tel-00434053, version 1 - 20 Nov 2009

HABILITATION À DIRIGER DES RECHERCHES

présentée devant

L'Université de Rennes 1
Institut de Formation Supérieure
en Informatique et en Communication

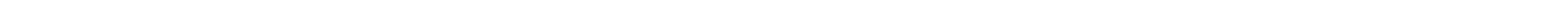
par

Christine Morin

Architectures et systèmes distribués tolérants aux fautes

soutenue le 5 mars 1998 devant le jury composé de

MM	Jean-Pierre Banâtre	Président
	Michel Banâtre	Examinateur
	Michel Diaz	Examinateur
	Sacha Krakowiak	Rapporteur
	Pete Lee	Rapporteur
	Kurt Rothermel	Rapporteur
	André Seznec	Examinateur



tel-00434053, version 1 - 20 Nov 2009

tel-00434053, version 1 - 20 Nov 2009

"*Le monde ne sera heureux que quand tous les hommes auront des âmes d'artistes, c'est-à-dire quand tous prendront plaisir à leur tâche.*" A. Rodin

à mon mari Denis,
à mes enfants Lucie, Cécile et Brice,
à mes parents.

tel-00434053, version 1 - 20 Nov 2009

tel-00434053, version 1 - 20 Nov 2009

Remerciements

Je tiens à remercier Jean-Pierre Banâtre, professeur à l'université de Rennes I, directeur de l'IRISA, qui me fait l'honneur de présider ce jury. Je lui suis reconnaissante pour les conseils qu'il m'a prodigués ces dernières années.

J'adresse également mes plus vifs remerciements à Michel Banâtre, directeur de recherche à l'INRIA, responsable du projet Solidor, pour la confiance qu'il m'a témoignée dans la conduite de mes travaux. Je le remercie pour la disponibilité dont il a toujours fait preuve à mon égard et pour le soutien qu'il m'a apporté tout au long de ces années.

Mes remerciements vont aussi à Sacha Krakowiak, professeur à l'université Joseph Fourier de Grenoble, à Pete Lee, professeur à l'université de Newcastle upon Tyne et à Kurt Rothermel, professeur à l'université de Stuttgart pour avoir accepter d'être les rapporteurs de ce travail.

Je suis reconnaissante à Pete Lee pour les fructueuses discussions que nous avons eues dans le cadre du projet FASST. Elles ont sans aucun doute influencé mes travaux de recherche.

Je remercie également Michel Diaz, directeur de recherche CNRS au LAAS de Toulouse, pour avoir accepté d'être membre du jury.

Enfin, j'exprime ma sincère reconnaissance à André Seznec, directeur de recherche de l'INRIA, pour sa participation au jury et pour l'intérêt qu'il a toujours porté aux travaux présentés dans ce document et ses conseils avisés.

Les travaux présentés dans ce document sont le fruit de plusieurs années de recherche au sein du projet LSP (Langages et Systèmes Parallèles) puis du projet Solidor qui lui a succédé en 1994. Ils ont été effectés en collaboration avec plusieurs chercheurs et étudiants en thèse parmi lesquels je tiens à remercier tout particulièrement : Alain Gefflaut, Philippe Joubert, Anne-Marie Kermarrec et Isabelle Puaut.

Je tiens également à remercier les membres du projet européen FASST et plus spécialement Cornelius Frankenfeld et les chercheurs du projet Model qui y ont participé.

tel-00434053, version 1 - 20 Nov 2009

tel-00434053, version 1 - 20 Nov 2009

Table des matières

1 Introduction **7**

2 Communications fiables dans les systèmes distribués **11**
2.1 Le problème des communications dans les systèmes distribués 11
2.2 Modèles de communication pour les applications distribuées 12
2.2.1 Communications biparties 12
2.2.2 Communications multiparties 15
2.3 Définition et mise en œuvre de l'appel de multiprocédure à distance 19
2.3.1 Le système Gothic et sa mémoire stable rapide 19
2.3.2 Définition du protocole RMPC 20
2.3.3 Le protocole RMC 22
2.3.4 Le protocole RBC 23
2.3.5 Evaluation de performance 24
2.3.6 Conclusion 25
2.4 Bilan 25

3 Multiprocesseurs à mémoire partagée tolérants aux fautes **27**
3.1 Introduction 27
3.2 Les multiprocesseurs à mémoire partagée 28
3.3 Tolérance aux fautes dans les multiprocesseurs à mémoire partagée 28
3.3.1 Masquage des erreurs 29
3.3.2 Détection d'erreur et recouvrement arrière 29
3.4 Mise en œuvre du recouvrement arrière dans un multiprocesseur à mémoire partagée 29
3.4.1 Principe 29
3.4.2 Impact des communications 30
3.4.3 Le stockage des données de récupération 34
3.4.4 Facteurs d'efficacité d'une stratégie de recouvrement arrière 36
3.5 Approche fondée sur une mémoire partagée récupérable 36
3.5.1 Protocole de récupération arrière 36
3.5.2 Mise en œuvre de la RSM 40
3.5.3 Evaluation de performance 42
3.6 Bilan 44

4 Architectures extensibles tolérantes aux fautes **45**
4.1 Introduction . . . 45
4.2 Architectures extensibles à mémoire partagée . . . 46
4.2.1 Les architectures NUMA . . . 46
4.2.2 Les systèmes à mémoire virtuelle partagée . . . 47
4.3 Recouvrement arrière dans les architectures à mémoire distribuée cohérente 49
4.3.1 Garantie d'un état global cohérent . . . 49
4.3.2 Stockage des données de récupération . . . 52
4.3.3 Gestion des informations de cohérence . . . 54
4.3.4 Synthèse . . . 55
4.4 Réplication pour l'efficacité et la disponibilité . . . 56
4.5 Extension du protocole de cohérence pour la mise en œuvre du recouvrement arrière . . . 57
4.5.1 Etablissement d'un point de récupération . . . 60
4.5.2 Restauration d'un point de récupération . . . 61
4.5.3 Avantages . . . 61
4.5.4 Contrôle de la réplication des données de récupération . . . 62
4.6 Mise en œuvre du protocole ECP . . . 63
4.6.1 Mise en œuvre dans un COMA . . . 63
4.6.2 Mise en œuvre dans un système à MVP . . . 64
4.7 Evaluation de performance . . . 66
4.7.1 Efficacité . . . 66
4.7.2 Extensibilité . . . 68
4.8 Bilan . . . 69

5 Conclusion **71**
5.1 Démarche de recherche . . . 71
5.2 Intérêt des approches à base de composants standard . . . 72
5.3 Intégration de mécanismes standard et de tolérance aux fautes . . . 72
5.4 Système d'exploitation pour architectures multiprocesseurs tolérantes aux fautes . . . 73
5.5 Systèmes distribués et architectures multiprocesseurs extensibles . . . 73
5.6 Applications . . . 74

tel-00434053, version 1 - 20 Nov 2009

tel-00434053, version 1 - 20 Nov 2009

Chapitre 1

Introduction

La tolérance aux fautes est née, il y a une cinquantaine d'années, avec les ordinateurs digitaux électroniques. La fiabilité des composants avec lesquels étaient construites les premières machines informatiques (tubes, relais, ...) était telle que la machine n'était pas capable d'exécuter un programme de plus d'une centaine de milliers de cycles sans être victime d'une défaillance. Ainsi, les programmeurs de l'ENIAC* étaient obligés de concevoir leurs applications scientifiques en étapes de calcul successives de telle sorte que l'application puisse être reprise à n'importe quelle étape intermédiaire pourvu qu'on lui fournisse les résultats des étapes précédentes [28]. L'idée du point de reprise était déjà née.

Le premier ordinateur commercial, l'UNIVAC I, faisait appel à plusieurs techniques de tolérance aux fautes telles que le contrôle de parité et une unité arithmétique et logique dupliquée intégrant un comparateur. A cette époque, l'identification d'un composant défaillant était effectuée par échange ou permutation d'éléments suspects ou à l'aide d'un oscilloscope. La facturation étant effectuée en fonction du temps de calcul, les programmeurs prévoyaient dans leurs applications de nombreux points de reprise sauvegardés sur bande magnétique de façon à éviter la réexécution de leurs programmes en cas de défaillance d'un des éléments de l'ordinateur.

L'invention des mémoires magnétiques et du transistor a considérablement amélioré la fiabilité matérielle des ordinateurs à partir de la fin des années 50. Mais en même temps, la complexité des machines s'est accrue rendant toujours nécessaires les techniques de tolérance aux fautes pour permettre aux utilisateurs des machines informatiques de placer une confiance justifiée dans les résultats fournis.

Les premiers ordinateurs *fiables* étaient utilisés pour des applications militaires (chiffrement, radar, lancement et guidage de missiles,...). D'autres domaines d'applications ont ensuite vu le jour dans les communications, l'industrie, l'aérospatiale et pour un usage commercial avec les systèmes transactionnels (par exemple, le système SABRE de réservation de places d'avion opérationnel en 1964). Aujourd'hui les ordinateurs sont utilisés dans tous les domaines professionnels et même à domicile. L'intérêt pour les techniques de tolérance aux fautes n'en est que renforcé du fait de l'impact économique non négligeable d'un *crash* informatique. Cependant les utilisateurs d'un système informatique ne sont en général pas prêts à faire des concessions sur les performances. En effet, les défaillances

tel-00434053, version 1 - 20 Nov 2009

ne se produisent qu'exceptionnellement tandis que les utilisateurs perçoivent au quotidien leur système à travers ses performances. Il est donc essentiel de concilier dans les systèmes informatiques efficacité et tolérance aux fautes.

Dans le projet Solidor, nous nous intéressons à la construction de systèmes et d'applications distribués. Les travaux menés vont de la programmation des applications réparties à la conception et la réalisation des supports d'exécution de ces applications. Les travaux de recherche que j'ai effectués à l'IRISA dans le projet Langages et Systèmes Parallèles puis dans le projet Solidor [1] entre 1987 et 1997 ont trait à la conception d'architectures et de systèmes pour l'exécution d'applications distribuées.

Ces architectures et systèmes sont avant tout conçus pour répondre aux exigences de qualité de service des applications telles que la sécurité, la fiabilité, la disponibilité, l'extensibilité ou le temps de réponse. Je me suis plus particulièrement intéressée à la conception d'architectures et systèmes satisfaisant les contraintes de disponibilité et de fiabilité des applications. Ma problématique a été de tenter d'apporter des éléments de réponse à la question suivante : comment concilier efficacité et tolérance aux fautes dans des systèmes construits à partir de composants standard tout en assurant la transparence de la tolérance aux fautes pour les applications ?

Les approches à base de composants standard constituent une tendance générale à la conception des systèmes informatiques actuels. Elles permettent de réduire les coûts de développement en permettant la réutilisation de composants et leur spécialisation dans le cas de composants logiciels. La réalisation d'architectures à base de composants matériels standard a pour conséquence leur grande diffusion à une large gamme d'utilisateurs.

La transparence de la tolérance aux fautes pour les applications est aussi souhaitable compte tenu de la complexité des applications actuelles. Les mécanismes de tolérance aux fautes sont transparents pour les applications si les programmeurs d'applications peuvent développer leurs programmes sans avoir à prendre en compte les défaillances. Il est en effet préférable que le programmeur se concentre sur la conception des composants intrinsèques de l'application plutôt que sur la gestion des défaillances qui s'avère complexe. Le programmeur peut se contenter de spécifier les contraintes de l'application en terme de qualité de service, la réalisation de ces contraintes étant à la charge des concepteurs des composants système ou de l'architecture sous-jacents. Les outils de génie logiciel du futur permettront à partir des contraintes de qualité de service spécifiées par le programmeur de sélectionner quasi automatiquement les composants systèmes qui permettent de les assurer [56].

Les applications distribuées sont extrêmement diverses : applications parallèles, applications transactionnelles, applications client/serveur, applications de travail coopératif, applications multimédias... Le support d'exécution des applications dépend de leur nature. Traditionnellement, les architectures multiprocesseurs servent de support aux applications scientifiques et transactionnelles. D'autres applications telles que les applications de travail coopératif ou les applications multimédias grand public sont distribuées par nature et s'exécutent donc sur des architectures distribuées qui vont des réseaux locaux de stations de travail à l'Internet. Par exemple, pour faire face à leur grand nombre d'utilisateurs potentiels, les services Internet peuvent être implantés sur un réseau de machines. Mes activités de recherche m'ont amenée à m'intéresser à une large gamme d'applications distribuées.

1. En avril 1994, le projet LSP s'est scindé en deux projets, Solidor et Lande.

Par conséquent, mes travaux concernent différentes architectures : architectures multiprocesseurs, réseaux de stations de travail. L'évolution de la technologie a aussi influencé la gamme d'architectures à laquelle je me suis intéressée : multiprocesseurs organisés autour d'un bus, multiprocesseurs extensibles à mémoire partagée, réseau local Ethernet, réseau local à haut débit (ATM).

Revenons à ma problématique. *Comment concilier efficacité et tolérance aux fautes dans des systèmes construits à partir de composants standard tout en assurant la transparence de la tolérance aux fautes pour les applications ?* J'ai initialement abordé cette question sous l'angle des communications. En effet, mes premiers travaux de recherche menés dans le cadre du projet Gothic ont consisté à concevoir un système de communication fiable pour mettre en œuvre efficacement l'appel de multiprocédure à distance. Ces travaux sont relatés dans le chapitre 2.

J'ai ensuite élargi mon domaine de recherche à la conception d'architectures tolérantes aux fautes tout en conservant un fort intérêt pour les problèmes liés aux communications. Ainsi, dans le contexte du projet européen FASST dont l'objectif était de concevoir une architecture à mémoire partagée tolérante aux fautes, je me suis intéressée à la conception et mise en œuvre d'un protocole de recouvrement arrière fondé sur une gestion précise des communications de manière à limiter la dégradation de performance due à la sauvegarde de points de reprise dans un système de processus communicants. Ces travaux sont présentés dans le chapitre 3.

J'ai poursuivi mes travaux dans le cadre de l'activité de recherche Aleth, que j'ai initialisée à la fin du projet FASST en collaboration avec Michel Banâtre. Mon objectif était de trouver une solution au problème de la disponibilité des architectures extensibles à mémoire partagée. Dans ce contexte, j'ai cherché à exploiter la réplication de données engendrée dans ce type d'architecture par les communications pour mettre en œuvre efficacement un protocole de recouvrement arrière. Ces travaux sont abordés dans le chapitre 4.

Dans le chapitre 5, je dresse un bilan de mes travaux et indique quelques perspectives de recherche.

L'expérimentation est essentielle dans mon domaine de recherche. Elle permet de vérifier l'applicabilité des idées et leur réalisme. En outre, elle permet bien souvent d'exhiber de nouveaux problèmes de recherche. Enfin, le développement de prototypes est un moyen de valoriser les résultats de recherche obtenus. Tout au long de mon parcours de recherche, j'ai accordé une grande importance à l'expérimentation. Tous les travaux présentés dans ce document ont été expérimentés, soit par simulation pour ceux relevant du domaine de l'architecture, soit par la réalisation de prototypes pour ceux relevant du domaine des systèmes distribués.

tel-00434053, version 1 - 20 Nov 2009

tel-00434053, version 1 - 20 Nov 2009

tel-00434053, version 1 - 20 Nov 2009

Chapitre 2

Communications fiables dans les systèmes distribués

2.1 Le problème des communications dans les systèmes distribués

Un système distribué est constitué d'un ensemble de postes de travail et de périphériques reliés entre eux par un système de communication. L'histoire des systèmes distribués est étroitement liée à l'évolution des outils de communication. Ainsi, l'apparition vers le milieu des années 70 du réseau Ethernet, réseau local à haut débit (sic) utilisant un réseau à diffusion, marque une étape importante : premiers postes de travail individuels dotés de facilités graphiques, organisation client/serveur pour l'accès aux ressources partagées. Les premiers systèmes répartis utilisant les réseaux locaux, qui apparaissent à la fin des années 70, sont réalisés en interconnectant plusieurs systèmes homogènes (notamment des systèmes Unix). La plupart de ces systèmes étendent simplement le système de fichiers pour offrir un accès transparent aux fichiers locaux ou distants ; quelques uns permettent la création et l'exécution de processus à distance. Ce n'est qu'au début des années 80 qu'apparaissent les systèmes distribués intégrés conçus au départ comme répartis. Ces systèmes donnent alors l'impression aux usagers qu'ils disposent d'un système d'exploitation aussi souple que les systèmes à temps partagé mais avec de nouvelles possibilités.

L'essor des systèmes distribués a entraîné le développement de nouvelles applications conçues pour s'y exécuter : les applications distribuées. Pour le programmeur d'applications distribuées se pose alors un problème nouveau : celui de la communication. La gestion des communications est une charge pour le programmeur du fait des imperfections de la transmission des messages (perte par exemple) et des défaillances possibles des sites du réseau de communication. Le développement de protocoles de communication fiable permet alors de décharger le programmeur de la lourde tâche du traitement des erreurs de transmission et des défaillances susceptibles de survenir pendant une communication.

C'est à ce problème que je me suis intéressée au cours de ma thèse. Mes travaux se sont déroulés dans le cadre du projet Gothic [11] qui visait à construire un système distribué in-

tégré tolérant les fautes sur une architecture matérielle constituée d'un réseau local de type Ethernet reliant un ensemble de machines multiprocesseurs. Polygoth, le langage de programmation de Gothic offre le concept de multiprocédure pour la construction d'applications distribuées. Mes travaux ont porté sur la conception de protocoles de communication fiable pour la mise en œuvre d'un protocole d'appel de multiprocédure à distance.

Dans la suite de ce chapitre, nous présentons les modèles de programmation d'applications distribuées et les travaux s'y rapportant concernant la gestion des communications. Nous présentons ensuite notre contribution au domaine et terminons par un bilan de ces travaux.

2.2 Modèles de communication pour les applications distribuées

Une application distribuée peut être modélisée par un ensemble de processus communicants. Plusieurs types de communications peuvent intervenir entre les processus. Lorsqu'une interaction est limitée à deux processus, on parle de communication bipartie. Les communications entre un processus et un groupe de processus sont qualifiées de communications multiparties.

2.2.1 Communications biparties

Les premières applications distribuées étaient fondées sur la communication par message. L'inconvénient de ce modèle est que la gestion des communications inter-sites ainsi que le traitement des défaillances sont à la charge du programmeur.

Au début des années 80, l'appel de procédure à distance prévaut sur la communication par message. L'appel de procédure à distance est une mise en œuvre de l'appel de procédure dans un environnement distribué qui respecte le schéma de contrôle de l'appel de procédure classique. Lors d'un appel de procédure local, la procédure appelante et la procédure appelée s'exécutent sur le même processeur tandis que dans le cas d'un appel de procédure à distance la procédure appelante et la procédure appelée s'exécutent sur deux processeurs distincts sur les sites *client* et *serveur*. De façon schématique, un appel de procédure à distance se déroule de la manière suivante (fig. 2.1). Le site client construit un message de type *APPEL* contenant l'identification de la procédure appelée à distance et les paramètres d'entrée, puis le transmet au site serveur. Le processus appelant est suspendu pendant l'exécution de la procédure. Le site serveur, à la réception du message *APPEL* crée un processus pour exécuter la procédure. A la fin de l'exécution de cette dernière, un message de type *RETOUR* contenant les résultats de l'exécution est transmis au site client. A la réception du message *RETOUR* sur le site client, l'appelant est débloqué.

L'intérêt du mécanisme d'appel de procédure à distance est de masquer au programmeur les détails de la communication inter-site et de le décharger en grande partie du traitement des défaillances. Le problème de la gestion des défaillances est un des problèmes les plus épineux dans la conception d'un mécanisme d'appel de procédure à distance. En effet, un appel de procédure à distance se déroule sur deux machines indépendantes. La défaillance de l'une des machines n'empêche pas l'exécution de se poursuivre sur l'autre.

tel-00434053, version 1 - 20 Nov 2009

tel-00434053, version 1 - 20 Nov 2009

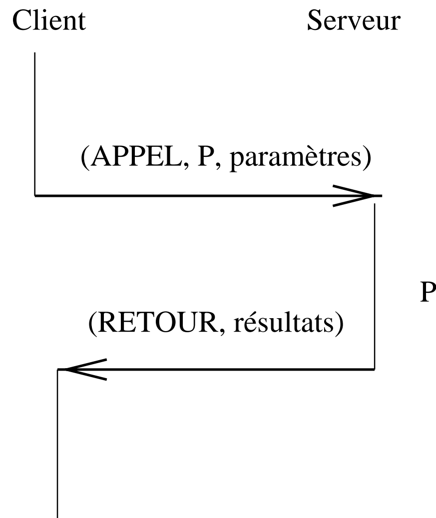


FIG. 2.1 – *Appel de procédure à distance*

Ainsi, l'exécution du processus serveur continue en dépit de la défaillance du site client. Même en l'absence de défaillance des sites client ou serveur, une et une seule exécution d'un appel de procédure à distance n'est pas garantie puisque le réseau de communication n'est pas fiable. En outre, un site ne peut pas toujours déterminer de façon certaine le type des défaillances qui surviennent. Ainsi comment un site S_i peut-il différencier la défaillance d'un site distant S_j et la défaillance du lien de communication entre S_i et S_j ? Dans les deux cas, toute communication entre S_i et S_j est impossible. Enfin, la durée d'une défaillance pouvant être longue, il apparaît qu'il est impossible pour le système de masquer complètement les défaillances matérielles aux applications. En effet pratiquement, il n'est pas envisageable d'attendre indéfiniment la terminaison d'un appel de procédure à distance perturbé par une défaillance de longue durée.

Comme nous venons de le voir, l'occurrence de défaillances donne naissance à de nombreuses configurations exceptionnelles pour lesquelles il est nécessaire de définir l'issue d'un appel de procédure à distance. Il est clair que les défaillances ne peuvent pas être totalement masquées aux utilisateurs mais il est essentiel qu'un mécanisme d'appel de procédure à distance garantisse aux programmeurs un comportement déterministe pour l'exécution des procédures à distance. Ces garanties définissent ce qui est couramment appelé par abus de langage la sémantique d'appel de procédure à distance. La sémantique d'appel caractérise combien de fois la procédure appelée à distance est exécutée suite à un appel en l'absence ou en présence de défaillances.

exécution incertaine (*peut-être une fois*) En l'absence de défaillance, la procédure est exécutée exactement une fois. En présence de défaillance, la procédure est exécutée zéro, une fois ou partiellement sans que le client puisse connaître le nombre d'exécutions

exécution au moins une fois En l'absence de défaillance, la procédure appelée est exécutée exactement une fois. En présence de défaillance, la procédure appelée est exécu-

tel-00434053, version 1 - 20 Nov 2009

tée une ou plusieurs fois. En cas de défaillance du site client ou du site serveur, la réception par le client du message de type *RETOUR* lui garantit que la procédure a été exécutée au moins une fois. En l'absence du message *RETOUR*, le client ignore combien de fois la procédure a été exécutée : zéro, une ou plusieurs fois avec la possibilité d'exécutions partielles de la procédure en cas de défaillance du site serveur. Ce type de mécanisme peut convenir à l'exécution d'une classe particulière de procédures : les procédures idempotentes. Une procédure est idempotente si quel que soit le nombre d'exécutions, elle délivre le même résultat et produit les mêmes effets.

Lampson propose dans [75] un protocole pour garantir que le résultat délivré à l'appelant est celui de la dernière exécution de la procédure. La sémantique d'appel assurée par ce protocole est appelée *last of many* dans la littérature. Cet algorithme garantit que, en l'absence de défaillance, la procédure est exécutée au moins une fois et que le résultat délivré à la procédure appelante est celui calculé lors de la dernière exécution de la procédure.

exécution au plus une fois Liskov [79] dans le cadre du système Argus et Svobodova [122] pour les calculs distribués résilients proposent des protocoles pour l'exécution de procédures à distance qui garantissent au plus une exécution de la procédure appelée en présence de défaillance. L'appel de procédure à distance est mis en œuvre par une action atomique. Une défaillance entraîne l'annulation de l'action atomique. L'appel de procédure à distance est dit atomique : soit la procédure est exécutée exactement une fois soit elle n'a aucun effet de bord (dans le cas où l'action atomique correspondante est annulée). En outre l'appelant connaît le nombre d'exécutions (zéro ou une) ayant eu lieu.

Dans le protocole décrit dans [122], un point de reprise est sauvegardé lors de chaque appel de procédure à distance. Les processus clients peuvent ainsi traiter les réponses leur parvenant après la défaillance de leur site et concernant les appels entrepris avant la défaillance. La sémantique *au plus une fois* est coûteuse à mettre en œuvre. Cependant, elle facilite le traitement des défaillances pour les applications puisqu'il est garanti qu'une procédure ne produit pas d'effets de bord en cas de défaillance.

La principale difficulté pour la mise en œuvre d'un mécanisme d'appel de procédure à distance en univers non fiable est le traitement des exécutions orphelines. Afin d'éviter l'écueil de l'anthropomorphisme, nous introduisons le néologisme *exéline* pour désigner un orphelin ou une exécution orpheline. Diverses circonstances conduisent à la création d'exélines. Les retransmissions du message *APPEL* peuvent provoquer plusieurs exécutions de la procédure appelée si aucune précaution particulière n'est prise sur le site serveur. Toutes les exécutions de la procédure sauf une sont des exélines. De même, le déclenchement du délai armé pour prévenir une attente infinie du message *RETOUR* par le client peut engendrer des exélines. En effet, le fait que le message *RETOUR* ne parvienne pas au site client n'empêche pas pour autant l'exécution de la procédure appelée sur le site serveur. Cette exécution devient une exéline à partir du moment où le client cesse d'attendre le message *RETOUR*. Enfin, en cas de défaillance du site client les exécutions de procédure à distance qu'il a initialisées avant de tomber en défaillance deviennent des exélines. Des mécanismes de numérotation des messages permettent facilement de résoudre les problèmes de perte ou de déséquencement des messages par le

Les exélines sont indésirables pour plusieurs motifs. En premier lieu, elles peuvent

interférer avec les exécutions valides en cours. En outre, les exélines acquièrent des ressources du système et par conséquent augmentent les conflits d'accès aux ressources. Elles consomment du temps d'unité centrale et contribuent ainsi à une baisse des performances. De nombreuses études ont été consacrées au problème des exélines [92, 75, 113, 114]. Il existe essentiellement trois approches pour ce problème : la prévention, l'élimination et la récupération des exélines.

Lorsque la création d'exélines est engendrée par la réémission du message *APPEL*, elle peut être évitée en numérotant les messages *APPEL* et en conservant quelques informations sur le site serveur. En revanche, la défaillance du site client entraîne inévitablement la création d'exélines. Dans ce cas, un protocole d'élimination ou de récupération des orphelins doit être mis en œuvre pour traiter les exélines. La méthode la plus simple et la moins coûteuse est l'élimination des exélines à l'initiative du site client lors du recouvrement après défaillance. Cependant, ce protocole présente l'inconvénient de ne pas détruire toutes les exélines si une défaillance survient pendant la phase de recouvrement.

2.2.2 Communications multiparties

Avec l'appel de procédure à distance, la communication est de type client/serveur et est donc restreinte à deux processus. Dans les systèmes distribués, les applications mettent souvent en jeu un ensemble de processus coopérants. Les communications n'ont plus seulement lieu entre deux processus mais aussi entre un processus et un groupe de processus. Citons quelques exemples.

- Un service, pour des raisons de sûreté de fonctionnement, peut être mis en œuvre par un groupe de processus (serveurs) s'exécutant sur des sites distincts. Un client du service adresse ses requêtes au groupe des serveurs.
- Une base de données peut être répliquée sur plusieurs sites de façon à en augmenter la disponibilité. Chaque copie est gérée par un gestionnaire. Un gestionnaire donné doit communiquer au groupe des gestionnaires les modifications qu'il apporte à sa copie locale de façon à les répercuter sur les autres sites.
- La localisation d'une ressource dans un système distribué peut entraîner l'envoi d'un message à tous les sites du système [32].

Pour répondre à ce besoin de communication entre un processus et un groupe de processus, plusieurs modèles de communication ont été développés. Nous détaillons dans la suite de ce paragraphe les modèles fondés sur la diffusion, les modèles reposant sur les appels multiples de procédure et le modèle multiprocédural mise en œuvre dans le système Gothic.

