



**Institut de Formation Supérieur en  
Informatique et Communication**  
Université de Rennes 1  
Campus de Beaulieu  
35042 RENNES



**Institut de Recherche en Informatique  
et Systèmes Aléatoires**  
Campus Universitaire de Beaulieu  
Avenue du Général Leclerc  
35042 RENNES

# Intégration d'un mécanisme de reprise d'applications parallèles dans un système d'exploitation pour grappe

Rapport de stage  
Matthieu Fertré  
Master 2 de Recherche en Informatique

Juin 2005

Encadrement : Christine Morin, Directeur de recherche INRIA



# Table des matières

<b>Introduction</b>	<b>5</b>
<b>1 État de l'art</b>	<b>7</b>
1.1 Introduction	7
1.2 Contexte	7
1.2.1 Applications parallèles communiquant par messages	7
1.2.2 Tolérance aux fautes	8
1.3 Sauvegarde et restauration de points de reprise d'applications parallèles	8
1.3.1 Protocoles de recouvrement arrière à base de points de reprise	9
1.3.2 Protocoles fondés sur la journalisation des messages	11
1.3.3 Comparaison des différentes stratégies	12
1.4 Mécanisme de base pour la mise en œuvre d'une stratégie de recouvrement arrière	13
1.4.1 Technique de détection des défaillances	13
1.4.2 Réalisation d'une mémoire stable pour le stockage des points de reprise	15
1.4.3 Point de reprise d'un processus	16
1.5 Mise en œuvre de la sauvegarde/restauration dans les environnements <i>MPI</i> et <i>PVM</i>	19
1.6 Conclusion	21
<b>2 Le système d'exploitation KERRIGHED</b>	<b>23</b>
2.1 Système à image unique	23
2.2 Fonctionnalités de KERRIGHED	23
2.3 Architecture du système KERRIGHED	24
2.4 État du prototype	25
<b>3 Sauvegarde et restauration d'applications parallèles dans le système KERRIGHED</b>	<b>27</b>
3.1 Vision système des applications parallèles s'exécutant sur grappe KERRIGHED	28
3.1.1 Les processus dans LINUX	28
3.1.2 Vision des processus dans KERRIGHED	29
3.2 Frontière d'application parallèle	31
3.2.1 Cas général	31
3.2.2 Cas des applications client/serveur	31
3.3 Organisation, désignation et stockage des points de reprise	31
3.3.1 Stockage physique des points de reprise	32
3.3.2 Organisation et désignation des points de reprise	32
3.4 Interface utilisateur	33
3.4.1 Types d'interfaces	33
3.4.2 Interface existante dans KERRIGHED	33

3.4.3	Proposition d'une nouvelle interface pour les points de reprise dans KERRIGHED	34
3.4.4	Exemples . . . . .	34
3.5	Sauvegarde d'un point de reprise d'application parallèle . . . . .	35
3.5.1	Déterminer quels sont les processus de l'application et les endormir . . . . .	35
3.5.2	Sauvegarder les processus de l'application . . . . .	36
3.6	Restauration d'un point de reprise d'application parallèle . . . . .	37
3.6.1	Gestion des PID . . . . .	38
3.7	Traitement des entrées-sorties . . . . .	40
3.7.1	Les flux dynamiques . . . . .	40
3.7.2	Les fichiers réguliers . . . . .	41
<b>4</b>	<b>Mise en œuvre et expérimentation</b>	<b>45</b>
4.1	Mise en œuvre . . . . .	45
4.2	Conditions d'expérimentations . . . . .	45
4.3	Analyse des résultats . . . . .	46
	<b>Conclusion</b>	<b>51</b>
	<b>Bibliographie</b>	<b>52</b>
	<b>Annexes</b>	<b>57</b>

# Introduction

Le calcul parallèle est aujourd'hui utilisé dans de nombreux domaines et particulièrement dans le monde scientifique. Parmi ces applications, on trouve par exemple des simulateurs biologiques, des modèles météorologiques. Pour ce type d'applications, il est nécessaire de disposer d'une puissance de calcul et d'une capacité de stockage très importantes.

Les *grappes* de calculateurs, constituées d'un ensemble d'ordinateurs indépendants (les *nœuds*) interconnectés par un réseau et vues comme une ressource de calcul unique, constituent désormais des architectures de choix pour répondre aux besoins en terme de ressources de ces applications. Apparues au milieu des années 90, elles prédominent maintenant parmi les machines les plus puissantes du monde (cf. Top 500 [1]) et sont de plus en plus utilisées dans le monde industriel.

Par ailleurs, la tendance est également à l'augmentation de la taille de ces grappes et à la fédération de grappes. Or, plus le nombre d'équipements augmente, plus la probabilité de défaillance augmente elle aussi.

Dans ce contexte, la tolérance aux fautes est un sujet critique. En effet, certains calculs peuvent durer plusieurs heures, plusieurs jours voire plusieurs semaines. Il n'est pas souhaitable qu'en cas de défaillance d'un des ordinateurs de la grappe le calcul doive reprendre depuis le début. Dès lors, la disponibilité d'un mécanisme de reprise est indispensable. Actuellement, celui-ci est souvent pris en charge par le programmeur de l'application scientifique.

L'objectif de ce stage de Master de Recherche en Informatique est la conception d'un mécanisme, intégré au système d'exploitation pour grappes KERRIGHED, de tolérance aux fautes pour les applications parallèles communiquant par messages. L'originalité de ces travaux vient de la transparence recherchée vis-à-vis des applications et des bibliothèques de communication en intégrant le mécanisme au sein du système d'exploitation. La première partie du rapport porte sur les travaux préalablement existant permettant la restauration de processus ou d'applications à partir de points de reprise. La partie suivante présente le système d'exploitation KERRIGHED. La troisième partie explique les choix de conception réalisés. Puis la quatrième partie valide la mise en œuvre actuelle et analyse les résultats de tests d'expérimentation. Enfin, nous terminons par un bilan et une mise en évidence des perspectives.



# Chapitre 1

## État de l'art

### 1.1 Introduction

Dans cette étude, nous nous intéressons aux stratégies de sauvegarde et restauration de points de reprise d'applications parallèles fondées sur le modèle de communication par messages.

La première partie de cette étude bibliographique précise le contexte des travaux présentés dans la suite. La partie 2 compare différentes stratégies de sauvegarde et restauration de points de reprise d'applications parallèles communiquant par message. La troisième partie porte sur les techniques utilisées pour la détection des défaillances, la réalisation d'une mémoire stable et la sauvegarde de l'état d'un processus. Enfin, la quatrième partie présente une comparaison de différents logiciels de sauvegarde et restauration d'applications parallèles et s'applique à exposer les problèmes courants et les solutions apportées.

### 1.2 Contexte

Dans cette partie, nous présentons le contexte de l'étude. Dans un premier temps, nous introduisons la notion d'application parallèle communiquant par message. Nous exposons ensuite le modèle de fautes considéré dans les travaux étudiés.

#### 1.2.1 Applications parallèles communiquant par messages

Une application parallèle est composée de plusieurs processus communicants, chacun pouvant s'exécuter sur un ordinateur différent de la grappe.

Différents processus faisant partie d'une même application s'échangent des informations. Plusieurs techniques existent pour ce faire : la mémoire ou les variables partagées, et la communication par message.

La communication par message passe généralement par au moins deux primitives : envoi (`message`, `processus`) et réception (`message`). Ce modèle de programmation est utilisé par de très nombreuses applications de calcul scientifique dans le cadre de leur exécution sur grappe. C'est pourquoi, il est particulièrement intéressant de permettre à ces applications d'intégrer de manière transparente un mécanisme de reprise en cas de défaillance.

Depuis quelques années, plusieurs réalisations des environnements de communication par message ont pris en compte cette nécessité de reprise. Pour les deux principaux utilisés, les bibliothèques *PVM* (*Parallel Virtual Machine*) [30] et *MPI* (*Message Passing Interface* [2]), des implémentations sont proposées. Nous nous sommes donc intéressés plus particulièrement dans cette étude aux stratégies conçues dans ces environnements.

## 1.2.2 Tolérance aux fautes

L'objectif de la tolérance aux fautes est de permettre au système de continuer à fournir le service malgré l'occurrence de fautes. En fonction des contraintes applicatives, un système tolérant aux fautes doit être apte à supporter un nombre donné de fautes (qui peuvent être simultanées), et différents types de fautes.

