeurs. Nous appliquons également des techniques de pré-chargement pour réduire la latence d'accès aux données partagées. Tous ces travaux ont fait l'objet de la thèse de Luciana Arantes.

Il y a d'autres points, qui limitent l'extensibilité de la MPR TreadMarks, mais que nous n'avons pas abordés. Parmi ceux ci, nous pouvons signaler : l'hétérogénéité entre les machines, la distribution et l'envoi des pages.

La section 7.2 présente brièvement les caractéristiques des mémoires partagées réparties et détaille le principe du protocole LRC. Dans la section 7.3, nous définissons les *horloges vectorielles barrière-verrou* et montrons comment elles peuvent être utilisées pour contrôler la causalité des modifications. Puis, la section 7.4 décrit le prototype de MPR multi-réseau basé sur le cache réseau. La section 7.5 présente des optimisations pour le multi-réseau des opérations de synchronisation. Dans la section 7.6, nous présentons des mesures de performance. Enfin, dans la section 7.7, nous évoquons les autres approches dans le domaine des MPR extensibles.

7.2 Caractéristiques des Mémoires Partagées Réparties

Il existe de nombreux systèmes de MPR dont la conception et l'implémentation impliquent des choix multiples comme la taille de l'unité de partage, le protocole de cohérence ou la façon de propager les modifications.

7.2.1 Granularité

Le choix du grain de partage d'un système de mémoire partagée est délicat car il conditionne directement le rapport entre le surcoût des protocoles de maintien de cohérence et la performance.

De nombreuses MPRs sont structurées en pages. Il faut alors choisir la granularité de la page. Plus la page est grosse, moins il y aura de surcoûts dus à la pagination, mais plus de problèmes de faux partages existeront. Ce problème survient quand deux tâches accédant à la même page mémoire, mais sur des zones différentes, rentrent en compétition pour l'acquérir. A l'inverse, plus la page est petite, plus le nombre de défauts de page est grand et plus la quantité de méta-données est importante. Ainsi, les implantations logicielles de MPR font varier la taille des pages entre 1 Ko et 8 Ko. Les implantations matérielles permettent d'avoir des granularités de l'ordre 16 ou 32 octets.

D'autres MPRs ont choisi une structuration par type. On ne partage plus de la mémoire non typée mais des variables et des structures. Cette méthode est implantée de façon logicielle. C'est la technique adoptée par Munin [53] et Midway [36]. Avec cette approche, la

MPR connaissant la structure des données échangées, il est alors possible de partager des données entre machines ayant une représentation interne des données différente. Enfin, les MPR peuvent être structurées par objet. Les variables ne sont accédées que par les méthodes de cet objet pour améliorer les performances de la mémoire partagée. C'est le cas de Linda [52] et Orca [25].

7.2.2 Classification des protocoles de cohérence

Pour des raisons de performance, une page (ou un objet) mémoire peut être présente physiquement sur plusieurs machines. Il y a alors un mécanisme pour maintenir la cohérence des différentes copies. Il existe plusieurs types de cohérence :

- la *cohérence atomique (Atomic consistency)* : une opération de lecture retourne la valeur de l'écriture la plus récente ;
- la *cohérence séquentielle (Sequential consistency)* : tous les accès à la mémoire sont perçus dans le même ordre par tous les processus ;
- la *cohérence causale (Causal consistency)* : seules les opérations d'écriture liées causalement sont perçues dans le même ordre par tous les processus ;
- la *cohérence pram (pram consistency)* : les opérations d'écriture d'un même processus sont perçues dans le même ordre par tous les autres processus ;
- la *cohérence faible (Weak consistency)* : le programmeur contrôle la cohérence au travers de variables de synchronisation ;
- la *cohérence relâchée (Release consistency)* : au travers d'opérations de synchronisation (acquérir (acquire) et relâcher (release)) le programmeur contrôle la cohérence ;
- la *cohérence à l'entrée (Entry consistency)* : la donnée partagée est seulement cohérente à l'acquisition de la variable de synchronisation qui lui est associée.

7.2.3 Faux partage et écrivains multiples

Les effets du faux partage peuvent être très dommageables particulièrement pour les MPRs basées sur les unités de partage de taille importante comme les pages. Le taux de faux partages peut être réduit de manière significative par l'utilisation de protocoles à *écrivains multiples*. Le concept d'écrivains multiples dans les MPRs a été introduit par le système Munin [53].

Alors que les protocoles à écrivain unique exigent un accès exclusif à une page avant une écriture, les protocoles à écrivain multiples permettent à plusieurs processus de modifier simultanément des parties différentes de la copie locale d'une même page. Ces modifications, appelées *diffs*, sont ensuite envoyées et appliquées aux différentes copies de la page alors que dans le protocole à écrivain unique les pages entières sont transférées. Ainsi l'effet du faux partage est minimisé. Cependant, les protocoles à écrivains multiples sont complexes à implanter et consommateurs de ressources mémoires.

Certaines MPRs ont donc adoptées une approche adaptative [5, 136] dans laquelle le protocole (écrivain unique ou multiple) est automatiquement choisi en fonction du mode d'accès à chaque page.

7.2.4 La cohérence relâchée

Dans le modèle de *cohérence relâchée* [92], les accès normaux à la mémoire sont isolés des synchronisations. Les synchronisations imposent un ordonnancement des processus et peuvent être divisées en séquences d'acquisition et de libération de verrous. Une opération d'acquisition permet d'obtenir l'accès à une variable partagée tandis que la libération transmet le droit d'accès. Les modifications effectuées par un processus sur les variables partagées sont propagées aux autres uniquement lorsque celui-ci exécute l'opération de libération. Toutes les opérations sur les verrous et les barrières peuvent être réalisées à l'aide des opérations d'acquisition et de libération : un verrouillage correspond à une acquisition, un déverrouillage à une libération, l'arrivée à une barrière peut être modélisée comme une libération tandis que son départ peut être représenté par une acquisition.

7.2.4.1 La cohérence relâchée paresseuse

La *cohérence relâchée paresseuse* [112] (LRC : Lazy Release Consistency) définie par TreadMarks adopte les mêmes principes que la cohérence relâchée. La principale différence réside dans le fait que l'envoi des données modifiées à un autre processeur est reporté jusqu'à ce que ce dernier fasse une opération d'acquisition. La figure 7.1 illustre les deux protocoles relâchés.

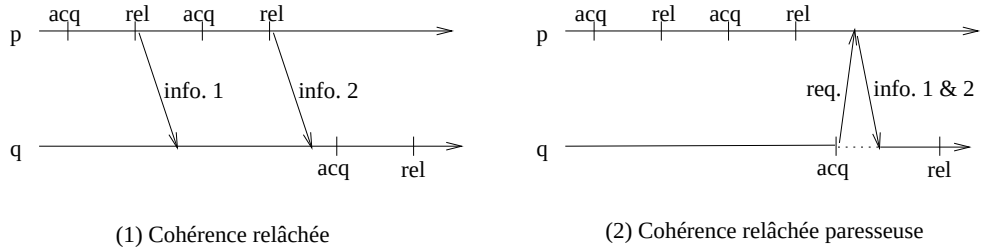

FIGURE 7.1 – Protocoles de cohérence relâchée.

LRC évite de transmettre des mises à jour inutiles en attendant que les données modifiées soient effectivement réclamées par les processus de l'application. Cependant, ce protocole génère plus de messages et à une latence naturellement plus forte lorsque les données sont demandées.

L'exécution de chaque processus est divisée en *intervalles*. Un nouvel intervalle commence à chaque acquisition ou libération. Les intervalles sont ordonnés partiellement de la manière suivante :

1. si i_1 et i_2 sont deux intervalles d'un même processus et que i_1 a lieu avant i_2 dans l'ordre du programme alors i_1 précède i_2, i.e., $i_1 \rightarrow i_2$;
2. si i_1 est un intervalle associé à la libération d'une variable de synchronisation et que i_2 est l'intervalle lié à la prochaine opération d'acquisition sur cette variable alors $i_1 \rightarrow i_2$;
3. si $i_1 \rightarrow i_2$ et $i_2 \rightarrow i_3$, alors $i_1 \rightarrow i_3$.

Selon cet ordre partiel, lorsqu'un intervalle i_1 précède un intervalle i_2, toutes les mises à jour effectuées durant i_1 sont visibles par i_2. Deux intervalles sont concurrents $(i_1 \| i_2)$ si $\neg(i_1 \rightarrow i_2)$ et $\neg(i_2 \rightarrow i_1)$.

L'ordre partiel est contrôlé en utilisant une horloge vectorielle associée à chaque intervalle [129]. Chaque processeur maintient un vecteur d'entiers de N entrées, où N est le nombre total de processeurs du système. Un processeur j contrôle les intervalles qu'il crée en utilisant la *jème* entrée (correspondant à son horloge locale). Les autres entrées mémorisent la vision que le processeur a des mises à jour des autres sites. Le processeur j met à jour son vecteur v_j à chaque opération de synchronisation de la manière suivante :

1. Lors d'une opération d'acquisition sur j, si j est différent du processeur libérateur l, j met à jour son vecteur local avec le maximum de son vecteur v_j et celui du vecteur v_l, transmis par l :
   $1 \leq k \leq N$; $v_j[k] = max(v_j[k], v_l[k])$;
2. Le nouvel intervalle est enregistré :
   $v_j[j] = v_j[j] + 1$.

Ces deux règles garantissent l'ordre partiel des intervalles. En effet, **Mattern** définit la propriété de cohérence forte qui peut être appliquée aux intervalles. Soit $v(s_1)$ la valeur de l'horloge vectorielle associée à l'opération de synchronisation (acquisition/libération) de l'intervalle i_1 et $v(s_2)$ celle associée à l'intervalle i_2, la cohérence forte définit que :
$v(s_1) < v(s_2) \Leftrightarrow s_1 \rightarrow s_2 \Leftrightarrow i_1 \rightarrow i_2$
et
$\neg(v(s_1) < v(s_2)) et \neg(v(s_2) < v(s_1)) \Leftrightarrow s_1 \| s_2 \Leftrightarrow i_1 \| i_2$.

Lors d'une acquisition, le processeur j doit recevoir les mises à jour de tous les intervalles du processeur libérateur supérieurs à son horloge vectorielle. Pour cela, j envoie la valeur courante de son horloge vectorielle au processeur libérateur, l. Ce dernier, renvoie alors la liste des mises à jour connues de l et ignorées de j. Les mises à jour sont envoyées sous la forme de notifications d'écriture (*write notices*).

Une notification d'écriture est une structure, créée à la fin d'un intervalle, qui stocke l'identification des pages qui ont été modifiées pendant l'intervalle. Les modifications sont elles mêmes envoyées soit sous forme de *diffs* (comparaison mot à mot entre une copie de la page et sa dernière version) soit directement sous forme de pages modifiées. Lors d'une acquisition, le processeur demandeur ne reçoit que les notifications d'écriture invalidant les copies locales des pages correspondantes. Le prochain accès à l'une de ces pages provoquera un défaut de page et le processus demandera alors les modifications. Dans le cas de *diffs*, l'acquéreur demande tous les *diffs* qu'il ne possède pas encore, puis il les applique dans l'ordre causal des intervalles.

7.3 Une horloge extensible : l'horloge barrière-verrou

Les horloges vectorielles capturent précisément la causalité entre les opérations d'acquisition et de libération. Cependant, leur taille est proportionnelle au nombre de processeurs du système ce qui limite fortement leur extensibilité. Dans une MPR adaptée aux multi-réseaux, le nombre de processeurs peut être très important exigeant donc des vecteurs très grand. Dans ce contexte, le surcoût des communications peut être très élevé car le vecteur est inclus dans

de nombreux messages (lors des opérations de synchronisation ou lors des accès aux données partagées).

Nous proposons une nouvelle logique d'horloge, *l'horloge vectorielle barrière-verrou*, dont la taille est indépendante du nombre de noeuds. Pour contrôler complètement l'ordre partiel des intervalles, l'horloge vectorielle traditionnelle (avec une entrée par noeud) est remplacée par un simple compteur de barrière et un vecteur avec une entrée par verrou. Comme dans la version originale du protocole LRC, des *diffs* et des notifications d'écriture sont associés à chaque intervalle.

Le nombre de verrous utilisés par les applications étant souvent très petit (un ou deux ou même aucun comme dans l'application SOR [24]), l'horloge vectorielle barrière-verrou réduit considérablement la taille de l'estampillage par rapport aux horloges vectorielles traditionnelles. Ceci est d'autant plus vrai si le nombre de sites est important. Par exemple, si une application s'exécute sur une plate-forme avec un grands nombre de site, mais réclame au maximum deux verrous, un vecteur avec deux entrées plus une simple variable scalaire est suffisant pour implémenter le protocole LRC. Naturellement, notre solution n'est pas adaptée aux applications utilisant un grand nombre de verrous comme l'application *Water* du benchmark SPLASH [176].

7.3.1 Gestion des verrous

Pour contrôler les mouvements de verrous entre les processeurs, nous utilisons l'ordre du diagramme événementiel ("poset-diagram") défini par **Mattern** [129]. Ce diagramme reflète la relation logique entre les événements en termes de causalité. Graphiquement, si $a \rightarrow b$, il existe "un chemin de causalité" de a vers b dans le diagramme temporel qui est visible dans le diagramme événementiel. Les deux diagrammes sont isomorphes.

Nous pouvons utiliser l'idée du diagramme événementiel pour contrôler l'ordre partiel des intervalles, c'est à dire qu'il est possible de tracer des "chemins de causalité" des opérations d'acquisition et de libération. Avec un seul verrou, l'acquisition et la libération tracent un simple flux des mises à jour des données partagées. Ceci représente, en fait, la façon dont les processeurs "sérialisent" leur accès aux variables partagées. Ainsi, chaque verrou trace les accès à des variables partagées indépendantes.