Diffusion d'un message à un groupe

Plusieurs types de diffusion existent. Déjà, les réseaux de communication de la famille IEEE802 (réseau Ethernet, Token Ring, FDDI) permettent la transmission d'un message vers un groupe de machines [125]. Cependant, ces primitives ne sont pas fiables au sens où elles ne garantissent pas que toutes les machines du groupe adressé reçoivent le message.

De nombreux travaux ont été conduits ces dernières années dans le domaine de la conception de protocoles de diffusion fiable tant dans les systèmes distribués asynchrones

tel-00434053, version 1 - 20 Nov 2009

[112, 30, 39, 17] que dans les systèmes distribués synchrones [6, 7, 39, 37, 98]. Les divers protocoles proposés diffèrent sur les hypothèses de défaillances ou sur la topologie du réseau qu'ils supposent, l'ordonnancement des messages diffusés, le caractère dynamique ou non des groupes de processus destinataires d'un message.

Le terme diffusion fiable est très fréquemment employé dans la littérature mais ne recouvre pas toujours exactement la même chose. Une première distinction s'impose ; elle concerne la définition de l'ensemble des destinataires. Dans le cas d'une diffusion générale (*broadcast*), il peut être composé de tous les processus du système ou tous les sites du réseau (l'ensemble des destinataires est dans ce cas implicite). Dans le cas d'une diffusion (*multicast*), le message diffusé s'adresse à un groupe (ou une liste) déterminé de processus.

Tous les protocoles de diffusion fiable sont des protocoles de diffusion atomique c'est-à-dire qu'ils possèdent au moins la propriété d'atomicité définie comme suit :

> *Tous les destinataires corrects reçoivent le message diffusé ou aucun d'entre eux ne le reçoit (tout ou rien).*

Plusieurs protocoles garantissent en plus de la propriété d'atomicité, un ordre sur l'ensemble des messages diffusés. Garcia-Molina caractérise dans [48] trois propriétés d'ordre :

1. ordonnancement avec source unique (*single source ordering*)
   Si un site diffuse deux messages m_1 et m_2 vers un groupe destinataire alors tous les processus destinataires reçoivent m_1 et m_2 dans le même ordre.
2. ordonnancement avec sources multiples (*multiple source ordering*)
   Si m_1 et m_2 sont destinés à un même groupe de processus, les processus destinataires recevront m_1 et m_2 dans un ordre identique même si m_1 et m_2 proviennent de deux sources différentes.
3. ordonnancement avec groupes destinataires multiples (*multiple group ordering*)
   Si deux messages m_1 et m_2 sont délivrés à deux processus P et P', ils sont délivrés dans un ordre identique même si m_1 et m_2 proviennent de sources différentes et sont destinés à des groupes différents (mais non disjoints).

D'autres protocoles garantissent la propriété d'ordre causal introduite par Birman [17] et mise en œuvre dans le protocole CBCAST d'Isis. Cette propriété est fondée sur la relation d'ordre partiel *se produire avant* [72]. Si deux émissions de messages vers une même destination sont liées par la relation *se produire avant* alors les réceptions de messages correspondantes sont liées par la même relation.

Les appels de procédure multiples

Une autre approche à la communication multipartie est d'étendre le mécanisme d'appel de procédure à distance afin de permettre l'exécution d'appels de procédure multiples en parallèle. Certains de ces travaux tels que l'appel de procédure à distance parallèle [84] et le multiRPC [108] visent à augmenter le parallélisme en permettant l'exécution d'une même procédure sur plusieurs sites en parallèle. Par exemple, il est intéressant de pouvoir exécuter en parallèle une procédure de recherche d'un fichier dans un répertoire réparti.

Un appel de multiprocédure à distance parallèle (de même qu'un multiRPC) donne lieu à l'exécution d'une procédure sur N sites en parallèle. La liste des sites sur lesquels doit s'exécuter la procédure est passée en paramètre de la procédure appelée. L'appelant est

tel-00434053, version 1 - 20 Nov 2009

tel-00434053, version 1 - 20 Nov 2009

bloqué pendant l'exécution des N procédures. Les résultats sont traités par l'appelant au fur et à mesure de leur arrivée et il peut décider de ne pas attendre tous les résultats. Le cas particulier d'une procédure sans paramètres de résultat s'apparente à une diffusion. La défaillance du site client entraîne la défaillance de l'appel de procédure à distance parallèle. Si l'appel n'aboutit pas sur un des processus serveurs, ce serveur est ignoré dans la suite de l'appel de procédure à distance qui est poursuivi avec les autres serveurs. Ce type de comportement s'apparente à la sémantique *peut-être une fois* puisque la procédure est susceptible de pas être exécutée sur certains sites.

D'autres travaux tels que les appels de procédure répliqués mis en œuvre dans Circus [33] ont été motivés par le besoin de fiabiliser les applications distribuées en exécutant une même procédure simultanément sur plusieurs sites. Dans Circus, la tolérance aux fautes est fondée sur la redondance active. Une application distribuée est constituée de modules interagissant répliqués sur plusieurs sites. L'ensemble des exemplaires d'un module est appelé une troupe. Un appel de procédure répliqué est exécuté une fois et une seule par tous les membres de la troupe. Tant qu'il existe au moins un membre dans une troupe l'appel répliqué peut être exécuté. Les membres d'une troupe sont indépendants et ne communiquent pas entre eux. Chaque membre d'une troupe agit exactement comme si les autres exemplaires n'existaient pas. Le degré de réplication peut donc varier dynamiquement. Le protocole d'appel de procédure répliqué est mis en œuvre au dessus d'un mécanisme d'appel de procédure à distance similaire à [18]. Dans le cas particulier de troupes réduites à un seul élément, un appel de procédure répliqué est réduit à un simple appel de procédure à distance.

Le concept de multiprocédure

Le concept de multiprocédure [10], introduit dans le langage Polygoth, est une extension de la procédure. Il permet de structurer les applications parallèles. La procédure est une abstraction du bloc ; la multiprocédure est une abstraction de la proposition parallèle.

Il existe deux types d'appel d'une multiprocédure : l'appel simple et l'appel coordonné. Nous décrivons chaque type d'appel sur l'exemple de la multiprocédure définie sur la figure 2.2.

La multiprocédure *mf* est constituée de trois composants. Le composant (1) prend un paramètre en entrée (p_1) et produit le résultat res_1 ; le composant(2) prend en entrée le paramètre p_2 et rend res_2 en résultat ; le composant (3) prend p_1 et p_2 comme paramètres d'entrée et produit res_3. Un appel simple est l'invocation de la multiprocédure par un seul appelant : une procédure ou un composant d'une multiprocédure. Un appel simple est dit de type 1-P car il fait intervenir un appelant et P appelés (les P composants de la multiprocédure appelée). Connaissant la définition de la multiprocédure *mf* et supposant la déclaration des variables m, n, a, b, c, un appel simple (1-3) à *mf* s'écrit :

$$(a, b, c) := mf(m, n).$$

Dans cet appel, m et n sont les paramètres effectifs en entrée tandis que a, b et c sont les variables destinées à recevoir les résultats élaborés pendant l'exécution de *mf*.

L'appel simple de la multiprocédure *mf* se déroule de la manière suivante :

– distribution des paramètres d'entrée aux divers composants,

– exécution parallèle des composants,

```
multiproc mf (p1 : t1, p2 : t2) : (res1 : tr1, res2 : tr2, res3 : tr3);
cobegin
begin (p1) : res1
   /* corps du composant (1) */
end //
        begin (p2) : res2
   /* corps du composant (2) */
end //
begin (p1, p2) : res3
   /* corps du composant (3) */
end
coend mf ;
```

FIG. 2.2 – *Définition de la multiprocédure mf*

– synchronisation des composants pour la construction et le transfert du résultat à l'appelant,
– reprise de l'exécution de l'appelant.

Une multiprocédure peut également être appelée par une autre multiprocédure. On parle alors d'appel coordonné ou d'appel de type N-P où N est le nombre de composants de la multiprocédure appelante et P le nombre de composants de la multiprocédure appelée. L'appel coordonné permet d'exprimer l'imbrication de propositions parallèles sous la forme la plus générale. Nous définissons sur la figure 2.3 un programme réalisant un appel coordonné 2-3 à la multiprocédure *mf*. L'appel coordonné se déroule de la manière suivante.

```
cobegin
   /* Déclarations */
   a: tr1 ; b: tr2 ; c : tr3 ;
   begin
var m : t1 ;
(a, b) := mf (p1 = m).(res1, res2)
   end \\
   begin
var n : t2 ;
        (c) := mf (p2 = n).(res3)
   end
coend
```

FIG. 2.3 – *Un appel coordonné 2-3 de multiprocédure*

Les composants (1) et (2) du programme appelant se synchronisent pour le transfert des paramètres. Ces derniers sont distribués aux trois composants de la multiprocédure appelée *mf*. Ces trois composants s'exécutent en parallèle puis se synchronisent pour la construction et le transfert du résultat. Finalement, le résultat est distribué aux deux composants du

tel-00434053, version 1 - 20 Nov 2009

tel-00434053, version 1 - 20 Nov 2009

programme appelant.

Discussion

Le principal intérêt de l'appel de procédure à distance parallèle et du multiRPC est de remplacer *N* appels de procédure à distance consécutifs par un seul appel réalisant les *N* exécutions en parallèle. Cependant, le traitement des défaillances n'est pas satisfaisant. Le recouvrement d'un serveur défaillant n'est pas prévu. Aucun traitement n'est prévu pour éliminer les exécutions orphelines engendrées par la défaillance du site client. Le principal intérêt des appels de procédure répliqués est d'offrir la transparence de la tolérance aux fautes aux applications. Le concept de multiprocédure permet d'exprimer le parallélisme dans sa forme la plus générale. On peut donc voir les appels multiples comme le cas particulier d'un appel 1-N de multiprocédure dont tous les composants sont identiques. Un appel de procédure est exécuté exactement une fois en l'absence de défaillance et donne lieu à exactement 0 ou une exécution en cas de défaillance. Un tel comportement n'est pas garanti pour les appels multiples. Les règles de synchronisation sont strictes pour les appels de multiprocédure pour lesquels le contrôle est rendu aux composants appelants qu'une fois l'exécution de tous les composants appelés terminés. Dans le cas des appels multiples, l'appelant peut reprendre son exécution dès la terminaison d'un des appels parallèles.

Comparé aux protocoles d'appel de procédure multiple, les principaux intérêts d'un protocole d'appel de multiprocédure à distance sont d'une part d'offrir une forme de communication plus générale (communications inter-groupales) et d'autre part de décharger le programmeur de la gestion des défaillances.

2.3 Définition et mise en œuvre de l'appel de multiprocédure à distance

Mes travaux de thèse [87] ont consisté à définir l'appel de multiprocédure à distance et à en proposer une mise en œuvre efficace dans le système Gothic.

2.3.1 Le système Gothic et sa mémoire stable rapide

L'architecture matérielle de Gothic est constituée d'un ensemble de machines multiprocesseurs interconnectées par un réseau local de type Ethernet (voir figure 2.4).

Chaque processeur peut accéder sur son bus local une mémoire locale et une mémoire stable rapide. Les processeurs d'une même machine sont interconnectés par le bus global.

L'originalité de l'architecture réside dans l'association d'une mémoire stable rapide, constituée de deux bancs de mémoire vive associés à un contrôleur, à chaque processeur. La mémoire stable rapide est normalement accédée par le processeur auquel elle est associée. En cas de défaillance de ce dernier, elle émet un signal de secours sur le bus global. Ce signal est acquitté par l'un des autres processeurs de la machine qui peut alors accéder le contenu de la mémoire stable rapide (en particulier, pour redémarrer les processus du processeur défaillant à partir de leur point de reprise). Son alimentation distincte de l'alimentation du reste de la machine est secourue par une batterie pour masquer les coupures

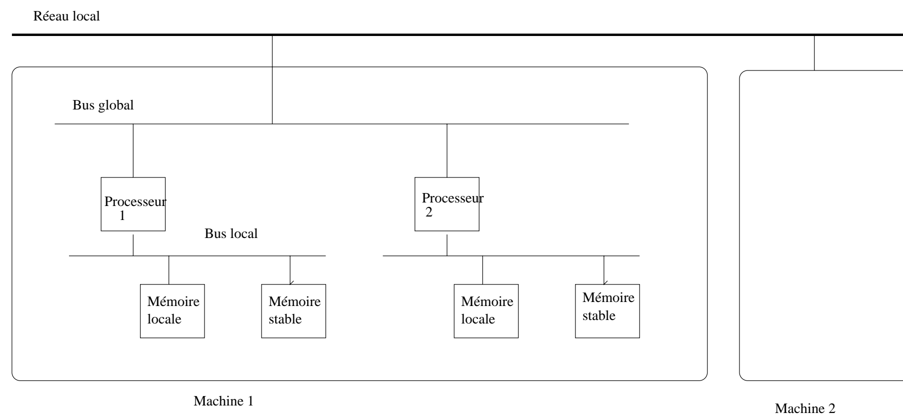


FIG. 2.4 – *Architecture matérielle de Gothic*

d'alimentation.

Le système d'exploitation de Gothic est un noyau modulaire conforme à la norme Sceptre [20] autour duquel vienne se greffer des agences offrant les services système de base. Une agence est dédiée aux primitives d'accès à la mémoire stable rapide permettant la lecture et l'écriture atomique d'une donnée ou d'un groupe de données ainsi que la sauvegarde ou la restauration d'un point de reprise de processus. Le système de communication est structuré en trois couches hiérarchiques implantant respectivement le protocole RMPC (appel de multiprocédure à distance), le protocole RBC (diffusion atomique ordonnée), le protocole RMC (communication fiable par message). Ce dernier est mis en œuvre au-dessus des protocoles de communication standard du système Gothic (protocole IP entre deux machines). Le système de communication fiable a été implanté comme une nouvelle agence de Gothic.

2.3.2 Définition du protocole RMPC

Nous devons définir une sémantique pour le protocole d'appel de multiprocédure à distance comme cela est fait pour les protocoles d'appel de procédure à distance. Notre étude des protocoles d'appel de procédure à distance montre que pour la construction d'applications distribuées fiables la sémantique *au plus une fois*, choisie notamment dans le système Argus [79], s'avère la mieux adaptée. En effet, les traitements d'exception sont considérablement simplifiés puisqu'en cas de défaillance la procédure appelée n'est exécutée que 0 ou 1 fois et le nombre d'exécutions réalisées est connu. Le principal inconvénient des protocoles d'appel de procédure à distance qui respectent la sémantique *au plus une fois* est leur coût élevé du à la mise en œuvre de transactions. Nous pensons que cet inconvénient peut être diminué dans le système Gothic puisque nous disposons de mémoires stables rapides. C'est pourquoi, nous avons choisi de mettre en œuvre dans le protocole RMPC la sémantique *au plus une fois*. Le protocole d'appel de multiprocédure à distance respecte la

sémantique *au plus une fois* si et seulement si les propriétés suivantes sont vérifiées :

1. En l'absence de défaillance, la multiprocédure appelée est exécutée exactement une fois c'est-à-dire que tous ses composants s'exécutent exactement une fois.
2. En présence de défaillance, la multiprocédure appelée est exécutée exactement 0 ou 1 fois et les composants de la multiprocédure appelante connaissent le nombre d'exécutions ayant eu lieu. 0 exécution de la multiprocédure appelée signifie qu'aucun de ses composants ne s'est exécuté.

Il est intéressant de noter que le cas particulier d'un appel 1-1 de multiprocédure à distance est équivalent à un appel de procédure à distance. Par ailleurs, le cas particulier d'un appel 1-N d'une multiprocédure dont tous les composants sont identiques s'apparente à un appel de procédure parallèle [84] ou à un MultiRPC [108] mais à la différence près de la sémantique. La propriété 2 énoncée ci-dessus n'est garantie ni pour un appel de procédure parallèle ni pour un MultiRPC. Il ne faut pas non plus négliger le fait qu'un mode d'imbrication général est défini pour les multiprocédures alors que ce n'est pas le cas pour l'appel de procédure parallèle et le MultiRPC. Le protocole RMPC est tolérant aux fautes dans le sens où il assure la résilience des applications. La propriété de résilience définie dans [121] peut s'exprimer ainsi :

> *Une application est résiliente si elle peut terminer son exécution en dépit des défaillances qui peuvent survenir dans les divers composants du système.*

Une méthode pour rendre une application résiliente est de sauvegarder périodiquement l'état courant de l'application dans un point de reprise. L'application peut ainsi en cas de défaillance reprendre son exécution à partir de son dernier point de reprise soit sur un autre site soit sur le site défaillant après le recouvrement. Les calculs effectués avant le dernier point de reprise n'ont pas besoin d'être réexécutés. Dans le système Gothic, nous avons suivi cette approche. Le protocole RMPC est exécuté au-dessus du noyau de processus stables. Un processus stable est un processus dont l'état est sauvegardé en mémoire stable. Le protocole RMPC sauvegarde des points de reprise des processus communicants de façon cohérente et garantit la résilience des applications. La sauvegarde de points de reprise et l'hypothèse de défaillances de durée finie permettent de traiter le problème des exélines de façon simple. En effet, puisque tout processus reprend son exécution au bout d'un temps fini à partir d'un point de reprise qui a été sauvegardé de façon consistante un processus n'est orphelin que pendant la durée de la défaillance d'un des sites clients (dans le cas des multiprocédures, il peut y avoir plusieurs sites clients puisque les multiprocédures peuvent être imbriquées). Toute exéline est récupérée lors du recouvrement. Aucun algorithme d'élimination n'est nécessaire. La récupération des exélines est automatique lors du recouvrement après défaillance.

L'appel simple de multiprocédure

Un appel simple de multiprocédure à distance se déroule en quatre étapes :

1. Transmission du contrôle et des paramètres aux composants de la multiprocédure appelée par la diffusion atomique ordonnée d'un message de type *APPEL_MP* puis attente du résultat de l'exécution de la multiprocédure.
2. Exécution des composants de la multiprocédure appelée.

tel-00434053, version 1 - 20 Nov 2009

3. Synchronisation des composants de la multiprocédure appelée pour la construction du résultat. Le coordinateur collecte les résultats partiels calculés par les autres composants de la multiprocédure appelée. Ceux-ci lui envoient un message fiable de type *RESUL_PARTIEL* contenant la partie du résultat qu'ils ont calculée.
4. Retour du résultat à l'appelant. Le coordinateur lui envoie un message fiable de type *RESUL_MP*. A la réception du message *RESUL_MP*, l'appelant reprend son exécution.

L'appel coordonné de multiprocédure

L'appel coordonné, préalablement à la diffusion fiable du message *APPEL_MP*, comporte une phase de synchronisation des composants de la multiprocédure appelante. Le coordinateur attend un message de type *APPEL_PARTIEL* de chacun des autres composants appelants. Un tel message contient les paramètres d'entrée fournis par le composant émetteur.

Une fois collectés tous les messages de type *APPEL_PARTIEL*, le coordinateur de la multiprocédure appelante envoie (par une diffusion atomique ordonnée) à tous les sites sur lesquels vont s'exécuter les composants de la multiprocédure appelée un message de type *APPEL_MP* comme dans le cas de l'appel simple.

Lorsque les composants appelés ont terminé leur exécution, ils se synchronisent comme dans le cas d'un appel simple pour la construction du résultat. Le coordinateur de la multiprocédure appelée diffuse alors atomiquement le message de type *RESUL_MP* à tous les composants de la multiprocédure appelante.

Primitives de communication

Trois types de primitives de communication sont utilisées dans le protocole RMPC :

– l'envoi d'un message fiable à un site,
– la diffusion atomique d'un message à un ensemble de sites,
– la diffusion atomique ordonnée d'un message à un ensemble de sites.

Ces primitives de communication fiable ont été mises en œuvre dans le cadre du système Gothic. Le protocole RMC implante les deux premières tandis que la dernière est à la charge du protocole RBC, implanté au dessus du protocole RMC.

2.3.3 Le protocole RMC

Le protocole RMC est chargé de la transmission fiable de messages en dépit des défaillances des nœuds (considérés *fail-silent*) et du système de communication (perte, duplication et déséquencement des messages).

Ports stables Les messages fiables sont échangés entre des ports stables. Un port stable est une file de messages stockée en mémoire stable. Un port stable est créé par un processus qui en est le propriétaire. Le propriétaire d'un port est le seul processus autorisé à en extraire des messages. Tout processus qui connaît l'identificateur d'un port stable peut émettre un message à destination de ce port.

tel-00434053, version 1 - 20 Nov 2009

tel-00434053, version 1 - 20 Nov 2009

Principes du protocole Le protocole RMC garantit à l'expéditeur d'un message *m* que *m* atteindra son port stable de destination *p* dès que possible sachant que des défaillances peuvent retarder l'introduction de *m* dans *p*.

Le protocole RMC associe un numéro de séquence *s* à chaque message *m*. *s* est généré par incrémentation, à chaque envoi de message, d'un compteur situé en mémoire stable. Les numéros de séquence sont utilisés pour détecter la perte de message et détecter la duplication des messages due aux retransmissions. Un nœud *Si* détecte les messages dupliqués en provenance d'un nœud *Sj* grâce à un compteur stocké en mémoire stable qui mémorise le numéro de séquence du dernier message accepté en provenance de *Sj*. *Si* accepte les messages en provenance des autres nœuds dans l'ordre de leur numéro de séquence.

L'exécution des primitives *envoyer-msg-fiable* et *recevoir-msg-fiable* offertes par le service de communication fiable est atomique. La primitive *envoyer-msg-fiable* affecte un numéro de séquence au message émis, stocke ce dernier en mémoire stable et établit un point de reprise du processus appelant. La primitive *envoyer-msg-fiable* n'émet pas le message sur le réseau. Un serveur sur chaque nœud envoie les messages stockés en mémoire stable. Un message reste en mémoire stable tant qu'il n'a pas été acquitté par son destinataire.

La primitive *recevoir-msg-fiable* ôte le premier message présent dans le port stable et sauvegarde un point de reprise du processus appelant. Le stockage d'un point de reprise lors de chaque opération de communication assure la cohérence de l'état global du système de processus communicants.

Chaque nœud conserve une vision de l'état des autres nœuds : opérationnel ou défaillant. L'état d'un nœud passe à défaillant après un nombre maximum de retransmissions d'un message donné sans réception d'acquittement.

Recouvrement après défaillance Quand un nœud *Sk* redémarre suite à une défaillance, il doit être réinséré correctement dans le système. Les tâches suivantes sont exécutées :

1. *Sk* doit signaler aux autres nœuds qu'il est opérationnel en envoyant un message de recouvrement contenant pour chaque site le numéro de séquence du dernier message accepté par *Sk* en provenance de ce site.

   A la réception de ce message, un nœud *Si* retransmet à *Sk* tous les messages qu'il a émis et dont le numéro de séquence est supérieur à la valeur reçue. *Si* élimine de sa mémoire stable tous les messages implicitement acquittés par le message de recouvrement.
2. Chaque processus stable de *Sk* est redémarré à partir de son point de reprise.

Du fait des imperfections du système de communication, un nœud peut à tort considérer qu'un autre nœud est défaillant. Cette situation pose problème car un nœud cesse d'émettre des messages à destination des sites considérés défaillants. Pour remédier à cette situation, chaque nœud envoie périodiquement un message "*hello*" aux autres nœuds qui peuvent ainsi corriger leur vision erronée le cas échéant.

2.3.4 Le protocole RBC

Le protocole RBC est construit au-dessus du protocole RMC. Il offre une primitive de diffusion sélective permettant d'envoyer un message à un groupe de destinataires. Il garantit

tel-00434053, version 1 - 20 Nov 2009

les deux propriétés suivantes :

atomicité *m* sera délivré à tous ses destinataires au bout d'un temps fini ou à aucun d'entre eux. Cette définition de l'atomicité diffère de celle généralement admise. Les protocoles de diffusion atomique ne font en général pas la distinction entre les deux situations suivantes : (1) un site reçoit un message diffusé et est défaillant, (2) un site est défaillant avant de recevoir le message diffusé. Ces protocoles garantissent uniquement que les destinataires corrects reçoivent un message diffusé. Notre protocole assure que tous les destinataires recevront le message diffusé. Cette propriété d'atomicité ne peut bien entendu être assurée que si les processus défaillants redémarrent au bout d'un temps fini. Dans le système Gothic les processus redémarrent à partir de leur point de reprise sauvegardé en mémoire stable.

ordonnancement avec groupes destinataires multiples Si deux messages *m1* et *m2* sont délivrés à deux processus, ils sont délivrés dans le même ordre relatif (même s'ils proviennent de sources différentes et sont adressés à des groupes distincts non disjoints).

La première propriété garantit que le contrôle est transféré à tous les composants de la multiprocédure ou à aucun d'entre eux. La seconde assure que si deux multiprocédures ont des composants s'exécutant sur le même ensemble de nœuds, ils sont exécutés dans le même ordre sur tous ces nœuds.

La propriété d'atomicité peut être assurée grâce à la primitive de diffusion sélective offerte par le protocole RMC. Cette primitive est une simple extension de la primitive *envoyer-msg-fiable* qui consiste à ranger en mémoire stable le message émis dans les listes de messages en attente de chacun des sites destinataires ainsi qu'un point de reprise du processus appelant en une seule opération atomique. Ceci garantit que le message émis sera reçu par tous ses destinataires ou aucun d'entre eux.

Un protocole similaire au protocole ABCAST du système Isis [16] est utilisé pour garantir l'ordre total des messages.

2.3.5 Evaluation de performance

Le système de communication fiable a entièrement été mis en œuvre dans le système Gothic. Le prototype réalisé a permis d'évaluer quantitativement les performances des protocoles de communication fiable proposés [87]. Ces protocoles ayant été mis en œuvre au dessus du protocole IP, les performances brutes mesurées ont été très largement influencées par les médiocres performances obtenues par le protocole IP dans le système Gothic, aucun effort n'ayant été consenti pour l'optimisation du système de communication de base (IP et surtout les protocoles de communication sous-jacents qui provoquent de nombreuses commutations de contexte et utilisent intensivement le bus global). Compte tenu de cette situation, nous n'avons pas jugé intéressant de mesurer directement les performances du protocole RMPC. Nous avons préféré mener une étude approfondie du protocole RMC afin de mesurer l'impact de l'utilisation de la mémoire stable dans ce protocole. Une évaluation globale du protocole RBC a également été effectuée dont il ressort que ses performances sont directement dépendantes de celles du protocole RMC.

Le tableau de la figure 2.1 détaille la quantité d'information lues et écrites en mémoire stable dans les différentes phases du protocole RMC ainsi que le temps passé pour ces opérations.

	quantité de données lues en mémoire stable (mots de 32 bits)	quantité de données écrites en mémoire stable (mots de 32 bits)	temps de lecture en mémoire stable (ms)	temps d'écriture en mémoire stable (ms)
envoyer-msg-fiable (+ pt de reprise)	13/19 9	269/275 8305	0.023 / 0.030 0.016	0.673/0.687 20,769
recevoir-msg-fiable (+ pt de reprise)	269 9	13 8305	0.377 0.016	0.039 20,769
envoi d'un message fiable par le serveur	266	5	0.372	0.019
Réception d'un message fiable par le serveur	13/19	270/276	0.023 / 0.030	0.675/0.690
Traitement d'un acquittement par le serveur	13	12	0.023	0.036

TAB. 2.1 – *Evaluation de performance du protocole RMC*

Il apparaît clairement que le temps passé dans les primitives *envoyer-msg-fiable* et *recevoir-msg-fiable* est dominé par le temps de sauvegarde d'un point de reprise. Celui-ci est très élevé compte tenu du fait que le processus tout entier est sauvegardé lors de chaque point de reprise. Si seules les données modifiées du processus étaient sauvegardées au moment d'un point de reprise, les performances du protocole RMC seraient sensiblement améliorées. La mémoire stable de Gothic n'autorisant pas les accès concurrents, le temps passé dans les primitives *envoyer-msg-fiable* et *recevoir-msg-fiable* comprend en outre le temps nécessaire à l'obtention de la mémoire stable en exclusion mutuelle qui est variable en fonction de la charge du système.