Dans la suite de cette étude, le modèle de fautes considéré est le modèle *fail-Stop*. Dans ce modèle, soit un processus fonctionne correctement par rapport à ses spécifications, soit il est défaillant et s'arrête totalement de fonctionner. Les fautes byzantines, qui correspondent à une exécution non respectueuse vis à vis des spécifications, ne sont pas considérées dans la suite de l'étude.

Parmi les fautes à l'origine d'une défaillance, on distingue les fautes déterministes et les fautes non déterministes. Une faute est déterministe si celle-ci se produit systématiquement si l'on exécute le même programme avec les mêmes données en entrée. Celle-ci correspond à une erreur de programmation et est facilement reproductible. Une faute non déterministe est quant à elle difficile à reproduire. En effet, celle-ci n'est pas due à une erreur de programmation de l'application mais s'est produite à la suite d'un contexte d'exécution particulier. Une ré-exécution dans un contexte différent n'aboutit donc pas à cette même faute. La suite de cette étude porte sur la tolérance de ce type de fautes.

Des problèmes de communication peuvent se produire : des paquets réseaux peuvent être perdus, corrompus ou déséquilibrés. Les différents modèles étudiés posent des hypothèses sur la fiabilité des infrastructures de communication sous-jacente. La connexion réseau peut être interrompue suite à un problème sur la liaison physique. Lorsque la connexion réseau vers un nœud est perdue, la déconnexion est assimilée à une défaillance du nœud.

Il existe plusieurs stratégies de tolérance aux fautes. Le recouvrement arrière consiste à sauvegarder périodiquement l'état du système dans une mémoire stable. En cas de défaillance, l'état du système est restauré à partir du dernier état préalablement sauvegardé. Cette technique est particulièrement adaptée aux applications dans le domaine du calcul scientifique.

## 1.3 Sauvegarde et restauration de points de reprise d'applications parallèles

Comme indiqué en partie 1.2.1, depuis quelques années plusieurs études ont été réalisées pour l'implémentation de stratégie de sauvegarde et restauration pour les applications parallèles communiquant par messages. Nous nous intéressons dans cette partie aux différentes stratégies qui ont été proposées par les environnements *MPI* et *PVM*.

Toutes ces stratégies visent à recréer un état global cohérent lors de la reprise.

Un *état global* correspond à l'ensemble des états individuels des processus participants et des états des canaux de communication. Un *état global cohérent* est un état qui aurait pu se produire en l'absence de fautes, durant une exécution correcte d'une application parallèle. Ce n'est pas nécessairement un état qui s'est produit avant la reprise.

Dans un état global cohérent, si un processus possède un état local indiquant qu'un message  $m$  a été reçu, l'état du processus émetteur du message  $m$ , doit indiquer que le message  $m$  a été envoyé [11]. La figure 1.1 montre deux exemples d'états globaux, l'un cohérent, l'autre pas.

L'évènement  $b$  de réception d'un message sur un site  $P_i$  dépend causalement de l'évènement  $a$  d'émission de ce message par  $P_j$ . On dit que  $a$  précède causalement  $b$ . Sur un même site, tout évènement ultérieur à un évènement  $c$  dépend causalement de l'évènement  $c$ . La relation de dépendance causale est transitive. Un état global est cohérent si et seulement si il ne met pas en défaut de dépendances causales.

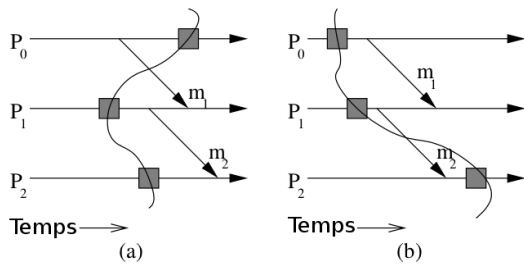


FIG. 1.1 – Une application communiquant par message constituée de trois processus. (a) montre un exemple d'état global cohérent où le message  $m_1$  est enregistré comme étant envoyé par le processus  $P_0$  mais non reçu par le processus  $P_1$ . (b) montre un exemple d'état global non cohérent où le message  $m_2$  est enregistré comme étant reçu par  $P_2$  mais non envoyé par  $P_1$ .

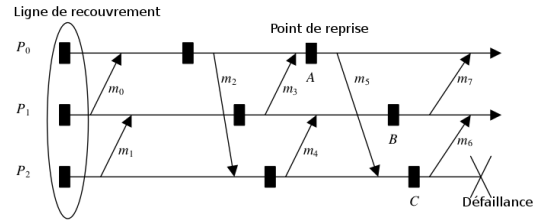


FIG. 1.2 – Points de reprise non coordonnés et retour arrière avec effet domino

Les mécanismes de reprise étudiés ici mettent tous en œuvre le recouvrement arrière. Celui-ci consiste à rétablir un processus défaillant dans un état local antérieur et à reproduire un état global cohérent. Toutes ces techniques, détaillées dans [13], se classent principalement en deux grandes familles de stratégie. La première s'appuie sur des points de reprises locaux, la deuxième est fondée sur la journalisation des messages.

### 1.3.1 Protocoles de recouvrement arrière à base de points de reprise

Lors d'une défaillance, les protocoles à base de points de reprise restaurent l'état global du système à partir de points de reprise locaux de chacun des processus formant l'état global cohérent le plus récent appelé ligne de recouvrement [13].

Pour former cette ligne de recouvrement, trois stratégies sont possibles : points de reprise non coordonnés, coordonnés, ou induits par les communications.

#### Points de reprise non coordonnés

La technique de sauvegarde non coordonnée de points de reprise vise à maximiser les performances en fonctionnement normal. L'objectif est de minimiser le surcoût lié à la sauvegarde des points de reprise lors d'une exécution sans fautes.

Chaque processus sauvegarde de manière indépendante son état local dans un point de reprise, il en conserve plusieurs. Durant l'exécution, les messages sont accompagnés d'une estampille temporelle qui permet de percevoir des dépendances causales entre les états des différents processus [18].

Lors de la reprise, la ligne de recouvrement est calculée à l'aide des estampilles temporelles. Le processus défaillant recouvre à partir de l'un de ses points de reprises. Les autres processus peuvent ou non être amenés à faire un recouvrement arrière, en fonction des dépendances causales.

Cette technique a comme principal avantage que chaque processus sauvegarde son point de reprise quand cela est le plus avantageux pour lui-même. Ainsi, il est possible pour un processus de faire cette opération quand la taille des informations sur son état est petite, minimisant ainsi la taille du point de reprise et limitant la dégradation des performances lors d'une exécution sans fautes.

Cependant, cette technique possède différents inconvénients. Premièrement, le calcul de la ligne de recouvrement peut s'avérer coûteux, surtout si le nombre de processus est grand et les communications importantes. De plus, il y a un risque d'effet domino qui a pour conséquence de ramener l'application dans son état initial, perdant ainsi tout le calcul déjà effectué. La figure 1.2 présente une exécution entraînant un tel effet. Ainsi, compte-tenu des dépendances causales entre processus et des points de reprise sauvegardés, la ligne de recouvrement est ici l'état initial. D'autre part, pour chaque processus, il est nécessaire de conserver plusieurs points de reprise, ce qui entraîne un surplus d'espace de stockage.

D'une manière générale, cette technique est peu utilisée dans les environnements *MPI* et *PVM*. Quelques systèmes la proposent en complément d'une autre, c'est le cas par exemple de *Starfish* [4].

### Points de reprise coordonnés

La technique de sauvegarde coordonnée des points de reprise vise la simplicité de mise en œuvre et l'assurance d'obtenir un état global cohérent lors de la sauvegarde.

Cette approche repose sur une coordination globale de tous les processus de l'application. Un processus, le coordinateur (désigné dynamiquement ou statiquement) sauvegarde son point de reprise et diffuse un message *CKPT* demandant à tous les autres processus de l'application d'établir leur point de reprise. Quand un processus reçoit le message, il stoppe son exécution, vide ses canaux de communication, prend un point de reprise provisoire, et envoie un message d'acquiescement au coordinateur. Une fois que le coordinateur a reçu l'ensemble des acquiescements, celui-ci diffuse un message de validation du point de reprise. Chaque processus remplace alors, de manière atomique, son ancien point de reprise permanent par le point de reprise provisoire.