La figure 7.2 montre le diagramme temporel lié à des opérations d'acquisition et de libération sur deux verrous faites par cinq processeurs. La figure 7.3 montre le diagramme événementiel correspondant. Pour simplifier, les opérations d'écriture ou de lecture ainsi que les mises à jour des variables partagées ne sont pas représentées. La notation $a_{ij}(l)$ et $r_{ij}(l)$ désigne respectivement la *jème* acquisition ou libération faite par le processeur i du verrou l, tandis que v_i est la valeur de l'horloge vectorielle de i.

Les verrous étant utilisés pour contrôler les sections critiques lors des accès aux variables partagées, le protocole de cohérence relâchée assure qu'à un instant donné un verrou n'est possédé que par un seul processeur. Le comportement est similaire à un jeton qui ne peut être possédé que par un processeur à la fois. Ceci signifie que si deux intervalles sont indépendants, alors ils correspondent à des acquisitions/libérations sur des verrous différents. Ainsi, pour estampiller les intervalles du diagramme événementiel, nous pouvons utiliser un vecteur où chaque entrée est associée à un verrou et non pas à un processeur. Grâce à la propriété

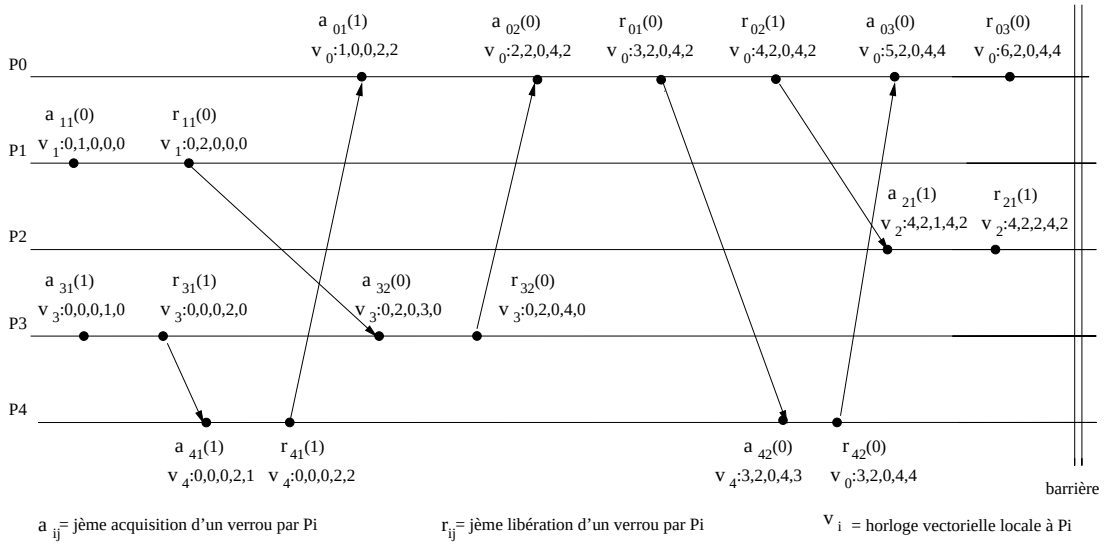

FIGURE 7.2 – Diagramme temporel des acquisitions/libérations.

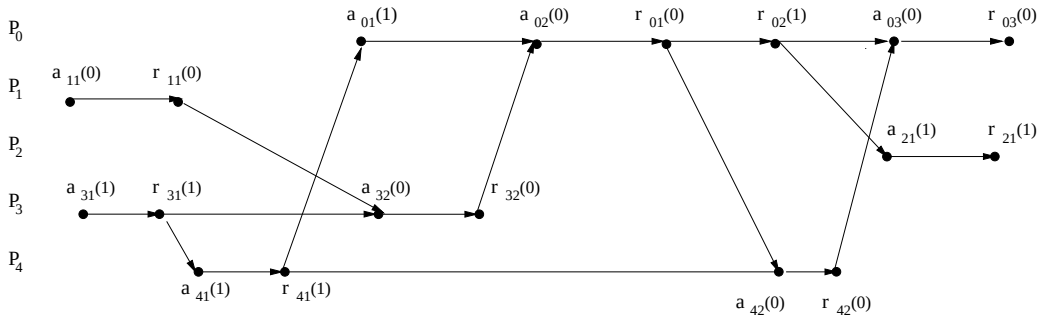

FIGURE 7.3 – Diagramme événementiel.

d'exclusivité dans l'utilisation des verrous, il est possible de garantir que la même entrée sur deux vecteurs différents ne peut pas être incrémentée parallèlement. Ainsi, l'ordre partiel des intervalles peut être tracé. Le processeur j met à jour son vecteur de verrous v_j à chaque opération de synchronisation de la manière suivante :

1. Lors d'une opération d'acquisition du verrou i, si le processeur acquéreur est différent du libérateur, celui-ci met à jour les entrées de son vecteur local avec le maximum de son vecteur et celui du vecteur v_l transmis par le libérateur :
   $1 \leq k \leq N_v$; $v_j[k] = max(v_j[k], v_l[k])$; N_v étant le nombre de verrous ;
2. Le nouvel intervalle est enregistré :
   $v_j[i] = v_j[i] + 1$.

La figure 7.4 reprend le même exemple en utilisant ces nouvelles horloges logiques. Dans la suite, nous appellerons ces horloges, les *horloges de verrous*.

De manière similaire aux horloges vectorielles classiques, les *horloges vectorielles de verrous* permettent de capturer complètement la causalité des intervalles. Dans [12], nous montrons

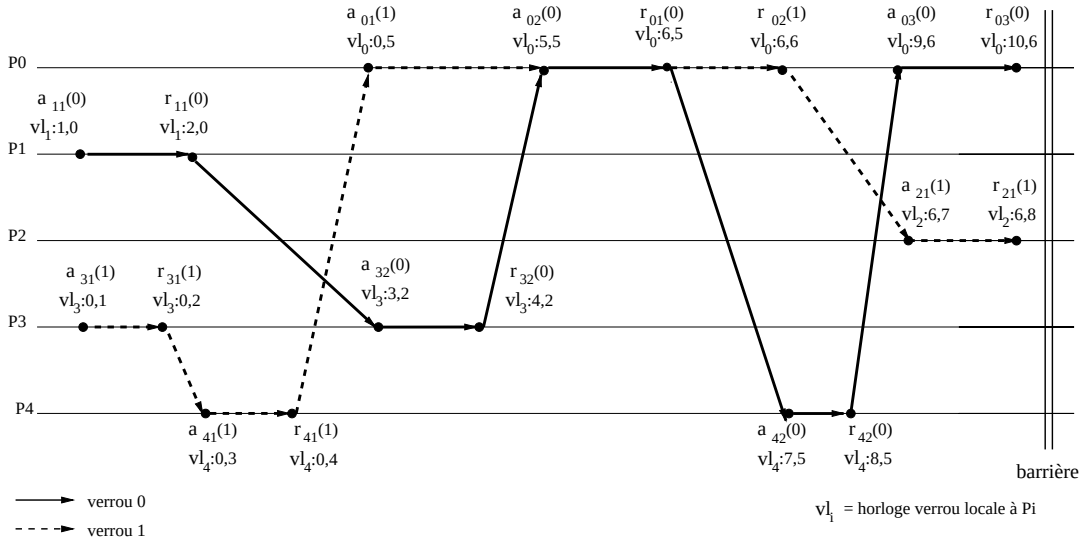

FIGURE 7.4 – Diagramme événementiel en utilisant les horloges vectorielles de verrous.

qu'il y a un isomorphisme entre l'ordre partiel des intervalles et leur estampille sous forme d'*horloge vectorielle de verrous*.

7.3.2 Gestion des barrières

Une barrière est un point de synchronisation où tous les processeurs égalisent leur vecteur à la même valeur et collectent les informations de cohérence (les notifications d'écriture) sur tous les accès en écriture. Cette nouvelle valeur du vecteur de temps couvre donc toutes les valeurs précédentes.

Puisqu'au moment des barrières tous les processeurs égalisent leur vecteur de temps et qu'entre deux barrières les verrous sont responsables de l'ordre partiel, l'exécution peut être divisée en intervalles de barrière, eux même subdivisé en intervalles de verrous. L'ordre des intervalles de barrière peut être simplement contrôlé en lui associant une variable scalaire. Ainsi, à chaque occurrence de barrière, les vecteurs de verrous sont remis à zéro et la variable barrière est incrémentée.

Comme il est mentionné dans [159], une seule barrière est nécessaire pour les synchronisations d'un programme. Si des bibliothèques permettent d'utiliser plusieurs barrières, c'est pour un simple soucis de clarté. Ainsi, dans notre solution, nous donnons la possibilité au programmeur de définir une seule barrière qu'il pourra, en revanche, utiliser à plusieurs reprises dans le programme.

La figure 7.5 illustre un exemple où une barrière est exécutée à deux reprises définissant 3 intervalles de barrières. Arrivés à la barrière, tous les processeurs mémorisent leur propre "chemin" ainsi que les parties de chemin qu'ils ignoraient des autres processus. Tous ont donc la même vision de l'ensemble des mises à jour. *L'horloge de verrous* est alors réinitialisée et un nouvel intervalle de barrière commence.

Le compteur de barrière et le vecteur de verrous forment ce nous avons appelé l'horloge

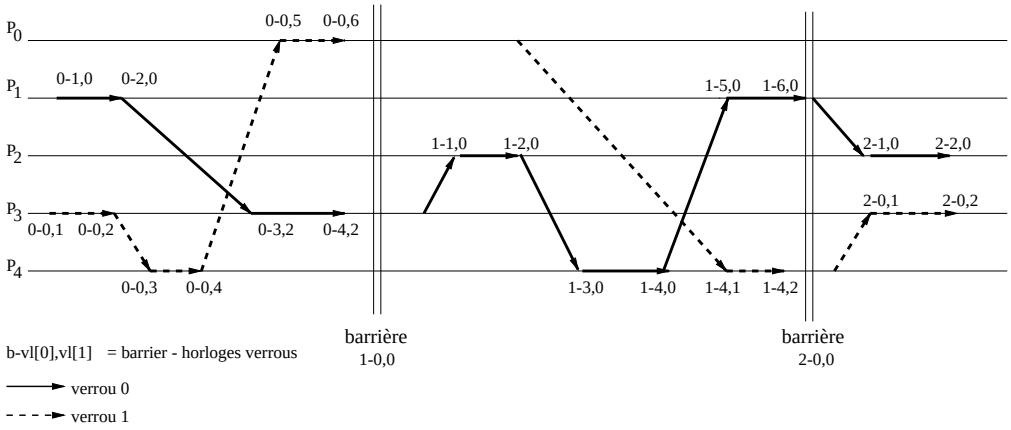

FIGURE 7.5 – LRC basé sur les horloges vectorielles barrière-verrou.

barrière-verrou.

7.4 Cache réseau

L'exécution d'une application sur plusieurs réseaux n'est envisageable que si les synchronisations entre des processus situés sur des réseaux distants ont une latence raisonnable malgré l'éloignement.

Nous avons modifié la MPR TreadMarks pour y introduire la notion de groupe de machines ou *cluster*. Chaque cluster se comporte comme une machine SMP à mémoire partagée. Dans ce type de machine, les données stockées dans la mémoire physique partagée sont automatiquement disponibles pour tous les processeurs. Ainsi, toutes les informations sur l'ordre partiel et la localisation des copies de pages disponibles dans un cluster sont utilisées au maximum pour éviter les accès entre clusters. Ainsi, chaque processeur demandant l'acquisition d'un verrou détenu sur un cluster distant ne reçoit du cluster que les informations qu'il ne possède pas sur son cluster local. Les informations locales sont directement envoyées à l'acquéreur par l'un des processeurs du cluster local (en fait le dernier processeur ayant libéré le verrou à un des processeurs d'un réseau distant).

Sur chaque cluster, nous avons introduit deux types de caches : le cache implicite et le cache étendu. Le cache implicite utilise directement les informations déjà maintenues en mémoire par le protocole LRC tandis que le cache étendu stocke des informations supplémentaires sur les clusters distants dans un processus additionnel dédié.

7.4.1 Cache implicite

Dans le protocole LRC standard, tous les *diffs* (i.e., les modifications des pages) reçus sont gardés en mémoire même lorsqu'ils ont été appliqués sur les copies locales des pages. Cette information est maintenue car d'autres processus peuvent réclamer ces diffs. Dans notre approche multi-cluster, nous utilisons cette information redondante pour éviter des accès à

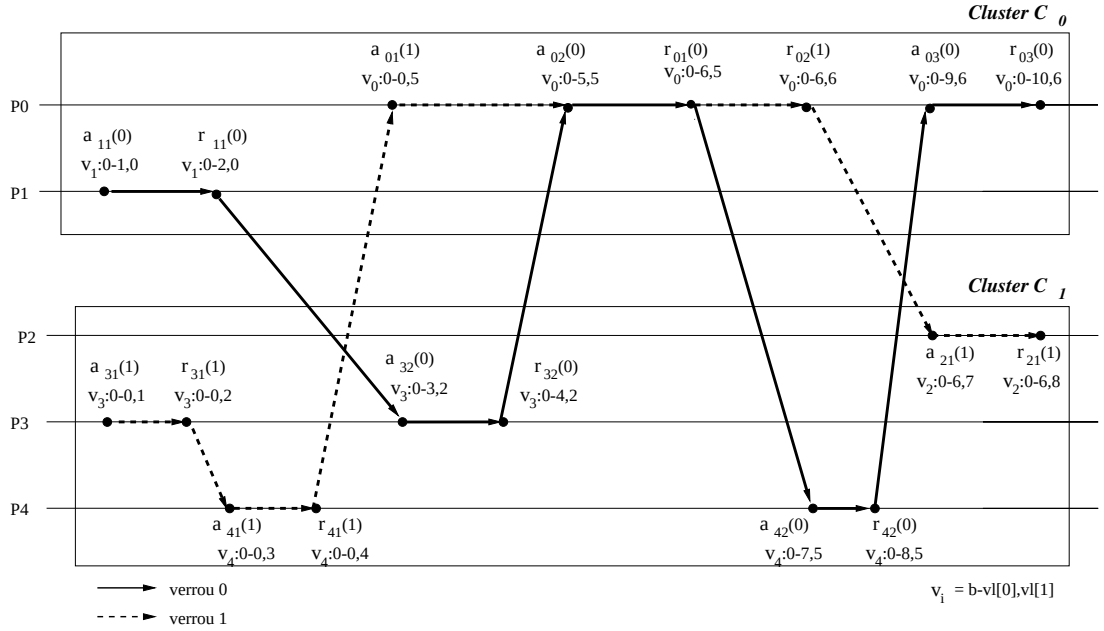

FIGURE 7.6 – Principe du cache implicite.

des réseaux distants. Ces informations pré-existantes dans TreadMarks original constitue ce nous avons appelé le *cache implicite*.