2.3.6 Conclusion

Cette étude a permis de montrer que la technologie de mémoire stable rapide offre une solution simple et réaliste à de nombreux problèmes de communication [87]. Le système de communication fiable de Gothic dépasse en effet le cadre du protocole RMPC. J'ai en particulier défini un protocole de rendez-vous atomique s'appuyant sur le protocole RMC et qui a été mis en œuvre dans le système Gothic [85, 86, 9].

2.4 Bilan

L'objectif de la transparence de la tolérance aux fautes a été atteint dans le système Gothic pour les applications réparties programmées à l'aide du langage Polygoth. Cependant, la transparence a été offerte au prix de nombreux et volumineux points de reprise et du développement d'un composant matériel spécifique. En effet, au moment de la conception

tel-00434053, version 1 - 20 Nov 2009

tel-00434053, version 1 - 20 Nov 2009

du système Gothic, un point de reprise de processus impliquait systématiquement la sauvegarde complète de son état. En outre, la méthode que j'ai proposée pour garantir un état global cohérent du système de processus communicants est extrêmement simple mais elle consiste à sauvegarder un point de reprise lors de chaque communication. La technologie de mémoire stable rapide a permis d'obtenir des performances acceptables à l'époque en dépit de la taille et de la fréquence des points de reprise. Cependant, la mémoire stable rapide est un composant matériel spécifique dont le coût de développement est non négligeable. Ce coût serait prohibitif actuellement, la tendance étant à l'utilisation de composants standard du marché.

Dans le système Gothic, la gestion mémoire et celle du recouvrement arrière ne sont pas intégrées. Ceci résulte en un usage non optimal de la ressource mémoire puisque les bancs de la mémoire stable sont dédiés aux données de récupération.

A l'issue du projet Gothic est née l'idée d'utiliser une mémoire stable rapide dans une architecture multiprocesseur à mémoire partagée et d'intégrer la gestion mémoire et la gestion du recouvrement arrière de manière à mieux utiliser la ressource mémoire et à éviter la sauvegarde inutile des données non modifiées lors de l'établissement d'un point de reprise.

Par ailleurs, il est intéressant de noter l'intérêt présenté par la structure modulaire du noyau de système Spart qui a servi de base au développement du système d'exploitation de Gothic. L'ajout de nouveaux services système tel que le service de communication ou le service de reprise de processus en a été très largement facilitée. En cela, le noyau SPART préfigurait la technologie micro-noyau.

Les travaux dans le domaine des communications de groupe se sont poursuivis très activement dans la communauté scientifique internationale au cours de ces dernières années [91, 41, 104, 105, 110, 38]. Ces travaux concernent essentiellement la conception de protocoles avec des propriétés adaptées à des classes d'applications peu étudiées au moment de la conception du système de communication fiable de Gothic telles que les applications coopératives, les applications s'exécutant dans les systèmes à grande échelle comme Internet, les applications présentant des contraintes temporelles fortes. D'autres protocoles ont été conçus pour la gestion de la réplication à des fins de tolérance aux fautes ou pour garantir des contraintes de sécurité.

Les travaux dans le domaine de la conception de systèmes distribués intégrés se sont également poursuivis. Actuellement le partage de ressources est étudié dans le contexte des réseaux de stations de travail (NOW) dans le but d'exécuter efficacement sur ce type d'architectures des applications de calcul parallèle traditionnellement mises en œuvre sur des machines parallèles. Ces applications sont caractérisées par la grande taille de l'ensemble des données manipulées et leur besoin élevé en puissance de calcul. Des travaux récents portent sur le partage de la ressource mémoire [5, 83] disponible dans le réseau. Le partage de ressources est également étudié dans le cadre de la conception de serveurs distribués à usage des services implantés sur Internet. Les contraintes particulières à ce type d'environnement sont liées au grand nombre d'utilisateurs et à la variabilité de ce nombre. Plusieurs études portent sur l'équilibrage de charge [95].

tel-00434053, version 1 - 20 Nov 2009

Chapitre 3

Multiprocesseurs à mémoire partagée tolérants aux fautes

3.1 Introduction

Les applications transactionnelles demandent à être exécutées sur des calculateurs à haute performance et tolérants aux fautes pour satisfaire les exigences des utilisateurs. Plusieurs types d'architectures peuvent répondre à ces besoins, en particulier, les multiprocesseurs à mémoire partagée qui présentent l'avantage du partage automatique des données et de l'équilibrage de charge.

A la suite des résultats obtenus dans le cadre du projet Gothic, j'ai entrepris des travaux de recherche dont l'objectif était de concevoir une architecture multiprocesseur à mémoire partagée tolérante aux fautes et efficace. Pour concilier efficacité et tolérance aux fautes dans ce type d'architecture, nous avons proposé de remplacer la mémoire partagée par une mémoire stable rapide offrant des mécanismes adaptés à la mise en œuvre d'une stratégie de retour arrière. Un des principaux problèmes à résoudre est alors de garantir, en cas de retour arrière suite à la défaillance d'un des processeurs, un état global cohérent du système de processus communiquant par partage de mémoire.

Ces travaux ont été effectués dans le cadre du projet européen FASST. Ils ont donné lieu à la thèse de Philippe Joubert [65].

Dans la suite de ce chapitre, nous présentons tout d'abord les caractéristiques des architectures multiprocesseurs à mémoire partagée. Dans le paragraphe 3.3, nous introduisons brièvement les techniques de traitement d'erreurs dans ce type d'architecture. Nous détaillons ensuite les problèmes que pose la mise en œuvre d'une stratégie de retour arrière dans un multiprocesseur à mémoire partagée et donnons un état de l'art des solutions existantes. Nous présentons dans le paragraphe 3.5 notre approche. Le paragraphe 3.6 conclut par un bilan de nos travaux.

3.2 Les multiprocesseurs à mémoire partagée

Les multiprocesseurs à mémoire partagée sont constitués d'un ensemble de processeurs et de leur cache associé [1] interconnectés par un bus ou un réseau multi-étage sur lequel est également connectée la mémoire partagée. Dans une telle architecture, lorsqu'un processeur accède à une donnée de la mémoire, celle-ci est amenée dans son cache. La mise à jour de la mémoire lorsque des données sont modifiées dans un cache peut être effectuée soit par une stratégie d'écriture simultanée soit par une stratégie d'écriture retardée.

La réplication des données dans les caches, engendrée par le partage de mémoire, pose le problème du maintien de la cohérence des différents exemplaires d'un même bloc qui peuvent exister dans des caches distincts. Seules les opérations de modification sur des blocs existants en plusieurs exemplaires sont susceptibles d'introduire des incohérences. Il existe principalement deux classes de protocoles de cohérence qui se distinguent par la gestion des opérations d'écriture sur les données partagées. Dans les protocoles à diffusion des écritures, tels que le protocole Firefly [126], la modification d'un bloc est immédiatement diffusée à l'ensemble des mémoires en possédant un exemplaire. Dans les protocoles à invalidation sur écriture, la modification d'une donnée entraîne immédiatement l'invalidation des exemplaires existants de cette donnée. Les protocoles Berkeley [66] et Illinois [97] sont représentatifs de cette seconde classe de protocoles.

Les protocoles de cohérence de caches les plus répandus dans les architectures multiprocesseurs à mémoire partagée sont à invalidation sur écriture. Les mises en œuvre à base d'espionnage sont privilégiées en vertu de leur simplicité. La stratégie d'écriture retardée permet d'éviter la contention sur le bus. La plupart des microprocesseurs possède dans leur gamme multiprocesseur un contrôleur de cache conçu pour supporter un protocole de cohérence à espionnage de bus. Plusieurs multiprocesseurs commerciaux utilisent ce type de protocole tels le Sequent Symmetry [82] ou l'Alliant FX [3].

3.3 Tolérance aux fautes dans les multiprocesseurs à mémoire partagée

La tolérance aux fautes dans les multiprocesseurs à mémoire partagée repose sur deux types de techniques de traitement d'erreurs. Les techniques de *compensation d'erreurs* effectuent un traitement particulier sur l'état du système pour fournir un service conforme en dépit des erreurs qui peuvent affecter cet état. La plupart du temps ces techniques répliquent de façon matérielle ou logicielle les calculs de manière à réaliser un traitement systématique des erreurs même en l'absence d'erreur ; on parle alors de *masquage d'erreur*. Au contraire, les techniques fondées sur la détection suivie du *recouvrement d'erreur* attendent la détection d'une erreur pour entreprendre des actions visant à substituer un état exempt d'erreur à l'état erroné.

1. Les processeurs actuels possèdent bien souvent plusieurs niveaux de cache. On entend alors par cache le cache de niveau supérieur.

tel-00434053, version 1 - 20 Nov 2009

tel-00434053, version 1 - 20 Nov 2009

3.3.1 Masquage des erreurs

Les techniques de masquage d'erreurs sont fondées sur la comparaison et le vote entre un ensemble de composants répliqués de façon matérielle ou logicielle. La réplication active, habituellement utilisée pour des systèmes communiquants par messages, masque les erreurs en répliquant les calculs sur plusieurs processeurs. Dans les multiprocesseurs à mémoire partagée, une réplication matérielle de type nMR (*n Modular Redundancy*) peut être employée tel que dans l'architecture Tandem Integrity S2 [62]. Le masquage d'erreurs est assuré par des techniques de vote entre composants matériels répliqués. L'architecture Stratus [130] repose également sur de la redondance matérielle statique. Chaque composant (carte processeur, contrôleur d'E/S,...) d'un module multiprocesseur est doublé. Une carte processeur est elle-même composée de deux processeurs dont les résultats sont comparés en permanence. En fonctionnement normal, le calcul est effectué sur quatre processeurs simultanément. La défaillance d'un processeur entraîne la déconnexion de la carte processeur fautive, le calcul se poursuivant sur la seconde carte.

Ce type d'approche est très coûteux en matériel. En outre, les techniques de masquage d'erreurs sont inopérantes pour traiter les fautes logicielles ou les fautes temporaires multiples. Elles sont cependant intéressantes dans le cas où des contraintes temps réel fortes sont exigées.

3.3.2 Détection d'erreur et recouvrement arrière

Lorsque les applications considérées ne sont pas critiques, on peut accepter des baisses temporaires de puissance pourvu que le système soit toujours en service. Les techniques de détection et de recouvrement d'erreurs fournissent ce type de fonctionnalité en impliquant un surcoût important au moment de la détection d'erreur. Deux classes de méthodes sont applicables : la *récupération avant* encore appelée *poursuite* et *la récupération arrière* appelée aussi *reprise*.

La récupération avant vise à reconstruire un état sain à partir d'un état erroné en y apportant les modifications nécessaires pour éliminer les erreurs. Elle s'adresse à des fautes anticipées et reste inutilisable dans le cas de défaillances de processeurs. Son alternative, la récupération arrière, est bien adapté aux systèmes multiprocesseurs à mémoire partagée pour tolérer les défaillances transitoires ou permanentes d'un processeur.

3.4 Mise en œuvre du recouvrement arrière dans un multiprocesseur à mémoire partagée

3.4.1 Principe

La récupération arrière procède par restauration d'un état du système préalablement sauvegardé. Cette restauration d'état simule un retour dans le temps en ramenant le système dans un état qu'il occupait avant la manifestation de la faute.

La restauration complète de l'état du système permet de se passer d'une estimation précise de l'étendu des dommages subis par celui-ci et rend donc possible la récupération d'erreurs provoquées par n'importe quel type de faute. De ce fait, la récupération arrière

est applicable à tous les systèmes pour lesquels un état correct peut être restauré, ce qui en fait un mécanisme général.

Nous précisons maintenant la terminologie employée tout au long de ce chapitre [76]. Un *point de récupération* est un instant passé auquel il pourra être nécessaire de faire revenir le système en restaurant l'état que le système occupait à cet instant. Un point de récupération est *établi* en sauvegardant toutes les informations utiles à la restauration future de ce point de récupération. Les informations sauvegardées sont appelées *données de récupération*. Un point de récupération est *restauré* en utilisant les données de récupération pour ramener le système dans l'état qu'il occupait au moment de l'établissement de ce point de récupération. Pour limiter la taille des données de récupération nécessaire à la restauration des différents points de récupération, il est nécessaire de prévoir une possibilité de *suppression* des points de récupération devenus inutiles. Lorsqu'un point de récupération est supprimé, le système effectue une *validation*. Un point de récupération est *actif* à partir de l'instant où il est établi jusqu'à l'instant oùil est validé. La période durant laquelle un point de récupération est actif constitue la *région de récupération* de ce point de récupération.

La mise en œuvre d'un protocole de récupération arrière soulève essentiellement deux problèmes : la prise en compte des communications pour garantir l'existence d'une ligne de récupération et le stockage des données de récupération sur un support assurant la propriété de stabilité.

Stabilité d'une donnée La stabilité d'une donnée est assurée si les trois propriétés suivantes sont respectées [74]:

1. *accessibilité :* une donnée stable reste accessible quelle que soit la défaillance survenue,
2. *inaltérabilité :* une donnée stable n'est pas altérée par une défaillance,
3. *atomicité de mise à jour :* la mise à jour d'une donnée stable est une opération qui, ou bien réussit totalement, ou bien échoue et laisse la donnée dans son état initial.

La propriété de *permanence* est la conjonction des deux premières propriétés. La mise en œuvre effective des propriétées de stabilité introduit nécessairement une réplication des données concernées, le degré de réplication dépendant du nombre de fautes à tolérer.

3.4.2 Impact des communications

Dans un système constitué d'un ensemble de processus communicants, il faut éviter que la restauration du point de récupération d'un processus n'introduise des incohérences dans l'exécution des autres processus. L'impact de la défaillance d'un processus sur l'activité des autres processus du système a principalement été étudié pour des processus communiquant par échange de messages [44]. Les solutions proposées s'appliquent aussi aux processus communiquant par variables partagées en assimilant l'envoi d'un message à l'écriture d'une variable partagée et la réception d'un message à la lecture d'une valeur précédemment écrite par un autre processus.

Considérons l'exemple de la figure 3.1 où deux processus, P_1 et P_2, communiquent par échange de message. Suite à l'envoi du message *m* à destination de P_2, P_1 subit une défaillance provoquant son retour à son point de récupération le plus récent, C_1. Le processus P_2 se trouve alors dans une situation incorrecte puisqu'il a reçu un message non émis par

tel-00434053, version 1 - 20 Nov 2009

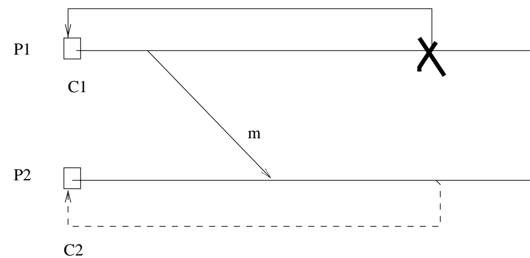


FIG. 3.1 – *Influence des communications sur le retour arrière*

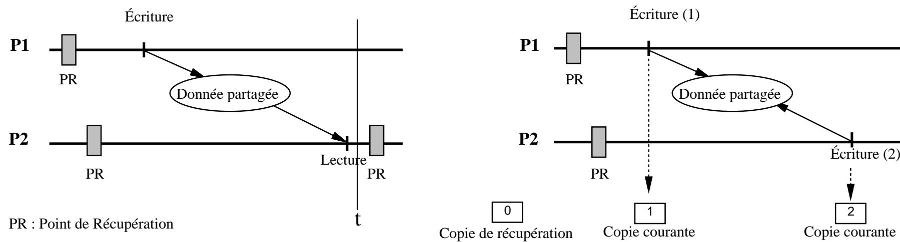


FIG. 3.2 – *Interactions entre deux processus*

P_1. Cette situation peut être évitée en forçant un retour arrière de P_2 coordonné à celui de P_1. Une telle stratégie peut souffrir de l'*effet domino* [103] dans lequel on assiste à une cascade de retours arrière des processus pouvant éventuellement ramener le système dans son état initial. Il est donc nécessaire de prendre en compte les communications entre les processus pour restaurer un *état global cohérent* [29] du système en cas de retour arrière. La situation précédente est équivalente à l'écriture d'une variable partagée par un processeur suivie d'une lecture de cette variable par un autre processeur (voir figure 3.2) dans un système à mémoire partagée.

En outre, dans un système à mémoire partagée, les écritures concurrentes sur une variable partagée posent problème dans le cas de la restauration des points de récupération d'un sous-ensemble des processus du système en cas de défaillance de l'un d'entre eux. Considérons l'exemple de la figure 3.2, où le processus P_1 a écrit la valeur 1 dans la variable partagée, la valeur précédente, 0, étant conservée comme donnée de récupération. Supposons que le processus P_2 fasse un retour arrière au temps t après qu'il ait écrit la valeur 2 dans la variable partagée. Après le retour arrière de P_2, le processus P_1 devrait lire la valeur 1 pour se conformer à la propriété de cohérence séquentielle [73]. Or, la seule valeur qui peut être restaurée pour la variable partagée est celle du dernier point de récupération (valeur 0) puisque la valeur écrite par P_1 a été écrasée par celle écrite par P_2. P_1 est donc potentiellement dans un état incohérent. Bien qu'il ne s'agisse pas d'une communication, une telle interaction, de type *écriture après écriture* doit être prise en compte pour restaurer un état global cohérent du système en cas de retour arrière.

La notion d'état global cohérent permet de déterminer quels sont les processus affectés par un retour arrière du fait des communications. En effet, quand un processus effectue un

retour arrière en restaurant un de ses points de récupération, il est nécessaire que les états de l'ensemble des processus après la restauration forment un état global cohérent pour permettre la reprise correcte du calcul. Si tel n'est pas le cas, d'autres processus doivent restaurer un de leurs points de récupération de façon à aboutir à un état global cohérent. Un ensemble de points de récupération actifs dont la restauration ramène le système dans un état global cohérent est appelé *ligne de récupération*. Trois stratégies ont été proposées pour garantir que l'ensemble des points de récupération des processus forme une ligne de récupération dans un multiprocesseur à mémoire partagée.

Synchronisation globale

La stratégie la plus simple pour identifier une ligne de récupération est que tous les processus établissent simultanément leur point de récupération. Si un processus effectue un retour arrière, tous les processus doivent restaurer leur point de récupération. Cette approche a été proposée pour le multiprocesseur CARER [2] avec la méthode *full-synchronized*. L'inconvénient majeur de cette méthode est que tous les processeurs sont impliqués dans l'établissement d'un point de récupération.

Synchronisation des communications et des points de récupération

L'objectif des approches fondées sur la synchronisation des communications et des points de récupération est de permettre de restaurer le point de récupération d'un processus indépendamment des autres processus.

Dans CARER [2, 133], la création de dépendances inter-processeurs est empêchée en forçant l'établissement d'un point de récupération par un processeur avant qu'il ne délivre une donnée modifiée à un autre processeur. Dans les deux cas de la figure 3.2, le processeur P_1 établirait un point de récupération avant de fournir la donnée partagée à P_2.

Une autre méthode pour empêcher la formation de dépendances entre est de ne pas fournir de mémoire partagée. Dans l'architecture Sequoia [14], les accès aux structures de données partagées doivent être effectués au sein de sections critiques explicites protégées par des verrous *test-and-set*. Pour empêcher la création de dépendances inter-processus, une nouvelle région de récupération est établie autour de chaque section critique. Ceci complexifie le système d'exploitation qui doit convenablement établir et valider les points de récupération et vider (*flush*) le cache pour assurer une sémantique correcte de la mémoire partagée.

Ces deux solutions évitent les dépendances inter-processeurs en synchronisant les communications et les points de récupération. Toute incohérence est évitée du fait qu'un processeur ne communique que les données qui ne peuvent être restaurées. La fréquence d'établissement des points de récupération est directement dépendante du nombre de communications engendrées par l'application et peut être la cause d'une importante dégradation de performance [59, 12].

Coordination dynamique

Un compromis entre la coordination globale et la synchronisation des points de récupération et des communications consiste à enregistrer les communications inter-processus afin

tel-00434053, version 1 - 20 Nov 2009

tel-00434053, version 1 - 20 Nov 2009

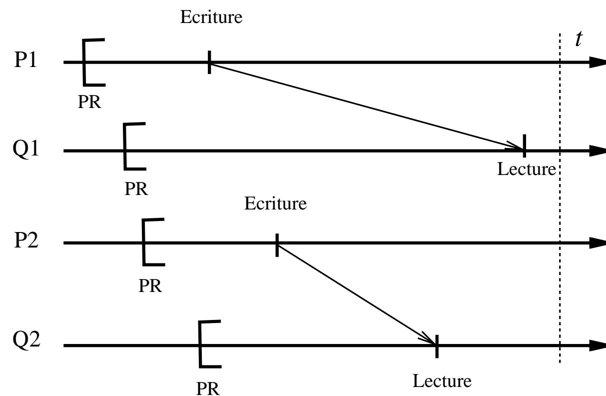


FIG. 3.3 – *Exemple de communications*

de calculer dynamiquement une ligne de récupération. Considérons les deux interactions de la figure 3.2. Les communications entre les processus imposent des actions minimales pour préserver la cohérence de l'état du système en cas de retour arrière. Dans le cas de l'interaction *lecture après écriture*, le retour arrière du processus P_1 au temps t nécessite le retour arrière de P_2 puisque P_1 a restauré l'état précédant son écriture. L'établissement d'un point de récupération par P_2 au temps t nécessite l'établissement d'un point de récupération par P_1 pour garantir l'existence d'une ligne de récupération. Toutefois, le processus P_1 peut établir un point de récupération au temps t sans impliquer P_2 et P_2 peut restaurer son point de récupération sans que P_1 doive faire de même.

Symétriquement, dans le cas d'une interaction *écriture après écriture*, le retour arrière du processus P_2 au temps t nécessite celui de P_1 être restaurée. L'établissement d'un point de récupération à l'instant t oblige P_2 à faire de même puisque le processus P_1 ne peut être restauré dans un état qui existait avant son écriture dans le cas où P_2 effectue un retour arrière. En revanche, le processus P_2 peut établir un point de récupération à l'instant t sans impliquer P_1 et le processus P_1 peut restaurer son point de récupération sans que P_2 soit obligé de faire de même.

Pour mettre en œuvre une telle méthode, il est nécessaire d'enregistrer les communications entre processeurs comme cela est effectué dans le schéma *flagged synchronized* de CARER [2]. Cette technique s'appuie sur un drapeau d'un bit présent dans le cache de chaque processeur qui indique s'il a communiqué avec au moins un autre processeur depuis l'établissement de son dernier point de récupération. L'ensemble des processeurs dont le drapeau est positionné forme une ligne de récupération. La détection des communications s'effectue de la façon suivante : lorsqu'un processeur tente d'accéder une donnée modifiée se trouvant dans le cache d'un autre processeur, le contrôleur de cache positionne son drapeau ainsi que tous les caches possédant une copie de cette donnée.

L'inconvénient de cette approche est qu'elle donne seulement une approximation de l'ensemble des processeurs qui doivent établir ou restaurer un point de récupération. Considérons l'exemple de la figure 3.3 où les flèches représentent les communications entre les processeurs. Au temps t, le processeur P_1 a communiqué avec le processeur Q_1 et le pro-

cesseur P_2 a communiqué avec Q_2. Par conséquent, tous les processeurs ont positionné leur drapeau. Si un processeur quelconque établit un point de récupération au temps t, alors les quatre processeurs doivent établir un point de récupération. Cependant, d'après l'étude de l'impact de l'interaction *lecture après écriture* sur le protocole de récupération, les processeurs P_1 et P_2 pourraient établir un point de récupération au temps t indépendamment de tout autre processseur tout en préservant la cohérence de l'état du système en cas de retour arrière. Pour la même raison, si le processeur Q_1 (respectivement Q_2) établit un point de récupération, seul le processeur P_1 (respectivement P_2) est obligé d'établir un point de récupération.

Une situation similaire se produit en cas de retour arrière. Les quatre processeurs doivent restaurer leur point de récupération si l'un d'entre eux effectue un retour arrière. Néanmoins, pour assurer la cohérence de l'état global du système après le retour arrière, les processeurs Q_1 et Q_2 auraient pu effectuer un retour arrière indépendamment de tout autre processeur. De façon analogue, le retour arrière de P_1 (respectivement P_2) devrait seulement impliquer le retour arrière de Q_1 (respectivement Q_2).

La méthode précédente est insuffisante car elle n'enregistre pas l'identité des processeurs communicants. Un mécanisme plus précis d'enregistrement des dépendances possède potentiellement l'avantage de minimiser le surcoût en terme de performance du protocole de récupération pour deux raisons. D'une part, il élimine le problème de la fréquence élevée d'établissement de points de récupération observée dans les approches dans lesquelles communications et points de récupération sont synchronisés. D'autre part, il minimise le nombre de processeurs impliqués dans l'établissement de points de récupération. Dans le cadre du projet FASST, nous avons proposé une architecture qui enregistre plus finement les communications entre les processeurs et permet de calculer l'ensemble minimal des processeurs communicants impliqués dans l'établissement ou la restauration d'un point de récupération. Notre proposition est décrite dans le paragraphe 3.5.

3.4.3 Le stockage des données de récupération

La sauvegarde d'un point de récupération nécessite une recopie des données actives sur un support permettant de les conserver intactes en vue d'un éventuel retour arrière.

Schémas utilisant la hiérarchie mémoire

Dans les schémas utilisant la hiérarchie mémoire, les données actives et de récupération sont stockées dans des niveaux distincts de la hiérarchie mémoire. Les niveaux les plus bas contiennent les données modifiées (données actives) et les niveaux les plus hauts, plus éloignés du processeur, contiennent les données de récupération. Les différents niveaux de la hiérarchie mémoire fournissent alors un moyen simple pour identifier les données de récupération.

Un schéma de ce type est proposé dans les architectures multiprocesseurs Sequoia [14] et CARER [2, 133] (voir figure 3.4). Les caches et les registres des processeurs contiennent les données actives. Les données de récupération sont stockées dans la mémoire partagée de la machine. Dans une région de récupération, toute modification de donnée doit être réalisée dans un cache sans mise à jour de la mémoire qui détruirait le point de récupération. Cette solution suppose donc l'utilisation de caches à copie retardée dans lesquels les

tel-00434053, version 1 - 20 Nov 2009

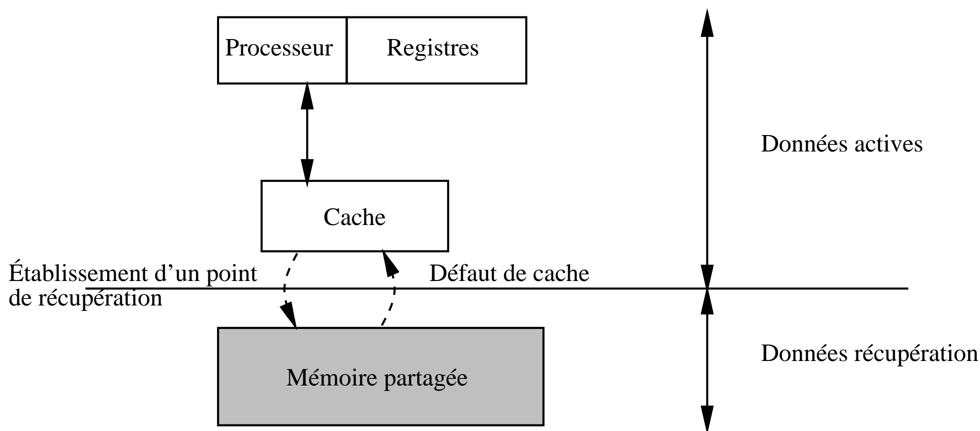


FIG. 3.4 – *Exemple de schéma hiérarchique*

modifications ne sont pas envoyées immédiatement vers la mémoire. La mise à jour de la mémoire est réalisée lorsqu'une ligne modifiée doit être évacuée d'un cache. Cette opération déclenche alors l'établissement d'un point de récupération qui met à jour la mémoire en recopiant l'ensemble des lignes modifiées du cache vers la mémoire. Le retour arrière d'un processeur est réalisé en invalidant le cache et les registres du processeur. Les données sont alors fournies par la mémoire.