Lors de la reprise, il suffit de restaurer l'ensemble des processus de l'application à partir de leur point de reprise respectif.

Cette technique assure que l'ensemble des points de reprise forme un état global cohérent et évite ainsi l'effet domino. De plus, la reprise est très simple. Un autre avantage de cette technique est que l'espace de stockage nécessaire pour conserver les points de reprise est minimisé puisqu'il n'est nécessaire que d'en conserver un seul par processus.

Le principal inconvénient de cette stratégie est la latence importante qu'elle implique lors de la sauvegarde du point de reprise, étant donné qu'elle nécessite une coordination globale de tous les processus. Deux problèmes se posent donc ici, la perte de performance y compris en l'absence de fautes, et la gestion du passage à l'échelle. En effet, plus le nombre de processus augmente, plus la latence risque d'être importante, chaque processus devant attendre la fin de l'opération avant de pouvoir reprendre son exécution normale.

Cette approche, utilisée par les premières mises en œuvre [19, 4], peut être optimisée. Ainsi, il est possible d'utiliser un algorithme de coordination non bloquant [13] comme celui de *Chandy-Lamport* [11]. Lors de la réception du message *CKPT*, le processus récepteur crée son point de reprise provisoire et ré-émet le message *CKPT* vers tous les autres processus avant de reprendre son exécution normale et l'envoi de messages de l'application. En présence de canaux de communication FIFO, cette technique assure qu'aucun message de l'application ne soit traité par un processus *A* entre l'émission par le coordinateur du message *CKPT* et la réalisation du point de reprise par le processus *A*.

Une autre optimisation possible est l'utilisation d'une coordination partielle dynamique [17]. Il faut pour cela prendre en compte les communications entre processus pour définir dynamiquement l'ensemble des processus concernés par la sauvegarde d'un point de reprise. Seuls les processus ayant communiqué avec l'initiateur d'un point de reprise directement ou indirectement depuis le dernier point de reprise doivent sauvegarder un point de reprise.

La technique des points de reprise coordonnés, parce qu'elle est simple et malgré tout relativement

efficace, est utilisée par de nombreux protocoles [19, 29, 27, 34, 9].

### **Points de reprise induits par les communications**

La technique des points de reprise induits par les communications [5] a pour idée d'initier les points de reprise en fonction des messages émis et reçus entre les processus. Aucun message spécifique de coordination n'est envoyé, mais les points de reprise sont réalisés lors d'évènements particuliers.

Dans chaque message de l'application, des informations du protocole sont encapsulées. À partir de ces informations encapsulées qui contiennent des estampilles temporelles sur les messages envoyés et les points de reprises créés, le processus récepteur du message décide s'il doit ou non créer un point de reprise. Il faut noter que certains points de reprise peuvent également être pris de manière indépendante mais ces derniers ne garantissent pas la progression de la ligne de recouvrement.

La reprise se passe de la même manière que pour la stratégie des points de reprise non coordonnés.

Les techniques de sauvegarde de points de reprise induits par les communications évitent l'effet domino tout en ne nécessitant pas la coordination de tous les processus de l'application.

La difficulté tient ici dans la mise en œuvre de la méthode de décision quant à la nécessité de créer un point de reprise. De plus, si les messages de l'application sont très petits, l'encapsulation dans chaque message des informations du protocoles entraîne un surcoût qui peut être important.

Cette technique ne semble pas avoir été mise en pratique dans le cadre des environnement *MPI* et *PVM*.

### **1.3.2 Protocoles fondés sur la journalisation des messages**

Un protocole à journalisation est fondé sur le fait qu'un état peut être reconstruit en rejouant les messages ayant été échangés entre le dernier point de reprise et la défaillance.

Dans le cas idéal, seul le processus défaillant doit donc être restauré, les autres processus peuvent continuer leur exécution normalement. Cependant un mécanisme sur chaque nœud doit pouvoir prendre en charge la ré-émission des messages.

Dans le cadre d'un protocole à journalisation des messages, la sauvegarde de points de reprise n'est pas obligatoire mais est fortement conseillée puisqu'elle permet de diminuer le nombre de messages journalisés. Par la suite, nous considérerons donc uniquement les protocoles à journalisation des messages qui utilisent en complément une sauvegarde non coordonnée de points de reprise locaux.

Les protocoles à journalisation se classent en plusieurs catégories que nous détaillons par la suite : journalisation pessimiste, optimiste et causale.

#### **Journalisation pessimiste**

La journalisation pessimiste repose sur l'hypothèse suivante : une défaillance peut se produire juste après un envoi ou une réception de message.

Tout les messages émis doivent être conservés au cas où il est nécessaire de les rejouer. Deux techniques de stockage peuvent ici s'affronter, le serveur centralisé [6] ou la conservation des messages par l'émetteur des messages [8]. Pour tout message, il faut savoir si celui-ci a été reçu. On associe à chaque message un identifiant et une estampille temporelle pour les archiver en mémoire stable dès la réception du message avant qu'il ne puisse être exploité par l'application. Ces informations constituent le journal qui permet de connaître dans quel ordre et par quel processus ont été reçus les messages.

Lors de la reprise, le processus défaillant est restauré à partir du dernier point de reprise et les messages lui ayant été envoyés ultérieurement sont rejoués.

Cette technique propose un retour arrière très limité et ne demande de conserver qu'un seul point de reprise. De plus, la restauration est simple, tout du moins, s'il n'y a qu'une seule faute simultanée. Cependant, la nécessité de stocker l'acquittement de réception des messages de l'application en mémoire stable entraîne une latence importante et donc une perte de performance. D'autre part, le stockage des messages déjà émis peut s'avérer coûteux en espace.

La technique de la journalisation pessimiste est utilisée dans les implémentations *MPICH-V* [6] et *MPICH-V2* [8].

### **Journalisation optimiste**

La journalisation optimiste est fondée sur l'hypothèse que la journalisation se termine avant l'occurrence d'une défaillance.

À la différence de la journalisation pessimiste, le journal n'est pas écrit directement en mémoire stable mais est d'abord écrit en mémoire volatile pour être vidé périodiquement vers la mémoire stable.

Le calcul de la ligne de recouvrement est plus compliqué que pour la journalisation pessimiste. En effet, comme le journal est partiellement écrit, certains événements ne peuvent être rejoués et il peut être nécessaire de restaurer plusieurs processus de l'application alors qu'un seul est défaillant.

Cette technique a pour avantage un faible surcoût en exécution sans faute. En effet, celle-ci n'est pas affectée par la latence produite à chaque message en journalisation pessimiste par l'écriture du journal en mémoire stable. Toutefois, les inconvénients de cette approche sont majeurs : la reprise est délicate et aucune garantie n'est fournie quant à la possibilité de retrouver un état global cohérent à partir du dernier point de reprise. Il faut donc en conserver plusieurs, d'où un surcoût en espace disque.

Cette technique ne semble pas avoir été mise en pratique dans le cadre des environnements *MPI* et *PVM*.

### **Journalisation causale**

La dernière catégorie est la journalisation causale qui essaie de combiner les avantages de la journalisation optimiste (faible surcoût en fonctionnement normal) et de la journalisation pessimiste.

Le journal est sauvegardé en mémoire volatile mais la décision de l'écrire en mémoire stable est prise à partir des dépendances causales entre les événements.

Cette stratégie, complexe à mettre en œuvre, élimine le problème de la latence en journalisation pessimiste et permet un recouvrement limité au pire au dernier point de reprise de chaque processus défaillant.

*MPICH-VCausal* [9] est l'un des seuls environnements *MPI* à journalisation causale.

### **1.3.3 Comparaison des différentes stratégies**

Le tableau 1.1 présente un résumé des implications liées aux différentes stratégies de recouvrement arrière.