Le protocole LRC a donc été modifié pour profiter au maximum des *diffs* existants dans chaque cluster. Nous avons utilisé le principe des *chemins de verrou*, décrit dans la section 7.3. Le principe est illustré dans la figure 7.6. Les synchronisations sont identiques à celles de le figure 7.5, mais les processeurs P_0 et P_1 appartiennent au même cluster C_0, tandis que P_2, P_3 et P_4 font partie du cluster C_1. Par exemple, l'opération d'acquisition $a_{42}(0)$ du processus P_4 demande le verrou 0 libéré par le processeur P_0. Même si le vecteur de verrou de P_4 (v_4) a la valeur $0,4$ avant l'acquisition, le cluster C_1 informera C_0 que son horloge est égale à $4,2$ car P_3, le dernier processeur de C_1 ayant possédé le verrou 0, possède la liste des modifications jusqu'à $4,2$ qu'il peut transmettre directement à P_4. P_0 enverra uniquement à P_4 les informations de causalité supérieures à $4,2$. De cette manière, le nombre et la taille des messages échangés entre les réseaux sont réduits de manière conséquente comme nous le verrons dans la section 7.6.1.

7.4.2 Cache étendu

Le cache implicite repose sur le fait que chaque processus peut identifier, lors d'un défaut de page, les relations de causalité des *diffs* manquants. Cependant, il est possible qu'un processus ne sache pas qu'un autre processus du même cluster a déjà demandé les mêmes diffs. Ceci arrive par exemple en cas de faux partage, où deux processus demandent les *diffs* sur une page contenant deux variables non liées causalement.

Pour réduire la latence des demandes distantes, nous avons introduit dans chaque cluster

un processus supplémentaire qui contient dans sa mémoire toutes les informations en provenance des clusters distants. Ce processus, que nous appelons *processus cache*, est dédié aux communications, il n'exécute pas le code de l'application.

Les données en provenance de clusters distants sont envoyées également au processus cache. Lorsqu'un processus réclame un *diff* appartenant à un cluster distant, il génère deux messages : un premier vers son processus cache local et un second vers le processus possédant le *diff*. Si la cache connaît le *diff*, il lui envoie directement la réponse sinon la réponse proviendra directement du cluster distant.

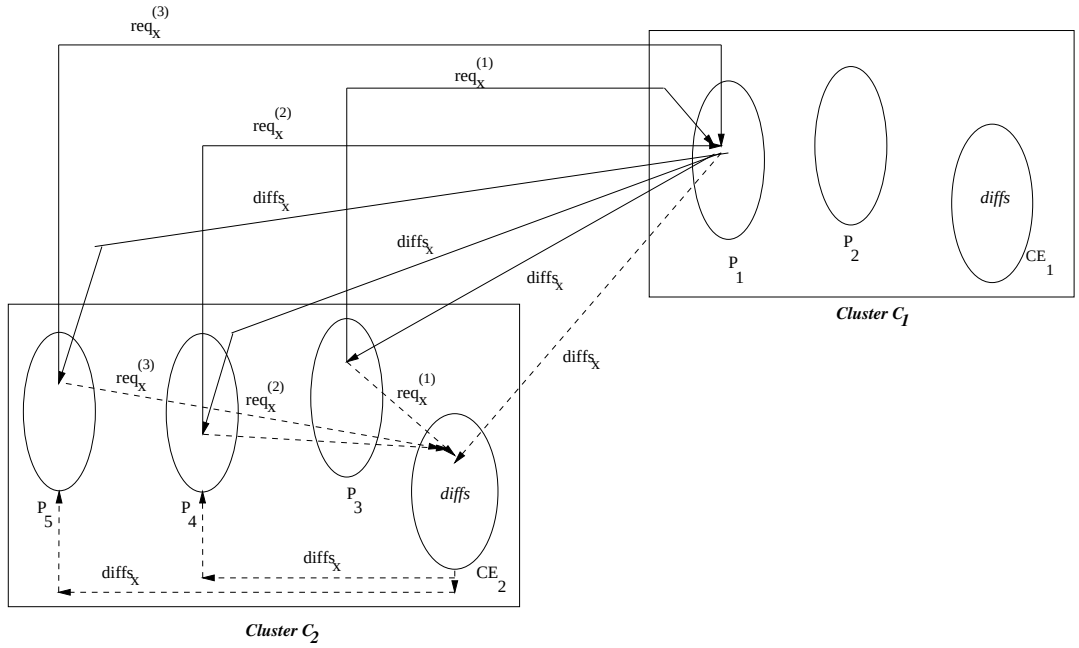

FIGURE 7.7 – Le cache étendu

La figure 7.7 illustre le fonctionnement du cache étendu dans une configuration à deux clusters avec six processus applicatifs (P_i) et deux processus dédiés au cache (CE_i). Supposons que le processus P_1 (le producteur) modifie une variable x qui est lue ensuite par P_3, P_4 et P_5 (les consommateurs). Lorsque P_3 tente de lire x, un défaut de page est généré. Une requête (req_x^1) est alors envoyée au producteur P_1. P_1 appartenant à un cluster distant, P_3 envoie également sa requête au cache du son cluster CE_2. Lorsque la requête arrive sur P_1, ce dernier renvoie les *diffs* correspondant à P_3. De plus, comme c'est la première fois que P_1 reçoit une demande sur ces diffs, P_1 envoie également les diffs à CE_2. Ensuite, lorsque P_4 ou P_5 essayent de lire la variable x, CE_2 ayant déjà une copie des *diffs*, il pourra directement renvoyer la réponse à P_4 ou P_5.

L'efficacité du cache étendu est directement liée aux données (diffs) contenues dans le processus cache. Le pré-chargement permet d'augmenter le taux de succès en anticipant les futures demandes. De nombreuses études ont démontré que la localité temporelle permet de concevoir des stratégies de pré-chargement efficaces [111, 38, 39].

Nous avons donc conçu une heuristique de pré-chargement où chaque processus compte

le nombre de demandes de *diffs* émanant de chaque cluster distant. Lors des barrières, si un même cluster a réclamé plus de *MinDiffReq diffs* sur une page locale p, le processus envoie au cache étendu du cluster distant les nouveaux *diffs* de p sans attendre que le cluster distant les réclame. Ainsi les *diffs* sur les pages souvent réclamées deviennent directement accessibles dans le cluster distant.

7.4.3 Agrégation des données

Pour les requêtes de *diffs*, nous appliquons une optimisation supplémentaire. Avec l'algorithme à écrivains multiples, une page peut avoir plusieurs copies mises à jour simultanément. Dans ce cas, lors d'un défaut de page, un processus demande en parallèle les *diffs* de la page à chaque processus possédant une copie (i.e., les derniers écrivains). Cette parallélisation des requêtes réduit de manière conséquente la latence du défaut de page. Cependant, si les écrivains sont situés dans le même cluster distant, plusieurs requêtes sont envoyées au processus cache local. Pour limiter l'engorgement de ce processus, l'émetteur assemble dans une seule requête les *diffs* destinés aux écrivains d'un même cluster distant.

Cette agrégation permet de réduire le trafic intra-cluster et limite la contention du processus cache. Elle apporte des gains conséquents pour des applications comme Barnes-Hut de Spash-2 [22] qui utilisent de manière intensive des requêtes concurrentes.

7.4.4 Cache adaptatif

Si le cache étendu permet de réduire la latence lors des demandes de *diffs* distantes, celui-ci à un coût non négligeable en nombre de messages échangés entre clusters. En effet, chaque nouveau *diff* réclamé est envoyé au demandeur et au processus cache du cluster distant. Cette approche est utile si le *diff* envoyé profite à d'autres processus du cluster. Dans le cas contraire, le cache étendu risque de diminuer les performances de l'application. Pour traiter ce problème, nous avons réalisé une heuristique similaire à celle employée dans le préchargement. En fonction du nombre de requêtes de *diffs* satisfaites sur une page par le cache, un processus peut décider de désactiver le cache pour la page en question. Dans ce cas, les informations de *diffs* de la page n'alimentent plus les processus caches. Le cache d'une page peut être réactivée lors d'une barrière si le taux de *diffs* d'une page en provenance d'un cluster a augmenté significativement.

7.5 Optimisations des opérations de synchronisation

Les caches permettent de diminuer le coût des échanges inter-cluster. Cependant, pour avoir une gestion plus efficace, il est important de modifier l'implantation des opérations de synchronisation pour les adapter à la structure multi-cluster. A cette fin, nous proposons les deux optimisations suivantes.

7.5.1 Barrières parallèles

Lors des barrières, TreadMarks propage des informations de cohérence (pages modifiées et intervalles) de manière centralisée. Un gestionnaire de barrière regroupe les informations et

les diffuse à tous les processeurs. Les processus de l'application sont bloqués tant qu'il n'ont pas reçu le message du gestionnaire. Ce protocole en deux phases présente une forte latence ce qui limite grandement ses performances lorsque le nombre de noeuds devient important.

Nous avons donc conçu et implémenté un protocole distribué qui permet de réduire la latence des barrières [15]. Arrivé à une barrière, chaque processus envoie *à tous les processus cache* les informations de cohérence. Dès que le cache a collecté les informations de tous les processus, il diffuse aux processus de son cluster les informations de cohérence. Pour des raisons de performance ce message est diffusé en utilisant IP-multicast. En fait, l'ensemble des processus caches joue le rôle du gestionnaire de barrière.

7.5.2 Migration des gestionnaires de verrous

Dans TreadMarks, tous les verrous ont, parmi les processus de l'application, un gestionnaire fixé statiquement au démarrage de l'application. Toutes les demandes d'acquisition sont ensuite envoyées au gestionnaire correspondant. Ceci peut poser de graves problèmes de performance dans une approche multi-réseau. Ainsi, même si le verrou est possédé par un processus situé dans le même cluster, il peut arriver que le gestionnaire soit localisé sur un autre cluster.

Pour remédier à ce type de situation, nous avons introduit dans TreadMarks la migration des gestionnaires de verrous. Pendant l'exécution, chaque gestionnaire trace les demandes de verrous. Si le nombre de requêtes issues d'un même cluster distant dépasse un seuil ($MaxRemReq$) et que les requêtes en provenance des processus du cluster local sont en dessous d'un second seuil ($MinLocReq$ avec $MinLocReq << MaxRemReq$), le gestionnaire est déplacé dans le cluster distant. L'utilisation de deux seuils permet de limiter un effet de ping-pong où un gestionnaire ne cesserait de migrer.

7.6 Performances

Cette section présente une évaluation des performances des principales solutions proposées. De nombreuses mesures ont été réalisés, nous ne mentionnons que quelques résultats, parmi les plus significatifs.

7.6.1 Horloge barrière-verrou

Pour valider notre approche, nous avons remplacé, dans TreadMarks (version 0.10.1), les horloges traditionnelles par les horloges *barrière-verrous*. Des tests de non régression ont été faits en utilisant cinq applications : SOR et TSP, applications fournies par TreadMarks [92] ; IS et 3D FTT du benchmark NAS [23] ; Barnes-Hut du benchmark SPLASH [176].

Le nombre de verrous acquis et d'appels faits à la barrière par chaque application est indiqué dans le tableau 7.1.

Nous avons comparé la taille des messages de synchronisation de notre MPR, la MPR *barrière-verrous*, avec celle de la MPR TreadMarks original. Les tests ont été effectués sur 7 stations de travail Sun Sparc-5 reliées par un réseau Ethernet à 100 Mbit/s.

Dans la figure 7.8, nous indiquons le rapport : *nombre d'octets des messages de synchronisation échangés en utilisant la MPR barrière-verrous/ nombre d'octets des messages*

TABLE 7.1 – Comportement des applications

applications	nombre de verrous	nombre d'appels à la barrière
SOR	0	401
TSP	2	3
IS	0	82
3D FFT	0	104
Barnes-Hut	0	13

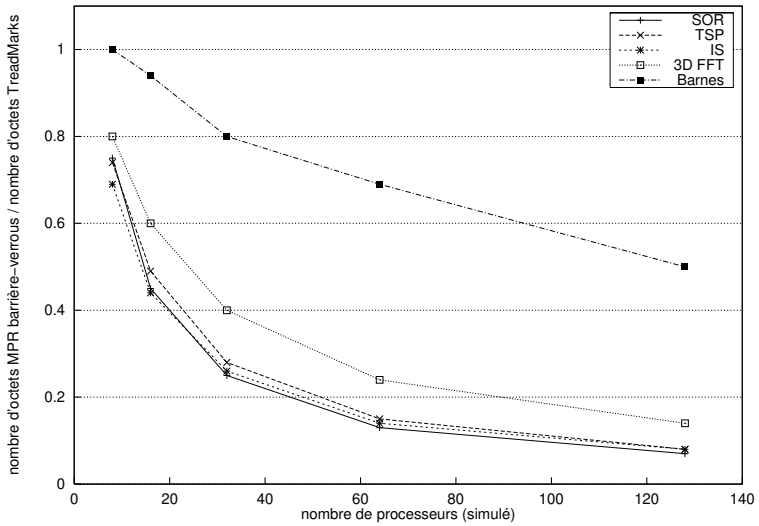

FIGURE 7.8 – Volume de données échangées aux opérations de synchronisation.

de synchronisation en utilisant TreadMarks. Pour illustrer l'extensibilité de la taille des horloges, nous avons fait varier la constante qui définit le nombre de processeurs du système dans TreadMarks. Cela permet de simuler des opérations de gestion du protocole de cohérence avec un nombre variable de machines.