L'intérêt majeur de cette approche est sa simplicité de mise en œuvre. Elle tire parti de la réplication de données déjà existante dans les différents niveaux d'une hiérarchie mémoire pour faciliter la sauvegarde et la restauration des points de récupération. Son défaut tient au caractère indécidable du nombre d'établissements de points de récupération qu'elle engendre. Chaque fois qu'une donnée active modifiée doit être envoyée vers le niveau de la hiérarchie contenant les données de récupération, un nouveau point de récupération doit être établi. Dans le cas de caches, le nombre de ces opérations dépend des caractéristiques du cache (taille, associativité,...) ainsi que des références mémoires générées par l'application. Il a été montré que cette approche peut sévèrement limiter les performances d'une architecture qui l'utilise [59, 12].

Schémas mixtes avec localisation physique fixe des données de récupération

Les schémas mixtes autorisent la cohabitation des données de récupération et des données actives dans un même niveau de la hiérarchie mémoire. Le stockage des données de récupération sur des supports indépendants de ceux utilisés pour les données actives permet de les identifier simplement par leur localisation physique.

Ces schémas corrigent le principal défaut des schémas hiérarchiques en rendant la fréquence de sauvegarde des points de récupération indépendante de l'application et des caractéristiques matérielles de la machine. C'est la raison pour laquelle nous avons retenu cette approche pour l'architecture que nous avons proposée dans le cadre du projet FASST.

tel-00434053, version 1 - 20 Nov 2009

3.4.4 Facteurs d'efficacité d'une stratégie de recouvrement arrière

Il ressort des paragraphes précédents que la durée et la fréquence des opérations d'établissement d'un point de récupération ont un impact important sur les performances d'une architecture fondée sur une stratégie de retour arrière.

La durée d'une opération de sauvegarde d'un point de reprise dépend de plusieurs facteurs : support de stockage des données de récupération, quantité de données de récupération qui est fonction du nombre de processeurs impliqués dans une sauvegarde de point de reprise et de la quantité de données qui ont été modifiées depuis le dernier point de récupération, du grain de la mémoire.

La fréquence d'établissement d'un point de récupération peut être fixée par l'utilisateur ou contrainte par les communications selon la stratégie retenue pour garantir l'existence d'une ligne de récupération.

Pour concilier efficacité et tolérance aux fautes dans un multiprocesseur à mémoire partagée, nous avons proposé une approche qui minimise la durée d'une opération d'établissement point de récupération en agissant sur les différents facteurs qui l'influencent et qui laisse le choix de la fréquence des points de récupération à l'utilisateur.

3.5 Approche fondée sur une mémoire partagée récupérable

Nous présentons dans ce paragraphe le multiprocesseur à mémoire partagée tolérant aux fautes que nous avons conçu dans le cadre du projet européen FASST. Dans cette architecture, la mémoire partagée est une mémoire partagée recouvrable (RSM pour *Recoverable Shared Memory*) qui met en œuvre un protocole de retour arrière pour tolérer les fautes transitoires ou permanentes simples des processeurs. Le module RSM offre les fonctionnalités habituelles d'une mémoire ainsi que le mécanisme de récupération arrière présenté ci-après. Les processeurs sont supposés *fail-stop* [111]. Un bus doublé peut être utilisé pour tolérer les défaillances du bus. Dans les paragraphes qui suivent, nous présentons le protocole de récupération arrière utilisé dans l'architecture et la façon dont il est mis en œuvre par la RSM. Nous donnons quelques résultats de performance.

3.5.1 Protocole de récupération arrière

Le mécanisme de base de la RSM est d'enregistrer des données de récupération pour chaque bloc mémoire en en conservant deux copies. Quand un point de récupération est établi, les deux copies du bloc contiennent la même donnée. Les mises à jour ultérieures du bloc sont effectuées sur une seule des copies et par conséquent, l'état du bloc au moment de l'établissement du dernier point de récupération est conservé dans la seconde copie (et peut donc être utilisé comme donnée de récupération). Un point de récupération est validé quand un nouveau point de récupération doit être établi, le protocole de récupération offrant une seule opération de *validation/établissement* d'un point de récupération au lieu de deux opérations distinctes. Pour identifier une ligne de récupération, le protocole (i) détecte et enregistre l'occurrence de communications inter-processeurs qui ont lieu en cas de partage

tel-00434053, version 1 - 20 Nov 2009

tel-00434053, version 1 - 20 Nov 2009

de données, et (ii) utilise cette information pour synchroniser les opérations d'établissement et de restauration d'un point de récupération des processeurs ayant communiqué. Les communications inter-processeurs sont enregistrées à un grain très fin pour minimiser le nombre de processeurs impliqués dans les opérations d'établissement et de restauration de points de récupération.

Relation de dépendance Comme expliqué au paragraphe 3.4.2, deux types d'interactions entre les processeurs ont un impact sur l'identification d'une ligne de récupération : les interactions *lecture après écriture* et les interactions *écriture après écriture*. Les effets de ces interactions sur les opérations de récupération sont prises en compte par la RSM par le biais d'une relation de dépendance. Un processeur P_1 dépend d'un processeur P_2 si :

– la restauration du point de récupération de P_1 nécessite celle de celui de P_2 (pour garantir que l'exécution reprend depuis une ligne de récupération),
– si P_2 effectue une opération de validation/établissement d'un point de récupération alors P_1 doit le faire également (pour assurer l'existence d'une ligne de récupération).

Dans le cas d'une interaction de type *lecture après écriture* entre les processeurs P_1 et P_2, comme sur l'exemple de la figure 3.2, P_1 dépend de P_2 dès que P_2 lit la valeur écrite par P_1. Dans le cas d'une interaction de type *écriture après écriture* entre les processeurs P_1 et P_2, comme indiqué sur la figure 3.2, P_2 dépend de P_1 dès que P_2 modifie la valeur écrite par P_1.

Opérations du protocole de récupération Il est clair que la relation de dépendance est transitive. Pour assurer l'existence d'une ligne de récupération, quand un processeur valide/établit (respectivement restaure) un point de récupération, ses ancêtres (respectivement ses descendants) directs et ceux obtenus par transitivité selon la relation de dépendance doivent effectuer la même opération. Ces opérations coordonnées doivent être atomiques vis à vis des défaillances des processeurs. Les processeurs impliqués dans une opération de validation/établissement ou de restauration d'un point de récupération constituent un *groupe de dépendances*. Quand un processeur effectue une opération du protocole de récupération, la RSM calcule la fermeture transitive de la relation de dépendance pour déterminer le groupe de dépendance. Ensuite la RSM informe les membres du groupe de dépendance qu'ils doivent participer à l'opération en cours.

Enregistrement des communications inter-processeurs

Quand un bloc mémoire est accédé par un processeur, la RSM a besoin des informations suivantes pour enregistrer les dépendances : l'identité du processeur réalisant l'accès, le type d'accès (lecture ou écriture) et l'identité du processeur qui est l'*écrivain actif* du bloc s'il existe. Un processeur est appelé *écrivain actif* d'un bloc mémoire *b* si ce processeur a modifié *b* dans sa région de récupération courante et si *b* n'a pas ensuite été modifié par un autre processeur.

Bien que les requêtes de lecture et écriture peuvent concerner des mots mémoire, la RSM enregistre les dépendances en considérant le bloc comme grain. La RSM conserve donc pour chaque bloc mémoire (i) les valeurs courantes de toutes les lignes de caches qui

ont été recopiées en mémoire, (ii) un champ contenant l'identité de l'écrivain actif du bloc (s'il existe), et (iii) les données de récupération du bloc.

Comme la RSM est connectée uniquement au bus commun, les dépendances sont enregistrées en espionnant les transferts sur le bus. Pour enregistrer toutes les dépendances, la RSM doit détecter sur le bus tous les accès à des données modifiées. Elle doit également détecter tout changement d'écrivain actif. La présence de caches et protocoles de cohérence de caches complique la tâche de la RSM puisqu'un certain nombre d'accès ne génèrent pas de trafic sur le bus. Par exemple, avec un protocole à diffusion sur écriture, les mises à jour sont diffusées à tous les processeurs ayant une copie de la ligne de cache qui fait l'objet de la modification. Un processeur est alors susceptible de lire une valeur modifiée sans accéder au bus. Cette situation est prise en compte dans les mécanismes de détection des dépendances de la RSM. Le mécanisme de détection et enregistrement des dépendances peut être mis en œuvre pour tout protocole de cohérence de caches. Toutefois, avec les protocoles à diffusion sur écriture, une approche préventive doit être adoptée pour l'enregistrement des dépendances qui conduit à un plus grand nombre d'opérations de récupération que strictement nécessaire si l'écrivain actif défaille. Les protocoles à invalidation sur écriture sont mieux adaptés au mécanisme précis d'enregistrement des dépendances puisque toutes les interactions *lecture après écriture* provoquent des accès au bus que la RSM peut exploiter afin d'assurer l'enregistrement d'un ensemble minimal de dépendances inter-processeurs. Le lecteur intéressé pourra se reporter à [65] pour la description détaillée des problèmes posés pour l'enregistrement des dépendances et la mise à jour de l'écrivain actif en présence de protocoles de cohérence.

Validation/établissement de points de récupération

Une opération de validation/établissement d'un processeur est mise en œuvre par un simple protocole de validation à deux phases distribué [52]. Les processeurs sont les participants tandis que la RSM est le coordinateur du protocole.

Initialisation du protocole Un processeur qui désire valider/établir un point de récupération doit d'abord s'assurer que les valeurs courantes de la mémoire qu'il a modifiées au sein de sa région de récupération courante sont bien à jour dans la RSM. Le contexte interne du processeur (registres, TLB,...) est recopié dans la RSM. Ensuite, toutes les lignes modifiées du cache sont recopiées en mémoire et marquées non modifiées. Une fois l'opération de vidage effectuée, le processeur envoie une commande *valider* à la RSM et se met en attente d'une interruption signifiant la terminaison de l'opération de validation/établissement. Le processeur peut alors reprendre ses calculs.

Première phase du protocole A la réception d'une commande *valider* en provenance du processeur P_i, la RSM consulte l'information de dépendance enregistrée au cours de la région de récupération courante de P_i pour déterminer le groupe de dépendance, *ie* le groupe des processeurs qui doivent valider/établir un point de récupération atomiquement avec P_i. Une fois le groupe de dépendance calculé, chaque processeur du groupe est informé de sa participation à l'opération de validation/établissement au moyen d'une interruption *it-validation*. Sur réception de cette interruption, un processeur dépendant peut copier son

tel-00434053, version 1 - 20 Nov 2009

contexte interne et vider ses lignes de cache dans la RSM avant d'envoyer une commande *valider* faisant office d'acquittement, pour indiquer qu'il a terminé la première phase du protocole. Pendant l'intervalle entre la commande *valider* initiale et la réception des acquittements de tous les processeurs dépendants, de nouvelles dépendances peuvent avoir été créées puisque la RSM continue de servir les requêtes de lecture et d'écriture des processeurs qui ne sont pas bloqués en attente de la terminaison du protocole. Ces processeurs doivent être ajoutés au groupe de dépendance. Il se peut aussi qu'un processeur ne faisant pas partie du groupe décide de valider/établir et envoie une commande *valider* à la RSM. Ce processeur est ajouté au groupe courant de même que tous les processeurs qui en dépendent.

Pour qu'il soit correct, il est important que le calcul du groupe de dépendance soit atomique vis à vis des accès en lecture et écriture à la RSM. Un moyen simple de mettre en œuvre cette atomicité est de sérialiser le calcul du groupe et les accès en lecture et écriture.

Seconde phase du protocole Quand tous les acquittements des processeurs participants ont été reçus, la RSM démarre la seconde phase du protocole. Les actions qui doivent alors être effectuées sont les suivantes :

1. remplacement de la valeur de récupération de tous les blocs dont l'écrivain actif participe au protocole par leur valeur courante et réinitialisation du champ écrivain actif de ces blocs,
2. effacement des dépendances des processeurs du groupe,
3. envoi d'une interruption à destination de tous les participants pour qu'ils reprennent leur activité.

Ces opérations doivent être réalisées de manière indivisible vis à vis des accès des processeurs à la mémoire pour éviter la modification par un processeur d'une valeur courante non encore sauvegardée. La mise en œuvre de cette phase a un impact considérable sur les performances de l'architecture. Pour éviter les inconvénients de solutions fondées sur un parcours séquentiel des blocs mémoire, la RSM met en œuvre un mécanisme de copie-sur-écriture, permettant la reprise des opérations mémoire à l'issue de la première phase du protocole de récupération. Ce mécanisme permet en effet de différer la copie effective d'un bloc jusqu'à ce que la copie soit nécessaire *ie* quand un processeur tente de modifier le bloc considéré. Il permet d'entrelacer la seconde phase avec les accès normaux des processeurs, ces derniers ne sont donc pas bloqués en attente de la terminaison de la copie. Il est nécessaire de marquer les blocs de la RSM afin de déterminer si un bloc donné doit être copié ou pas lors d'un accès en écriture. Une approche fondée sur des identificateurs de points de récupération [133] est utilisée à cet effet.

Restauration des points de récupération

Par souci de clarté, nous considérons que la défaillance d'un processeur P_i se traduit par l'envoi d'une commande *restaurer(i)* à la RSM. A la réception de cette commande, la RSM calcule le groupe de dépendance de P_i. Ce groupe contient l'ensemble des processeurs qui doivent effectuer un retour arrière pour que le système retrouve un état global cohérent. Une fois le groupe de dépendances calculé, la RSM recopie les données de récupération de tous les blocs modifiés par les processeurs membres du groupe de dépendance dans les

tel-00434053, version 1 - 20 Nov 2009

valeurs courantes et réinitialise l'écrivain actif de ces blocs. La RSM efface ensuite les dépendances puis envoie une interruption *it-restauration* à chaque processeur pour l'informer de la restauration de son point de récupération et le forcer à abandonner les calculs en cours. Quand un processeur reçoit une interruption *it-restauration*, les valeurs des blocs ont été restaurées seulement en RSM et les lignes de son cache peuvent donc contenir des valeurs incorrectes. Le processeur doit donc invalider le contenu de son cache pour acquérir les valeurs restaurées, en particulier les valeurs de ses registres sauvegardées lors du dernier point de récupération.

3.5.2 Mise en œuvre de la RSM

La RSM est un composant critique de l'architecture tolérante aux fautes, sa fiabilité est donc essentielle. Nous nous concentrons ici sur les mécanismes permettant de mettre en œuvre les fonctionnalités de la RSM liées au protocole de récupération sans détailler les techniques permettant d'assurer la fiabilité de la RSM. Comme toujours, la conception de la RSM est un compromis entre fiabilité, complexité et coût.

La figure 3.5 présente la structure de la RSM. Celle-ci est composée de deux bancs mémoire de taille identique appelés *banc1* et *banc2*. L'espace d'adresses de la RSM est divisé en un ensemble de blocs contigus de même taille. Chaque bloc comprend une valeur courante dans le *banc1*, une valeur de récupération dans le *banc2* et deux champs lui sont associés : l'écrivain actif et l'identificateur de point de récupération. Ce second champ est utilisé par le mécanisme de copie sur écriture qui met en œuvre la seconde phase du protocole de validation/établissement. L'unité de gestion des dépendances enregistre les dépendances dans une matrice $n*n$, n étant le nombre maximal de processeurs dans l'architecture. L'interface bus fournit l'information nécessaire à l'enregistrement des dépendances (identificateur de processeur, adresse, type de l'accès ou de la transaction du protocole de cohérence). Pour les requêtes de lecture et écriture servies par la RSM, les dépendances sont enregistrées sans dégradation de performance puisque toutes les informations nécessaires à cette opération sont stockées en SRAM (mémoire à accès rapide) et la plupart des actions pour enregistrer les dépendances peuvent être effectuées en parallèle avec l'accès mémoire normal.

La RSM met en œuvre des registres de commande et d'état pour gérer les communications avec les processeurs. Ces derniers communiquent avec la RSM en envoyant des commandes tandis que la RSM communique avec les processeurs en générant des interruptions sur le bus. Pour le prototype, la capacité de chaque banc est fixée à 32 Moctets. Des mémoires SRAM sont utilisées pour le stockage de la matrice dépendance, les compteurs de points de récupération et les champs identificateur de point de récupération et écrivain actif associés aux blocs mémoire. En supposant un nombre maximal de seize processeurs, une taille de banc de 32 Moctets, des blocs de 32 octets et des compteurs et identificateurs de points de récupération sur 32 bits, la taille de SRAM est de 96 octets pour la matrice de dépendance et les compteurs de points de récupération et de 4,5 Moctets pour les informations associées aux blocs. Ces surcoûts peuvent être réduits en utilisant des blocs de plus grande taille. Enfin, les unités de gestion des dépendances et de copie-sur-écriture sont mises en œuvre avec des FPGAs.

tel-00434053, version 1 - 20 Nov 2009

tel-00434053, version 1 - 20 Nov 2009

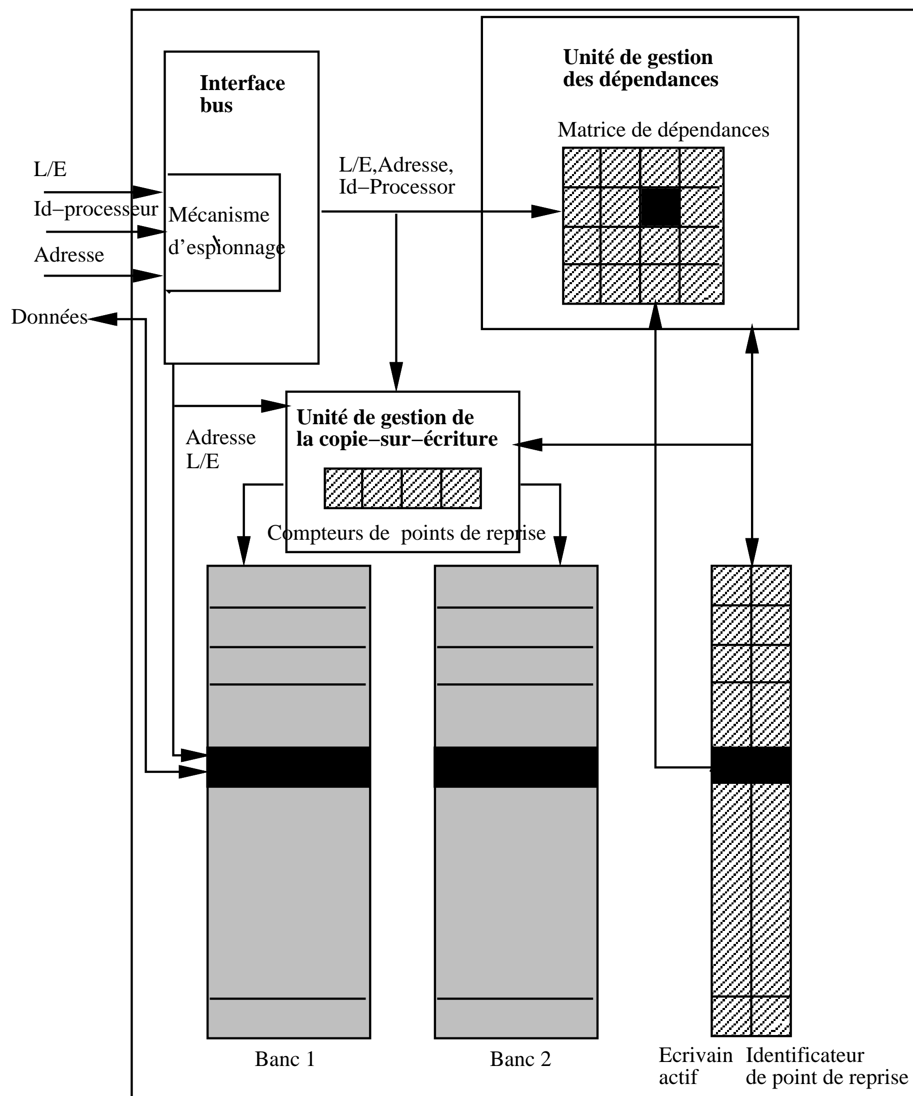


FIG. 3.5 – *La* RSM

tel-00434053, version 1 - 20 Nov 2009

3.5.3 Evaluation de performance

Méthodologie et charge de travail

L'évaluation de performance est une étape clé dans la conception d'une architecture. Elle permet de déterminer les paramètres permettant d'obtenir une configuration performante et de déceler les goulets d'étranglement de l'architecture. Il n'est guère envisageable de mettre en œuvre une architecture sans avoir fait au préalable une évaluation de performance. Nous avons choisi de réaliser notre évaluation par simulation plutôt que d'utiliser des modèles analytiques. En effet, ces derniers ne permettent pas de modéliser un système complexe tel qu'un multiprocesseur à mémoire partagée sans d'importantes approximations. Nous avons utilisé le noyau de simulation SPAM [51] qui met en œuvre une technique de simulation à événements discrets coordonnée à l'exécution [35]. Une telle technique présente l'avantage d'éviter le phénomène de décalage de trace (*trace shifting*) [42].

Ne disposant pas de traces d'applications transactionnelles (les industriels ne les divulguent pas) représentatives de la charge de travail d'une architecture telle que celle que nous étudiée, nous avons réalisé notre évaluation avec des traces d'applications scientifiques parallèles de l'ensemble SPLASH [115] (*cholesky*, *mp3d*, *pthor*, *water*). Ces applications sont utilisées classiquement pour l'évaluation des performances des multiprocesseurs à mémoire partagée. Elles présentent une grande variété de comportements vis à vis de l'utilisation des caches et des accès à la mémoire partagée et constituent donc une base d'étude intéressante.

Un aspect important des simulations est la fréquence des opérations de validation/établissement. Tout protocole de récupération arrière est forcé de valider/établir un point de récupération lors des opérations d'entrée/sortie, pour éviter leur perte ou duplication [76]. Dans les simulations, les opérations d'E/S sont modélisées par des interruptions. Les dates d'arrivée des interruptions sont tirées aléatoirement. Ces dernières sont distribuées équitablement entre tous les processeurs de la machine simulée. Pour choisir une valeur appropriée pour la fréquence des opérations d'E/S, nous avons tenu compte de mesures effectuées sur un serveur de fichier NFS qui font apparaître un taux compris entre 150 et 650 opérations d'E/S par seconde. Pour les simulations, des fréquences variant entre 100 et 5000 opérations d'E/S par seconde ont été utilisées.

Mesures effectuées

Nous avons comparé notre proposition fondée sur la RSM avec les protocoles de récupération de CARER et Sequoia, mesuré l'efficacité du mécanisme de dépendance, et évalué la dégradation de performance engendrée par le protocole de récupération arrière sur une architecture standard.

Comparaison avec CARER et Sequoia Pour effectuer une comparaison objective de notre approche fondée sur la RSM avec les protocoles de récupération de CARER et Sequoia, il n'était pas possible de se fonder sur les performances brutes des trois architectures car elles dépendent très largement des options de mise en œuvre et du modèle de défaillance considérés. En revanche, la quantité de données de récupération générée par chacune des

approches est un critère indépendant de la mise en œuvre du protocole de récupération. Pour un schéma donné, la quantité des données de récupération sur une période de temps donnée correspond au nombre de lignes de caches modifiées dans le système au cours d'une région de récupération et au nombre de points de récupération établis.

Les mesures effectuées ont montré que la taille des données de récupération est toujours supérieure pour Sequoia et CARER à celle de notre approche. Ceci est du à l'effet combiné des caches bloquants (un point de récupération est validé/établi quand une ligne de cache est remplacée) et de la synchronisation des communications et des points de récupération. Pour notre approche, c'est uniquement la fréquence des opérations d'E/S qui détermine le rythme auquel les points de récupération sont établis. Pour CARER et Sequoia, les opérations d'E/S représentent une toute petite fraction du nombre total de points de récupération, tout spécialement lorsque leur fréquence est faible. Notre approche génère une quantité minimum de données de récupération puisqu'elle a le plus faible nombre de points de récupération et l'ensemble minimal de processeurs impliqués dans une opération de validation/établissement grâce au mécanisme de gestion de dépendances. Les résultats obtenus pour CARER et Sequoia sont proches de ceux de la RSM uniquement pour les applications ayant un petit ensemble de travail (ce qui réduit l'effet des caches bloquants) et peu de communications entre les processeurs (l'effet du mécanisme de gestion de dépendances est moins sensible). Dans ces cas, le nombre de points de récupération peut être comparable à celui de notre approche.

Dans les approches CARER et Sequoia, l'établissement des points de récupération est imposé par les paramètres du cache (taille, associativité, politique de remplacement). Le nombre de points de récupération est très difficilement contrôlable et prévisible car il dépend du comportement microscopique de l'application. Comparativement, le principal intérêt de notre approche, est de rendre la fréquence des points de récupération indépendante des paramètres architecturaux et du schéma de communication des applications.

Efficacité du mécanisme de dépendance Concernant le mécanisme de gestion des dépendances, les mesures effectuées ont montré qu'il est d'autant plus efficace que la fréquence d'établissement d'un point de récupération est élevée. Ceci n'est pas surprenant car les processeurs ont une forte probabilité d'avoir communiqué entre eux quand le temps entre deux points de récupération successifs est long. L'impact du mécanisme de gestion des dépendances sur la taille des données de récupération est non négligeable. Pour les applications exhibant un faible taux de partage la quantité de données de récupération est très sensiblement réduite par la gestion des dépendances. Ceci peut avoir un effet important dans un multiprocesseur exécutant une charge de travail générale avec plusieurs programmes indépendants avec peu de partage de données.

Performance d'une architecture fondée sur la RSM Afin d'évaluer les performances d'un multiprocesseur mettant en œuvre notre approche, nous avons comparé le temps d'exécution de plusieurs applications sur une architecture standard et la même architecture mettant en œuvre la RSM et le protocole de récupération associé. Selon les applications et la fréquence des opérations d'E/S, le surcoût de la tolérance aux fautes varie de moins de 1 % à 35 %. Trois facteurs l'expliquent : le temps d'arrêt des processeurs, la bande passante du bus occupée par les opérations de vidage des caches et le mécanisme de copie sur écriture

tel-00434053, version 1 - 20 Nov 2009

de la RSM. La gestion des dépendances contribue à réduire le temps d'occupation du bus lié au vidage des caches par les processeurs. Le surcoût observé est intimement lié à la taille des données de récupération.

3.6 Bilan

Un des avantages majeurs de notre approche fondée sur la RSM pour la mise en œuvre d'un protocole de récupération arrière dans un multiprocesseur à mémoire partagée est qu'elle minimise la fréquence des opérations de validation/établissement en enregistrant les dépendances inter-processeurs tout en utilisant des caches et protocoles de cohérence de caches standard contrairement à Sequoia et CARER. Un autre avantage dérive du premier : la mémoire peut être partagée librement entre les processeurs sans aucune complication logicielle. Enfin, si un retour arrière est nécessaire suite à la défaillance d'un processeur, un nombre minimal de processeurs sont impliqués, ces derniers étant déterminés à partir des communications effectuées. L'évaluation de performance que nous avons effectuée par simulation montre que l'architecture que nous avons proposée obtient de meilleures performances que les autres propositions Carer et Sequoia du fait du mécanisme de gestion des dépendances qui permet de diminuer la quantité de données de récupération sauvegardées et de rendre la fréquence de sauvegarde des points de reprise indépendante des accès des processus aux variables partagées.

Dans l'architecture proposée, la tolérance aux fautes est transparente pour les applications. En outre, peu de modifications sont nécessaires dans le système d'exploitation pour l'intégration du protocole de retour arrière, le problème le plus délicat étant celui de la gestion des entrées/sorties qui sont des opérations non recouvrables par nature [90]. J'ai étudié ce problème dans le cadre de mes activités liées au projet européen FASST. Seule une étude au cas par cas des périphériques permet de traiter les opérations d'entrées/sorties.

Ces travaux ont permis une nette évolution par rapport aux résultats obtenus dans le cadre du projet Gothic. La gestion de la mémoire et celle du recouvrement arrière ont pu être intégrées au sein de la RSM. La RSM est nettement plus élaborée que la mémoire stable rapide de Gothic. Elle offre des fonctionnalités permettant de mettre en œuvre efficacement une stratégie de retour arrière : (1) sauvegarde incrémentale de points de reprise, (2) mécanisme de gestion de dépendances pour réduire à la fois le nombre de processeurs impliqués dans une sauvegarde de point de reprise et la fréquence de ces derniers.