Toutes ces techniques différentes offrent donc des services différents. Cependant, de nombreux concepts sont partagés. Le projet *Egida* [26] a pour objectif de créer des environnements *MPI* tolérants aux fautes. Par rapport à d'autres projets, son intérêt réside dans la possibilité de construire n'importe lequel des protocoles vus précédemment au dessus de l'architecture présentée en figure 1.3. On voit ainsi sur la figure le découpage en modules. Chaque module est responsable de plusieurs tâches et offrent l'accès à

	Points de reprise			Journalisation		
	Non coordonnés	Coordonnés	Induits par communications	Pessimiste	Optimiste	Causale
Effet domino	Possible	Non	Non	Non	Non	Non
Nb PR par processus	Plusieurs	1	Plusieurs	1	Plusieurs	1
Étendu du retour arrière	Illimité	Dernier PR global	Plusieurs PR possibles	Dernier PR	Plusieurs PR possibles	Dernier PR
Protocole de reprise	Distribué	Distribué	Distribué	Local	Distribué	Distribué

TAB. 1.1 – Comparaison de différentes stratégies de recouvrement arrière

des méthodes. Plusieurs implémentations de ces méthodes peuvent être réalisées en fonction des besoins. *Egida* est prévu pour fonctionner avec *MPICH* mais son architecture le rend intéressant à étudier même si l'objectif est l'implémentation d'un mécanisme de sauvegarde/restauration à un autre niveau.

L'équipe Grand Large du Laboratoire de Recherche en Informatique a comparé l'efficacité des protocoles à points de reprises coordonnés et ceux à journalisation [9]. Cette étude a été réalisée en opposant *MPICH-V2*, *MPICH-VCausal* et *MPICH-VCL* sur les mêmes ordinateurs exécutant les mêmes programmes. *MPICH-V2* utilise la journalisation pessimiste avec journalisation des messages au niveau de l'émetteur. *MPICH-VCL* implémente l'algorithme de points de reprise coordonnés de Chandy-Lamport, tandis que *MPICH-VCausal* utilise la journalisation causale. Les résultats, présentés sur la figure 1.4 [7] montre l'efficacité des points de reprise coordonnés lorsque des fautes ne se produisent pas trop souvent.

Les auteurs concluent que les points de reprise coordonnés sont à préférer si l'on considère des grappes pour lesquelles le *MTBF* (*Meant Time Before Failure*) est supérieur à 9 heures. Ce chiffre est évidemment à nuancer en fonction du nombre et des capacités des ordinateurs utilisés et de la taille des données manipulées (ici 1 Go) mais il reste très faible.

## 1.4 Mécanisme de base pour la mise en œuvre d'une stratégie de recouvrement arrière

Cette partie présente les différents mécanismes de base qui sont nécessaires à l'implémentation d'un mécanisme de sauvegarde/reprise pour les applications parallèles.

Ces mécanismes sont la détection des défaillances, la réalisation d'une mémoire stable et la sauvegarde d'un processus.

### 1.4.1 Technique de détection des défaillances

La première étape d'une stratégie de recouvrement arrière est la détection de la défaillance d'un processus.

En l'absence de communication en provenance d'un processus pendant une durée  $t$ , celui-ci est considéré potentiellement défaillant. Un temporisateur est amorcé et un message spécial est envoyé à



FIG. 1.3 – L'architecture d'Egida [26]

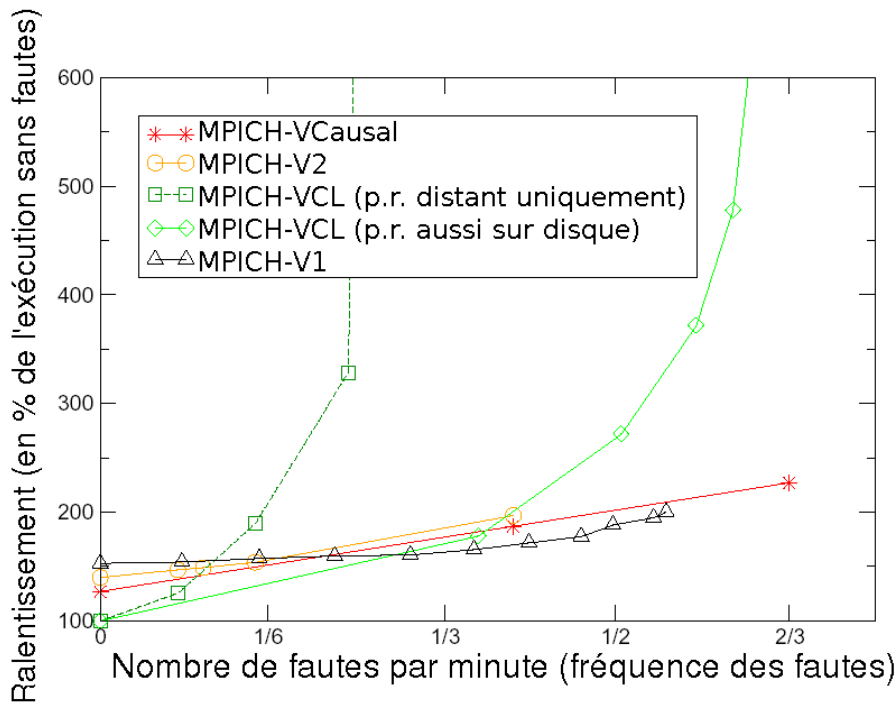


FIG. 1.4 – Comparaison de performances entre différents protocoles coordonnés ou journalisés

ce processus. En l'absence de réponse avant la fin du temporisateur, le processus est jugé défaillant et le recouvrement arrière peut commencer.

Cette technique simple n'est pas optimale, puisqu'elle a pour inconvénient de distinguer difficilement un processus lent d'un processus mort. D'autres techniques, comme le protocole *HeartBeat*, existent pour pallier ce type de problèmes et sont présentées dans [10].

Dans le cadre d'applications communiquant par messages, cette détection peut en pratique être réalisée de manière distribuée ou centralisée, comme c'est le cas dans les systèmes *MPI-FT* [22] et *MPICH-V\** [6, 8].

#### 1.4.2 Réalisation d'une mémoire stable pour le stockage des points de reprise

Pour supporter les fautes, les points de reprise doivent être sauvegardés sur un support stable.

Celui-ci doit être accessible quelque soit les défaillances, et les données ne doivent pas être altérées. Cela suppose une redondance des données à conserver. Par ailleurs, lors de la création d'un nouveau point de reprise, le précédent ne doit être invalidé qu'après finalisation du nouveau. Ainsi, s'il y a échec au cours de la sauvegarde d'un point de reprise, le précédent est toujours disponible et considéré comme point de reprise de référence. On dit qu'il y a atomicité des mises à jour des points de reprise.

En fonction du nombre de fautes à tolérer, des performances et des moyens disponibles, plusieurs solutions sont possibles pour réaliser une mémoire stable.

La solution la plus simple, l'utilisation d'un serveur considéré comme fiable, est utilisée notamment par *MPICH-V\** [6, 8]. Cette technique transforme le serveur en un goulot d'étranglement du réseau, particulièrement si les points de reprise sont calculés de manière coordonnée. Le passage à l'échelle est rendu difficile et il est parfois nécessaire de dupliquer le serveur, le système perdant alors une bonne

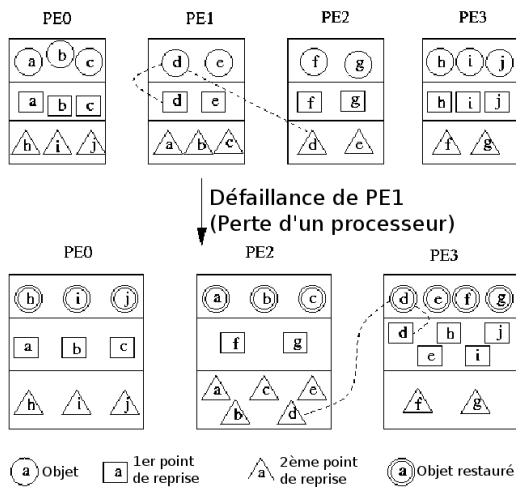


FIG. 1.5 – Double in-memory checkpointing

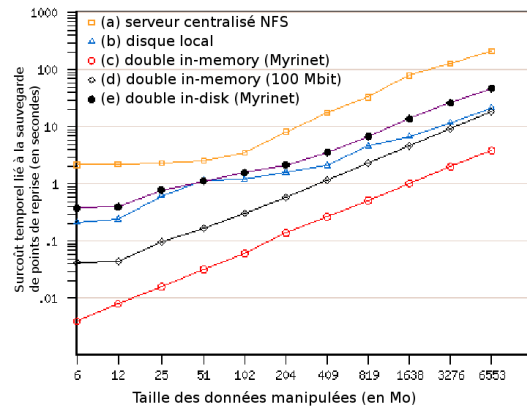


FIG. 1.6 – Comparaison de performances entre différentes techniques de mémoire stable - Utilisation de FTC-Charm++[34]

partie de sa simplicité.