Nous pouvons observer une diminution importante du volume des données échangées lors des opérations de synchronisation. Naturellement, notre modèle est moins adapté aux applications qui utilisent un grand nombre de verrous, surtout si les données sont partagées avec un fort taux de migration. Par exemple, pour l'application Water du benchmark SPLASH [176], nous avons observé une nette augmentation de la taille des messages de synchronisation. Dans ce cas, le chevauchement des estampillages non concurrents ou l'utilisation de la technique différentielle de **Singhal-Kshemkalyani** [177] peuvent être appliqués afin de réduire la taille des messages de synchronisation.

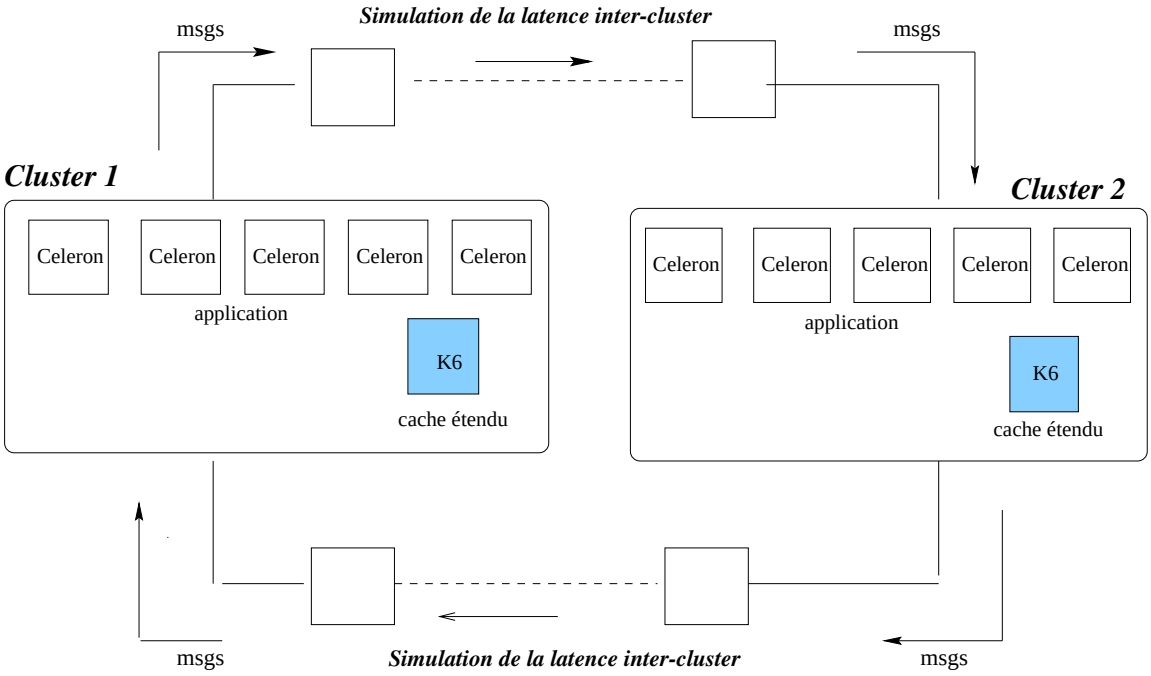

FIGURE 7.9 – Plate-forme d'évaluation.

7.6.2 Approche multi-réseau

7.6.2.1 Plate-forme d'expérimentation

Nous avons fait nos tests un ensemble de PC exécutant Linux reliés par un réseau Ethernet à 100 Mbit/s. Il y a deux types de processeurs : des Celerons cadencés à 433 MHz avec 128 Mégaoctets de mémoire et des K6 cadencés à 233 MHz et 64 Mégaoctets. Nous avons divisé les machines en groupes logiques pour former des clusters différents. Chaque cluster est composé de machines Celerons, plus rapides, dédiées au code applicatif et un K6 dédié au processus cache.

Pour simuler la latence entre ces clusters logiques, les communications inter-clusters transitent par des processus situés sur des machine supplémentaires. La figure 7.9 montre l'architecture de notre plate-forme d'expérimentation.

En faisant varier le nombre de machines intermédiaires, nous pouvons simuler plusieurs latences. Dans la suite nous présentons les résultats avec et sans latence simulée.

7.6.2.2 Performances

Nous avons modifié TreadMarks version 0.10.1 pour intégrer le cache réseau. Nous avons évalué l'efficacité de notre solution avec des applications de granularité moyenne et faible. La table 7.2 donne les caractéristiques de ces applications.

TABLE 7.2 – Caractéristiques des applications

Application	Taille/Itérations
TSP	19 villes
SOR	1000x1000, 200
IS	16x16, 10
3D-FFT	256x256x16,30
Barnes-Hut	4K, 3
Water	343,10
MG	64x64x64

7.6.2.3 Impact du cache implicite

En limitant le nombre d'informations demandées aux clusters distants lors des défauts de page, le cache implicite est directement responsable de la réduction du nombre d'octets des messages échangés entre les clusters.

La table 7.3 illustre ce phénomène. Dans cette mesure aucune latence n'est simulée. On remarque que les applications avec des données migratoires telles que IS, TSP et Water profitent vraiment de cette réduction.

TABLE 7.3 – Nombre d'octets par message inter-cluster

Applications	Sans cache	Cache implicite
Barnes-Hut	400	400
FFT	1879	1879
Gauss	1263	1263
IS	6757	4876
MG	239	237
SOR	312	312
TSP	452	274
Water	1511	1326

7.6.2.4 Impact du cache étendu

Pour évaluer l'efficacité du cache étendu, nous avons mesuré le pourcentage de requêtes de *diff* satisfaites par le cache (*hit*). La table 7.4 présente ces résultats sur différentes configurations de clusters (nous faisons varier le nombre de nœuds des deux clusters). On peut observer que plus la taille des clusters augmente, plus l'efficacité du cache est importante. De plus, les applications fortement parallèles dont les processus partagent peu de données profitent peu du cache.

Nous avons également mesuré le temps de réponses des applications s'exécutant sur 10 machines et en simulant une latence entre deux groupes de 5 machines. La figure 7.10 présente

TABLE 7.4 – Efficacité du cache

Appli.	Latence uniforme			Latence inter-cluster plus élevée		
	3 nœuds	4 nœuds	5 nœuds	3 nœuds	4 nœuds	5 noeuds
Barnes-Hut	56%	63%	71%	55%	62%	72%
3D- FFT	15%	18%	39%	15%	18%	39%
MG	9%	18%	20%	10%	17%	21%
IS	23%	37%	42%	24%	35%	38%
SOR	0	0	0	0	0	0
TSP	31%	38%	46%	32%	39%	44%
Water	35%	44%	53%	34%	42%	52%

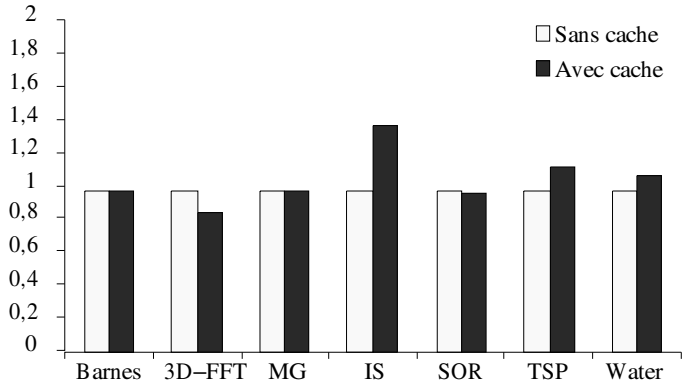

FIGURE 7.10 – Gain du cache.

le gain observé par la configuration avec cache. Cette dernière montre l'efficacité de notre approche pour la plusieurs applications. Nous avons réalisé d'autres mesures en augmentant notamment la latence inter-cluster. Les résultats obtenus sont particulièrement encourageants.

7.7 Positionnement

7.7.1 Horloge

TreadMarks fut la première MPR à implanter le modèle de cohérence relâchée paresseuse. AURC [208] et HLRC [208] sont des variantes de ce modèle. Ces trois approches utilisent des horloge vectorielle pour contrôler la causalité des opérations de synchronisation.

Plusieurs auteurs ont proposé des implantations optimisées des horloges de Mattern et Fidge qui visent à réduire dans les messages le nombre d'octets nécessaires à l'estampillage. Par exemple, dans [177] Singhal et Kshemkalyani propose une approche incrémentale où seule

les entrées modifiées depuis la dernière émission vers le même destinataire sont ajoutées dans les messages. Fowler et Zwaenepoel ont également proposé de maintenir uniquement une trace des dépendances causales directes leur permettant de réduire la taille de l'estampillage. Cependant, pour capturer les relations de causalité transitive, il est nécessaire de tracer récursivement les dépendances.

Les horloges plausibles [194] définies par Torres-Rojas et Ahamad ont un nombre d'entrées constant indépendant du nombre de nœuds. Cependant, même si elles offrent des bonnes propriétés pour ordonner les événements, ces horloges ne permettent pas d'ordonner certains événements concurrents ce qui rend leur utilisation délicate dans le protocole LRC.

7.7.2 Multi-réseaux

Plusieurs travaux ont proposé d'étendre des MPR à des réseaux de machines SMP. Les principaux projets dans ce domaine sont **Shasta** de l'Université de Rutgers [162], **Cashmere-2L** de l'Université de Lancaster [182], **HLRC-SMP** de Princeton [161], **SoftFlash** de Standford [74] et **MGS** du MIT [204]. Ces approches profitent des protocoles de cohérence cache matériels au sein de chaque nœud tandis que des protocoles implantés de façon logicielle gèrent les échanges entre nœuds. Ainsi Hu et al. présentent dans [60] une version de TreadMarks pour réseaux SMPs. Cashmere-2L implémente un protocole à écrivains multiples tandis que Shasta supporte une mémoire partagée avec une granularité plus fine. HLRC-SMP propose une extension du protocole HLRC aux réseaux de machines SMP.

Notre prototype présente des similarités certaines avec ces approches. En effet au sein d'un même cluster toutes les informations de cohérence en provenance de l'extérieur sont disponibles pour tous les nœuds du clusters. La différence est que, dans notre cas, cette disponibilité est assurée de façon logicielle.

Plusieurs auteurs [38, 140, 39, 111] ont proposé le pré-chargement des *diffs* pour des protocoles LRC dans le but d'absorber en partie la latence des communications. Ces travaux visent à rapprocher les données des processeurs qui les utilisent. Comme dans notre approche, les opérations de synchronisation, particulièrement les barrières, sont utilisées pour diviser l'exécution des programmes en intervalles sur lesquels sont définis des historiques d'accès. Dans ces approches les données pré-chargées sont stockées dans la mémoire des processeurs susceptibles d'exiger ces données dans un future proche. Cela diffère de notre approche où un processus dédié, le processus cache, stocke toutes ces données. Ainsi, nous exploitons l'architecture multi-cluster : tous les processus d'un même cluster profitent des données pré-chargées du cache.

7.8 Conclusions et Perspectives

Nous avons présenté un prototype de mémoire partagée adapté à des multi-réseaux en étendant la MPR TreadMarks.

Pour rendre plus extensible le protocole de maintien de cohérence LRC, nous avons introduit les *horloges vectorielles barrière-verrou* pour contrôler l'ordre partiel des mises à jour. Ces horloges sont extensibles car elles sont indépendantes du nombre de sites. Elles sont donc particulièrement adaptées pour une mémoire partagée répartie à large échelle. Les premiers

résultats montrent une diminution conséquente dans la taille de messages de contrôle par rapport à l'implémentation de LRC basée sur les horloges vectorielles traditionnelles.

Nous avons également défini une architecture de cache réparti, pour limiter les échanges inter-réseau et diminuer la latence des opérations de synchronisations. Deux caches ont été introduits. Le cache implicite exploite les informations de protocole LRC présentes dans chaque réseau local pour éviter des accès distants. Le cache étendu permet, sur chaque réseau, de collecter les informations de cohérence des autres réseaux. Ce cache met en œuvre des stratégies de pré-chargement. De mesures de performance montrent l'intérêt de nos solutions.

Nous envisageons plusieurs perspectives à ce travail. Dans un premier temps, nous visons à améliorer l'efficacité des caches en mettant en œuvre une stratégie de coopération entre les caches. Nous aborderons également le problème de partitionnement de réseau en envisageant des stratégies à partitions primaires.

Chapitre 8

Conclusions et Perspectives

Sommaire

Nous avons présenté des solutions pour la conception de supports d'exécution performants pour des applications réparties en environnement non fiable.

Leur base commune est la réplication des données et/ou des traitements. Nous avons montré qu'à partir de concepts communs comme la duplication et la migration des composants applicatifs (que ce soient des processus, des agents ou des pages mémoires) il est possible de concevoir des supports d'exécution efficaces.