Néanmoins, la RSM reste un composant matériel spécifique. Composant critique de l'architecture tolérante aux fautes, sa conception est un compromis entre fiabilité, complexité et coût. Compte tenu du temps de développement d'un tel composant, il est difficile de suivre les évolutions technologiques rapides. Cet aspect est clairement un inconvénient de notre approche, qui reste néanmoins digne d'intérêt pour les architectures ciblées. Cependant, une telle approche à base de matériel spécifique n'est pas envisageable dans le cadre des architectures extensibles à mémoire partagée car elle serait beaucoup trop coûteuse compte tenu du grand nombre de nœuds.

tel-00434053, version 1 - 20 Nov 2009

tel-00434053, version 1 - 20 Nov 2009

Chapitre 4

Architectures extensibles tolérantes aux fautes

4.1 Introduction

Dans le cadre du projet FASST, je me suis intéressée à la conception d'une architecture multiprocesseur tolérante aux fautes. L'architecture considérée, organisée autour d'un bus commun, ne peut comporter qu'un petit nombre de processeurs (moins de vingt) du fait de la bande passante limitée du bus et de la vitesse croissante des processeurs. De nombreux travaux sont en cours depuis le tout début des années 90 pour concevoir des architectures multiprocesseurs extensibles à mémoire partagée pouvant contenir un très grand nombre de processeurs [1, 53, 77, 80, 47, 70, 109]. A la suite de mes travaux sur les multiprocesseurs à mémoire partagée traditionnels, je me suis tout naturellement intéressée au problème de disponibilité posé par ces nouvelles architectures.

En effet, bien que les progrès de la micro-électronique aient permis d'améliorer de façon très importante la fiabilité des composants matériels des ordinateurs, les problèmes de fiabilité et de disponibilité d'une machine, souvent ignorés du fait de la rareté des fautes quand le nombre de composants est limité, deviennent cruciaux pour une machine extensible pour au moins deux raisons. D'une part, avec l'augmentation du nombre de composants d'une architecture, la probabilité d'une défaillance dans le système augmente aussi. On peut estimer qu'un système comprenant 1000 nœuds ayant chacun un MTBF (*Mean Time Between failures*) de 30000 heures aura lui même un MTBF de 30 heures soit en moyenne presque qu'une défaillance par jour. D'autre part, les applications s'exécutant sur ce type de machine nécessitent des temps d'exécution longs qui ne peuvent être envisagés avec les taux de défaillance annoncés précédemment. Il est donc indispensable que ces architectures intègrent dès leur conception, des mécanismes leur assurant une disponibilité et une fiabilité suffisantes pour être réellement utilisables. Cette préoccupation apparaît chez tous les constructeurs de ce type de machine qui proposent des mécanismes de tolérance aux fautes minimaux autorisant habituellement une reconfiguration de l'architecture en inhibant un composant défaillant. Bien entendu, les calculs en cours d'exécution sont perdus et doivent être intégralement réexécutés. Des calculs de longue durée ne sont pas

envisageables sans l'aide de mécanismes de tolérance aux fautes permettant de tolérer la défaillance d'un nœud de l'architecture tout en assurant la continuité des calculs en cours (*graceful degradation*).

Mon objectif a donc été de proposer une solution simple, efficace et peu coûteuse en développement matériel pour tolérer la défaillance d'un élément dans des architectures extensibles à mémoire partagée. La solution proposée est fondée sur une technique de retour arrière. Ma contribution porte sur la proposition de mécanismes originaux pour la gestion des données de récupération qui exploitent la réplication engendrée par les communications pour mettre en œuvre efficacement une technique de retour arrière. Ces travaux ont été effectués dans le cadre de la convention de recherche Aleth, financée par la DGA/DRET. Ils ont donné lieu aux thèses d'Alain Gefflaut [50] et d'Anne-Marie Kermarrec [68] que j'ai dirigées.

Dans le paragraphe 4.2, je présente les architectures extensibles à mémoire partagée qui constituent le contexte des travaux. L'état de l'art des architectures extensibles à mémoire distribuée cohérente (dont la définition est donnée dans le paragraphe 4.2) est présenté dans le paragraphe 4.3. J'explique dans le paragraphe 4.4 les idées qui m'ont conduit à la conception de mécanismes originaux de gestion des données de récupération pour les architectures à mémoire distribuée cohérente. Le protocole proposé est présenté dans le paragraphe 4.5. Les mises en œuvre de ce protocole que j'ai étudiées dans le cadre d'une architecture COMA d'une part et d'un système à mémoire virtuelle partagée d'autre part sont brièvement décrites dans le paragraphe 4.6. Les résultats de l'évaluation de performance sont discutés dans le paragraphe 4.7. Je termine par un bilan de ces travaux.

4.2 Architectures extensibles à mémoire partagée

Deux grandes classes d'architectures extensibles à mémoire partagée peuvent être distinguées : les architectures NUMA et les systèmes à mémoire virtuelle partagée (MVP).

4.2.1 Les architectures NUMA

Les architectures NUMA (*Non Uniform Memory Access*) sont organisées en un ensemble de nœuds reliés par un réseau d'interconnexion à haut débit. Chaque nœud est constitué d'un ou plusieurs processeurs, de leurs caches associés et d'une mémoire. Les processeurs de chaque nœud ont accès directement aux données stockées dans n'importe quel module mémoire. L'unité de transfert et de cohérence est la ligne mémoire contenant généralement une centaine d'octets. Les architectures NUMA se caractérisent par des temps d'accès non uniformes aux données, qui varient selon la localisation du bloc référencé. L'extensibilité de ce type d'architecture tient à l'extensibilité du réseau d'interconnexion et à la distribution de la mémoire dans les nœuds. On peut classer les architectures NUMA en trois catégories.

Les NCC-NUMA (*Non Cache-Coherent NUMA*)

Dans les NCC-NUMA, seul l'accès direct à des données distantes est géré par le matériel. La gestion de la cohérence des caches relève de la responsabilité du compilateur ou du

tel-00434053, version 1 - 20 Nov 2009

programmeur. Les premières architectures NUMA étaient de ce type [99].

Les CC-NUMA (*Cache-Coherent NUMA*)

Les CC-NUMA se caractérisent par un espace d'adressage unique et statique. Les adresses sont distribuées statiquement sur l'ensemble des nœuds du système. Dans les CC-NUMA, l'accès aux données distantes et le protocole de cohérence des caches sont mis en œuvre par matériel. Le protocole de cohérence est fondé sur l'utilisation de répertoires. Chaque nœud possède un répertoire qui contient, pour chacune des lignes de sa mémoire, une entrée dans laquelle sont conservées les informations de cohérence. L'architecture Dash [77] est représentative de cette classe d'architecture. Parmi les architectures commerciales, on peut citer la machine T3D de Cray [96] et la machine STiNG [81] de Sequent.

Les COMA (*Cache Only Memory Architecture*)

Dans les architectures COMA, la mémoire de chaque nœud est utilisée comme un cache de grande taille pour les processeurs du nœud. Ainsi, chaque fois qu'un processeur référence une donnée présente dans la mémoire d'un nœud distant la donnée est copiée (par une migration ou réplication) dans la mémoire du nœud contenant le processeur à l'origine de la requête. Pour cette raison, la mémoire d'un nœud dans une architecture COMA est appelée *mémoire attractive*.

L'ensemble des mémoires attractives constitue la mémoire partagée distribuée dans une architecture COMA. Une ligne mémoire pouvant se trouver dans plusieurs mémoires attractives simultanément, un protocole de cohérence est mis en œuvre par matériel à ce niveau. La localisation d'une ligne mémoire dans une architecture COMA est délicate puisque qu'une ligne n'a pas de localisation fixe en mémoire. Dans les prototypes KSR1 [47] et DDM [53], elle est effectuée grâce à une organisation hiérarchique de l'architecture. Des répertoires présents à chaque niveau de la hiérarchie contiennent les informations de cohérence concernant les données stockées dans les mémoires du sous-système sous-jacent. L'inconvénient de cette solution est le coût important de la résolution des défauts qui nécessite le parcours de toute la hiérarchie des répertoires. Une approche intermédiaire, proposée pour l'architecture COMA-F [117] permet de pallier cet inconvénient. Cette architecture non hiérarchique repose sur des répertoires statiquement distribués pour la localisation des lignes. Contrairement à une architecture CC-NUMA, le nœud gestionnaire d'une ligne ne possède pas obligatoirement un exemplaire de cette ligne dans sa mémoire.

Le protocole de cohérence d'une architecture COMA doit assurer l'existence d'au moins une copie de chaque ligne mémoire dans l'architecture car les mémoires attractives sont gérées par matériel indépendamment du système d'exploitation. A cet effet, les COMAs disposent d'un mécanisme d'exportation de données, appelé injection, qui permet lorsque le dernier exemplaire d'une ligne doit être remplacé de le faire migrer (injecter) dans la mémoire d'un autre nœud.

4.2.2 Les systèmes à mémoire virtuelle partagée

Un système à MVP met en œuvre une mémoire partagée sur des architectures à communication par message telles les architectures parallèles à mémoire distribuée (IPSC/2 [54]

tel-00434053, version 1 - 20 Nov 2009

tel-00434053, version 1 - 20 Nov 2009

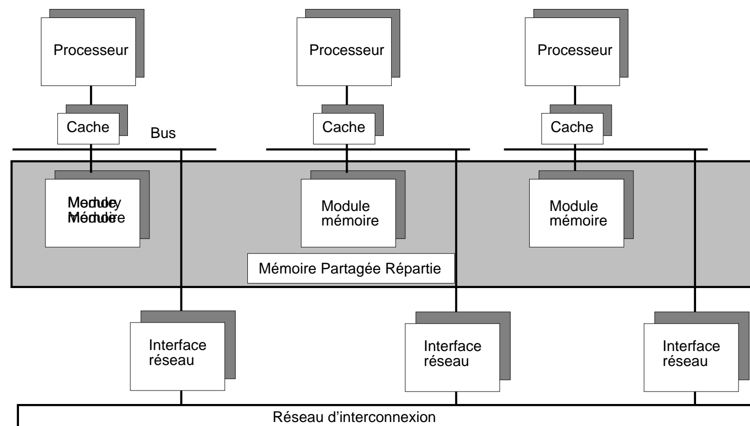


FIG. 4.1 – *Architecture à mémoire distribuée cohérente*

ou la machine Paragon [55]) ou des réseaux de stations de travail. Dans les systèmes à MVP, la mémoire partagée est mise en œuvre de façon essentiellement logicielle au-dessus de l'ensemble des mémoires locales des sites de l'architecture. Les MVP, qui ont été introduites il y a une dizaine d'années, ont donné le jour à de nombreux prototypes académiques Ivy [78], Munin [25], KOAN [71] ainsi qu'à quelques produits commerciaux comme TreadMarks [4].

Les différentes MVP se distinguent par la structure et le grain de leur unité de partage. Le plus souvent, la page (d'une taille de l'ordre de quelques kilo-octets) constitue l'unité de partage et de cohérence. Dans ce cas, la mise en œuvre de la MVP s'appuie sur les mécanismes de translation d'adresse du système d'exploitation. Cette approche est commune à Ivy [78], KOAN [71], Myoan [23] et Mirage [45]. Dans d'autres systèmes tels que Munin [25] et Orca [8], ce sont les objets qui constituent l'unité de partage. Dans Linda [24], une approche langage est adoptée.

Comme dans les COMAs, la gestion de la cohérence est effectuée au niveau des mémoires. Les protocoles de cohérence privilégiés dans les systèmes à MVP sont à invalidation sur écriture et reposent sur l'utilisation de répertoires. Les plus utilisés sont ceux proposés dans Ivy [78] à base de répertoire distribué statiquement ou de répertoire distribué dynamiquement. Par ailleurs, dans le domaine des MVP, beaucoup de travaux portent sur les modèles de cohérence affaiblie afin de diminuer le nombre de communications et le problème du faux partage [27, 36, 135]. Munin [25], TreadMarks [4], Midway [135] et DiSOM [93] mettent en œuvre des modèles de cohérence faible.

Il apparaît que les systèmes à MVP et les COMAs ont un fonctionnement similaire même si les mécanismes mettant en œuvre la mémoire partagée distribuée sont essentiellement logiciels pour les premiers et purement matériel pour les secondes. Nous regroupons ces deux architectures sous le terme d'architecture à mémoire distribuée cohérente (voir figure 4.1). Une telle architecture met en œuvre la cohérence des données au niveau des mémoires. Un bloc mémoire n'a pas de localisation physique fixe et peut être migré ou répliqué selon les accès des processeurs. L'étude présentée ci-après dans le paragraphe 4.5 s'adresse à ce

tel-00434053, version 1 - 20 Nov 2009

modèle d'architecture.

4.3 Recouvrement arrière dans les architectures à mémoire distribuée cohérente

Dans les architectures extensibles à mémoire distribuée cohérente la technique du recouvrement arrière est la plus utilisée pour le recouvrement d'erreur. Nous mettons l'accent dans cette partie sur les propositions relatives aux systèmes à mémoire virtuelle partagée recouvrable car il n'y a pas eu de travaux menés dans le cadre d'architectures COMA tolérantes aux fautes. Dans le paragraphe 4.3.1, nous présentons les stratégies optimistes et pessimistes mises en œuvre dans les architectures à mémoire partagée cohérente pour garantir la cohérence des points de récupération de processus communicants. Nous étudions les techniques de stockage des données de récupération dans le paragraphe 4.3.2. Enfin, nous abordons la gestion des informations de cohérence en cas de défaillance dans le paragraphe 4.3.3. Une synthèse est présentée dans le paragraphe 4.3.4.

4.3.1 Garantie d'un état global cohérent

Il est possible d'assurer un état global cohérent au moment de l'établissement d'un point de récupération, il s'agit alors d'une stratégie pessimiste, ou à la reprise après une défaillance à partir de points de récupération indépendants, on parle alors de stratégie optimiste.

Stratégies optimistes

Les approches optimistes, également appelées non planifiées ou asynchrones, se caractérisent par le fait qu'elles considèrent l'occurrence d'une défaillance comme un événement rare. Elles tendent à minimiser le coût introduit en fonctionnement normal par l'établissement de points de récupération. Les différents processus établissent des points de récupération de manière complètement indépendante. Il est alors nécessaire de coordonner les processus au moment du retour arrière afin de construire une ligne de récupération à partir des points de récupération sauvegardés. Généralement, l'historique des communications entre processus est utilisée par le système pour restituer un état global cohérent [64]. Cette approche est essentiellement utilisée dans le cadre des systèmes distribués à échange de messages.

Reconnue pour son efficacité à l'établissement d'un point de récupération, cette méthode présente néanmoins plusieurs inconvénients. Ainsi, il est nécessaire de sauvegarder plusieurs points de récupération par processus. En effet, étant donné que les points de récupération sont sauvés périodiquement par chaque processus sans précaution par rapport à l'état global cohérent, les différents points de récupération sont conservés pour le cas de propagation au moment du retour arrière. Ainsi, la place mémoire utilisée pour stocker ces points de récupération peut s'accroître rapidement. En outre, cette approche est sujette à l'*effet domino* [102] qui se manifeste lorsque le premier état global cohérent détecté d'un système n'est autre que l'état initial.

Réduction du nombre de points de récupération par processus La *journalisation des messages* permet de limiter le nombre de points de récupération sauvegardés pour chaque processus et d'éviter l'effet domino. Elle consiste à ne sauvegarder qu'un seul point de récupération par processus et à enregistrer les différents messages entre chaque sauvegarde de point de récupération. En cas de défaillance, ce point de récupération est restauré et les messages enregistrés sont *rejoués* par le système. Cette technique a été largement utilisée dans les systèmes distribués comme dans Auragen [19] ou Publishing [101], dans [118, 64, 128, 63, 134] et Manetho [43].

La clause de déterminisme des exécutions nécessaire afin que la journalisation des messages puisse être utilisée sans entraîner de décalage par rapport à une exécution réelle, rend cette technique particulièrement difficile à mettre en œuvre dans les systèmes à MVP. En effet, la plupart des programmes parallèles sont indéterministes et l'ordre des accès mémoire peut varier d'une exécution à une autre.

Dans les systèmes à mémoire virtuelle partagée, le nombre de messages est particulièrement important car la résolution de défauts de page entraîne l'échange de plusieurs messages. La taille du journal peut rapidement devenir très importante et occuper une place mémoire considérable. Il existe néanmoins des approches optimistes pour les MVP recouvrables [61, 106, 120]. Par exemple, dans [61], les différents processus établissent des points de récupération à intervalles fixes, le même intervalle étant appliqué à chaque processus. Ce système postule que les processus établissent un point de récupération approximativement au même moment (simulation d'une horloge globale) créant ainsi avec une forte probabilité un état global cohérent.

De manière générale, le problème du non déterminisme des accès à la mémoire partagée est résolu en journalisant, non pas les messages entres sites, mais le contenu des pages. En effet, les communications susceptibles de provoquer des dépendances dans une MVP sont les accès à des pages partagées. Dans [106], le contenu des pages est sauvegardé. En cas de défaillance, les accès mémoire sont à nouveau effectués à partir des informations contenues dans le journal. Des estampilles associées aux pages lors de chaque accès permettent de retrouver l'ordre chronologique des accès à la mémoire. Outre les accès aux données, l'approche adoptée dans [120] sauvegarde également les invalidations. Cette approche permet de limiter la taille du journal. Seul le site défaillant effectue un retour arrière et utilise son journal pour effectuer à nouveau les accès partagés. Les défauts de pages sont alors résolus à partir du journal sans solliciter les autres sites, qui sont bloqués jusqu'à l'expiration du journal.

Stratégies pessimistes

Les stratégies pessimistes (également dites synchrones ou planifiées) sont prédominantes dans les systèmes à mémoire virtuelle partagée. L'avantage principal d'une stratégie pessimiste est qu'il n'est nécessaire de sauvegarder qu'un seul point de récupération par processus. Ceci limite la quantité de mémoire utilisée par les données de récupération. Cet élément est crucial dans les architectures extensibles à mémoire partagée cohérente car les ensembles de travail des processus ont, pour la plupart, une taille relativement importante. Afin de limiter la quantité de données à sauvegarder à chaque point de récupération, il est possible de ne sauvegarder que les données modifiées entre deux points de récupération

tel-00434053, version 1 - 20 Nov 2009

tel-00434053, version 1 - 20 Nov 2009

[22, 131, 57]. On parle alors de *point de récupération incrémental*.

Les stratégies présentées pour les architectures multiprocesseurs à mémoire partagée dans le paragraphe 3.4.2 ont été aussi étudiées dans le cadre des systèmes à mémoire virtuelle partagée.

Synchronisation gloable Dans un système utilisant une technique de synchronisation globale [124], tous les processus sont synchronisés de manière à établir simultanément un point de récupération. De nombreuses MVP reposent sur une telle synchronisation [26, 58]. Un atout majeur de cette technique dans ce cadre est qu'aucune perturbation n'est à déplorer pendant les phases de calcul. Les interactions entre nœuds du système provoqués par la tolérance aux fautes ont lieu pendant les phases d'établissement de points de récupération. En particulier, les communications entre les différentes phases de calcul ne nécessitent pas d'être enregistrées. Le coût du mécanisme de tolérance aux fautes peut alors être limité par un intervalle d'établissement des points de récupération judicieusement choisi. L'inconvénient de cette méthode est le coût élevé qu'entraîne la coordination des processus au moment de l'établissement d'un point de récupération. Une solution pour limiter ce temps est de profiter des barrières de synchronisation des applications scientifiques [22]. Cette technique permet d'y intégrer le temps de synchronisation lié à l'établissement des points de récupération.

Synchronisation des communications et des points de récupération Dans bon nombre de MVP recouvrables, un processus établit un point de récupération avant de créer une dépendance, c'est-à-dire juste avant de fournir une donnée modifiée à un autre processus [119, 123, 131]. Établir un point de récupération à chaque communication engendre un surcoût souvent inacceptable dans le cas des systèmes à MVP dans lesquels les défauts provoquent de nombreux messages pour la transmission des pages et la gestion de la cohérence. En outre, en cas de faux partage, la fréquence des points de récupération peut s'en trouver extrêmement accrue. Cependant, il est également possible de profiter d'un modèle de cohérence relâchée, mis en œuvre à des fins de performance dans une MVP, pour diminuer le nombre de points de récupération et de processus simultanément concernés. Ainsi, dans [60], un processus établit un point de récupération quand une variable de synchronisation est lue par un autre processus.

Coordination dynamique Le principe de la coordination dynamique est d'enregistrer les communications entre processus au cours de l'exécution. Le système tient à jour pour chaque nœud l'ensemble des nœuds avec lesquels il a interagit depuis le dernier point de récupération. Ainsi, seuls les processus qui en sont dépendants sont simultanément sollicités pour établir un point de récupération ou effectuer un retour arrière. Une stratégie de coordination dynamique d'établissement de points de récupération est mise en œuvre dans la MVP proposée dans [57]. Un bit **dsc** (*dirty since checkpoint*) associé à une page indique si elle a été modifiée depuis le dernier point de récupération. L'ensemble des processus interdépendants est construit en considérant les transferts de pages dont le bit **dsc** est à 1.

tel-00434053, version 1 - 20 Nov 2009

4.3.2 Stockage des données de récupération

Le support de stockage des données de récupération doit garantir les *propriétés de stabilité*. Un tel support est appelé un *support stable*. Dans les systèmes à mémoire virtuelle partagée le support stable est réalisé en utilisant soit les disques, soit les mémoires vives, soit une combinaison des deux. Les différentes approches proposées sont présentées dans la suite de ce paragraphe.

Stockage sur disque

La plupart des MVP recouvrables reposent sur l'utilisation de disques [22, 57, 123, 131] pour stocker les données de récupération. Dans [22], le support stable est formé par un système de fichiers reposant sur la technologie RAID, accessible depuis n'importe quel site par l'intermédiaire du réseau d'interconnexion. Les hypothèses de fiabilité du réseau d'interconnexion et des disques garantissent les propriétés d'accessibilité et de permanence. La mise à jour atomique est assurée par un protocole de validation à deux phases.

La technique de pagination reposant sur les *pages jumelles* proposée dans [131] offre une mise en œuvre simple et efficace du retour arrière. La sauvegarde d'une page de la mémoire partagée, lors de l'établissement d'un point de récupération, est assurée par son propriétaire. Les données de récupération sont sauvées sur le disque et le système ne tolère pas la défaillance du serveur de disque. Pour chaque page modifiable, deux pages physiques contiguës sont allouées sur le disque. Les pages sur le disque peuvent être dans l'un des quatre états suivants : *working* (modifiée depuis le dernier point de récupération), *invalide*, *out of date* (appartenant au point de récupération précédent) ou *checkpoint* (appartenant au point de récupération courant). L'intérêt de ces états est de permettre une restauration à la demande des données de récupération. Augmentée de compteurs de point de récupération et de retour arrière, cette approche permet de disposer d'un mécanisme de point de récupération incrémental : il devient en effet aisé d'identifier les données modifiées depuis le dernier point de récupération à l'aide des états et compteurs. La figure 4.2 représente deux pages jumelles allouées sur le disque correspondant à une même page de la mémoire partagée. Après une écriture, la page appartenant au point de récupération le plus récent (C) est conservée et la seconde est transformée en (W). L'établissement d'un point de récupération transforme la page C en O et la dernière version de la page est désormais une page de récupération. Lors d'un retour arrière, les pages appartenant au dernier point de récupération sont conservées et les autres invalidées. Ce mécanisme assure qu'il existe toujours un exemplaire de récupération de chaque page. Ceci ajouté à la fiabilité supposée des disques garantit la permanence et l'accessibilité des données de récupération.

L'utilisation des compteurs alliée à celle des états autorise qu'une page puisse exister en deux versions de récupération correspondant à deux points de récupération différents (*checkpoint* et *out of date*). Un retour arrière ne nécessite pas de transfert massif au moment de la reprise car le disque est capable de traiter les défauts de page en lecture et en écriture. Il connaît en effet la version correcte de la page grâce à la comparaison des compteurs et l'utilisation des états. Pages courantes et pages de récupération cohabitent dans la hiérarchie mémoire. Toutefois, elles sont facilement identifiables car leur état traduit leur statut. L'utilisation de disques pour le stockage des données de récupération simplifie leur *localisation* puisqu'elles y sont placées exclusivement et systématiquement.

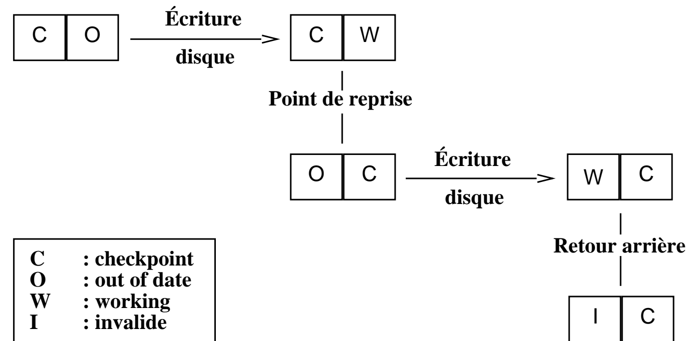


FIG. 4.2 – *Mécanisme des pages jumelles [131]*

La propriété de permanence est généralement assurée dans ces approches en posant des hypothèse fortes de fiabilité des disques ainsi que des serveurs qui en fournissent l'accès. En dépit de leur faible coût, ces solutions souffrent du faible débit et de la latence élevée des disques.

Stockage en mémoire vive

Lorsque les données de récupération sont sauvegardées dans les mémoires vives, sujettes aux défaillances, il est nécessaire de les répliquer dans la mémoire de nœuds distincts et mutuellement indépendants vis-à-vis des défaillances. Le nombre de répliques dépend du nombre de défaillances que l'on souhaite tolérer. Ceci permet d'assurer la propriété de permanence des données de récupération. Cette solution de stockage permet de ne pas limiter l'extensibilité puisque le nombre de modules mémoires augmente avec la taille du système. De plus, les modules mémoires bénéficient d'un temps d'accès beaucoup plus court que les disques. Ceci permet d'accélérer tant l'établissement des points de reprise que le retour arrière. Les MVP proposées dans [46, 119] et [129] stockent les données de récupération en mémoire vive.

Dans [129], les données qui ont été modifiées depuis le dernier point de reprise sont identifiées et estampillées au moment de l'établissement d'un point de récupération. Leur création effective se déroule en arrière-plan grâce à un mécanisme de *copie sur écriture*.

Dans [119], les données de récupération sont sauvegardées en mémoire principale. Quand une page est transmise d'un nœud N_1 à un nœud N_2, un exemplaire est conservé localement avant le transfert pour assurer la réplication de chaque page sur deux sites. Quand il s'agit d'une page modifiée, un point de récupération de toute la mémoire locale est établi : toutes les pages modifiées sont conservées localement sur le nœud N_1 et transmises au nœud N_2. L'inconvénient d'une telle solution est la fréquence élevée d'établissement de points de récupération.

La MVP proposée dans [21] associe à chaque page un nœud *espion* chargé de gérer l'un des deux exemplaires de récupération de la page, le second étant assuré par le propriétaire (différent du nœud espion). Dans l'un des algorithmes proposé, chaque nœud est le nœud espion d'un ensemble de pages déterminé dynamiquement.

tel-00434053, version 1 - 20 Nov 2009

tel-00434053, version 1 - 20 Nov 2009

La localisation des données de récupération est plus délicate lorsqu'elles sont sauvegardées en mémoire, car les données courantes y cohabitent avec les données de récupération. Il est donc indispensable de disposer d'un dispositif permettant de les identifier clairement.

Stockage mixte

Certaines MVP utilisent conjointement les disques et les mémoires vives pour stocker les points de récupération [93, 106, 120, 123]. C'est le cas en particulier des MVP reposant sur la journalisation des messages où le journal est provisoirement stocké en mémoire principale jusqu'à la migration d'une page sur un autre nœud. Il est ensuite recopié sur le disque. Toutefois, ces méthodes ne font qu'une utilisation provisoire de la mémoire vive pour limiter le nombre d'accès, souvent pénalisant, au disque.