Une autre solution est la sauvegarde du point de reprise dans la mémoire vive d'un nœud distant. Il n'y a pas ici de redondance des points de reprise, ceci ne constitue donc pas réellement une mémoire stable. Cette solution suppose qu'il est peu probable que le nœud de calcul et celui conservant son point de reprise subisse une défaillance simultanée. Ceci ne permet de résister qu'à une seule faute mais évite la saturation d'un serveur. L'autre inconvénient de ce système est la forte augmentation de l'utilisation mémoire. Pour éviter ce surplus d'utilisation mémoire, une variante consiste à sauvegarder le point de reprise sur un disque local d'un nœud distant. Le défaut est alors le coût en latence dû aux accès en écriture sur disque.

FTC-Charm++ [34] utilise le *double checkpointing* (voir figure 1.5). L'idée est de sauvegarder les points de reprise sur deux nœuds différents, offrant ainsi une meilleure résistance aux fautes. Ce système propose le choix entre le *double in-memory checkpointing* (sauvegarde dans les mémoires vives locales de deux nœuds différents) et le *double in-disk checkpointing* (sauvegarde sur les disques locaux de deux nœuds différents). Cette dernière technique a l'avantage de pas accroître considérablement la mémoire vive utilisée comme le fait le *double in-memory checkpointing*. Elle est donc à favoriser lorsque les applications manipulent de gros volumes de données.

La figure 1.6, réalisée dans le cadre de l'étude de performance de FTC-Charm++ [34], présente une comparaison de performance entre différentes techniques de mémoire stable. Cette comparaison a été réalisée sur une grappe de 32 nœuds exécutant le benchmark *Jacobi3D*. La première constatation est que quelque soit la solution utilisée, le surcoût en temps lié à la sauvegarde augmente linéairement avec la taille des données manipulées par l'application. Par ailleurs, on remarque les différences de performances liées aux réseaux utilisés. Malheureusement, aucune précision n'est donnée sur le réseau utilisé lors de la sauvegarde sur le serveur centralisé *NFS* (a). La figure montre l'avantage considérable en terme de performance qu'offre le mécanisme de *double in-memory* par rapport aux autres solutions. Il ne faut cependant pas oublier le surcoût en mémoire qui s'y attache et qui rend cette solution inutilisable lorsque la taille des données manipulées devient très importante.

### 1.4.3 Point de reprise d'un processus

Cette partie s'intéresse à la sauvegarde de l'état d'un processus. La sauvegarde d'un processus a pour objectif de pouvoir relancer ce dernier dans l'état dans lequel il était lors de la sauvegarde. Nous nous

intéressons tout d'abord aux besoins des utilisateurs. Ensuite, les différentes stratégies de sauvegardes sont présentées et comparées. La dernière partie expose les contraintes techniques liées à la création d'un point de reprise d'un processus.

#### 1.4.3.1 Besoins des utilisateurs

Le premier besoin utilisateur consiste en la définition de l'état du processus. Cette définition doit être suffisamment complète pour qu'un arrêt du processus suivi d'une reprise préserve toutes les caractéristiques du processus et permettent notamment le rétablissement des communications et des entrées/sorties.

Pour réaliser un point de reprise, deux possibilités sont offertes au programmeur. La première consiste à fournir à ce dernier des primitives de programmation qu'il faut insérer dans le code de l'application. La deuxième est de lui permettre de se reposer entièrement sur le système pour initier les points de reprise. Cette dernière solution offre la transparence et rend facile l'utilisation de points de reprise. Dans la suite de cette étude, nous ne nous intéressons donc qu'à cette possibilité.

La fiabilité de cette sauvegarde tient à son atomicité. Ainsi, un point de reprise ne doit en aucun cas être supprimé avant qu'un point de reprise ultérieur ne soit initié et finalisé.

Dans le cadre de notre étude, nous nous intéressons aux applications parallèles communiquant par messages. Ces applications ont des contraintes fortes en terme de performance, la réalisation du point de reprise doit donc se faire de manière efficace. Ainsi, les points de reprises incrémentaux sont préférés aux points de reprise complets. Un point de reprise incrémental contient uniquement les données mises à jour depuis le dernier point de reprise antérieur. Pour opérer une restauration, il est alors nécessaire de disposer de tous les points de reprise ultérieurs au dernier point de reprise complet réalisé. Par ailleurs, la création du point de reprise doit bloquer le processus le moins longtemps possible et donc être réalisée en arrière-plan.

#### 1.4.3.2 Niveau de mise en œuvre des points de reprise de processus

Les points de reprise peuvent être implémentés de trois manières : par l'application elle-même, par une librairie liée à l'application, ou par le système d'exploitation.

La sauvegarde au niveau applicatif consiste en l'insertion, au sein de l'application, de code permettant la réalisation des points de reprise. Cette technique promet les meilleurs performances. En effet, le programmeur sait parfaitement quelles données sauvegarder. Les inconvénients de la sauvegarde au niveau applicatif sont cependant nombreux. Premièrement, il est nécessaire de modifier le code source, ce qui n'est pas forcément possible. De plus, les sauvegardes se font à des moments précis dans l'exécution du programme. Ainsi, il paraît délicat d'effectuer une sauvegarde au milieu d'une boucle. Dès lors, les sauvegardes risquent d'être espacées, perdant beaucoup de leur intérêt. Enfin, en plus de programmer la génération des points de reprise, le programmeur doit programmer également leur restauration.

Une librairie liée à l'application peut initier les points de reprise. Dans le cadre des bibliothèques *PVM* et *MPI*, l'interface de programmation n'est pas modifiée. Les points de reprise sont alors sauvegardés par la librairie lors des appels de fonctions. Les implémentations au niveau librairie posent aussi quelques problèmes. Ainsi, s'il n'y a pas besoin de modifier le code source, il est nécessaire de réeffectuer l'édition de liens comme c'est le cas avec *Condor* [31]. La contrainte principale est le fait que l'utilisation d'une telle librairie limite les appels systèmes que l'application peut utiliser. Il en résulte que bon nombre d'applications ne peuvent être sauvegardées.

La dernière solution, est l'intégration dans le système d'exploitation de la génération des points de reprise. Le système d'exploitation peut fournir une primitive d'initiation des points de reprise ou les initier de manière transparente. Dans ce dernier cas, aucune modification des applications ou bibliothèques n'est nécessaire. Les mises en œuvre système suppriment les limitations des approches précédentes. Le

noyau a en effet accès à toutes les structures de données représentant l'état du processus. Dès lors, il y a très peu de restrictions sur les applications qui peuvent être restaurées. Cependant, peu d'implémentations ont été réalisées, compte-tenu de la complexité de la mise en œuvre de points de reprise dans le système d'exploitation. Parmi ces implémentations, distinguons *CRAK* [35] et *BLCR* [12].

### 1.4.3.3 Contenu à sauvegarder dans un point de reprise

Dans la suite de cette partie, il est admis que le point de reprise est pris au niveau système permettant ainsi le plus de flexibilité.

#### Sauvegarde générale

Pour réaliser un point de reprise, il est nécessaire d'extraire l'état du processus. Cet état doit également être extrait pour les fonctionnalités de duplication de processus et migration de processus. C'est pourquoi, les différentes implémentations de ces fonctionnalités au niveau système utilise la notion de virtualisation de processus.

Un fichier de point de reprise doit contenir tout ce qui est nécessaire pour redémarrer le processus, éventuellement sur un autre nœud. Il faut donc normalement conserver le tas, la pile, le code et l'état des registres dans le système d'exploitation. Il est également nécessaire de sauvegarder le répertoire courant, les comptes à rebours, l'identité de l'utilisateur (*uid*, *gid*, etc.), celle du processus (*pid*, *pgrp*, *ppid*, etc.), l'arborescence du processus. Le mécanisme de *Ghost Process*[33] mis en œuvre dans *Kerrighed* permet l'extraction de toutes ces informations.

Une première optimisation peut être faite en proposant de ne pas sauvegarder le code puisque généralement, celui-ci est déjà disponible sur le disque.

De la même manière, il faut conserver les bibliothèques partagées liées à l'application. Cependant, celles-ci sont généralement présentes sur le disque du nœud de reprise, il n'est donc pas obligatoire de les sauvegarder.