Nos recherches se sont concentrées selon trois axes : (1) l'analyse et l'intégration de la tolérance aux fautes en environnement réparti, (2) l'unification de la répartition de charge et des techniques de gestion des fautes et (3) la gestion de la ressource mémoire pour réduire les temps de réponse des applications.

8.1 Tolérance aux fautes

Afin de comparer l'efficacité de différentes techniques de tolérance aux fautes, nous avons développé la plate-forme d'exécution STAR. STAR est dédiée aux applications parallèles de calcul scientifique. L'objectif était de montrer qu'une approche purement logicielle sur une plate-forme banalisée est une alternative performante aux architectures dédiées. Nous avons expérimenté et évalué les techniques de tolérance aux fautes à base de points de reprise.

Pour que le coût d'une gestion logicielle des fautes soit acceptable, nous avons d'abord optimiser les temps de sauvegardes et la latence d'accès au support stable. Pour réduire la quantité de données à sauvegarder, nous avons conçu une technique de sauvegarde incrémentale de l'espace d'adressage des processus. Pour diminuer la latence des sauvegardes, nous

avons également réalisé une sauvegarde asynchrone. Nous avons porté une attention particulière à la réalisation du support stable qui est le principal goulot d'étranglement des techniques de points de reprise. Notre support (SFS) est particulièrement optimisé pour les accès en écriture qui sont les plus fréquents. Des mesures de performances ont montré l'importance prédominante de ces optimisations.

Un constat de l'expérience de Star est que l'assignation statique (au démarrage de l'application) des stratégies de tolérance aux fautes est, dans certains cas, un réel frein pour optimiser la gestion des fautes. En effet, deux critères dynamiques influencent grandement le choix de la stratégie : (1) le modèle de communication de l'application, (2) la fréquence des fautes de l'environnement.

Pour étudier l'adaptation dynamique de la tolérance aux fautes, nous avons participé à la conception la plate-forme DarX. DarX est une plate-forme pour répartir des agents qui permet en cours d'exécution d'adapter et changer les stratégies de réplication de chaque agent. La réplication est totalement masquée aux agents. Les systèmes multi-agents sont particulièrement propices pour une gestion dynamique des fautes. En effet, le comportement des agents peut être très variable. Ainsi, dans certaines applications [51], les agents peuvent exprimer des degrés de "criticité" variables en cours d'exécution. Les premières expérimentations sur le coût des mécanismes de base de DarX (ajout d'un réplicat, changements dynamiques de stratégie, diffusion des messages) sont particulièrement encourageantes.

8.2 Intégration de la répartition de charge

Le choix de la bonne stratégie de tolérance aux fautes au bon moment n'est certainement pas suffisant pour absorber le coût de gestion des fautes. Nos expériences nous ont convaincus qu'il fallait associer à ces mécanismes coûteux une gestion globale afin de maximiser l'utilisation des ressources (processeur, mémoire, disque) disponibles sur le réseau.

Avec Bertil Folliot, nous avons donc conçu la plate-forme GatoStar qui est l'unification de Star et du répartiteur de charge Gatos. GatoStar intègre des algorithmes de placement de programmes multi-critères qui prennent en compte non seulement la charge de machines mais également le comportement des applications en termes d'utilisation de ressources processeur ou/et mémoire. Nous avons montré qu'il existe une réelle synergie entre les techniques de placement et de tolérance aux fautes. Les points de reprise servent, en effet, non seulement au recouvrement de fautes mais également à la migration de processus pour réagir aux variations importantes de charge.

GatoStar a été une première étape qui a montré l'intérêt du placement dynamique et de la migration sur des réseaux de travail. Une des difficultés que nous avons rencontrées réside dans la conception, la mise au point et l'évaluation des heuristiques de placement. En effet, l'environnement cible étant réel et non dédié, les mesures sont peu reproductibles et chaque évaluation nécessite de nombreuses mesures.

A partir de cette constatation, nous avons conçu l'outil d'aide à la conception de stratégies de

placement SIGAP (il s'agit de la thèse de Yanal Haj-Mamoud). SIGAP est un modèle à base de files d'attente développé avec QNAP2 qui permet d'évaluer les performance en simulant des algorithmes de placement sur différentes configurations d'environnements et d'applications. SIGAP est largement paramètrable et modélise finement les ressources (chaque site est modélisé par 27 files). Nous modélisons non seulement les algorithmes de répartition de charge mais également les processeurs, les disques, le réseau ainsi que la surcharge induite par les algorithmes et les utilisateurs. Ce degré de finesse rend trop complexe l'utilisation de méthodes analytiques. Pour valider notre modèle, nous faisons intervenir une phase de calibrage et de raffinement où les mesures simulées sont comparées à des mesures réelles. Grâce à SIGAP, nous avons pu concevoir rapidement et ensuite intégrer avec succès dans GatoStar une stratégie à base de seuils dynamiques. SIGAP nous a également permis de tester GatoStar sur de nouvelles configurations matérielles (avec un plus grand nombre de machines et dans de fortes conditions de charges) et ainsi de mettre en évidence les limites de notre système.

8.3 Gestion de la mémoire

La gestion de la mémoire est une composante essentielle des supports d'exécution. En effet, un rapport de l'ordre de 100000 entre une entrée/sortie disque et un accès mémoire fait des gestionnaires de pagination le principal goulot d'étranglement des systèmes. Une gestion optimisée de la mémoire s'avère donc indispensable pour implanter des supports d'exécution performants.

Nous avons abordé la gestion de la mémoire selon deux angles : (1) dans le cadre du projet Maîs (thèse de Philippe Cadinot en cours), nous avons étudié l'intégration dans le noyau des systèmes de stratégies d'évincement de pages en mémoires distantes ; (2) pour les supports d'exécution, nous nous sommes intéressés au problème ouvert de l'extension des plates-formes à base de mémoire partagée répartie (MPR) sur des réseaux à très large échelle (thèse de Luciana Arantes).

Maîs permet d'évincer de manière transparente les pages sur les machines inutilisées en mémoire. Maîs se distingue des autres approches comme GMS [77] et le système décrit dans [128] dans le rôle actif que prennent les périphériques de swap (les serveurs de pagination distants). En effet, grâce à des politiques de pré-chargement, ces serveurs coopèrent pour anticiper les futurs accès des clients. Nous avons également introduit plus d'asynchronisme entre les clients (qui exécutent les applications) et les serveurs de mémoire. Ainsi, contrairement au swap standard, nous avons conçu un pré-chargement asynchrone. Les première études de performances montrent le réel gain que peut apporter ces stratégies (nous avons mesuré une accélération de défaut de page entre 1,88 et 35).

Au niveau des supports d'exécution, nous avons étendu la mémoire partagé répartie (MPR) TreadMarks [6] pour l'adapter à un ensemble de réseaux interconnectés. TreadMarks est une plate-forme, développée l'Université de Rice, qui est connue pour ses bonnes performances grâce principalement au protocole de maintien de cohérence relâché paresseux qui a été introduit. Une des difficultés de son extension réside dans le goulot d'étranglement que présentent

les communications inter-réseaux. Pour réduire la taille des messages du protocole de cohérence, nous avons proposé une nouvelle technique d'estampillage indépendante du nombre de sites permettant cependant de capturer la causalité des mises à jour des données partagées. Nous avons également défini une architecture multi-cluster où, dans chaque cluster, les machines partagent leurs informations liées au protocole de cohérence. Des expérimentations sur plusieurs applications parallèles ont montré l'efficacité des solutions que nous proposons. Cependant, sans une réelle réflexion pour restructurer les applications, l'apport des architectures multi-cluster reste limité. Nos travaux sont donc complémentaires à ceux sur la restructuration d'applications menés notamment dans le projet Albatros [24]. L'alliance d'un support optimisé pour le multi-réseau et d'applications qui exploitent une forte localité intra-cluster est prometteuse.

8.4 Perspectives

Les supports d'exécution modernes doivent être :

- *Adaptables* : la diversité des besoins applicatifs et la variabilité de l'environnement font que les algorithmes de gestion des fautes, et plus généralement les gestionnaires de ressources, doivent pouvoir s'adapter en cours d'exécution à des nouvelles contraintes.
- *Extensibles* : les supports d'exécution ont atteint une maturité suffisante pour passer à une échelle plus large. Il faut donc d'une part étendre les supports à une plus grand nombre de sites et d'autre part modifier les protocoles (de maintien de cohérence ou de gestion de groupes) pour les adapter à des latences et débits variables.
- *"Transversales"* : la diversité des ressources à gérer et leur imbrication nécessitent une étroite coopération entre les gestionnaires des différentes ressources (processeur, mémoire, réseau). Les supports doivent donc combiner plusieurs gestionnaires et particulièrement mettre en œuvre des algorithmes "multi-critères".

Nous avons déjà initié des recherches dans chacun de ces points. Nous comptons poursuivre nos travaux dans ces axes d'avenir.

Pour la tolérance aux fautes, nous visons une extension de la plate-forme DarX pour gérer des réseaux à large échelle.

- Dans le cadre de la thèse d'Olivier Marin, nous avons entamé une collaboration avec le laboratoire d'informatique du Havre (LIH) pour fiabiliser une application de simulation d'éco-système. Cette application possède un très grand nombre d'agents (environ 100000) qu'il faudra répartir sur plusieurs centaines de machines reliées par des réseaux hétérogènes. Cela nécessitera d'adopter dans DarX une architecture hiérarchique similaire à celle de notre MPR multi-réseau.
- Nous visons également une application de gestion de crise développée au LIP6 dans l'équipe OASIS. Cette application a des fortes propriétés de dynamicité : le niveau de fiabilité exigé par les agents peut énormément varier en cours d'exécution. L'objectif est de définir des heuristiques pour adapter automatiquement la stratégie de réplication en fonction des besoins exprimés par chaque agent et de l'environnement d'exécution.

Pour la répartition de charge, nous visons à affiner les algorithmes de placement. La plupart

des systèmes de placement qui adoptent des approches multi-critères [31, 149, 207] utilisent des informations globales sur l'utilisation des ressources consommées par chaque tâche (taux d'utilisation) ; nous avons donc défini un modèle qui permet de caractériser plus finement le comportement des application parallèles et de l'environnement matériel. A partir de ces modèles, nous définissons de nouvelles heuristiques de type Tabou [187, 197] pour optimiser le placement de tâches. Actuellement, nous évaluons notre modèle et l'heuristique de placement associée en nous comparant à des outils d'optimisation de système linéaire tel que CPLEX [61].

Dans notre prototype de MPR multi-réseau, nous nous sommes concentrés sur la définition d'une architecture multi-cluster basée sur des caches répartis. A court terme, nous envisageons d'améliorer les performances de notre cache et mettant en œuvre une stratégie de coopération inter-caches. L'approche est similaire à celle que nous envisageons pour Maîs. En effet, dans chaque cluster, une machine est dédiée au cache. Nous visons donc à donner à ces machines un rôle plus actif pour un meilleur recouvrement entre les traitements applicatifs et le protocole de pré-chargement du cache. Nous envisageons également le traitement du partitionnement où des clusters peuvent se retrouver isolés. Nous avons déjà procédé à quelques expérimentations, nous choisirons une stratégie où une partition primaire reste active.

Enfin, dans le prototype actuel du système Maîs, le pré-chargement reste en grande partie contrôlé par les clients ce qui peut ralentir l'application et la coopération entre les serveurs de pages reste limitée. Notre objectif est donc de rendre plus actifs les serveurs de pages pour décharger les sites d'exécution des applications. En nous appuyant sur nos expériences des heuristiques de pré-chargement, nous envisageons également d'implanter un mécanisme de migration de processus. Nous avons constaté qu'il est en général moins coûteux de déplacer un processus sur un serveur possédant une grande partie de son espace d'adressage plutôt que d'importer les pages du processus sur son site d'exécution. Maîs gérant la répartition des pages sur le réseau, il est assez naturel qu'il serve de support à l'implantation d'une mémoire partagée répartie. Notre approche "interne au noyau" permet d'espérer de meilleures performances qu'une MPR au niveau "middleware".

Bibliographie

[1] G. Agha, S. Frølund, R. Panwar, and D. Sturman. A linguistic framework for dynamic composition of dependability protocols. In *Dependable Computing for Critical Applications III (DCCA-3)*, IFIP Transactions, pages 197–207. Elsevier, New York (NY), USA, 1993.

[2] R. Alonso and L. L. Cova. Sharing jobs among independently owned processors. In *The 8th International Conference on Distributed Computing Systems*, pages 365–372, 1988.

[3] R. Alonso and K. Kyrimis. A Process Migration Implementation for a Unix System. In *USENIX Technical Conference Proceedings*, pages 365–372, Dallas, TX, February 1988.

[4] O. Amir, Y. Amir, and D. Dolev. A highly available application in the Transis environment. *Lecture Notes in Computer Science*, 774 :125– ? ?, 1994.

[5] C. Amza, A. Cox, K. Dwarkadas, Rajamani, and W. Zwaenepoel. Adaptive protocols for software distributed shared memory. *IEEE Proceedings*, 87(3) :467–475, March 1999.

[6] C. Amza, A. L. Cox, S. Dwarkadas, P. Keleher, H. Lu, R. Rajamony, and W. Zwaenepoel. TreadMarks : Shared memory computing on networks of workstations. *IEEE Computer*, 29(2) :18–28, February 1996.

[7] T.E. Anderson, D.E. Culler, and D.A. Patterson. A case for NOW (networks of workstations). *IEEE Micro*, February 1995.

[8] ARAGO. *Informatique tolérante aux fautes*. Edition Masson, 1994.

[9] L. Arantes, B. Folliot, and P. Sens. Problématique de la conception d'une MPR hétérogène. In *Deuxièmes Journées de Recherche sur Le Placement Dynamique et la Répartition de Charge*, 1998.