La MVP proposée dans [129], outre le fait d'être l'une des rares MVP à utiliser les mémoires vives comme support stable, utilise le disque pour tolérer les double fautes et des fautes qualifiées de catastrophiques. Ce second mécanisme de point de récupération, dit *permanent* coexiste avec le premier décrit ci-dessus et assure le stockage des points de récupération appelés *persistants*. Ce second mécanisme peut être utilisé à la demande ou périodiquement. Il permet de limiter la place utilisée en mémoire par les données de récupération au détriment des données courantes. En effet, le pendant d'un point de récupération persistant ne subsiste pas en mémoire vive.

4.3.3 Gestion des informations de cohérence

Nous traitons dans ce paragraphe de la restauration des informations de cohérence en cas de défaillance.

Sauvegarde des informations de cohérence

Lorsque les informations de cohérence, c'est-à-dire les répertoires, sont sauvegardées au même titre que le reste des données, elles sont accessibles directement lors du retour arrière. Dans [123], une base de données distribuée stockée sur disque contient toutes les informations de cohérence. Toutefois lors de leur sauvegarde, il convient de s'assurer que leur contenu et celui des mémoires sont cohérents. Dans [94], les répertoires sont entièrement sauvegardés. L'état local d'un nœud qu'il est nécessaire de sauvegarder lors d'un point de reprise est formé des pages qu'il a modifiées depuis le dernier point de récupération ainsi que des informations de cohérence.

Reconstruction de répertoires à la demande

Il est également possible de ne pas sauvegarder les données de cohérence. Outre sa simplicité en fonctionnement normal, cette solution permet de limiter la quantité de données de récupération sauvegardées. Les répertoires sont alors reconstruits à la demande après le retour arrière. La cohérence entre répertoires et contenu des mémoires est assurée automatiquement. Un nœud défaillant est susceptible d'être *gestionnaire* ou *propriétaire* d'un ensemble de pages. Le gestionnaire d'une page est le nœud responsable de la gestion de l'entrée de répertoire relative à cette page. Le nœud propriétaire d'une page est le nœud

tel-00434053, version 1 - 20 Nov 2009

qui détient les droits d'écriture sur la page. La fonction de propriétaire étant dynamique par définition, il est aisé de déterminer un nouveau propriétaire à la demande. Au contraire, la fonction de gestionnaire, statique, doit être réattribuée explicitement au moment du retour arrière.

Dans [57], les répertoires permettant de stocker les informations de cohérence ne sont pas sauvegardés en support stable. Ils sont mis à jour lors du retour arrière afin de refléter le contenu réel de la mémoire physique. La mémoire du nœud défaillant est restaurée en plaçant toutes les données dans l'état *invalide*. Ces données sont accessibles depuis le dernier point de reprise stocké sur disque ou depuis les autres mémoires saines. Le répertoire est ainsi reconstruit à la demande.

Il en est de même dans [131], où, en cas de perte du gestionnaire d'une page, le nœud à l'origine de la requête diffuse une demande dans tout le système. Si le propriétaire existe, le répertoire est à nouveau mis à jour et la requête satisfaite. Dans le cas où aucun site ne revendique la propriété de la page demandée, le nœud à l'origine de la requête en devient le propriétaire.

Dans [61], où une approche optimiste est adoptée, tous les sites ne sont pas forcément concernés par un retour arrière. Pour chaque page, une estampille indique le dernier instant auquel un site était propriétaire d'une page. Elle est sauvée avec l'état de chaque processus au moment de l'établissement d'un point de récupération. En cas de défaillance, toutes les informations de cohérence sont perdues exceptées les estampilles de propriété appartenant au point de récupération. En cas de défaut de page, deux cas sont susceptibles de se présenter. Si le gestionnaire n'a pas été concerné par le retour arrière, l'information de propriété n'a pas été perdue et le défaut de page est résolu de manière traditionnelle. Si le gestionnaire a effectué un retour arrière, il réclame toutes les estampilles de propriété relative à la page demandée existant dans le système. En les comparant, il peut ainsi déterminer le propriétaire de la page. Le cas où l'information de propriété du gestionnaire est erronée à cause du retour arrière des autres nœuds est également traité. Lorsqu'un nœud reçoit une requête alors qu'il n'est plus propriétaire de la donnée, les estampilles de propriété sont à nouveau utilisées. Ainsi, très peu d'information nécessitent d'être sauvegardées pour reconstruire les répertoires à la demande après un retour arrière.

4.3.4 Synthèse

Le recouvrement arrière est une méthode particulièrement appropriée pour assurer la tolérance aux fautes dans un système à mémoire virtuelle partagée. En outre, cette technique est parfaitement adaptée à des applications scientifiques qui tolèrent une diminution provisoire de la performance au moment de la reconfiguration.

La stratégie optimiste de sauvegarde de points de récupération est porteuse intrinsèquement du risque de l'effet domino et est gourmande en place mémoire. La technique de journalisation, qui permet de pallier ces deux effets néfastes, repose essentiellement sur le déterminisme des exécutions, hypothèse beaucoup trop forte pour être envisagée dans le cadre d'une architecture extensible à mémoire distribuée cohérente. C'est pourquoi, la méthode la plus appropriée dans le cadre d'une telle architecture reste l'approche pessimiste qui permet de limiter la quantité des données de récupération mais nécessitent une syn-

tel-00434053, version 1 - 20 Nov 2009

<table>
<tr><th>Type de MVP</th><th>MVP de
Wu & al. [132]</th><th>MVP de
Wilkinson [129]</th><th>MVP de
Janakiraman & al. [57]</th></tr>
<tr><td rowspan="4">**Modèle de système**</td><td>réseau de stations de travail connectées par ethernet</td><td>système Arius</td><td>multiprocesseur à mémoire distribuée</td></tr>
<tr><td colspan="3">processeurs de type *silence sur défaillance*</td></tr>
<tr><td colspan="3">disques et réseaux supposés fiables</td></tr>
<tr><td>répertoire centralisé</td><td colspan="2">répertoire distribué</td></tr>
<tr><td rowspan="2">**Point de récupération**</td><td colspan="3">approche pessimiste</td></tr>
<tr><td>forcé à chaque communication</td><td colspan="2">coordination dynamique</td></tr>
<tr><td>**Support de stockage**</td><td>disques</td><td>mémoires ou disques</td><td>disques</td></tr>
<tr><td>**Évaluation**</td><td>simulation</td><td>mise en œuvre</td><td>simulation guidée par la trace</td></tr>
</table>

TAB. 4.1 – *Synthèse*

chronisation (globale ou non) des processus au moment de l'établissement d'un point de récupération. Son avantage majeur est qu'elle permet d'éviter l'effet domino. De plus, si la phase d'établissement des points de récupération est plus coûteuse que dans une approche optimiste, la phase de reconfiguration s'en trouve grandement accélérée.

Le support utilisé pour stocker les données de récupération doit assurer les propriétés de stabilité. Du support de stockage dépend largement le coût lié à l'intégration d'un mécanisme de tolérance aux fautes au sein d'une architecture extensible à mémoire distribuée cohérente. Les disques permettent de stocker les données de récupération de manière particulièrement économique dans le cadre d'une MVP mais peu performante. Le matériel spécifique possède l'avantage de l'efficacité au détriment d'un prix particulièrement élevé qui devient prohibitif lorsque le système considéré est de grande taille. L'alternative est de stocker les données de récupération en mémoire vive, cette méthode beaucoup moins répandue assure de bien meilleures performances mais complique l'établissement des points de récupération car il est nécessaire d'y gérer leur cohabitation avec les données courantes. Cette technique repose sur la *réplication* des données dans deux modules mémoires indépendant vis à vis des défaillances.

Les systèmes à MVP étudiés pêchent souvent par le peu d'évaluation proposée. Il existe en effet très peu des propositions réellement mises en œuvre. On peut toutefois citer les notables exceptions que représentent les MVP recouvrables proposées dans [129] et [34]. Les mesures de performances sont, dans les autres cas, obtenues soit par simulation, soit par modélisation (voir tableau 4.1).

Pour ce qui concerne les architectures extensibles à mémoire partagée (NUMA ou COMA), peu de travaux ont été réalisés en tolérance aux fautes. On note toutefois le système d'exploitation *Hive* [31] qui ne traite cependant pas du recouvrement arrière mais du confinement de l'erreur dans l'architecture FLASH.

4.4 Réplication pour l'efficacité et la disponibilité

Dans les architectures extensibles à mémoire partagée, la réplication de données est utilisée pour réduire les temps d'accès mémoire. L'utilisation de caches (exploitant la localité

spatiale et temporelle des accès des processeurs aux données) permet alors d'assurer cette réplication de façon automatique mais nécessite l'introduction de protocoles de cohérence pour maintenir à jour l'ensemble des répliques des blocs mémoire.

La réplication de données est aussi largement utilisée dans le domaine de la tolérance aux fautes. Ainsi une technique de récupération arrière est fondée sur la réplication de l'état mémoire courant d'un système lors de la sauvegarde d'un point de récupération. Elle peut également utiliser une réplication des données de récupération conservées pour leur assurer des propriétés de stabilité et limiter ainsi les hypothèses sur les fautes tolérées.

L'idée à la base de mes travaux a été de chercher à exploiter les copies de données existantes du fait du mécanisme de cache pour obtenir la redondance de données nécessaire à la mise en œuvre de la tolérance aux fautes. Les gains envisagés étaient d'une part d'éviter la création de nouvelles copies de données en mémoire pour les données de récupération pour faire un usage optimal de la ressource mémoire et d'autre part d'éviter des transferts de données sur le réseau et surtout vers des disques à latence élevée.

J'ai alors étudié les caractéristiques des différentes architectures extensibles à mémoire partagée en vue de la mise en œuvre d'un protocole de récupération arrière. J'ai constaté que le caractère statique des mémoires d'une architecture CC-NUMA ne permet pas d'envisager une mise en œuvre qui garantisse à la fois l'efficacité et la simplicité de la reconfiguration [13]. J'ai donc considéré pour mes travaux les architectures à mémoire distribuée cohérente (COMA et systèmes à mémoire virtuelle partagée) dans lesquelles l'absence de localisation physique fixe des données simplifie considérablement la reconfiguration.

La mise en œuvre d'une stratégie de retour arrière nécessite d'assurer la stabilité des données de récupération. Pour tolérer une faute simple, deux copies des données de récupération doivent exister. Compte tenu de l'indépendance vis à vis des défaillances des nœuds d'une architecture à mémoire distribuée cohérente, ces deux copies peuvent être sauvegardées dans la mémoire de deux nœuds distincts. Comme les données actives et de récupération sont stockées dans le même support et dans le but de tirer partie de ces dernières pour le calcul, j'ai proposé d'étendre le protocole de cohérence de manière à ce qu'il gère les deux types de données [88]. Le protocole ainsi défini est décrit dans le paragraphe suivant.

4.5 Extension du protocole de cohérence pour la mise en œuvre du recouvrement arrière

Nous considérons dans la suite de ce chapitre des architectures à mémoire distribuée cohérente qui mettent en œuvre un protocole de cohérence à invalidation sur écriture pour gérer les multiples exemplaires d'une donnée en mémoire. Par souci de simplification, nous considérons une architecture avec des nœuds monoprocesseurs. Nous considérons une technique coordonnée globale d'établissement de points de récupération où tous les processeurs se synchronisent pour établir un point de récupération. La sauvegarde des données de récupération est incrémentale c'est-à-dire que des données de récupération ne sont créées que pour les données qui ont été modifiées depuis le dernier point de récupération.

La mise en œuvre des principes évoqués dans le paragraphe précédent peut être réalisée en étendant le protocole de cohérence de l'architecture pour combiner de façon transparente

tel-00434053, version 1 - 20 Nov 2009

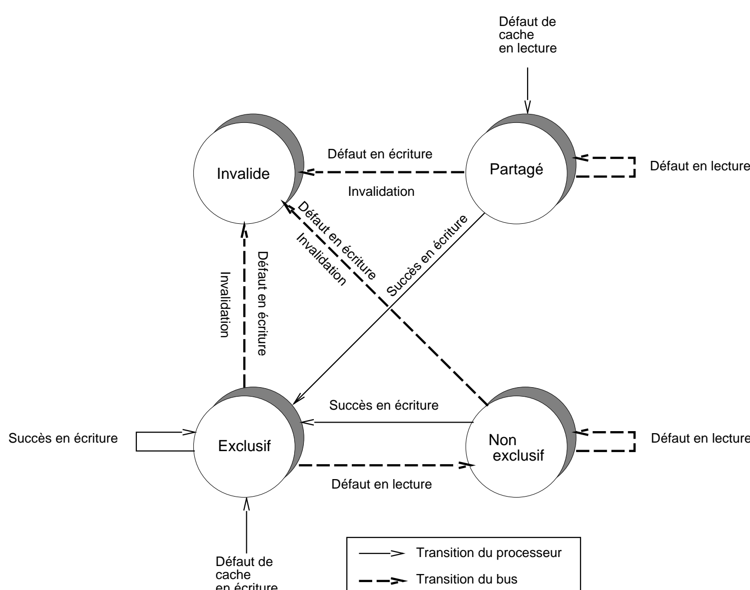


FIG. 4.3 – *Protocole de cohérence de base*

la gestion des données actives et de récupération. Pour identifier les données de récupération, deux nouveaux états sont ajoutés au protocole de cohérence de base présenté à la figure 4.3. L'état *partagé-PR* identifie les deux copies de récupération d'un bloc qui n'a pas été modifié depuis le dernier point de récupération. Une telle copie peut être lue par le processeur local et peut être utilisée pour servir les défauts en lecture. L'état *inv-PR* identifie les deux copies de récupération d'un bloc qui a été modifié depuis le dernier point de récupération. Une telle copie ne peut pas être accédée par les processeurs et est conservée à des fins de tolérance aux fautes. Par conséquent, les succès en lecture ou écriture sur une copie *inv-PR* sont traités comme des défauts.

Pour éviter toute violation de cohérence (propriétaires multiples), deux états *partagé-PR* différents (*partagé-PR1* et *partagé-PR2*) doivent être distingués de telle sorte que seule l'une des deux copies (*partagé-PR1*) peut délivrer les droits d'accès exclusifs au bloc. Comme les copies *inv-PR* sont susceptibles d'être restaurées en copies *partagé-PR*, deux états *inv-PR* sont également distingués. Dans la présentation du protocole, l'état *partagé-PR* (respectivement *inv-PR*) représente les états *partagé-PR1* et *partagé-PR2* (respectivement les états *inv-PR1* et *inv-PR2*).

Deux nouvelles transitions représentent l'établissement et la restauration d'un point de récupération. Le protocole résultant, présenté sur la figure 4.4 est appelé protocole de cohérence étendu (ECP[1]). Ce protocole assure qu'à tout instant, il existe pour chaque bloc mémoire soit deux copies dans l'état *partagé-PR*, soit deux copies dans l'état *inv-PR* dans deux mémoires distinctes.

Le protocole de cohérence étendu reste similaire au protocole standard. Nous ne dé-

1. ECP pour *Extended Coherence Protocol*.

tel-00434053, version 1 - 20 Nov 2009

tel-00434053, version 1 - 20 Nov 2009

Invalide PR
Écriture
Partagé PR
Succès en lecture
Succès en lecture
+ création d'un exemplaire invalide–PR
Succès en écriture
Si exemplaire choisi
Si exemplaire non choisi
Invalide
Défaut en écriture
Partagé
Succès en lecture
Défaut en écriture
Défaut en écriture
Succès en écriture
Succès en lecture
Exclusif
Succès en écriture
Non Exclusif
Défaut en lecture
Succès en lecture
Défaut en écriture
Défaut en lecture

Transition distante
Transition locale
Établissement d'un point de récupération
Retour arrière

FIG. 4.4 – *Protocole* ECP

taillons ici que les situations liées aux nouveaux états du protocole. Après l'établissement d'un point de récupération, il existe dans le système pour un bloc donné uniquement deux copies *partagé-PR* et éventuellement des copies *partagé*. Quand un défaut en lecture se produit, la copie *partagé-PR1* est utilisée pour servir la requête. Le nœud à l'origine de la demande reçoit une copie du bloc qu'il met dans l'état *partagé*. Une requête en écriture sur un bloc qui n'a pas été modifié depuis le dernier point de récupération est traité quasiment de la même façon que par le protocole standard. Les messages d'invalidation sont envoyés à tous les nœuds qui possèdent une copie du bloc et le nœud ayant la copie *partagé-PR1* envoie une copie du bloc au nœud demandeur. L'état des deux copies *partagé-PR* est changé en *inv-PR* et toutes les copies *partagé* sont invalidées. A la fin de l'opération, le nœud demandeur contient l'unique copie active du bloc dans l'état *exclusif*. L'architecture contient alors une copie active du bloc et exactement deux copies de récupération dans l'état *inv-PR*. A partir de cet instant, le protocole standard est utilisé pour toutes les requêtes concernant ce bloc et les copies *inv-PR* ne sont conservées que pour faire face à un éventuel retour arrière.

4.5.1 Etablissement d'un point de récupération

Un protocole classique de validation à deux phases [43] est utilisé pour établir un nouveau point de récupération. Tous les nœuds sont synchronisés pour l'exécution de ce protocole. Le but de la première phase, appelée phase de création, est de créer un nouveau point de récupération tandis que la seconde phase, appelée phase de validation, a pour objectif d'éliminer l'ancien point de récupération et de confirmer le nouveau.

Avant de commencer la phase de création, chaque nœud termine ses requêtes en cours. Pendant cette phase, deux copies des données de récupération doivent être créées pour les blocs qui ont été modifiés depuis le dernier point de récupération puisqu'une approche incrémentale est utilisée. L'état *pré-validé* est un état transitoire utilisé pour ditinguer les nouvelles données de récupération des anciennes. La première copie de récupération est obtenue en changeant simplement l'état des copies *exclusif* et *non-exclusif* en l'état *pré-validé*. La seconde résulte de la réplication du bloc dans l'état *pré-validé* dans la mémoire d'un autre nœud. Pour les blocs déjà répliqués (*ie* les blocs pour lesquels il existe une copie *non-exclusif* et au moins une copie *partagé*), une optimisation consiste à choisir une des copies *partagé* et à la transformer en copie de récupération (dans l'état *pré-validé*). Ceci évite un transfert de donnée sur le réseau et la création d'une copie supplémentaire du bloc mémoire.

Une fois que tous les nœuds ont terminé la phase de création, la phase de validation démarre sur chacun des nœuds. Cette phase est locale. Chaque nœud parcourt sa mémoire et change l'état de toutes les copies *inv-PR* en *invalide* et transforme toutes les copies qui sont dans l'état *pré-validé* en des copies *partagé-PR*. A la fin l'établissement d'un point de récupération, chaque bloc de la mémoire possède deux copies dans l'état *partagé-PR* et éventuellement une ou plusieurs copies dans l'état *partagé*.

Toute défaillance au cours de la première ou seconde phase de l'algorithme peut être traitée correctement. Pendant la phase de création, l'ancien point de récupération (composé de l'ensemble des copies *inv-PR* et *partagé-PR*) n'est pas altéré et peut donc être restauré en cas de défaillance. Au cours de la phase de validation, le nouveau point de récupération

tel-00434053, version 1 - 20 Nov 2009

tel-00434053, version 1 - 20 Nov 2009

(constitué de l'ensemble des copies *pré-validé* et *partagé-PR*) est complet et persistant puisque tous les blocs ont été répliqués au cours de la première phase. Comme la phase de validation est locale, toute défaillance survenant pendant son exécution peut être traitée à la fin de cette phase, *ie* comme si la défaillance s'était produite au cours d'une phase de calcul.

4.5.2 Restauration d'un point de récupération

Quand la défaillance d'un nœud a été détectée, le but de la phase de restauration est de restaurer le dernier point de récupération constitué de toutes les copies *partagé-PR* et *inv-PR* des blocs. Toutes les autres copies doivent être invalidées. Comme un point de récupération global est restauré, un message est diffusé à tous les nœuds pour les informer qu'une opération de restauration doit être effectuée. Chaque nœud parcourt sa mémoire et invalide toutes les copies actives des blocs (*ie* copies dans l'état *partagé*, *exclusif* ou *non-exclusif*) de même que toutes les copies *pré-validé*. Les copies *inv-PR* sont restaurées en *partagé-PR*. Aucune action ne doit être effectuée pour les copies *partagé-PR*. A la fin du retour arrière, seulement deux copies *partagé-PR* existent pour chaque bloc de la mémoire partagée.

Si la faute est une faute permanente, une reconfiguration de la mémoire doit aussi être effectuée. Elle consiste à dupliquer les blocs dont une des copies *partagé-PR* se trouve sur le nœud fautif de façon à ce que la propriété de permanence soit de nouveau satisfaite. A l'issue du retour arrière, chaque nœud vérifie pour chacune de ses copies *partagé-PR* si la seconde copie de récupération existe toujours. Si ce n'est pas le cas, une nouvelle copie *partagé-PR* est créée sur un des nœuds sains. L'identité du propriétaire est mise à jour dans le répertoire le cas échéant.

Le protocole de cohérence étendu a été vérifié à l'aide de la technique d'analyse d'accessibilité décrite dans [100]. Cette vérification a permis de montrer que les propriétés de cohérence du protocole ECP et les propriétés de stabilité des données de récupération [50].

4.5.3 Avantages

Les avantages du protocole sont multiples. Il permet de mettre en œuvre une technique de récupération arrière de façon simple et efficace tout en limitant les hypothèses sur les fautes tolérées. La simplicité de l'approche réside essentiellement dans l'utilisation des mémoires locales standard sans mécanisme particulier de tolérance aux fautes, ce qui minimise le développement matériel. L'efficacité est assurée par l'utilisation d'un schéma mixte ne contraignant pas le nombre d'établissements de points de récupération et surtout par l'utilisation des mémoires et du réseau d'interconnexion de l'architecture pour le stockage et la réplication des données de récupération.

Un autre avantage de cette approche est de pouvoir exploiter la réplication de donnée existante dans les mémoires locales pour simplifier la création des données de récupération. Ainsi, lors de l'établissement d'un point de récupération, les blocs actifs déjà répliqués sur plusieurs nœuds ne nécessitent pas de transfert de données pour créer de nouvelles copies de récupération. En outre, les données de récupération étant stockées dans les mémoires, notre protocole autorise leur consultation en lecture par les processeurs aussi longtemps

tel-00434053, version 1 - 20 Nov 2009

que le bloc correspondant n'est pas modifié dans la région de récupération courante. Pour un bloc mémoire, la création de véritables copies de récupération non consultables est repoussée jusqu'à la première modification.

Enfin, l'absence de localisation fixe des données de récupération rend aisée la reconfiguration de l'architecture.

4.5.4 Contrôle de la réplication des données de récupération

Lorsqu'une donnée est en exemplaire unique au moment de l'établissement d'un point de récupération, le protocole ECP entraîne la création d'une nouvelle copie de cette donnée. Cependant, la localisation de cette nouvelle copie n'est pas dictée par le protocole et peut être un site quelconque distinct du site sur lequel est situé le premier exemplaire de la donnée. Nous avons vu que les données de récupération peuvent être accédées par les processeurs pour leur calcul. Ceci est un facteur d'efficacité dans la mesure où la réplication de données effectuée lors de la création d'un point de récupération permet d'anticiper dans certains cas les accès des processeurs. Dans le cadre de l'utilisation du protocole ECP dans un système à MVP, nous avons proposé un mécanisme permettant de contrôler le lieu de création des données de récupération de façon à favoriser leur utilisation par les processeurs et de ce fait améliorer les performances globales du système. Ceci nous a amené à définir le concept d'affinité mémoire.

> **Définition (Affinité mémoire)** *Il existe de l'affinité mémoire entre un module mémoire M d'un nœud N et un processus p si M contient au moins une page référencée durant l'exécution de p sur N. Le degré d'affinité dépend du nombre de pages contenues dans M et accédées par p.*

Le concept d'affinité mémoire est issu de celui d'affinité de cache [116] utilisé dans l'ordonnancement des processus dans les multiprocesseurs à mémoire partagée. Cette stratégie consiste, au moment du réordonnancement des processus bloqués ou préemptés, à mesurer l'affinité entre les différents caches et le processus et à le réordonnancer de préférence sur un processeur dont le cache contient déjà un sous-ensemble des données nécessaires à son exécution. Dans le contexte d'un système à MVP implémentant le protocole ECP, l'affinité des processus avec les mémoires n'est pas mesurée mais créée sans surcoût au moment de l'établissement des points de récupération. L'efficacité d'un tel mécanisme dépend pour une large part du degré de localité processeur[2] des applications considérées. Il est toutefois nécessaire de disposer d'informations pour contrôler la réplication. La seule information disponible dynamiquement concernant les pages de la mémoires partagée, relève du comportement passé de l'application et plus précisément de l'histoire des références des pages. Nous considérons, en vertu des caractéristiques de localité processeur, qu'une page a une plus forte probabilité d'être référencée dans le futur par un nœud l'ayant déjà utilisée par le passé que par un nœud ne l'ayant jamais référencée.

La méthode que nous proposons pour collecter l'information permettant de créer l'affinité, repose sur l'historique récent d'une page. Ainsi, les exemplaires *partagé-PR2* sont répliqués de préférence sur des nœuds ayant déjà référencé la page correspondante dans un

2. Localité spatiale ou temporelle associée à un processeur en univers multiprocesseur.

tel-00434053, version 1 - 20 Nov 2009

passé récent. Le concept d'exploitation des pages déjà répliquées n'est bien entendu pas remis en cause par cette nouvelle stratégie. Il en représente un cas particulier.

À l'établissement d'un point de récupération, le nœud sur lequel est créée la seconde copie de récupération est déterminé en tenant compte des informations de l'historique. Cette méthode revient à effectuer un préchargement de page et au contraire de bon nombre de mécanismes d'affinité, est très peu coûteuse. En effet, l'information nécessaire pour contrôler la réplication est immédiate à collecter car elle est déjà disponible pour la gestion de la cohérence.

4.6 Mise en œuvre du protocole ECP

Nous donnons dans le paragraphe 4.6.1 quelques éléments de mise en œuvre du protocole dans un COMA. Dans le paragraphe 4.6.2, nous décrivons ICARE, un prototype de système à MVP tolérant aux fautes fondé sur le protocole ECP que nous mis en œuvre sur un réseau de stations de travail.

4.6.1 Mise en œuvre dans un COMA

Dans une architecture COMA non hiérarchique similaire à l'architecture COMA-F, toutes les modifications matérielles introduites par la mise en œuvre du protocole ECP concernent la mémoire attractive et son contrôleur qui met en œuvre le protocole de cohérence. Le processeur et son cache n'ont pas à être modifiés.

Dans la mémoire attractive, des bits supplémentaires doivent être prévus pour coder les nouveaux états du protocole ECP. Pour ce qui concerne les accès du processeur local, le contrôleur de la mémoire attractive doit permettre les accès en lecture sur les copies *partagé-PR* et générer une injection en cas d'accès en écriture sur une copie *partagé-PR* et d'accès quelconque à une copie *inv-PR*. D'autres modifications sont relatives aux requêtes distantes en lecture ou écriture sur la copie *partagé-PR1*, aux nouvelles injections des copies de récupération et aux nouveaux algorithmes pour l'établissement et la restauration d'un point de récupération.

Les requêtes en lecture et écriture sur la copie *partagé-PR1* sont traitées de manière analogue aux requêtes en lecture et écriture sur une copie *non-exclusif*. La seule différence est que dans le cas d'une requête en écriture, la copie *partagé-PR1* passe dans l'état *inv-PR1* et une invalidation est envoyée à la copie *partagé-PR2* qui passe dans l'état *inv-PR2*.