Ces deux optimisations mises en œuvre dans *CRAK* [35] et *Epckpt* [25], permettent d'économiser du temps et de l'espace.

La création du point de reprise pouvant avoir lieu à n'importe quel moment au cours de l'exécution du processus, il est nécessaire de conserver l'état des signaux et du gestionnaire de signaux. Le mécanisme de *Ghost Process* [33] propose de rajouter un état au noyau. Cet état supplémentaire est accessible après le traitement des signaux, ce qui simplifie cette gestion.

#### Gestion des entrées/sorties

La gestion des entrées/sorties est l'un des problèmes délicats pour la reprise d'un processus.

La gestion des fichiers ouverts en fait partie. Ainsi, il ne suffit pas de savoir quel fichier était ouvert lors du point de reprise, mais il s'agit bien de ramener le fichier dans son état lors de la restauration. Pour les fichiers ouverts en écriture, cela nécessite donc de sauvegarder ceux-ci alors que pour ceux ouverts en lecture il suffit les ouvrir à nouveau.

Lors de la sauvegarde d'un point de reprise d'un processus, des sockets de communication peuvent être ouverts. À la reprise, il est donc nécessaire de les restaurer. Le problème est alors que pour le processus qui ne recouvre pas, la socket est sensée n'avoir jamais été fermée. *CRAK* [35] propose la solution suivante. Du côté du processus encore en exécution, la socket est placée dans un état `TCP_TIME_WAIT` pour que celle-ci ne soit pas complètement fermée. Les structures de données du noyau correspondant

au socket peuvent alors être mises à jour (adresse de l'extrémité, etc.). Après cela, il suffit de replacer la socket dans son état normal.

Un tube (*pipe*) est un canal de communication permettant à deux processus de communiquer entre eux. Ce tube est reconnu par chaque processus comme un fichier. L'un des processus écrit dans le fichier, le deuxième lit le contenu du fichier. Le contenu du tube (files d'attente d'entrée/sortie) doit être sauvegardé lors de la création d'un point de reprise. Lorsque l'un des deux processus recouvre, il est nécessaire de créer un nouveau tube et le descripteur de fichier correspondant doit être mis à jour à l'autre extrémité. Cependant, l'arrêt de l'un des processus provoque normalement un *broken pipe*. Pour sauvegarder l'état des deux processus communiquant de manière correcte, il faut synchroniser leur sauvegarde respective. *CRACK* [35] utilise des signaux pour réaliser cette synchronisation.

Chaque processus est attaché à un terminal : une entrée standard et une sortie standard. Lors de la reprise, ce terminal peut exister ou pas. Il faut donc s'en assurer pour réaliser le recouvrement et mettre à jour les structures de données correspondantes. Il est important de souligner que certains processus paramètrent le mode du terminal. Il est essentiel de le sauvegarder également.

Tous ces mécanismes sont relativement complexes à gérer. Dès lors, les mécanismes de sauvegarde ne les prennent pas tous en charge. Ainsi, *BLCR* [12] ne prend pas en charge les tubes.

## 1.5 Mise en œuvre de la sauvegarde/restauration dans les environnements *MPI* et *PVM*

Le tableau 1.2 propose une synthèse du mode de fonctionnement de différents environnements *MPI* et *PVM* intégrant la tolérance aux fautes. Certaines implémentations n'ont pas été intégrées à ce tableau car elles sortaient du cadre de cette étude. Ainsi, *MPI-FT* [22] ne s'appuie pas sur un mécanisme de recouvrement arrière mais réalise de la duplication de processus pour réaliser la tolérance aux fautes. *FT-MPI* [14] a lui aussi été écarté. En effet, celui-ci propose aux développeurs de détecter les défaillances et de s'en affranchir par eux-même. Le mécanisme de tolérance aux fautes choisi n'est donc pas le recouvrement arrière.

Dans *PVM* [30], les processus de l'application communiquent et se synchronisent par l'intermédiaire de processus démons. Chaque ordinateur du système distribué héberge un de ses processus démons. Les démons coordonnent toutes les opérations globales et les processus de l'application ne réalisent que le calcul local.

*Fail-safe PVM* [19] est la seule implémentation étudiée ici décrivant comment elle s'intègre au dessus de *PVM* [30]. La détection de la défaillance est réalisée par un processus démon intégré dans les démons *PVM* existants et utilisant l'infrastructure de communication inter-démons déjà existante. La création d'un point de reprise global nécessite la prise en compte des canaux de communications. L'utilisation de processus démons dans *PVM* facilite cette gestion. Ainsi, à chaque fois qu'un processus de l'application est bloqué en attente de réception d'un message ou de finalisation d'une étape de synchronisation, celui-ci attend une réponse du processus démon. Le démon est donc un emplacement privilégié pour stopper les opérations en cours (vider les canaux de communications) avant de réaliser une sauvegarde locale. Tout ceci est possible car *PVM* assure des communications fiables et FIFO.

L'un des premiers problèmes posés est le mode choisi pour détecter les défaillances et comment est décidé la nécessité de recouvrir un processus ou pas. Deux possibilités sont envisageables : centralisé ou distribué. Un processus centralisé détermine si un processus est défaillant, décide son recouvrement si besoin est et affecte le processus restauré à un nœud. Cette technique est utilisée par *MPICH-V\** [6, 8, 9]. Dans le cas distribué, lorsqu'un processus détecte la défaillance d'un autre nœud, soit il se

	Failsafe PVM	CoCheck	Starfish	MP(CH-V1)	MP(CH-V2)	MP(CH-VCausal)	MP(CH-VQ)	LAM-MPI	Ediga	FTC-Charm++
<b>Environnements disponibles</b>										
PVM	x	x								
MPI		x	x	x	x	x	x	x	x	x
<b>Type de mémoire stable</b>										
			?							
centralisée				x	x	x	x		possible	
distribuée (sur disque)	x								possible	x
distribuée (dans mémoire vive)									possible	x
locale		x						?	possible	
<b>Sauvegarde d'un processus</b>										
	?	?	?						?	Applicatif
librairie Condor				x	x	x	x			
BLCR								x		
<b>Points de reprise</b>										
coordonnés	x	x	x				x	x	possible	x
non coordonnés	x		x					?	possible	
induits par les communications									possible	
<b>Journalisation</b>										
pessimiste				x	x				possible	
optimiste									possible	
causale						x			possible	
<b>Nb de fautes supportées</b>										
	1	Plusieurs	Plusieurs	Plusieurs	Plusieurs	Plusieurs	Plusieurs	Plusieurs	Au choix	Plusieurs
<b>Transparence</b>										
Modification du code source			x	x	x	x	x	x		x
Édition de liens	x	x		x	x	x	x		x	

TAB. 1.2 – Comparatif de différents environnements *MPI* et *PVM* tolérant aux fautes

charge lui-même de restaurer le processus, soit un protocole de consensus est alors lancé pour dégager un responsable.

Les sauvegardes de points de reprise coordonnés peuvent être initiées par une méthode centralisée ou distribuée. Dans *CoCheck* [29], l'initiateur de la sauvegarde est ainsi un processus fiable et assigné statiquement. La méthode distribuée est tout à fait utilisable et le cas de plusieurs processus différents décidant l'initiation d'un point de reprise global de manière quasi-simultanée est géré par le protocole de Chandy-Lamport [11].

Il est intéressant de noter que même pour les protocoles à journalisation, l'initiation d'un point de reprise peut être suggéré par un processus distant. Ceci peut se produire lorsqu'un processus n'a plus suffisamment de place pour maintenir différents points de reprise, il s'agit du mécanisme de ramasse-miettes. Pour assurer une progression de la ligne de recouvrement, ce dernier va alors s'assurer que les autres processus ont réalisé un point de reprise compatible avec le dernier réalisé. Cette fonction peut également être accomplie par un processus centralisé comme c'est le cas dans *MPICH-V2* [8].

Pour les protocoles uniquement à base de points de reprise non coordonnés, il est également possible de réaliser des points de reprise à partir d'une synchronisation lâche (horloge du système par exemple), afin de diminuer les risques d'effet domino.