[10] L. Arantes, B. Folliot, and P. Sens. An approach for a Multi-LAN DSM based on lazy release consistency. In *Proceedings of the third european reserch Seminar on Advances in Distributed Systems (ERSADS'99)*, Funchal, Portugal, April 1999.

[11] L. Arantes, B. Folliot, and P. Sens. A customised logical clock for timestamp-based relaxed consistency DSM systems. In *The 1999 Workshop on Software Distributed Shared Memory held in conjunction with the 1999 International Conference in SuperComputing*, pages 1–6, Rhode,Grèce, June 1999.

[12] L. Arantes, B. Folliot, and P. Sens. A node-count independent logical clock for scaling lazy release consistency protocol. In P. Amestoy, P. Berger, M. Dayd, I. Duff, V. Frayss, L. Giraud, and D. Ruiz, editors, *Euro-Par'99 - Parallel Processing*, volume 1685 of

Lecture Notes in Computer Science, pages 815–822, Toulouse, France, September 1999. Springer-Verlag.

[13] L. Arantes, B. Folliot, and P. Sens. A proposal for a parallel programming support for Multi-LAN platforms. In *Proceedings of the 11th Symposium on Computer Architecture an High Performance Computing*, Natal, Brazil, September 1999.

[14] L. Arantes, B. Folliot, and P. Sens. Une mémoire partagée repartie extensible basée sur la cohérence paresseuse relâchée. In *1ère Conférence Française sur les Systèmes d'exploitation*, pages 185–196, Rennes, France, June 1999.

[15] L. Arantes, P. Sens, and B. Folliot. The impact of caching in a loosely-coupled clustered software dsm system. In *Proceedings of the IEEE International Conference on Cluster Computing*, November 2000.

[16] L. Arantes, P. Sens, and B. Folliot. Enhancing the cache strategy of a cluster-based dsm system using an adaptive approach. In *Proc. of the 2001 International Conference on Parallel Processing (ICPP'01)*, Valencia, Spain, September 2001. IEEE Society Press.

[17] L. Bezerra Arantes, D. Poitrenaud, P. Sens, and B. Folliot. The barrier-lock clock : A scalable synchronization-oriented logical clock. *Parallel Processing Letters*, 11(1) :65–76, 2001.

[18] L. Bezerra Arantes, P. Sens, and B. Folliot. An effective logical cache for a clustered lrc-based dsm system. *Journal of Cluster Computing*, 5(1) :19–31, 2002.

[19] O. Babaoglu. Fault-tolerant computing based on Mach. *Operating Systems Review*, 24(1) :27–39, January 1990.

[20] O. Babaoglu, L. Alvisi, A. Amoroso, and R. Davoli. Paralex : An environment for parallel programming in distributed systems. *Proc. of the Int'l Conf. on Supercomputing*, pages 178–187, July 1992.

[21] O. Babaoglu, R. Davoli, L-A. Giachini, and M. G. Baker. RELACS : A communications infrastructure for constructing reliable applications in large-scale distributed systems. Technical Report BROADCAST-TR94-58, ESPRIT Basic Research Project BROADCAST, October 1994.

[22] D. Bailey, J. Barton, T. Lansinski, and H. Simon. The nas parallel benchmark. Technical Report Technical Report 103863, NASA, July 1993.

[23] D. Bailey, J. Barton, T. Lasinski, and H. Simon. The NAS parallel benchmark. Technical Report 103863, NASA, July 1993.

[24] H. Bal, A. Plaat, M. Bakker, P. Dozy, and R. Hofman. Optimizing parallel applications for wide-area clusters. In *The 12th International Parallel Processing Symposium*, April 1998.

[25] H. E. Bal, M. F. Kaashoek, and A. S. Tanenbaum. Orca : A language for parallel programming of distributed systems. *IEEE Transactions on Software Engineering*, 18(3) :190–205, March 1992.

[26] M. Banatre, G. Muller, B. Rochat, and P. Sanchez. Design decisions for the FTM : a general purpose fault tolerant machine. Technical Report RR-1400, Inria, Institut National de Recherche en Informatique et en Automatique.

[27] J. S. Banino. Parallelism and fault-tolerance in the Chorus. *The Journal of Systems and Software*, 6(1–2) :205–211, May 1986.

[28] A. Barak and A. Braverman. Memory Sharing in a Network of Workstations. Report 96-04, Institute of Computer Science, The Hebrew University, May 1996.

[29] A. Barak, A. Braverman, I. Gilderman, and O. Laadan. Performance of PVM with the MOSIX Preemptive Process Migration. In *Proc. 7th Israeli Conf. on Computer Systems and Software Engineering*, pages 38–45, Herzliya, June 1996.

[30] A. Barak, A. Braverman, I. Gilderman, and O. Laadan. The MOSIX Multicomputer Operating System for Scalable NOW and its Dynamic Resource Sharing Algorithms. Report 96-11, Institute of Computer Science, The Hebrew University of Jerusalem, July 1996.

[31] A. Barak, S. Guday, and R. G. Wheeler. *The MOSIX Distributed Operating System*, volume 672 of *Lecture Notes in Computer Science*. Springer, 1993.

[32] J. Baumann, F. Hohl, K. Rothermel, M. Schwehm, and M. Straßer. Mole 3.0 : A Middleware for Java-Based Mobile Software Agents. In *Proc. Middleware'98*. Springer-Verlag : Heidelberg, Germany, 1998.

[33] G. Bernard and B. Folliot. Caractéristiques générales du placement dynamique : synthèse et problématique. In *Acte de l'École thématique CNRS Placement dynamique et répartition de charge*, pages 3–22, 1996.

[34] G. Bernard and S. Hamma. Remote memory paging in networks of workstations. In *Proceedings of the SUUG International Conference on Open Systems : Solutions for Open World*, April 1994.

[35] G. Bernard and M. Simatic. A decentralized and efficient algorithm for networks of workstations. In *The European Conference for Open Systems Spring-91*, pages 139–148, 1991.

[36] B. N. Bershad and M. J. Zekauskas. The midway distributed shared memory system. In *93 COMPCON Conference*, pages 528–537, February 1993.

[37] B. Bhargava, P. J. Leu, and S. R. Lian. Experimental evaluation of concurrent checkpointing and rollback-recovery algorithms. In *Proc. IEEE Int'l. Conf. on Data Eng.*, page 182, Los Angeles, CA, February 1990.

[38] R. Bianchini, L. I. Kontothanassis, R. Pinto, M. De Maria, M. Abud, and C. L. Amorim. Hiding communication latency and coherence overhead in software DSMs. In *The 7th International Conference on Arcgitectural Ssupport for Programming Langua ge and Operating Systems*, October 1996.

[39] R. Bianchini, R. Pinto, and C. L. Amorim. Data prefetching for software dsms. In *The International Conference on Spercomputings*, July 1998.

[40] K. P. Birman. Replication and fault-tolerance in the ISIS system. *ACM Operating Systems Review*, 19(5) :79–86, 1985. Proc. 10th ACM Symp. on Operating System Principles, Orcas Island, WA, USA, December 1985.

[41] K. P. Birman and T. A. Joseph. Reliable communication in the presence of failures. *ACM Trans.Comp.Syst.*, 5(1) :47–76, February 1987.

[42] K. P. Birman and R. van Renesse. *Reliable Distributed Computing with the ISIS Toolkit.* IEEE Press, 1994.

[43] A. Borg, W. Blau, W. Craetsch, F. Herrmann, and W. Oberle. Fault tolerance under unix. *ACM Transactions on Computer Systems*, 7(1) :1–24, February 1989.

[44] R. Boutaba and B. Folliot. Load balancing in local area networks. In *In Proc. of The Networks'92 International Conference on Computer Networks, Architecture and Applications*, 1992.

[45] R. Boutaba, B. Folliot, and P. Sens. Efficient resources management in local area networks. In *Proc. of International Conference on Advanced Information Processing Techniques for LAN and MAN Management*, pages II/29–38, Versailles, France, April 1993. North Holland.

[46] D. Briatico, A. Cuiffoletti, and L. Simoncini. A distributed domino-effect free recovery algorithm. In *Symposium on Reliable Distributed Systems (SRDS '84)*, pages 207–215. IEEE Computer Society Press, October 1984.

[47] P. Cadinot and D. Collard. Maîs : un mécanisme de pagination en mémoire distante dans un réseau à haut débit, implémentation et évaluation. In *Acte de la Conférence Française sur les Systèmes d'Exploitation*, pages 243–246, 1999.

[48] P. Cadinot, N. Dorta, B. Folliot, and P. Sens. Swap réparti. In *Actes des Journées de Recherche sur la Mémoire Partagée Répartie*, Bordeaux, France, 1996.

[49] Ph. Cadinot and N. Dorta. MAÎS. Master's thesis, University Pierre & Marie Curie, September 1996.

[50] Ph. Cadinot, N. Dorta, B. Folliot, and P. Sens. MAÎS : Un système de pagination en mémoire distante dans un environnement réseau à haut débit. In *RenPar'9*, May 1997.

[51] A. Cardon. Système de gestion de crises coopératif : un processus d'interprétation de points de vues multiples. *Journal of Decision Systems*, 7 :39–67, 1999.

[52] Nicholas Carriero and David Gelernter. Linda in context. *Communications of the ACM*, 32(4) :444–458, April 1989.

[53] J. Carter, J. Bennett, and W. Zwaenepoel. Implementation and performance of Munin. In ACM, editor, *Proceedings of the 13th Symposium on Operating Systems Principles*, pages 152–164, 1991.

[54] T. Chandra and S. Toueg. Unreliable failure detectors for reliable distributed systems. Technical Report TR95-1535, Cornell University, Computer Science Department, August 1995.

[55] T. D. Chandra, V. Hadzilacos, and Sam Toueg. The weakest failure detector for solving consensus. In Maurice Herlihy, editor, *Proceedings of the 11th Annual ACM Symposium on Principles of Distributed Computing (PODC'92)*, pages 147–158, Vancouver, BC, Canada, August 1992. ACM Press.

[56] T. D. Chandra and S. Toueg. Unreliable failure detectors for reliable distributed systems. *Journal of the ACM*, 43(2) :225–267, March 1996.

[57] K. M. Chandy and Leslie Lamport. Distributed snapshots : determining global states of distributed systems. *ACM Transations on Computing Systems*, 3(1) :63–75, 1985.

[58] H. Clark and B. McMillin. DAWGS - a distributed compute server utilizing idle workstations. *Journal of Parallel and Distributed Computing*, 14 :175–186, February 1992.

[59] D. Comer and J. Griffioen. A new design for distributed systems : The remote memory model. In *Proceedings of the 1990 USENIX Summer Conference*, pages 127–135, June 1990.

[60] A. Cox, Y. Hu, H. Lu, and W. Zwaenepoel. OpenMP on networks of SMP. In *The 13th International Parallel Processing Symposium*, April 1999.

[61] CPLEX Optimization Inc. CPLEX Linear Optimizer and Mixed Integer Optimizer. Suite 279, 930 Tahoe Blvd. Bldg 802, Incline Village, NV 89541.

[62] X. Défago, A. Schiper, and N. Sergent. Semi-passive replication. In *Seventeenth IEEE Symposium on Reliable Distributed Systems (SRDS '98)*, pages 43–50, Washington - Brussels - Tokyo, October 1998. IEEE.

[63] D. Dolev, C. Dwork, and L. Stockmeyer. On the minimal synchronism needed for distributed consensus. *Journal of the ACM*, 34(1) :77–97, January 1987.

[64] Fred Douglis and John Ousterhout. Transparent Process Migration : Design Alternatives and the Sprite Implementation. *Software – Practice and Experience*, 21(8) :757–785, August 1991.

[65] George Dramitinos and Evangelos P. Markatos. Adaptive and reliable paging to remote main memory. *Journal of Parallel and Distributed Computing*, 58(3) :357–388, September 1999.

[66] C. Dwork, N. Lynch, and L. Stockmeyer. Consensus in the presence of partial synchrony. *Journal of the ACM*, 35(2) :288–323, April 1988.

[67] D. L. Eager, E. D. Lazowska, and J. Zahorjan. A Comparison of Receiver-Initiated and Sender-Initiated Adaptive Load Sharing. *Performance Evaluation*, 6(1) :53–68, March 1986.

[68] D. L. Eager, E. D. Lazowska, and J. Zahorjan. Adaptive load sharing in homogeneous distributed systems. *IEEE Trans. on Software Engineering*, 12(5) :662–675, 1986.

[69] D. L. Eager, E. D. Lazowska, and J. Zahorjan. Speedup versus efficiency in parallel systems. *IEEE Transactions. on Computers*, 8(3) :408–423, 1989.

[70] D. L. Eager, Edward D. Lazowska, and John Zahorjan. Dynamic load sharing in homogeneous distributed systems. *IEEE Transactions on Software Engineering*, 12(5) :662–675, May 1986.

[71] E. N. Elnozahy, Johnson D. B., and Wang Y.M. A survey of rollback-recovery protocols in message passing systems. Technical Report CMU-CS-96-181, Department of Computer Science, Carnegie Mellon University, September 1996.

[72] E. N. Elnozahy, D. B. Johnson, and W. Zwaenepoel. The performance of consistent checkpointing. *Proc. of the 11th Symp. on Reliable Distributed Systems*, October 1992.

[73] E. N. Elnozahy and W. Zwaenepoel. the use and implementation of message logging. *Proc. of the 24th Int'l Symp. on Fault-Tolerant Computing Systems*, pages 298–307, June 1994.

[74] A. Erlichson, N. Nuckolls, G. Chesson, and J. Henessy. SoftFLASH : Analysing the performance of clustered distributed virtual shared memory. In *The Seventh Internation Conference on Architectural Support for Programming Languages and Operating Systems*, pages 15–26, October 1996.

[75] D. J. Evans and W. U. N. Butt. Dynamic load balancing using task-transfer probabilities. *Parallel Computing*, 19 :897–916, 1993.