Cause	Etat de la copie locale	Action
Remplacement	*Partagé-PR*	Injection
Remplacement	*Inv-PR*	Injection
Accès en lecture	*Inv-PR*	Injection + défaut en lecture
Accès en écriture	*Inv-PR*	Injection + défaut en écriture
Accès en écriture	*Partagé-PR*	Injection + défaut en écriture

TAB. 4.2 – *Nouvelles injections introduites par le protocole* ECP

Dans un COMA standard, les injections ne sont utilisées que lorsqu'une copie dans l'état *exclusif* ou *non-exclusif* doit être remplacée. Le protocole ECP introduit cinq nouveaux cas d'injections présentés dans le tableau 4.2. Ces injections sont nécessaires pour assurer l'existence à tout moment de deux copies de récupération. La gestion de ces injections entraîne les modifications les plus complexes du protocole de cohérence standard. Comme dans les COMAs traditionnels, une architecture fondée sur le protocole ECP doit assurer qu'une copie injectée trouve toujours une place dans l'ensemble des mémoires attractives. Dans la machine KSR1, ce problème est résolu en allouant une page irremplaçable pour chaque page allouée dans l'architecture [47]. Un problème similaire se pose avec les copies de récupération lors de l'établissement d'un point de récupération. Quatre copies sont nécessaires pendant la phase de création. Une allocation statique de quatre pages irremplaçables au lieu d'une seule peut être effectuée pour garantir qu'il y a toujours suffisamment de mémoire lors de l'établissement d'un nouveau point de récupération.

La mise en œuvre des algorithmes de création/validation et de restauration d'un point de récupération est assez simple. La réplication d'une ligne mémoire pour l'établissement d'un point de récupération ne nécessite aucune nouvelle fonctionnalité puisque ces requêtes s'apparentent à des injections. La seule différence est que la copie de la ligne mémoire injectée n'est pas remplacée dans la mémoire attractive du nœud réalisant l'injection.

4.6.2 Mise en œuvre dans un système à MVP

Nous avons mis en œuvre sur la plate-forme ASTROLAB un système à mémoire virtuelle partagée tolérant aux fautes fondé sur le protocole ECP, appelé ICARE [69]. La plate-forme ASTROLAB est constituée d'un ensemble de PC/Pentium (133 MHz) exécutant le micro-noyau CHORUS [107] interconnectés par un réseau local ATM (155Mb/s). ICARE est mis en œuvre comme une extension d'un système à MVP fondé sur un protocole de cohérence à base de répertoires statiquement distribués. Le protocole ECP est entièrement mis en œuvre par logiciel. Par conséquent, l'ajout de nouveaux états est très simple.

Outre une mémoire virtuelle partagée recouvrable, ICARE met en œuvre un système de reprise de processus permettant de poursuivre l'exécution d'une application en dépit de la défaillance d'une site. Dans ICARE, les données privées relatives à l'état des processus ne sont pas placées en mémoire partagée. Il est donc nécessaire de sauvegarder également un *point de récupération privé* d'un processus contenant sa pile, son contexte et son segment de données afin de pouvoir en reprendre l'exécution en cas de retour arrière. Le code de l'application ne nécessite pas d'être sauvegardé car nous considérons des applications parallélisées par distribution du contrôle dans lesquelles tous les processeurs exécutent le même code. Le même algorithme à deux phases que celui des données partagées est utilisé. Lors de l'établissement d'un point de récupération, chaque nœud sauvegarde localement l'ensemble des données de récupération privées de ses processus, baptisé *point de reprise privé local* et en réplique un exemplaire, appelé *point de reprise privé distant*, sur un nœud distant.

ICARE est composé de deux acteurs CHORUS, l'un utilisateur, l'autre superviseur. Le besoin d'un acteur superviseur, appelé *serveur de contexte*, est lié à la reprise des processus. En effet, la primitive qui permet de sauvegarder et restaurer le contexte d'une activité ne peut être appelée que par un acteur superviseur. Ce serveur de contexte est chargé de

tel-00434053, version 1 - 20 Nov 2009

la sauvegarde des registres et de la pile pour chaque point de reprise privé des activités de l'application. Le second acteur d'ICARE est responsable de la gestion de la MVP recouvrable, c'est-à-dire de la résolution des défauts de pages, des communications avec les autres nœuds et avec le noyau ainsi que de l'établissement périodique de points de récupération et du retour arrière le cas échéant. Au sein de l'acteur utilisateur d'ICARE, outre l'activité de l'application, quatre autres activités existent (voir figure 4.5) : **l'activité noyau** qui gère les interactions avec le noyau, **l'activité de communication** qui gère les communications avec les autres nœuds ; **l'activité de tolérance aux fautes** (TAF) qui est chargée de l'établissement périodique de points de récupération et de leur traitement ; **l'activité de retour arrière** (RA) est chargée du retour arrière sur chaque nœud en cas de défaillance.

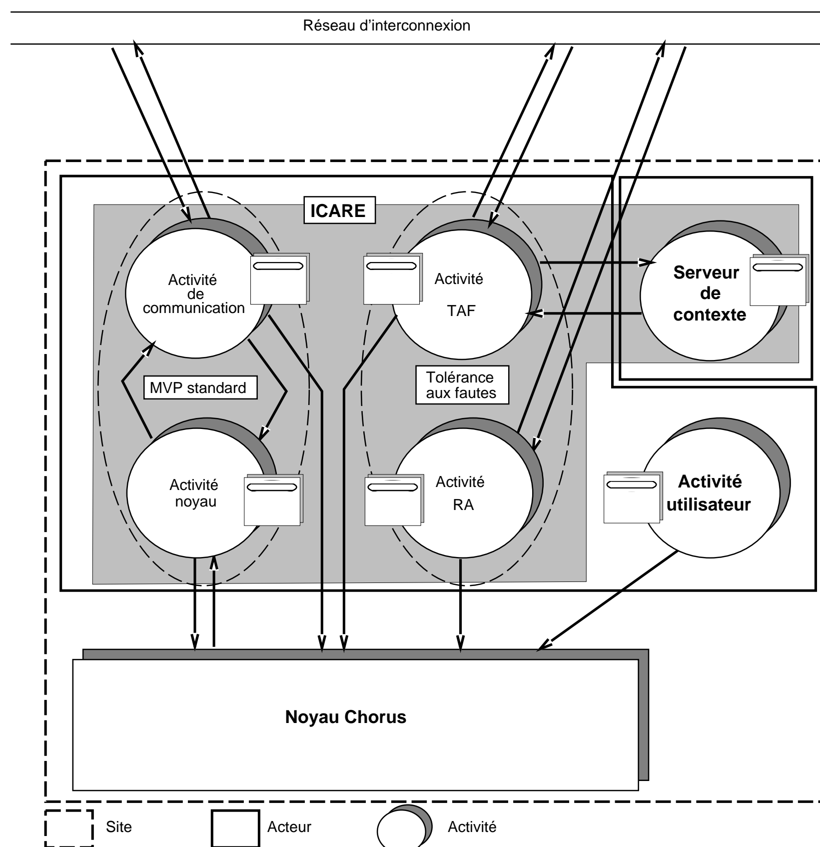


FIG. 4.5 – *Architecture logicielle d'*ICARE

tel-00434053, version 1 - 20 Nov 2009

4.7 Evaluation de performance

Afin de valider notre approche, nous avons évalué les performances de deux architectures : un COMA non hiérarchique et un système à MVP. Le protocole ECP a été évalué pour la machine COMA et le système à MVP décrits dans les paragraphes précédents. Les résultats de performance ont été obtenus par simulation pour le COMA et par des mesures effectuées sur le prototype ICARE pour le système à MVP.

Pour évaluer le protocole ECP dans un COMA, nous avons développé un simulateur d'architectures extensibles à mémoire partagée. Celui-ci utilise le noyau de simulation SPAM [51] qui met en œuvre de façon efficace une technique de *simulation coordonnée à l'exécution* (*Execution Driven Simulation*) [35, 40] qui permet d'obtenir des résultats de simulation réalistes. Les caractéristiques de l'architecture simulée s'inspirent de l'architecture KSR1. Dans notre évaluation, nous avons utilisé quatre applications de SPLASH [115] : *Barnes-Hut*, *Cholesky*, *Mp3d* et *Water*. Ces applications parallèles fournissent un panel de comportement vis à vis de la mémoire partagée qui donne à notre évaluation toute sa validité. Dans les simulations, nous avons fait varier la fréquence d'établissement des points de récupération de 5 à 400 points de récupération par seconde.

Nous avons évalué ICARE avec trois applications exécutées sur quatre PC reliés par un réseau local ATM [3] : *Matmul*, MGS et *Radix*.

Le lecteur intéressé trouvera une étude détaillée des performances dans [49, 50, 68]. Ici, je m'attache à montrer que le protocole ECP est à la fois efficace et extensible dans les architectures à mémoire distribuée cohérente.

4.7.1 Efficacité

Deux facteurs permettent d'évaluer l'efficacité du protocole ECP : le surcoût temporel dû à l'établissement périodique de points de récupération et celui dû à une modification du comportement de la mémoire du fait des changements d'état des blocs mémoire. Pour mesurer les surcoûts temporels, nous avons comparé les résultats obtenus avec une architecture implémentant le protocole ECP et ceux de la même architecture mettant en œuvre le protocole de cohérence standard de base.

Coût d'établissement des points de récupération

La dégradation de performance observée pour les différentes applications varie en fonction de la fréquence d'établissement d'un point de récupération. Le surcoût le plus important est lié à la phase de création des données de récupération et dépend essentiellement de la quantité de données de récupération à transférer lors des sauvegardes des points de récupération. Dans l'architecture COMA, selon les applications et les fréquences de sauvegarde de points de récupération, ce surcoût varie de 1 à 2 % dans le meilleur des cas, pour atteindre plus de 15% dans le pire des cas.

Le surcoût de la phase de création dépend aussi des caractéristiques des applications. Ainsi, les applications qui possèdent un taux d'écriture important exhibent une grande

3. La petite taille de la configuration est due au fait que seules quatre stations de travail de la plate-forme ASTROLAB étaient équipées de carte d'interface ATM au moment où les mesures ont été effectuées.

tel-00434053, version 1 - 20 Nov 2009

quantité de données répliquées. En outre, plus l'espace de travail d'une application est important, plus elle a la possibilité de modifier de lignes mémoires entre deux points de récupération. L'utilisation du réseau d'interconnexion et des mémoires attractives pour le transfert et le stockage des données de récupération dans un COMA est en grande partie responsable de la faible dégradation de performance engendrée par la phase de création malgré les fréquences élevées de sauvegarde des points de récupération. Par ailleurs, l'exploitation par le protocole ECP de la réplication de données existante permet de limiter les transferts de données. Pour certaines applications, nous avons constaté que plus de 50% des lignes à répliquer existaient déjà.

Le surcoût de la phase de validation ne dépend que du nombre de pages allouées sur les nœuds de l'architecture. La dégradation qu'elle engendre est donc directement proportionnelle à la taille de l'espace de travail et à la fréquence de sauvegarde des points de récupération. Il est possible de diminuer voire d'annuler le coût de cette phase en utilisant des techniques de compteurs de validation.

Coût lié à la modification du comportement mémoire

COMA L'introduction des données de récupération dans les mémoires de l'architecture entraîne un surcoût qui se caractérise par deux effets : (i) une variation des taux de défauts de cache et de mémoire attractive, (ii) une création de nouvelles injections de lignes. Suivant les fréquences de sauvegarde des points de récupération, le surcoût mémoire varie de 10% à moins de 2%, il reste donc limité. à chaque sauvegarde d'un point de récupération, les données modifiées des caches sont recopiées en mémoire. Le nombre de défauts de cache primaire en lecture reste stable alors que celui des défauts en écriture augmente légèrement avec la fréquence de sauvegarde des points de récupération.

La variation des taux de défauts de mémoire attractive est négligeable et ceci quelle que soit la fréquence de sauvegarde des points de récupération. La constance du nombre de défauts en lecture confirme un des avantages du protocole qui autorise la lecture des données de récupération non modifiées. Dans certains cas, le nombre de défauts peut même diminuer avec l'augmentation des fréquences des points de récupération si l'application tire parti de la réplication réalisée lors de la création des nouveaux points de récupération.

La dégradation de performance engendrée par le stockage des données de récupération dans les mémoires est en fait essentiellement dûe aux nouvelles injections de lignes. Le nombre global d'injection est faible. De plus, le nombre d'injections sur lecture varie peu avec la fréquence de sauvegarde des points de récupération. Ici aussi, ce comportement s'explique par le fait que le protocole ECP autorise la lecture des données de récupération non modifiées. Au contraire, le nombre d'injections sur écriture augmente de façon importante avec la fréquence de sauvegarde des points de récupération. La transformation fréquente de données actives modifiées en données de récupération explique ce comportement. Ce type d'injection constitue donc la principale cause du surcoût mémoire mesuré.

Icare L'introduction des données de récupération en mémoire entraîne trois types de changement de comportement mémoire : lectures sur des pages *partagé-PR2*, écritures sur des pages *partagé-PR2* et écritures sur des pages *partagé-PR1*. Les deux premiers cas ont trait à l'utilisation pour le calcul, de pages répliquées pour les besoins de la tolérance aux

tel-00434053, version 1 - 20 Nov 2009

fautes. Ils représentent un facteur de diminution du coût de la tolérance aux fautes. En revanche, la troisième situation représente l'inconvénient majeur de l'algorithme proposé et entraîne une dégradation de performance sur le fonctionnement normal.

Lectures sur des pages P-PR2 Ce type d'opération concerne les lectures effectuées sur des pages *créées* à l'établissement d'un point de récupération. Il permet d'anticiper des défauts de pages qui auraient eu lieu en fonctionnement normal, en utilisant des données répliquées à l'établissement d'un point de récupération. Ceci permet de réduire le coût de la tolérance aux fautes. En effet, une réplication à l'établissement d'un point de récupération est moins coûteuse que la résolution d'un défaut de page pour deux raisons. La première est que de nombreuses réplications sont effectuées simultanément à l'établissement d'un point de récupération, factorisant en quelque sorte le coût de la résolution d'un défaut de page. La seconde raison est que la réplication est effectuée de manière systématique à l'établissement d'un point de récupération sans échange préalable de messages avec le gestionnaire et le propriétaire d'une page. L'occurrence de telles opérations a un impact non négligeable sur la performance. Nous avons pu ainsi observer que le temps d'exécution de certaines applications avec ICARE est inférieur à celui obtenu avec une MVP sans tolérance aux fautes. En effet, dans ce cas, jusqu'a 32% des défauts de pages en lecture sont évités et transformés en succès en lecture, améliorant le temps d'exécution de près de 30%.

écritures sur des pages P-PR2 Lorsque des pages créées pour les besoins de la tolérance aux fautes à l'établissement d'un point de récupération sont utilisées ensuite pour des opérations en écriture, des défauts de page en écriture sont transformés en simples augmentations de droits d'accès. Etant donné qu'une augmentation de droit d'accès ne nécessite pas de transfert de page sur le réseau, ceci représente un facteur d'efficacité.

écritures sur des pages P-PR1 Ce type d'opération est relatif à des augmentations de droits d'accès exclusivement dues à l'établissement périodique de points de récupération. En effet, à l'établissement d'un point de récupération, les pages dans l'état *exclusif*, accessibles en écriture sont transformées en pages *partagé-PR1* et leur accès restreint en lecture. Si ces pages sont à nouveau référencées en écriture immédiatement après l'établissement d'un point de récupération, outre la création d'une réplique locale qui permet d'honorer la requête, les deux exemplaires de récupération doivent être invalidés et transformés en *invalide-PR*. Ceci entraîne un surcoût temporel non négligeable en raison du nombre de messages échangés à cette occasion.

Pour pallier cet inconvénient, une optimisation destinée aux programmeurs d'applications connaissant bien le comportement mémoire de leurs applications a été proposée dans [68].

4.7.2 Extensibilité

Il est important que notre approche préserve l'extensibilité des architectures considérées.

Nous avons évalué l'extensibilité du protocole ECP dans un COMA en faisant varier le nombre de nœuds de 9 à 56 et en mesurant les surcoûts de la phase de création et mémoire.

tel-00434053, version 1 - 20 Nov 2009

Le surcoût engendré par la phase de validation ne dépend pas du nombre de processeurs de l'architecture. Les résultats montrent que la dégradation de performance engendrée par la phase de création reste constante voire diminue avec le nombre de processeurs. Ce comportement a deux explications. La première est qu'avec des applications de taille fixe et une augmentation du nombre de processeurs, la quantité de données de récupération traitée par processeur, à chaque point de récupération, diminue. La seconde explication est une augmentation pratiquement linéaire du débit de réplication des données de récupération. Nous avons également observé que le surcoût mémoire reste constant ou diminue avec le nombre de processeurs.

La mise en œuvre actuelle d'ICARE ne permet pas d'en évaluer l'extensibilité. Cependant, nous avons mis en œuvre l'algorithme d'établissement de points de récupération sur une machine Intel Paragon à 56 nœuds exécutant le micro-noyau Mach. Les résultats obtenus dans ce contexte attestent de l'extensibilité du protocole dans un système à MVP [67].

4.8 Bilan

J'ai proposé une approche générique permettant de mettre en œuvre par une simple extension du protocole de cohérence une stratégie de retour arrière dans les architectures extensibles à mémoire partagée. Cette approche exploite les mécanismes de réplication existants de ce type d'architecture ainsi que la redondance de données nécessaire à la mise en œuvre d'une stratégie de retour arrière. La mise en œuvre du protocole proposé nécessite très peu de modifications matérielles dans une architecture COMA et est entièrement logicielle dans un système à MVP. Ceci est un atout de l'approche proposée qui peut être facilement implantée dans diverses architectures.

L'activité Aleth m'a permis d'obtenir des résultats importants dans les domaines de l'architecture, des systèmes distribués et de la tolérance aux fautes.

Dans le domaine de l'architecture, nous avons été les premiers à proposer une solution au problème de disponibilité des architectures extensibles à mémoire partagée et avons montré que parmi celles-ci les COMAs offrent des fonctionnalités qui permettent de mettre en œuvre simplement une stratégie de retour arrière avec un minimum de modifications matérielles.

Dans le domaine des systèmes distribués, le prototype ICARE est l'une des rares réalisations complètes de système à MVP recouvrable intégrant un mécanisme de reprise de processus. Les systèmes décrits dans la littérature sont pour la plupart évalués par simulation [89]. Il est à noter que la technologie micro-noyau utilisée pour le développement d'ICARE a grandement facilité l'intégration des mécanismes de tolérance aux fautes au sein du gestionnaire mémoire. La tâche aurait été plus difficile avec un système monolithique.

Ces travaux représentent en quelque sorte l'aboutissement de notre démarche. Avec ICARE, j'ai montré que la tolérance aux fautes peut être mise en œuvre de façon transparente aux applications à un coût raisonnable à partir de composants sur étagère par des mécanismes logiciels. J'ai aussi montré qu'une telle approche permet de concilier tolérance aux fautes et efficacité puisque les études de performance réalisées ont montré que les performances d'un système tolérant aux fautes peuvent être meilleures que celles d'un système équivalent sans mécanisme de tolérance aux fautes. Ceci a pu être obtenu grâce à l'intégration des mécanismes de gestion mémoire et de tolérance aux fautes permettant

tel-00434053, version 1 - 20 Nov 2009

de tirer bénéfice de l'ensemble des caractéristiques de l'architecture : fonctionnalités standard pour implanter les mécanismes de tolérance aux fautes et utilisation des mécanismes de tolérance aux fautes pour améliorer les performances du système. L'approche proposée permet en outre une utilisation optimale de la ressource mémoire, ce qui était l'un des objectifs poursuivis à l'issue du projet Gothic.

tel-00434053, version 1 - 20 Nov 2009

tel-00434053, version 1 - 20 Nov 2009

Chapitre 5

Conclusion

Je ne reviens pas ici sur les principaux résultats de recherche obtenus. Ils ont été soulignés tout au long du document. Je fais état de quelques constats et dresse quelques perspectives de recherche.

5.1 Démarche de recherche

La démarche que j'ai suivie pour mes travaux comporte trois étapes :

1. une étape de conception qui se traduit par la proposition d'un protocole répondant aux spécifications du problème posé,
2. une étape de vérification permettant de s'assurer de la correction du protocole avant sa mise en œuvre,
3. une étape d'expérimentation (par simulation ou réalisation d'un prototype) indispensable pour mesurer l'intérêt pratique et le réalisme de la solution élaborée.

J'ai toujours accordé une large place aux expérimentations au cours de mes travaux. Je considère en effet cette étape comme essentielle dans le domaine de la construction de machines et systèmes. Elle nécessite la mise en place d'une infrastructure souvent complexe (réseau de machines) et coûteuse. Le soutien d'un industriel se révèle indispensable pour se doter d'une infrastructure de taille réaliste permettant d'évaluer nos idées dans des configurations représentatives de celles utilisées dans le monde industriel. L'administration d'une plate-forme d'expérimentation constitue une lourde charge souvent dévolue à ses utilisateurs. Consciente de l'investissement demandé par ces tâches, j'ai initialisé une action de recherche sur ce thème dans le cadre du GIE BULL/INRIA Dyade en collaboration avec la *Business Unit* BULL ISM, leader sur le marché des plates-formes d'administration de réseaux. Dans ce cadre, je me suis attachée à concevoir et réaliser un environnement simple adapté aux particularités de notre environnement de recherche [15]. Deux thèses sont en cours sur ce thème.

Dans le domaine des systèmes distribués, la technologie micro-noyau s'est révélée être bien adaptée à nos besoins d'expérimentation de mécanismes système. L'utilisation d'un micro-noyau permet de disposer de tous les services de base sans avoir à les réécrire ce

qui serait extrêmement coûteux en temps. Compte tenu de la modularité d'un tel système, nos efforts de développement ont pu être concentrés sur le sous-système étudié sans avoir à modifier le reste du système. Ceci est très appréciable du fait de la complexité de mise au point des systèmes d'exploitation (en l'absence d'outils adaptés).

Dans le domaine de l'architecture, la simulation est une étape incontournable avant la réalisation d'un prototype. Nous avons été amenés à développer notre propre environnement de simulation. L'outil développé est modulaire permettant de simuler aisément de nombreuses variantes architecturales. Il permet de simuler des architectures extensibles comportant une cinquantaine de nœuds. La qualité des résultats obtenus dépend non seulement des outils utilisés et de la qualité du modèle d'architecture mais aussi des traces d'applications. Nos simulations ont été effectuées avec des traces d'applications parallèles standard dans le domaine de l'évaluation d'architectures multiprocesseurs. Malheureusement, au moment de nos travaux nous n'avons pas pu obtenir de traces d'applications transactionnelles, chasse gardée des constructeurs. Si les applications scientifiques ont permis d'étudier différents comportements vis à vis de la mémoire, elles ne constituent qu'un sous-ensemble des applications auxquelles les architectures étudiées sont destinées. En particulier, elles ne sont pas représentatives des applications transactionnelles qui sont très répandues.

5.2 Intérêt des approches à base de composants standard

Si mes premiers travaux m'ont conduit à concevoir des solutions fondées sur du matériel spécifique pour tolérer la défaillance d'un élément d'une machine, je suis désormais convaincue qu'il est possible de concilier tolérance aux fautes et efficacité en s'appuyant sur des composants sur étagère. Une telle approche possède indéniablement l'avantage d'un moindre coût. En outre, une solution conçue pour des composants standard est plus facilement adaptable à une large gamme d'architectures.

5.3 Intégration de mécanismes standard et de tolérance aux fautes

Mes travaux m'ont permis d'aboutir au système ICARE dans lequel les mécanismes de tolérance aux fautes sont intégrés à la gestion de la mémoire. Le même type d'approche peut être appliqué à d'autres mécanismes systèmes tels que la migration de processus et le système de reprise de processus. L'intégration de mécanismes habituellement disjoints permet une meilleure utilisation des ressources. Ces travaux ouvrent des perspectives dans le domaine des réseaux de stations de travail. Dans un tel système, les ressources processeur et mémoire d'un site peuvent être sous utilisées alors qu'un autre site souffre de surcharge. Du point de vue des applications, les ressources disponibles d'une station de travail peuvent être insuffisantes pour certaines applications nécessitant d'importants volumes de données ou une très grande puissance de calcul. Afin de permettre à de telles applications de profiter de l'ensemble des ressources du réseau, il est intéressant de concevoir un système d'exploitation permettant de rapprocher le contexte d'exécution d'un processus de ses données, soit

tel-00434053, version 1 - 20 Nov 2009

en faisant migrer le processus sur le site sur lequel sont situées les données, soit en faisant migrer les données sur le site d'exécution du processus selon l'état de la mémoire, la charge des processeurs et le patron d'accès aux données. Dans ce contexte où l'état d'un processus peut être distribué sur plusieurs sites, il est essentiel de mettre en œuvre des mécanismes de tolérance aux fautes pour tolérer de façon transparente aux applications la défaillance d'un site. L'intégration des mécanismes de gestion des processeurs, de la mémoire et de tolérance aux fautes devrait permettre un usage optimal des ressources disponibles permettant de répondre aux besoins des applications en terme de ressources et de performance. Il serait en outre intéressant de traiter dans ce cadre l'hétérogénéité des systèmes et du matériel.

5.4 Système d'exploitation pour architectures multiprocesseurs tolérantes aux fautes

Dans le domaine de la conception d'architectures multiprocesseurs tolérantes aux fautes, j'ai essentiellement travaillé sur des protocoles destinés à être mis en œuvre par matériel. Dans le cadre du projet FASST, j'ai abordé la conception du système d'exploitation de la machine, en particulier à travers la gestion des entrée/sorties. Cependant, peu d'expérimentations ont pu être effectuées en raison de contraintes liées au projet Esprit FASST dans lequel se sont déroulés ces travaux. De telles expérimentations seraient nécessaires pour valider complètement notre approche. Un problème potentiel est celui des dépendances générées par le système du fait des verrous. La compétition des processus sur les verrous est susceptible, si aucune précaution n'est prise, de rendre dépendants l'ensemble des processeurs de la machine. Dans le cadre de mes travaux relatifs aux architectures COMA, je ne me suis pas véritablement penchée sur la conception du système d'exploitation. En effet, une telle étude aurait été peu réaliste en l'absence de prototype. Il s'agit d'une perspective intéressante à mes travaux. Je peux mentionner les travaux menés à l'université de Stanford sur la machine Flash [70] et son système d'exploitation Hive [31] qui intègre des mécanismes de confinement d'erreur et à l'université d'Illinois où la tolérance aux fautes est clairement un objectif du projet I-ACOMA [127]. Dans les deux cas, des expérimentations sont possibles puisqu'un prototype de machine est développé.

5.5 Systèmes distribués et architectures multiprocesseurs extensibles

Au cours de ces dix dernières années, j'ai assisté au rapprochement de deux domaines auparavant disjoints : celui des architectures multiprocesseurs et celui des systèmes distribués. L'évolution de la technologie fait qu'un réseau de stations de travail fournit potentiellement une puissance de calcul comparable à celle des multiprocesseurs. En effet, la puissance des processeurs des stations de travail évolue à un rythme soutenu faisant passer en 10 ans du processeur Motorola 68020 à 16 MHz utilisé dans les machines BULL SPS7 utilisées dans le projet Gothic au processeur Pentium Pro à 200 MHz en 1997 (les processeurs de la plate-forme Astrolab sont des Pentium à 133 MHz). En parallèle, nous avons assisté au développement des réseaux haut débit. Ainsi, nous sommes passé d'un

tel-00434053, version 1 - 20 Nov 2009

tel-00434053, version 1 - 20 Nov 2009

réseau local Ethernet à 10 Mb/s dans le projet Gothic, à un réseau ATM à 155 Mb/s pour la plate-forme Astrolab. L'effet conjugué de l'évolution de la puissance des processeurs et du débit des réseaux laisse présager que les réseaux de stations de travail constitueront des architectures à haute performance concurrentes des architectures multiprocesseurs à condition toutefois que les systèmes d'exploitation évoluent pour tirer pleinement profit des technologies sous-jacentes.

Du côté des architectures multiprocesseurs, on est passé au début des années 90 des architectures multiprocesseurs à bus aux architectures multiprocesseurs extensibles constituées d'une interconnexion de nœuds. Les nœuds sont des stations de travail à haute performance intégrant des composants matériels spécifiques (contrôleur mémoire, interface réseau) destinés à mettre en œuvre efficacement le partage de mémoire. Les réseaux utilisés dans les prototypes sont quant à eux dérivés de ceux des machines parallèles. L'organisation distribuée des architectures multiprocesseurs extensibles les rapproche donc des systèmes distribués. L'évolution de la technologie et la tendance à l'utilisation de composants sur étagère laisse présager d'une intersection de plus en plus importante entre ces deux domaines. Les concepts développés dans l'un d'entre eux pourront s'appliquer dans l'autre. Nous en avons fait l'expérience dans le projet Aleth dans lequel nous avons transposé des idées ayant germé dans le cadre de l'étude d'architectures multiprocesseurs au domaine des systèmes distribués dans le contexte d'un système à mémoire virtuelle partagée.