Les processus distants échangent des messages par les canaux de communication. Comme un état global cohérent doit contenir l'état de ses canaux, une gestion spécifique doit être mise en place, et les processus applicatifs ne doivent pas communiquer directement. Une première possibilité, utilisée dans *MPICH-VI* [6], est d'envoyer le message vers une mémoire stable, celle-ci va ensuite le transmettre au destinataire réel. L'autre solution est de faire passer les communications par un processus démon à chaque extrémité. Les files d'attente de ce démon sont alors sauvegardées lors de la création du point de reprise du processus applicatif. Ce démon est en charge de la journalisation des messages émis dans le cas des protocoles journalisés comme *MPICH-V2* [8]. Dans le cadre des protocoles à point de reprise coordonnés, c'est grâce à ce démon que s'effectuent les diverses étapes de synchronisation et le vidage des canaux précédant la sauvegarde du point de reprise local.

Le processus de recouvrement peut démarrer alors même qu'il reste des messages en transit ou dans les files d'attente. Il est important de ne pas confondre ces messages avec les messages envoyés par le(s) processus restauré(s). Une estampille spéciale doit être ajoutée. Celle-ci est changée à chaque fois qu'une nouvelle défaillance a lieu. Ainsi, il suffit ensuite au démon d'ignorer les messages n'ayant pas la bonne estampille. Cette solution est notamment utilisée dans *FTC-Charm++* [34].

Le démon permet également une virtualisation des adresses, permettant ainsi de restaurer un processus sur un nœud différent du nœud d'origine. Pour cela, une simple table de correspondance doit être mise à jour au niveau de chaque nœud.

Lors de la création d'un point de reprise local, celui-ci est envoyé vers une mémoire stable. Celle-ci peut être locale, distribuée, ou centralisée. L'utilisation d'une mémoire stable centralisée en collaboration avec des points de reprise coordonnés est a priori une mauvaise idée. Il apparaît en effet que les communications réseau vers la mémoire stable sont ponctuelles (uniquement au moment de la sauvegarde) et dans ce cas, elles sont simultanées, l'accès à la mémoire devenant alors un goulot d'étranglement. C'est pourtant la combinaison utilisée par *MPICH-VCL* [9].

## 1.6 Conclusion

Cette étude bibliographique nous a permis de dresser un état de l'art des travaux de sauvegarde et restauration de points de reprise d'applications.

La plupart des mises en œuvre sont réalisées dans les environnements *MPI*. Elles utilisent généralement la coordination globale qui offre de bons résultats [9] bien que d'autres stratégies ont également été

implémentées malgré une complexité accrue et des gains en temps d'exécution pas forcément évidents.

On constate qu'un faible support est offert dans le système d'exploitation, uniquement pour la création de point de reprise d'un processus séquentiel [33, 35]. Le projet BLCR[12] constitue une exception mais seule la sauvegarde de l'état d'un processus est réalisée dans le système d'exploitation, le reste étant à la charge des couches supérieures [27].

Pour la mémoire stable, l'utilisation de la distribution en mémoire est la technique qui offre le meilleur compromis performance/sûreté en comparaison des autres solutions. Celle-ci a cependant l'inconvénient majeur de consommer une grande quantité de mémoire.

Il paraît important d'utiliser une approche modulaire comme celle du projet Egida [26] pour permettre la mise en place des différents protocoles permettant d'obtenir un état global cohérent après une ou plusieurs fautes. Nous constatons en effet que des blocs de base communs sont utilisables pour la gestion des estampilles notamment.

## Chapitre 2

# Le système d'exploitation KERRIGHED

Nous présentons dans ce chapitre le système KERRIGHED [23] qui a servi de base à notre étude.

KERRIGHED est un système d'exploitation pour grappe de calculateurs qui vise à donner l'illusion qu'une grappe est une machine unique. Un tel système est qualifié de système à image unique (*Single System Image - SSI*). Celui-ci est construit comme une extension d'un système d'exploitation LINUX (sous forme d'un ensemble de modules et de quelques modifications du noyau).

Ce système a été conçu et réalisé dans le cadre de l'activité de recherche KERRIGHED dirigée par Christine Morin au sein du projet de recherche PARIS de l'IRISA/INRIA Rennes.

### 2.1 Système à image unique

KERRIGHED est un système à image unique (*Single System Image - SSI*). Il vise à donner l'illusion à l'utilisateur d'un seul calculateur multiprocesseur à mémoire partagée (*SMP*) au-dessus d'un ensemble d'ordinateurs standard reliés par un réseau à haut débit et faible latence.

L'intérêt est de permettre la programmation d'application pour grappe de la même manière que pour les calculateurs SMP qui ont un coût de revient supérieur et sont peu évolutifs.

Un système à image unique doit donc offrir à l'utilisateur et au programmeur d'application les mêmes services que ceux disponibles sur un système multiprocesseur. Le système doit supporter les applications parallèles à mémoire partagée — ceci est rendu possible grâce à l'implémentation d'une mémoire virtuelle partagée (*Distributed Shared Memory - DSM*) — et les applications communiquant par messages.

Un *SSI* permet la gestion globale des ressources physiques (processeurs, mémoires, disques) présentes sur les différents nœuds et des ressources logiques offertes par le système d'exploitation (fichiers, segment de mémoire virtuelle, processus, flux de communication) dans le but de réaliser l'exécution d'un programme. Les qualités recherchées sont la transparence — la compatibilité binaire des applications pour calculateur *SMP* est recherchée —, le partage de ressources entre les nœuds, l'extensibilité, la haute disponibilité et bien évidemment la performance.

### 2.2 Fonctionnalités de KERRIGHED

KERRIGHED offrent de nombreuses fonctionnalités parmi lesquelles on distingue :

**Un ordonnanceur de tâches global à la grappe et personnalisable.** En utilisant la politique d'ordonnement par défaut de KERRIGHED, les processus et les threads sont automatiquement répartis sur les nœuds pour optimiser la charge processeur. KERRIGHED offre également un mécanisme pour écrire facilement des ordonnanceurs personnalisés et les charger dynamiquement.

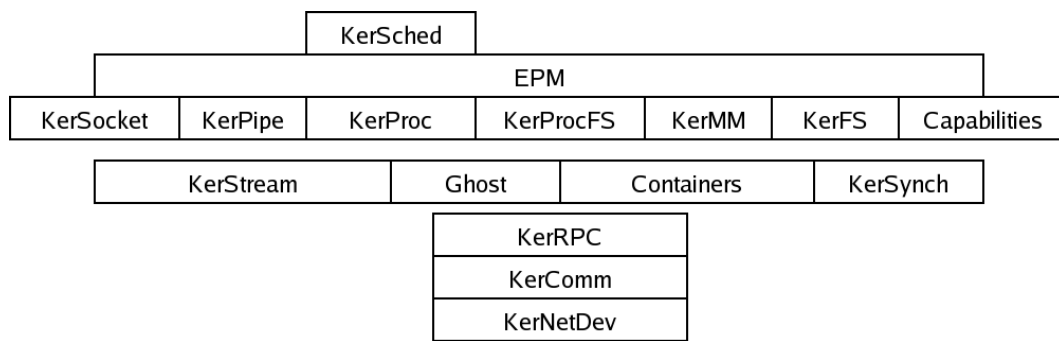


FIG. 2.1 – Architecture de KERRIGHED

**Une mémoire virtuelle partagée.** Les threads et les segments mémoires SYSTEM V fonctionnent à travers la grappe, de la même façon que sur une machine *SMP*.

**Des flux de communication déplaçables haute performance.** Les processus utilisant des flux de communication (*socket*, *fifo*, *tube*, etc.) peuvent être déplacés sans aucune pénalité sur les performances après la migration [16].

**Un système de gestion de fichiers distribué.** Un unique espace de nommage pour les fichiers est vu à travers toute la grappe. Tous les disques locaux des nœuds sont fusionnés en un unique disque virtuel offrant des fonctionnalités *RAID*.

**La réalisation de points de reprise de processus séquentiels.** Un point de reprise d'un processus peut être créé et restauré à partir de n'importe quel nœud de la grappe.

**L'interface complète de programmation POSIX pour les threads.** Toute l'interface de programmation des Threads POSIX est fonctionnelle sur la grappe.

**Une vision globale des processus.** Toutes les commandes traditionnelles UNIX de gestion de processus (*top*, *ps*, *kill*, etc.) fonctionnent de manière globale sur la grappe. De plus, les identifiants de processus (PID) sont uniques sur la grappe.