[76] M.J. Feeley. *Global Memory Management for Workstation Networks.* PhD thesis, University of Washington, 1996.

[77] M.J. Feeley, W.E. Morgan, F.H. Pighin, A.R. Karlin, and H.M. Levy. Implementing global memory management in a workstation cluster. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles*, December 1995.

[78] P. Felber, B. Garbinato, and R. Guerraoui. The design of a CORBA group communication service. In *Proceedings of the 15th Symposium on Reliable Distributed Systems (SRDS-15)*, pages 150–159, Niagara-on-the-Lake, Canada, October 1996.

[79] E.W. Felten and J. Zahorjan. Issues in the implementation of a remote memory paging system. Technical Report 91-03-09, Department of Computer Science & Engineering, University of Washington, March 1991.

[80] Domenico Ferrari and Songnian Zhou. An empirical investigation of load indices for load balancing applications. Technical Report CSD-87-353, University of California, Berkeley.

[81] C. Fidge. Logical time in distributed computing systems. *IEEE Computer*, pages 28–33, July 1991.

[82] M. J. Fischer, N. A. Lynch, and M. S. Paterson. Impossibility of distributed consensus with one faulty process. *Journal of the ACM*, 32(2) :374–382, April 1985.

[83] R. P. Fitzgerald. *A Performance Evaluation of the Integration of Virtual Memory Management and Inter-Process Communication.* PhD thesis, Carnegie Mellon University, October 1986. Available as CMU technical report CMU-CS-86-158.

[84] B. Folliot. *Méthodes et outils de partage de charge pour la conception et la mise en oeuvre d'applications dans les systèmes répartis hétérogènes.* PhD thesis, Université Pierre et Marie Curie, 1992.

[85] B. Folliot. Contribution à une approche système du placement dynamique dans les systèmes répartis hétérogènes, December 1996. Thèse d'habilitation de l'Université Pierre et Marie Curie.

[86] B. Folliot and P. Sens et P-G. Raverdy. Plate-forme de répartition de charge et de tolérance aux fautes pour applications parallèles en environnement réparti. *Calculateurs Parallèles*, 7(4) :345–366, 1995.

[87] B. Folliot, K. Foughali, and P. Sens. Placement d'applications parallèles dans les réseaux de stations de travail : l'expérience de gatostar. In *Actes du congrès sur le Placement Dynamique et la Répartition de Charge : Applications aux Systèmes Répartis et Parallèles*, pages 75–78, Paris, France, May 1995. GDR PRS / CNRS.

[88] B. Folliot and P. Sens. GATOSTAR : A fault-tolerant load sharing facility for parallel applications. In K. Echtle, D. Hammer, and D. Powell, editors, *Proceedings of the First European Dependable Computing Conference*, volume 852 of *Lectures Notes in Computer Science*. Springer Verlag, 1994.

[89] P. Folliot and P. Sens. Load sharing and fault tolerance manager. In *High Performance Cluster Computing*, chapter 22, pages 534–552. Rajkumar Buyya (ed), Prentice-Hall, 1999.

[90] K. Foughali and R. Boutaba. A distributed open platform and its federation for tools interworking in software engineering environments. In *Integrated Computer-Aided Engineering Journal*, 1995.

[91] B. Garbinato and R. Guerraoui. Flexible protocol composition in Bast. In *Proceedings of the 18th International Conference on Distributed Computing Systems (ICDCS-18)*, pages 22–29, Amsterdam, The Netherlands, May 1998. IEEE Computer Society Press.

[92] J. Gharachorloo, D. Lenoski, J. Laudon, Gupta A., and J. Henessy. Memory consistency and event ordering in scalable shared-memory multiprocessors. In *The 17th International Symposium on Computer Architecture*, pages 15–26, 1990.

[93] G. Glass. ObjectSpace voyager — the agent ORB for Java. *Lecture Notes in Computer Science*, 1368 :38– ? ?, 1998.

[94] Rachid Guerraoui, Benoît Garbinato, and Karim R. Mazouni. Garf : A tool for programming reliable distributed applications. *IEEE Concurrency*, 5(4) :32–39, October/ December 1997.

[95] Z. Guessoum. Dima : Une plate-forme multi-agents en smalltalk. *Revue Objet*, 3(4) :393–410, 1998.

[96] Olivier Gutknecht and Jacques Ferber. MadKit : A generic multi-agent platform. In *Proceedings of the Fourth International Conference on Autonomous Agents*, pages 78–79, Barcelona, Catalonia, Spain, June 2000. ACM Press. Poster announcement.

[97] V. Hadzilacos and S. Toueg. A modular approach to fault-tolerant broadcasts and related problems. Technical Report TR94-1425, Cornell University, Computer Science Department, May 1994.

[98] Y. Haj-Mahmoud. *Modélisation et évaluation de performances des systèmes de distribution de charge en environnements répartis.* PhD thesis, Université Pierre et Marie Curie, 1999.

[99] Y. Haj-Mahmoud, B. Folliot, and P. Sens. Performance evaluation of a load sharing system on a cluster of workstations. In *Proceeding of the International Conference on High Performance Computing*, Lecture Notes in Computer Science, Calcutta, Indes, December 1999.

[100] Y. Haj-Mahmoud, B. Folliot, and P. Sens. Quantifying the performance improvement of migration mechanism in load distributing systems. In *Proceedings of International Conference on Parallel and Distributed Processing Techniques and Applications*, Las Vegas, USA, July 1999.

[101] Y. Haj-Mahmoud, B. Folliot, and P. Sens. Simulation et evaluation de performances d'un système de Répartion de charge. *Technique et Science Informatiques*, 18(9) :1005–1028, November 1999.

[102] M. Harchol-Balter and A. B. Downey. Exploiting process lifetime distributions for dynamic load balancing. *ACM Transactions on Computer Systems*, 15(3), 1997.

[103] J. M. Hélary, A. Mostefaoui, R. H. B. Netzer, and M. Rayna l. Preventing useless checkpoints in distributed computation. *Proc. of the 27-th IEEE Symp. on Fault-Tolerant Computing Systems*, pages 68–77, June 1997.

[104] J. Hylton, K. Manheimer, F. L. Drake, Jr., B. Warsaw, R. Masse, and G. van Rossum. Knowbot programming : System support for mobile agents. In *Proceedings of the Fifth International Workshop on Object Orientation in Operating Systems*, pages 8–13, Seattle, Wash., October 1996.

[105] L. Iftode, K. Li, and K. Petersen. Memory servers for multicomputers. In *Proceedings of the IEEE Spring COMPCON '93*, pages 538–547, February 1993.

[106] IONA and Isis. An introduction to Orbix+Isis, 1994. IONA Technologies Ltd, and Isis Distributed Systems, Inc.

[107] Ayal Itzkovitz, Assaf Schuster, and Lea Shalev. Thread migration and its applications in distributed shared memory systems. *The Journal of Systems and Software*, 42(1) :71–87, July 1998.

[108] D. B. Johnson and W. Zwaenepoel. Sender-based message logging. *Proc. of the 7th Symp. on Fault Tolerant Computing Systems*, pages 97–104, June 1990.

[109] M. Frans Kaashoek, Raymond Michiels, Henri E. Bal, and A. Tanenbaum. Transparent fault-tolerance in parallel Orca programs. In USENIX Association, editor, *Symposium on Experiences with Distributed and Multiprocessor Systems (SEDMS), March 26–27, 1992. Newport Beach, CA*, pages 297–312, Berkeley, CA, USA, March 1992. USENIX.

[110] Z. Kalbarczyk, S. Bagchi, K. Whisnant, and R. Iyer. Chameleon : A software infrastructure for adaptive fault tolerance. *IEEE Transactions on Parallel and Distributed Systems*, June 1999.

[111] M. Karllson and P. Stenstrom. Effectiveness of dynamic prefetching in multiple-writers distributed virtual shared memory systems. *Journal of Parallel and Distributed Computing*, 42(7), July 1997.

[112] P. Keleher. *Lazy Release Consistency for Distributed Shared Memory.* PhD thesis, Texas University, Texas (EUA), 1995.

[113] A.-M. Kermarrec, C. Morin, and M. Banâtre. Design, implementation and evaluation of ICARE : an efficient recoverable DSM. *Software Practice and Experience*, 28(9) :981–1010, July 1998.

[114] R. Koo and S. Toueg. Checkpointing and rollback-recovery for distributed systems. *IEEE Transactions on Software Engineering*, SE-13(1) :13–21, January 1987.

[115] P. Krueger and M. Livny. The diverse objectives of distributed scheduling policies. In *Proceedings of The Seventh International Conference of Distributed Computing Systems*, pages 242–249, 1987.

[116] T. Lai and T. Yang. On Distributed Snapshots. *Information Processing Letters*, 25 :153–158, 1987.

[117] L. Lamport, R. Shostak, and M. Pease. The Byzantine generals problem. *ACM Transactions on Programming Languages and Systems*, 4(3) :382–401, July 1982.

[118] W. Leland and T. Ott. Load balancing heuristics and process behavior. *ACM Performance Evaluation*, 14 :54–69, 1986.

[119] P.Y. Leu and B. Bhargava. Concurrent Robust Checkpointing and Recovery in Distributed Systems. In *Proceedings of the 4th IEEE International Conference on Data Engeneering*, pages 154–163, 1988.

[120] H. Lin and C. S. Raghavendra. A dynamic load balancing policy with a central job dispatcher (LBC). *IEEE Transactions on Software Engineering*, 18(2) :264–271, 1991.

[121] M. J. Litzkow, M. Livny, and M. W. Mutka. Condor–a hunter of idle workstations. In *Proceedings of the 8th Internatinal Conference on Ditributed Computing Systems*, pages 104–111, 1988.

[122] Michael Litzkow, Todd Tannenbaum, Jim Basney, and Miron Livny. Checkpoint and migration of UNIX processes in the condor distributed processing system. Technical Report CS-TR-1997-1346, University of Wisconsin, Madison, April 1997.

[123] M. Livny and M. Melman. Load balancing in homogeneous broadcast distributed systems. *ACM Performance Evaluation*, 11(1) :47–55, 1982.

[124] S. Maffeis. Electra — making distributed programs object-oriented. In USENIX Association, editor, *Proceedings of the USENIX Symposium on Experiences with Distributed and Multiprocessor Systems (SEDMS IV) : September 22–23, 1993, San Diego, California, USA*, pages 143–156, Berkeley, CA, USA, September 1993. USENIX.

[125] C. P. Malloth. *Conception and Implementation of a Toolkit for Building Fault-Tolerant Distributed Applications in Large Scale Networks.* PhD thesis, École Polytechnique Fédérale de Lausanne, Switzerland, September 1996.

[126] E.P. Markatos. Issues in reliable network memory paging. In *Proceedings of MASCOTS '96*, February 1996.

[127] E.P. Markatos and G. Dramitinos. Implementation and evaluation of a remote memory pager. Technical Report 129, FORTH/ICS, 1995.

[128] E.P. Markatos and G. Dramitinos. Implementation of a reliable remote memory pager. In *Proceedings of the USENIX 96 Technical Conference*, January 1996.

[129] F. Mattern. Virtual time and global states in distributed systems. In *Workshop on Parallel and Distributed Algorithms*, Elsevier (Holland), October 1988.

[130] K. R. Mazouni. *Étude de l'invocation entre objets dupliqués dans un système réparti tolérant aux fautes.* PhD thesis, École Polytechnique Fédérale de Lausanne, Switzerland, January 1996.

[131] M. K. McKusick, K. Bostic, M. J. Karels, and J. S. Quarterman. *The design and implementation of the 4.4BSD operating system.* ADDISON-WESLEYT, 1996.

[132] M.K. McKusick, K. Bostic, M.J. Karels, and J.S. Quarterman. *Conception et implémentation du système 4.4BSD*. International Thomson Publishing, 1997.

[133] J. Menden and G. Stellner. Proving properties of PVM applications — a case study with CoCheck. In Arndt Bode, Jack Dongarra, T. Ludwig, and V. Sunderam, editors, *Parallel virtual machine, EuroPVM '96 : third European PVM conference, Munich, Germany, October 7–9, 1996 : proceedings*, volume 1156 of *Lecture Notes in Computer Science*, pages 134–??, Berlin, Germany / Heidelberg, Germany / London, UK / etc., 1996. Springer-Verlag.

[134] Sun Microsystem. Java development kit 1.2.

[135] Sun Microsystem. Java remote method invocation specification.

[136] L. R. Monnerat and R. Bianchini. Efficiently adapting to sharing patterns in software dsms. In *The Fourth International Symposium on High Performance Computer Architecture*, February 1998.

[137] L. E. Moser, P. M. Melliar-Smith, D. A. Agarwal, R. K. Budhia, and C. A. Lingley-Papadopoulos. Totem : A fault-tolerant multicast group communication system. *Communications of the ACM*, 39(4) :54–63, April 1996.

[138] L. E. Moser, P. M. Melliar-Smith, and P. Narasimhan. A fault tolerance framework for CORBA. In *Proceedings of the 29th Annual International Symposium on Fault-Tolerant Computing (FTCS)*, Washington, DC, June 1999. IEEE Computer Society.

[139] J. E. B. Moss. *Nested Transactions an Approach to Reliable Distributed Computing*. MIT Press, Cambridge, Mass., 1985.

[140] T. Mowry, C. Chan, and Lo. A. Comparative evaluation of latency tolerance techniques for software distributed shared memory. In *The Fourth International Symposium on High Performance Computer Architecture*, February 1998.

[141] G. Muller, M. Hue, and N. Peyrouze. Operating system : results of the FTM experiment. In Klaus Echtle, Dieter Hammer, and David Powell, editors, *Dependable computing–EDCC-1 : first European Dependable Computing Conference, Berlin, Germany, October 4–6, 1994 : proceedings*, volume 852 of *Lecture Notes in Computer Science*, pages 491–508, Berlin, Germany / Heidelberg, Germany / London, UK / etc., 1994. Springer Verlag.