5.6 Applications

Les applications considérées dans les travaux décrits dans ce document sont essentiellement des applications parallèles et des applications transactionnelles. Dans le domaine de la tolérance aux fautes, d'autres classes d'applications présentant d'autres types de contraintes que celles que nous avons considérées mériteraient d'être étudiées. Je peux citer en particulier les applications ayant des contraintes temps réel, les applications multimédias qui sont en pleine expansion et les applications grande échelle sur Internet.

tel-00434053, version 1 - 20 Nov 2009

Bibliographie

[1] A. Agarwal, D. Chaiken, K. Johnson, D. Kranz, J. Kubiatowicz, K. Kurihara, B. Lim, G. Ma, and D. Nussbaum. The MIT Alewife machine : A large-scale distributed memory multiprocessor. Rapport de recherche MIT/LCS/TM-454, MIT Laboratory for Computer Science, Juin 1991.

[2] R.E. Ahmed, R.C. Frazier, and P.N. Marinos. Cache-aided rollback error recovery (CARER) algorithms for shared-memory multiprocessor systems. In *Proc. of 20th International Symposium on Fault-Tolerant Computing Systems*, pages 82–88, Newcastle, Juin 1990.

[3] Alliant. Fx/series product summary. Technical report, Alliant Computer Systems Corporation, 1986.

[4] C. Amza, A.L. Cox, S. Dwarkadas, P. Keleher, H. Lu, R. Rajamony, W. Yu, and W. Zaenepoel. Treadmarks: Shared memory computing on networks of worstations. *Computer*, pages 18–28, Février 1996.

[5] T.E. Anderson, D.E. Culler, and D.A. Patterson. The Berkeley Networks of Workstations (NOW) project. In *CompCon Spring*, pages 322–326, 1995.

[6] Ö. Babaoğlu and R. Drummond. Streets of byzantium : Network architectures for fast reliable broadcast. *IEEE Transactions on Software Engineering*, SE-11(6), 1985.

[7] Ö. Babaoğlu, P. Stephenson, and R. Drummond. Reliable broadcasts and communication models : Tradeoffs and lower bounds. *Distributed Computing*, ?(2):177–189, 1988.

[8] H. E. Bal, M. F. Kaashoek, and A. S. Tanenbaum. Experience with distributed programming in Orca. *IEEE Transactions on Software Engineering*, 18:190–205, Mars 1992.

[9] J. P. Banâtre, M. Banâtre, and C. Morin. Implementing atomic rendezvous within a transactional framework. In *Proc. of 8th Symposium on Reliable Distributed Systems*, pages 119–128, Seattle, Octobre 1989. IEEE.

[10] J.P. Banâtre. *La programmation parallèle : outils, méthodes et éléments de mise en œuvre*. Eyrolles, 1990.

[11] J.P. Banâtre and M. Banâtre, editors. *Les systèmes distribués : l'expérience du projet Gothic*. InterEditions, Février 1991.

[12] M. Banâtre, A. Gefflaut, P. Joubert, P. Lee, and C. Morin. An architecture for tolerating processor failures in shared-memory multiprocessors. *IEEE Transactions on Computers*, 45(10):1101–1115, Octobre 1996.

tel-00434053, version 1 - 20 Nov 2009

[13] M. Banâtre, A. Gefflaut, and C. Morin. Scalable shared memory multiprocessors: Some ideas to make them reliable. In M. Banâtre and P.A. Lee, editors, *Hardware and Software Architectures for Fault Tolerance, Experiences and perspectives*, number 774 in LNCS. Springer Verlag, 1994.

[14] Ph. Bernstein. Sequoia: A fault-tolerant tightly coupled multiprocessor for transaction processing. *IEEE Computer*, 21(2):37–45, Février 1988.

[15] S. Billiart, A. Sahai, and C. Morin. Administration système à partir d'un navigateur Web. In *Proc. des 2eme journées réseaux JRES-97*, Octobre 1997.

[16] K. Birman and R. Van Renesse. *Reliable Distributed Computing with the Isis Toolkit*. IEEE Computer Society Press, 1994.

[17] K. P. Birman and T. Joseph. Reliable communication in the presence of failures. *ACM Transactions on Computer Systems*, 5(1):47–76, Février 1987.

[18] A.D. Birrell and B.J. Nelson. Implementing remote procedure call. *ACM Transactions on Computer Systems*, 2(1):39–59, Février 1984.

[19] A. Borg, J. Baumach, and S. Glazer. A message system supporting fault-tolerance. In *Proc. of 9th ACM Symposium on Operating Systems Principles*, pages 90–99, Bretton Woods, N.H., Octobre 1983.

[20] F. Browaeys, H. Derriennic, P. Desclaud, H. Fallour, C. Faulle, J. Febvre, J.E. Hanne, M. Kronental, J.J. Simon, and D. Vojnovic. Sceptre : proposition de noyau normalisé pour les exécutifs temps réel. *Techniques et Sciences Informatiques*, 3(1):45–62, Janvier 1984.

[21] L. Brown and J. Wu. Dynamic snooping in a fault-tolerant distributed shared memory. In *Proc. of the 14th International Conference on Distributed Computing Systems*, 1994.

[22] G. Cabillic, G. Muller, and I. Puaut. The performance of consistent checkpointing in distributed shared memory systems. In *Proc. of the 14th Symposium on Reliable Distributed Systems*, Septembre 1995.

[23] G. Cabillic, T. Priol, and I. Puaut. MYOAN: an implementation of the KOAN shared virtuel memory on the Intel Paragon. Rapport de recherche 812, IRISA, Mars 1994.

[24] N. Carriero and D. Gelernter. The S/Net's Linda kernel. *ACM Transactions on Computer Systems*, 4:110–129, Mai 1986.

[25] J. B. Carter, J. K. Bennett, and W. Zwaenepoel. Implementation and performance of Munin. In *Proc. of 13th ACM Symposium on Operating Systems Principles*, pages 152–164, Octobre 1991.

[26] J.B. Carter, A.L. Cox, S. Dwarkadas, E.N. Elnozahy, D.B. Johnson, P. Keleher, S.Rodrigues, W. Yu, and W. Zwaenepoel. Network multicomputing using recoverable distributed shared memory. In *Proc. of Spring Compcon93*, Février 1993.

[27] J.B. Carter, D. Khandekar, and L. Kamb. Distributed shared memory: Where we are and where we should be headed. In *Proc. of Fifth Workshop on Hot Topics in Operating Systems*, 1995.

[28] W.C. Carter. Experiences in fault tolerant computing, 1947-1971. In A. Avizienis, H. Kopetz, and J.C. Laprie, editors, *The Evolution of Fault-Tolerant Computing*, volume 1, pages 1–36. Springer Verlag, 1987.

[29] K. M. Chandy and L. Lamport. Distributed snapshots : Determining global states of distributed systems. *ACM Transactions on Computer Systems*, 3(1):63–75, Février 1985.

[30] J. M. Chang and N. F. Maxemchuk. Reliable broadcast protocols. *ACM Transactions on Computer Systems*, 2(3):251–273, Août 1984.

[31] J. Chapin, M. Rosenblum, S. Devine, T. Lahiri, D. Teodosiu, and A. Gupta. Hive: Fault containment for shared-memory multiprocessors. In *Proc. of 15th ACM Symposium on Operating Systems Principles*, 1995.

[32] D.R. Cheriton and W. Zwaenepoel. Distributed process group in the V Kernel. *ACM Transactions on Computer Systems*, 3(2):77–107, Mai 1985.

[33] E. Cooper. *Replicated Distributed Programs*. PhD thesis, Computer Science Division, University of California, Berkeley, Mai 1985.

[34] M. Costa, P. Guedes, M. Sequeira, N. Neves, and M. Castro. Lightweight logging for lazy release consistent distributed shared memory. In *Proc. of the second Symposium on Operating Systems Design and Implementation*, Novembre 1996.

[35] R. C. Covington, S. Madala, V. Metha, J.R. Jump, and J. B. Sinclair. The Rice parallel processing testbed. In *Proc. of ACM SIGMetrics International Conference on Measurement and Modeling of Computer Systems*. ACM, 1988.

[36] A.L. Cox, S. Dwarkadas, P. Keleher, H. Lu, R. Rajamony, and W. Zwaenepoel. Software versus hardware shared-memory implementation: A case study. In *Proc. of 21th Annual International Symposium on Computer Architecture*, pages pp 106–117, Mai 1994.

[37] F. Cristian. Synchronous atomic broadcast for redundant broadcast channels. Technical Report RJ 7203 (67682), IBM Almaden Research Center, Décembre 1989.

[38] F. Cristian. Synchronous and asynchronous group communication. *Communications of the ACM*, 39(4):88–97, Avril 1996.

[39] F. Cristian, H. Aghili, R. Strong, and D. Dolev. Atomic broadcast : From simple message diffusion to bizantine agreement. Technical Report RJ 5244, IBM Almaden Research Center, Juillet 1986.

[40] H. Davis, S.R. Goldschmidt, and J. Hennessy. Multiprocessor simulation using Tango. In *Proc. of 1991 International Conference on Parallel Processing*, volume II, pages 99–107, Août 1991.

[41] D. Dolev and D. Malki. The Transis approach to high availability cluster comunication. *Communications of the ACM*, 39(4):65–70, Avril 1996.

[42] M. Dubois, F.A. Briggs, I. Patil, and M. Balakrishnan. Trace-driven simulation of parallel and distributed algorithms in multiprocessors. In *Proc. of 1986 International Conference on Parallel Processing*, volume I, pages 505–508, Août 1986.

[43] E. Elnozahy, D. Johnson, and W. Zwaenepoel. The performance of consistent checkpoint. In *Proc. of 11th Symposium on Reliable Distributed Systems*, pages 39–47, Octobre 1992.

[44] E. N. Elnozahy, D.B. Johnson, and Y.M. Wang. A survey of rollback-recovery protocols in message passing systems. Technical Report CMU-CS-96-181, Department of Computer Science, Carnegie Mellon University, Septembre 1996.

tel-00434053, version 1 - 20 Nov 2009

[45] B. D. Fleisch and G. J. Popek. Mirage : a coherent shared memory design. In *Proc. of 12th ACM Symposium on Operating Systems Principles*, Operating System Review, pages 211–223, Décembre 1989.

[46] B.D. Fleisch. Reliable distributed shared memory. In *Proc. of the 2nd Workshop on Experimental Distributed Systems*, pages 102–105, 1990.

[47] S. Frank, H. Burkhardt, and J. Rothnie. The KSR1 : Bridging the gap between shared memory and MPPs. In IEEE Computer Society, editor, *Proc. of spring COMPCON'93*, pages 285–294, Février 1993.

[48] H. Garcia-Molina and A. Spauster. Ordered and reliable multicast communication. Technical Report CSTR 184-88, Princeton University Department of Computer Science, Octobre 1988.

[49] A. Gefflaut, C.Morin, and M.Banâtre. Tolerating node failures in Cache Only Memory Architectures. In *Proc. of Supercomputing'94*, Novembre 1994.

[50] Alain Gefflaut. *Proposition et évaluation d'une architecture multiprocesseur extensible à mémoire partagée tolérante aux fautes.* Thèse de doctorat, Université de Rennes I, Janvier 1995.

[51] Alain Gefflaut and Philippe Joubert. SPAM: a multiprocessor execution-driven simulation kernel. *International Journal in Computer Simulation*, 6(1):69–88, 1996. Special issue: computer architecture simulations.

[52] J. Gray. *Notes on Database Operating Systems.*, volume 60 of *Lecture Notes in Computer Science*. Springer Verlag, 1978.

[53] E. Hagersten, A. Landin, and S. Haridi. DDM - a cache-only memory architecture. *IEEE Computer*, 25(9):44–54, Septembre 1992.

[54] Intel. The Intel iPSC/2 system. In *Third Conference on Hypercube Concurrent Computers and Applications*, pages 843–846, Pasadena, California, 1988.

[55] Intel Corporation. *Paragon User's Guide*, 1993.

[56] V. Issarny. Architectures logicielles pour systèmes distribués. Habilitation à diriger des recherches, Octobre 1997. Université de Rennes I.

[57] G. Janakiraman and Y. Tamir. Coordinated checkpointing-rollback error recovery for distributed shared memory multicomputers. In *Proc. of the 13th Symposium on Reliable Distributed Systems*, 1994.

[58] B. Janssens and W. K. Fuchs. Reducing interprocessor dependence in recoverable distributed shared memory. In *Proc. of the 13th Symposium on Reliable Distributed Systems*, pages 34–41, Dana Point, CA, Octobre 1994.

[59] B. Janssens and W.K. Fuchs. Experimental evaluation of multiprocessor cache-based error recovery. In *Proc. of 1991 International Conference on Parallel Processing*, volume I, pages 505–508, Août 1991.

[60] B. Janssens and W.K. Fuchs. Relaxing consistency in recoverable distributed shared memory. In *Proc. of 23rd International Symposium on Fault-Tolerant Computing Systems*, 1993.

[61] B. Janssens and W.K. Fuchs. Ensuring correct rollback recovery in distributed shared memory systems. *Journal of Parallel and Distributed Computing*, Octobre 1995.

tel-00434053, version 1 - 20 Nov 2009

[62] D. Jewett. Integrity S2: A fault-tolerant Unix platform. In *Proc. of 21st International Symposium on Fault-Tolerant Computing Systems*, pages 512–519, Montréal, Canada, Juin 1991.

[63] D.B. Johnson. Efficient transparent optimistic rollback recovery for distributed application programs. In *Proc. of the 12th Symposium on Reliable Distributed Systems*, 1993.

[64] D.B. Johnson and W. Zwaenepoel. Sender-based message logging. In *Proc. of 12th International Symposium on Fault-Tolerant Computing Systems*, 1987.

[65] P. Joubert. *Conception et évaluation d'une architecture multiprocesseur à mémoire partagée tolérante aux fautes*. Thèse de doctorat, université de Rennes I, Janvier 1993.

[66] R. H. Katz, S. J. Eggers, D. A. Wood, C. L. Perkins, and R. G. Sheldon. Implementing a cache consistency protocol. In *Proc. of 12th Annual International Symposium on Computer Architecture*, pages 276–283, Boston, 1985. IEEE.

[67] A. Kermarrec, G. Cabillic, A. Gefflaut, C. Morin, and I. Puaut. A recoverable distributed shared memory integrating coherence and recoverability. In *Proc. of 25th International Symposium on Fault-Tolerant Computing Systems*, pages 289–298, Pasadena, USA, Juin 1995. IEEE Computer Society Press.

[68] A.-M. Kermarrec. *Une approche globale fondée sur la réplication pour la disponibilité et l'efficacité des systèmes extensibles à mémoire partagée*. Thèse de doctorat, Université de Rennes 1, 1996.

[69] A.-M. Kermarrec and C. Morin. *Fault-Tolerant Parallel and Distributed Systems*, chapitre 7 : An efficient recoverable DSM on a network of Workstations: design and implementation. Kluwer Academic Press, 1997.

[70] J. Kuskin, D. Ofelt, M. Heinrich, J. Heinlein, R. Simoni, K. Gharachorloo, J. Chapin, D. Nakahira, J. Baxter, M. Horowitz, A. Gupta, M. Rosenblum, and J. Hennessy. The Stanford FLASH multiprocessor. In *Proc. of 21th Annual International Symposium on Computer Architecture*, pages 302–313, Chicago, Illinois, Avril 1994.

[71] Z. Lahjomri and T. Priol. KOAN : a shared virtual memory for the iPCS/2 hypercube. In *CONPAR/VAPP92*, Septembre 1992.

[72] L. Lamport. Time, clocks and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, Juillet 1978.

[73] L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Transactions on Computers*, 28(9):690–691, Septembre 1979.

[74] B. Lampson. Atomic transactions. In *Distributed Systems and Architecture and Implementation : an Advanced Course*, volume 105 of *Lecture Notes in Computer Science*, pages 246–265. Springer Verlag, 1981.

[75] B.W. Lampson. Remote Procedure Call. In *Distributed Systems and Architecture and Implementation : an advanced course*, volume 105 of *Lecture Notes in Computer Science*, chapter 14, pages 365–370. Springer Verlag, New York, 1981.

[76] P.A. Lee and T. Anderson. *Fault Tolerance: Principles and Practice*, volume 3 of *Dependable Computing and Fault-Tolerant Systems*. Springer Verlag, second revised edition, 1990.

tel-00434053, version 1 - 20 Nov 2009

[77] D. Lenoski, J. Laudon, K. Gharachorloo, W. Weber, A. Gupta, J. Hennessy, M. Horowitz, and M. Lam. The Stanford DASH multiprocessor. *IEEE Computer*, 25(3):63–79, Mars 1992.

[78] K. Li and P. Hudak. Memory coherence in shared virtual memory systems. *ACM Transactions on Computer Systems*, 7(4):321–357, Novembre 1989.

[79] B. Liskov, D. Curtis, P. Johnson, and R. Scheifler. Implementation of Argus. In *Proc. of 11th ACM Symposium on Operating Systems Principles*, pages 111–122, 1987.

[80] D. Litaize, A. Mzoughi, C. Rochange, and P. Sainrat. Towards a shared memory massively parallel multiprocessor. In *Proc. of 19th Annual International Symposium on Computer Architecture*, pages 70–79, Mai 1992.

[81] T. Lovett and R.Clapp. STiNG : A CC-NUMA computer system for the commercial marketplace. In *Proc. of 23rd Annual International Symposium on Computer Architecture*, 1996.

[82] T. Lovette and S. Thakkar. The Symmetry multiprocessor system. In *Proc. of International Conference on Parallel Processing*, 1988.

[83] E. Markatos and G. Dramitinos. Implementation of a reliable remote memory pager. In *Proc. of Usenix 1996 Technical Conference*, Janvier 1996.

[84] B. Martin. Parallel remote procedure call and portable C stub compiler. In *Proc. Workshop on Design Principles for Experimental Distributed Systems*, Purdue University (Indiana), Octobre 1986.

[85] C. Morin. An efficient implementation of the rendezvous atomicity property. In D.J. Evans, G.R. Joubert, and F.J. Peters, editors, *Parallel Computing 89*, pages 603–608. Elsevier Science Publishers B.V (North-Holland), 1989.

[86] C. Morin. Fault-tolerant implementation of CSP input-output commands. In *Proc. of The Fourth International Conference on Fault-tolerant Computing Systems*, Baden-Baden (RFA), Septembre 1989.

[87] C. Morin. *Protocole d'appel de multiprocédure à distance dans le système Gothic : définition et mise en œuvre*. Thèse de doctorat, université de Rennes I, Décembre 1990.

[88] C. Morin, A. Gefflaut, M. Banâtre, and A.-M. Kermarrec. COMA: an opportunity for building fault-tolerant scalable shared memory multiprocessors. In *Proceedings of the 23rd international Symposium on Computer Architectures*, pages 56–65, Philadelphie, USA, Mai 1996.

[89] C. Morin and I. Puaut. A survey of recoverable distributed shared memory systems. *IEEE Transactions on parallel and Distributed Systems*, 8(9), Septembre 1997.

[90] Christine Morin, Michael Johnk, Wilfried Schwartz, Andrew Thomas, and Paula McGrath. Fasst microkernel specification. Technical report, Projet Esprit FASST, Septembre 1992. deliverable.

[91] L. Moser, R. Budhia P. Melliar-Smith, D. Agarwal, and C. Lingley-Papadopoulos. Totem : A fault-tolerant multicast group communication system. *Communications of the ACM*, 39(4):54–63, Avril 1996.

[92] B.J. Nelson. *Remote Procedure Call*. PhD thesis, Department of Computer Science, Carnegie-Mellon University, Pittsburgh, 1981.

tel-00434053, version 1 - 20 Nov 2009

[93] N. Neves, M. Castro, and P. Guedes. A checkpoint protocol for an entry consistent shared memory system. In *Proc. of 13th International Symposium on Principles of Distributed Computing*, Août 1994.

[94] T.P. Ng. Checkpointing in a virtual shared memory system. Technical Report UIUCDCS-R-91-1700, University of Illinois at Urbana-Champaign, Décembre 1991.

[95] J. Nieh and M. Lam. The design and implementation and evaluation of SMART : a scheduler for multimedia applications. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles*, pages 184–197, 1997.

[96] W. Oed. The Cray research massively parallel processor system CRAY T3D. Technical report, Cray Research GmbH, Novembre 1993.

[97] M. S. Papamarcos and J. H. Patel. A low overhead coherence solution for multiprocessors with private cache memories. In *Proc. of 11th Annual International Symposium on Computer Architecture*, pages 348–354, Ann Arbor, Juin 1984. IEEE.

[98] K.J. Perry and S. Toueg. Distributed agreement in the presence of processor and communication faults. *IEEE Transactions on Software Engineering*, 12(3):477–482, Mars 1986.

[99] G. Pfister, W. Brantley, D. George, S. Harvey, W. Kleinfelder, K. McAuliffe, E. Melton, V. Norton, and J. Weiss. The IBM research parallel processor prototype (RP3): Introduction and architecture. In *Proceedings of the 1985 International Conference on Parallel Processing*, pages 764–771. IEEE Computer Society, Août 1985.

[100] F. Pong and M. Dubois. A new approach for the verification of cache coherence protocols. Technical report, Department of Electrical Engineering - Systems, University of Southern California, Avril 1993.

[101] M.L. Powell and D.L. Presotto. Publishing: A reliable broadcast communication mechanism. In *Proc. of 9th ACM Symposium on Operating Systems Principles*, pages 100–109, 1983.

[102] B. Randell. System structure for software fault tolerance. *IEEE Transactions on Software Engineering*, 1(SE-2):220–232, Juin 1975.

[103] B. Randell. *Reliable Computing Systems*, volume 60. Lecture Notes in Computer Science, 1978.

[104] M. Reiter. Distributed trust with the Rampart toolkit. *Communications of the ACM*, 39(4):71–74, Avril 1996.

[105] R. Van Renesse, K. Birman, and S. Maffeis. Horus : a flexible group communication system. *Communications of the ACM*, 39(4):76–83, Avril 1996.

[106] G.G. Richard-III and M. Singhal. Using logging and asynchronous checkpointing to implement recoverable distributed shared memory. In *Proc. of 12th Symposium on Reliable Distributed Systems*, pages 58–67, Octobre 1993.

[107] M. Rozier, V. Abrossimov, F. Armand, I. Boule, M. Gien, M. Guillemont, F. Herrmann, P. Léonard, S. Langlois, and W. Neuhauser. The Chorus distributed operating system. *Computing Systems*, 1(4):305–370, 1988.

[108] M. Satyanarayanan and E.H. Siegel. MultiRPC : A parallel Remote Procedure Call mechanism. Technical Report CMU-CS-86-139, Department of Computer Science, Carnegie-Mellon University, Pittsburgh, Août 1986.

tel-00434053, version 1 - 20 Nov 2009

tel-00434053, version 1 - 20 Nov 2009

[109] A. Saulsbury, T. Wilkinson, J. Carter, and A. Landin. An argument for simple COMA. In *Proc. of 1st IEEE Symposium on High-Performance Computer Architecture*, Janvier 1995.

[110] A. Schiper and M. Raynal. From group communication to transactions in distributed systems. *Communications of the ACM*, 39(4):84–87, Avril 1996.

[111] F. B. Schneider. The fail-stop processor approach. In *Concurency control and reliability in distributed systems, Chapitre 13*, pages 370–394. Barghava, 1987.

[112] A. Segall and B. Awerbuch. A reliable broadcast protocol. *IEEE Transactions on Communications*, 31(7):896–901, Juillet 1983.

[113] S. K. Shrivastava and F. Panzieri. The design of a reliable remote procedure call mechanism. *IEEE Transactions on Computers*, C-31(7), Juillet 1982.

[114] S.K. Shrivastava. On the treatment of orphans in a distributed system. In IEEE Computer Society, editor, *Proc. of the 3rd Symposium on Reliability in Distributed Software and Database Systems*, pages 155–162, Florida, Octobre 1983.

[115] J.P. Singh, W.D. Weber, and A. Gupta. SPLASH : Stanford parallel applications for shared-memory. Technical Report CSL-TR-91-469, Computer Systems Laboratory, Stanford University, Avril 1991.

[116] M. Squillante and E. Lazowska. Using processor-cache affinity information in shared-memory multiprocessor scheduling. Technical report, IBM Research Division, 1991.

[117] P. Stenstrom, T. Joe, and A. Gupta. Comparative performance evaluation of cache-coherent NUMA and COMA architectures. In *Proc. of 19th Annual International Symposium on Computer Architecture*, pages 80–91, Mai 1992.

[118] R.E. Strom and S.A. Yemini. Optimistic recovery: An asynchronous approach to fault-tolerance in distributed systems. In *Proc. of 14th International Symposium on Fault-Tolerant Computing Systems*, pages pp 374–379, 1984.

[119] M. Stumm and S. Zhou. Fault tolerant distributed shared memory algorithms. In *Proc. of 2nd IEEE Symposium on Parallel and Distributed Processing*, pages 719–724, Dallas, Texas, Décembre 1990.

[120] G. Suri, B. Janssens, and W.K. Fuchs. Reduced overhead logging for rollback recovery in distributed shared memory. In *Proc. of 25th International Symposium on Fault-Tolerant Computing Systems*, 1995.

[121] L. Svobodova. File servers for network-based distributed systems. *ACM Computing Surveys*, 16(4), Décembre 1984.

[122] L. Svobodova. Resilient distributed computing. *IEEE Transactions on Software Engineering*, SE-10(3):257–268, Mai 1984.

[123] V. O. Tam and M. Hsu. Fast recovery in distributed shared virtual memory systems. In *Proc. of 10th International Conference on Distributed Computing Systems*, pages 38–45, Paris, France, Mai 1990.

[124] Y. Tamir and C. Sequin. Error recovery in multicomputers using global checkpoints. In *Proc. of 1984 International Conference on Parallel Processing*, pages 32–41, Août 1984.

[125] A.S. Tanenbaum. *Computer Networks*. Prentice-Hall International, 2nd edition edition, 1988.

tel-00434053, version 1 - 20 Nov 2009

[126] C. P. Thacker, L. C. Stewart, and E. H. Satterthwaite. Firefly : A multiprocessor workstation. *IEEE Transactions on Computers*, 37(8):909–920, Août 1988.

[127] J. Torrellas and D. Padua. The illinois aggressive coma multiprocessor project. In *Proc. of Sixth Symposium on the Frontiers of Massively Parallel Computing*, Octobre 1996.

[128] Y.M. Wang and K. Fuchs. Optimistic message logging for independant checkpointing in message-passing systems. In *Proc. of the 11th Symposium on Reliable Distributed Systems*, 1992.

[129] T. J. Wilkinson. *Implementing Fault Tolerance in a 64-bit Distributed Operating System*. PhD thesis, City University of London, Juillet 1993.

[130] D. Wilson. The stratus computer system. In T. Anderson, editor, *Resilient Computer Systems*, pages 208–231, 1985.

[131] K. L. Wu and W. K. Fuchs. Recoverable distributed shared memory: Memory coherence and storage structures. *IEEE Transactions on Computers*, 34(4):460–469, Avril 1990.

[132] K. L. Wu, W. K. Fuchs, and J. H. Patel. Cache-based error recovery for shared memory multiprocessor systems. In *Proc. of 1989 International Conference on Parallel Processing*, volume 1, pages 159–166, University Park, Pennsylvania, 1989.

[133] K.L. Wu, W.K. Fuchs, and J.H. Patel. Error recovery in shared memory multiprocessors using private caches. *IEEE Transactions on Parallel and Distributed Systems*, 1(2):231–240, Avril 1990.

[134] J. Xu and R.H.B. Netzer. Adaptative independant checkpointing for reducing rollback propagation. In *Proc. of the 5th IEEE Symposium on Parallel and Distributed Processing*, pages 754–761, 1993.

[135] M.J. Zekauskas, W.A. Sawdon, and B.N. Bershad. Software write detection for distributed shared memory. In *Proc. of the First Symposium on Operating System Design and Implementation*, pages 87–100, Novembre 1994.