[142] B. Natarajan, A. Gokhale, D. Schmidt, and S. Yajnik. Doors : Towards high-performance fault-tolerant corba. In *Proceedings of the 2 nd International Symposium on Distributed Objects and Applications*, Antwerp,Belgium, September 2000.

[143] OMG. Fault tolerant CORBA specification v1.0, April 2000.

[144] Scott Pakin, Vijay Karamcheti, and Andrew A. Chien. Fast Messages : Efficient, portable communication for workstation clusters and MPPs. *IEEE Concurrency*, 5(2) :60–73, April/June 1997.

[145] K. Petersen. *Operating System Support for Modern Memory Hierarchies*. PhD thesis, Princeton University, october 1993. also available as Technical Reports 431-1 and 431-2.

[146] N. Peyrouze and G. Muller. FT-NFS : an efficient fault tolerant NFS server designed for off-the-shelf workstations. In *International Symposium on Fault-Tolerant Computing (FTCS '96*. IEEE Computer Society Press, June 1996.

[147] James S. Plank, Micah Beck, Gerry Kingsley, and Kai Li. Libckpt : Transparent Checkpointing under Unix. In *USENIX Technical Conference Proceedings*, January 1995.

[148] James S. Plank and Kai Li. Ickp – A Consistent Checkpointer for Multicomputers. *IEEE Parallel and Distributed Technologies*, 2(2) :62–67, Summer 1994.

[149] Platform Computing Corporation. *LSF : Load Sharing Facility Administrator's Guide*, December 1994.

[150] D. Powell. *Delta-4 : a Generic Architecture for Dependable Distributed Computing.* 1991.

[151] David Powell. Failure mode assumptions and assumption coverage. In Dhiraj K. Pradhan, editor, *Proceedings of the 22nd Annual International Symposium on Fault-Tolerant Computing (FTCS '92)*, pages 386–395, Boston, MA, July 1992. IEEE Computer Society Press.

[152] M.L. Powell and B.P. Miller. Process migration in DEMOS/MP. *Operating Systems Review*, 17(5) :110–119, October 1983.

[153] David Leo Presotto. PUBLISHING : A reliable broadcast communication mechanism. Technical Report CSD-83-165, University of California, Berkeley, 1983.

[154] L. Prylli and B. Tourancheau. BIP : A new protocol designed for high performance networking on myrinet. *Lecture Notes in Computer Science*, 1388 :472– ? ?, 1998.

[155] I. Puaut, M. Banatre, and J-P. Routeau. Early experience with building and using the Gothic distributed operating system. In USENIX Association, editor, *Symposium on Experiences with Distributed and Multiprocessor Systems (SEDMS II), March 21–22, 1991. Atlanta GA*, pages 271–282, Berkeley, CA, USA, March 1991. USENIX.

[156] B. Randell. System Structure for Software Fault-tolerance. *IEEE Transactions on Software Engineering*, SE-1(2), June 1975.

[157] C. R. Reeves. *Modern Heuristic Techniques for Combinatorial Problems.* Blackwell Scientific Publications, 1993.

[158] V. Reibaldi. Rcube specification. Technical report, MASI CAO-VLSI UPMC Paris VI, 1997.

[159] Rice University. *Concurrent Program with TreadMarks - Users'guide*, 1994. version 0.9.8 and 0.10.1.

[160] David L. Russell. State restoration in systems of communicating processes. *IEEE Transactions on Software Engineering*, 6(2) :183–194, March 1980.

[161] R. Samanta, A. Bilas, L. Iftode, and J. Singh. Home-based SVM protocols for SMP clusters : Design and performance. In *The 4th Symposium on High Performance Computer Architecture*, February 1998.

[162] D. Scales, Gharachorloo, and A. Aggarwal. Fine-grain software distributed shared memory on smp clusters. In *The Fourth International Symposium on High Performance Computer Architecture*, February 1998.

[163] B.N. Schilit and D. Duchamp. Adaptive remote paging for mobile computers. Technical Report TR CUCS-004-91, Department of Computer Science, Columbia University, February 1991.

[164] Fred B. Schneider. Implementing fault-tolerant services using the state machine approach : A tutorial. *ACM Computing Surveys*, 22(4) :299–319, December 1990.

[165] P. Sens. Conception et mise en œvre d'une plate-forme logicielle de tolérance aux fautes pour le support d'applications réparties, December 1994. Thèse de doctorat Université Pierre et Marie Curie.

[166] P. Sens. The performance of independent checkpointing in distributed systems. In *Proceedings of the 28th Hawai International Conference on System Sciences*, Maui, Hawaii, January 1995. IEEE press.

[167] P. Sens and B. Folliot. STAR : A fault-tolerant system for distributed applications. In *Proceedings of the 5th IEEE Symposium on Parallel and Distributed Processing*, December 1993.

[168] P. Sens and B. Folliot. Performance evaluation of fault tolerance for parallel applications in networked environments. In *Proceedings of 26th International Conference on Parallel Processing*, pages 334–341. IEEE, August 1997.

[169] P. Sens and B. Folliot. The STAR fault manager for distributed operating environments : Design, implementation and performance. *Software Practice and Experience*, 28(10) :1079–1099, August 1998.

[170] P. Sens, F. Popentiu-Valdicesdu, and M. Adrian. Software reliability forecasting for adapted fault tolerance algorithms. In *Proceedings of the European Safety and Reliability Conference*, June 1998.

[171] G-W. Sheu, Y-S. Chang, D. Liang, S-M. Yuan, and W. Lo. A fault-tolerant object service on CORBA. In *Proceedings of the 17th International Conference on Distributed Computing Systems (ICDCS-17)*, pages 393–400, Baltimore, USA, May 1997. IEEE Computer Society Press.

[172] N. G. Shivaratri, P. Krueger, and M. Singhal. Load distributing for locally distributed systems. *Computer Journal*, 25 :33–44, 1992.

[173] G. C. Shoja. A distributed facility for load sharing and parallel processing among workstations. *The Journal of Systems and Software*, 14(3) :163– ? ?, March 1991.

[174] S. K. Shrivastava. Lessons learned from building and using the Arjuna distributed programming system. *Lecture Notes in Computer Science*, 938 :17– ? ?, 1995.

[175] Simulog S.A. *QNAP2 Reference Manual*, 1995.

[176] P. Singh, W. Weber, and A. Gupta. Splash : Stanford parallel applications for shared-memory. *Computer Architecture News*, 20(1) :5–44, March 1992.

[177] M. Singhal and A. Kshemkalyani. An efficient implementation of vector clocks. *Information Processing Letters*, 33 :47–53, August 1992.

[178] A. P. Sistla and J. L. Welch. Efficient distributed recovery using message logging. *Proc. of the 8th Annual ACM Symp. on Principles of Distributed Co mputing*, August 1989.

[179] N. A. Speirs and P. A. Barrett. Using passive replicates in delta-4 to provide dependable distributed computing. In *International Symposium on Fault-Tolerant Computing (FTCS '89*, pages 184–190, Washington, D.C., USA, June 1989. IEEE Computer Society Press.

[180] M. G. Sriram and M. Singhal. Measures of the potential for load sharing in distributed systems. *IEEE Transactions on Software Engineering*, 21(5) :468–475, 1995.

[181] G. Stellner. CoCheck : checkpointing and process migration for MPI. In IEEE, editor, *Proceedings of IPPS '96. The 10th International Parallel Processing Symposium : Honolulu, HI, USA, 15–19 April 1996*, pages 526–531, 1109 Spring Street, Suite 300, Silver Spring, MD 20910, USA, 1996. IEEE Computer Society Press.

[182] R. Stets, S. Dwarkadas, N. Hardavellas, G. Hunt, H. Kontothannassis, Parhasarathy, and M. Scott. Cashmere-2L : software coherent shared memory on a clustered remote-write network. In *The 16th ACM Symposium on Operating System Principles*, pages 170–183, October 1997.

[183] D. B. Stewart, R. A. Volpe, and P. K. Khosla. Design of dynamically reconfigurable real-time software using port-based objects. *IEEE Transactions on Software Engineering*, 23(12) :759–776, December 1997.

[184] R. E. Strom and S. A. Yemini. Optimistic recovery in distributed systems. *ACM Transactions on Computer Systems*, 3(3) :204–226, August 1985.

[185] D. C. Sturman and G. A. Agha. A protocol description language for customizing failure semantics. In *Symposium on Reliable Distributed Systems (SRDS '94)*, pages 148–159, Los Alamitos, Ca., USA, October 1994. IEEE Computer Society Press.

[186] E-G. Talbi, J-M. Geib, Z. Hafidi, and D. Kebbal. MARS : An adaptive parallel programming environment. In *High Performance Cluster Computing*, chapter 31, pages 722–739. Rajkumar Buyya (ed), Prentice-Hall, 1999.

[187] E. G. Talbi, Z. Hafidi, and J-M. Geib. A parallel adaptive tabu search approach. *Parallel Computing*, 24(14) :2003–2019, December 1998.

[188] Y. Tamir and H. Sequin. Error-recovery in multicomputers using global checkpoints. In *Proceedings of the International on Parallel Processing*, pages 32–41. IEEE Computer Society Press, October 1984.

[189] T. Tannenbaum and M. Litzkow. The Condor Distributed Processing System. *Dr. Dobb's Journal*, (2) :40–48, February 1995.

[190] A. N. Tantawi and D. Towsley. Optimal Static Load Balancing in Distributed Systems. *Journal of the ACM*, 32(5) :445–465, April 1985.

[191] M. M. Theimer and K. A. Lantz. Finding idle machines in a workstation-based distributed system. *IEEE Trans. on Software Engineering*, 15 :1444–1458, 1989.

[192] M. M. Theimer, K. A. Lantz, and D. R. Cheriton. Preemptable Remote Execution Facilities for the V-System. *Operating Systems Review*, 19(3) :2–12, December 1985.

[193] Z. Tong, R.Y. Kain, and W.T. Tsai. A lower overhead checkpointing and rollback recovery scheme for distributed systems. In *Symposium on Reliable Distributed Systems (SRDS '89)*, pages 12–20. IEEE Computer Society Press, October 1989.

[194] F. Torres-Rojas and M. Ahamad. Plausible clocks : Constant size logical clocks for distributed systems. In *10th International Workshop on Distributed Algorithms*, Bologna (Italy), October 1996.

[195] J. Tsai, Y-M. Wang, and S-Y. Kuo. Evaluations of domino-free communication-induced checkpointing protocols. *Information Processing Letters*, 69(1) :31–37, 15 January 1999.

[196] R. van Renesse, K. P. Birman, and S. Maffeis. Horus : A flexible group communication system. *CACM*, 39(4) :76–83, April 1996.

[197] D. Vigo and V. Maniezzo. A genetic/tabu thresholding hybrid algorithm for the process allocation problem. *Journal of Heuristics*, 3(2) :91–110, 1997.

[198] Thorsten von Eicken, David E. Culler, Seth Copen Goldstein, and Klaus Erik Schauser. Active messages : a mechanism for integrated communication and computation. Technical Report CSD-92-675, University of California, Berkeley, March 1992.

[199] Y.-M. Wang. Maximum and minimum consistent global checkpoints and their applications. In *Symposium on Reliable Distributed Systems (SRDS '95)*, pages 86–95, Los Alamitos, Ca., USA, September 1995. IEEE Computer Society Press.

[200] Y.-M. Wang, P. Chung, I. Lin, and W. K. Fuchs. Checkpoint space reclamation for uncoordinated checkpointing in message-passing systems. *IEEE Transactions on Parallel and Distributed Systems*, 6(5) :546–554, May 1995.

[201] Y.-M. Wang and W. K. Fuchs. Optimistic message logging for independent checkpointing in message-passing systems. In *Symposium on Reliable Distributed Systems (SRDS '92)*, pages 147–154. IEEE Computer Society Press, October 1992.

[202] M. Welsh, A. Basu, and T. von Eicken. ATM and fast ethernet network interfaces for user-level communication. In *Proceedings of the Third International Symposium on High Performance Computer Architecture (HPCA '97)*, pages 332–342, Los Alamitos, Ca., USA, February 1997. IEEE Computer Society Press.

[203] J. Xu and R. H. B. Netzer. Adaptive independent checkpointing for reducing rollback adaptive independent checkpointing for reducing rollback. *Proc. of the 5th IEEE Symp. on Parallel and Distributed*, pages 754–761, December 1993.

[204] D. Yeung, J. Kubiatowicks, and A. Agarwal. Mgs : A multi-grain shared memory system. In *The 23rd Annual International Symposium on Computer Architecture*, pages 44–55, April 1996.

[205] Yasuhiko Yokote. Kernel structuring of object-oriented operating systems : The apertos approach. *Lecture Notes in Computer Science*, 742 :145– ? ?, 1993.

[206] Y. Zhang, K. Hakozaki, H. Kameda, and K. Shimizu. A performance comparison of adaptive and static load balancing in heterogeneous distributed systems. In *28th Annual Simulation Symposium*, pages 332–340, 1995.

[207] S. Zhou, X. Zheng, J. Wang, and P. Delisle. Utopia : a load sharing facility for large, heterogeneous distributed computer systems. *Software-practice and experience*, 23(12) :1305–1336, 1993.

[208] Y. Zhou, L. Iftode, and K. Li. Performance evaluation of two home-based lazy release consistency protocols for shared irtual memory systems. In *The 2nd Symposium on Operating Systems Design and Implementation (OSDI 96)*, October 1996.

[209] J. Zimmerman. Dima agent replication extension, June 1999. Rapport de Stage de DEA Systèmes Informatique.

Index