**La personnalisation des fonctionnalités SSI.** Toutes les fonctionnalités *SSI* (mémoire partagée, ordonnanceur de tâches global, flux migrables, etc.) peuvent être activées ou désactivées pour chaque processus.

## 2.3 Architecture du système KERRIGHED

La figure 2.1 présente l'architecture logicielle de KERRIGHED.

Pour la mise en œuvre de propriétés SSI, KERRIGHED fournit trois concepts de base : les conteneurs, le mécanisme de ghost et les flux dynamiques.

**Les conteneurs** — offrent un mécanisme assurant le partage cohérent de données à travers la grappe [21]. Cette fonctionnalité est implémentée dans le module CONTAINERS.

**Le GHOST** — offre un mécanisme de capture d'état d'un processus et des flux de communication qui lui sont rattachés [33],

**Les flux dynamiques** — sont une abstraction pour les flux de communication dont les extrémités peuvent être déplacées entre les nœuds [16, 15]. Cette fonctionnalité est implémentée dans le module KERSTREAM.

Les services de plus haut niveau sont construits au-dessus de ces abstractions.

KERMM et KERFS, fondés sur les conteneurs, offrent respectivement une mémoire virtuelle partagée [20] avec espace global d'adressage et un système de gestion de fichiers distribué.

KERPROC étend les mécanismes de gestion de processus (filiation, signaux, etc.) à la grappe. KERPROCFS, associé à KERPROC, met en œuvre le répertoire système UNIX `/proc` global à la grappe. Il s'agit d'un système de gestion de fichiers (SGF) virtuel qui exporte au niveau utilisateur des informations système sur les processus. Ce SGF est utilisé par les commandes de gestion de processus comme `ps` ou `top`. La migration de processus et la sauvegarde / restauration de points de reprise de processus sont accomplies par le module *Enhanced Processus Management* (gestion avancée de processus, EPM), construit au-dessus du module GHOST. KERSCHED offre un ordonnanceur de tâches global à la grappe [32] ainsi que les mécanismes pour sa personnalisation.

KERSOCKET et KERPIPE, au-dessus de KERSTREAM, offrent les flux dynamiques sur les interfaces de programmation UNIX classiques comme les *sockets* et les tubes.

Tous les services distribués de KERRIGHED s'appuient sur un système de communication à haute performance découpé en trois modules. KERCOMM offre le protocole réseau et les interfaces de communication noyau à noyau. Le module KERNETDEV fournit l'abstraction d'une carte réseau. KERRPC offre les appels de procédures à distance, bloquants ou non.

Le module KERSYNCH offre les outils de synchronisation nécessaires (verrous, barrières, etc.) à la mise en œuvre d'algorithmes distribués.

Le module CAPABILITIES permet l'activation ou la désactivation de fonctionnalités *SSI*, à l'échelle d'un processus, pour des raisons de sécurité ou de performance.

## 2.4 État du prototype

La dernière version stable de KERRIGHED est la version 1.0.2 fondée sur le noyau LINUX 2.4.29 (un portage vers le noyau LINUX 2.6 est en cours). Le code source de KERRIGHED est constitué d'environ 100000 lignes pour les modules et 7000 lignes de modification du noyau LINUX. KERRIGHED offre toutes les fonctionnalités du système à image unique à l'exception de la haute disponibilité du système d'exploitation et des mécanismes de sauvegarde et restauration de point de reprise d'applications parallèles.



## Chapitre 3

# Sauvegarde et restauration d'applications parallèles dans le système KERRIGHED

Dans les chapitres précédents, nous avons présenté les différents travaux existant dans le domaine de la sauvegarde et restauration de points de reprise d'applications parallèles, et le système KERRIGHED, cette partie s'intéresse à la mise en place efficace d'un tel mécanisme dans le système d'exploitation à image unique KERRIGHED.

L'originalité des travaux effectués ici vient de la transparence recherchée vis-à-vis des applications et des bibliothèques de communication.

Dans le cadre du stage, nous nous limitons aux applications parallèles communiquant par messages. Parmi la variété des protocoles assurant la restauration d'un état global cohérent à partir des points de reprise des processus d'une application parallèle, nous nous sommes restreints à la coordination globale. L'accent a été mis sur l'efficacité de la mise en œuvre et sur la transparence pour l'utilisateur et le programmeur (*system initiated checkpoint*).

Les travaux effectués ont porté principalement sur trois points. Le premier est l'identification de l'ensemble des processus d'une application parallèle à partir de l'un d'entre d'eux. Cet ensemble est construit en exploitant les liens de filiation et de communication entre processus. Le deuxième point est le stockage distribué des points de reprise effectués en parallèle sur les nœuds. Le troisième point porte sur la définition d'une interface utilisateur dans KERRIGHED pour le mécanisme de sauvegarde et restauration de points de reprise.

Des réflexions ont également été menées sur les problèmes d'entrées-sorties disque, sur la suppression des points de reprise inutiles et sur la possibilité de restaurer des applications sur une autre grappe.

Dans cette partie, nous présentons tout d'abord en quoi consiste une application parallèle déployée au-dessus du système d'exploitation KERRIGHED et comment calculer la frontière de l'application. Nous introduisons ensuite la politique de désignation choisie pour conserver les différents fichiers constituant un point de reprise global d'une application. Le troisième paragraphe porte sur l'interface dont dispose l'utilisateur pour réaliser un point de reprise et restaurer une application. Les deux parties suivantes portent sur les algorithmes utilisés respectivement lors de la création d'un point de reprise et pour la restauration de l'application à partir d'un point de reprise. Enfin la dernière partie se focalise sur les problèmes liés aux entrées-sorties.

## 3.1 Vision système des applications parallèles s'exécutant sur grappe KERRIGHED

KERRIGHED s'appuie sur un noyau LINUX standard. Nous étudions d'abord le fonctionnement classique de LINUX puis les extensions apportées dans le cadre de KERRIGHED pour le support de l'exécution d'applications parallèles sur grappe.

### 3.1.1 Les processus dans LINUX

Une application parallèle s'exécutant sur un système LINUX (cf. figure 3.1) consiste en :

- des processus ;
- des flux de communication entre les processus ;
- des liens de filiation entre processus ;
- des fichiers réguliers ouverts en lecture ou en écriture, par un ou plusieurs processus ;

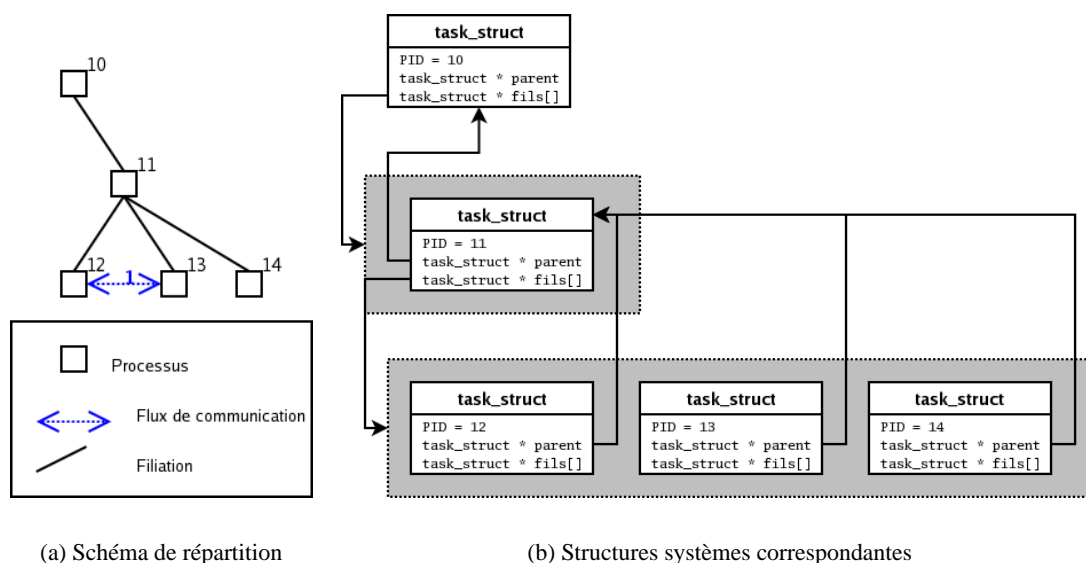


FIG. 3.1 – Application parallèle sous LINUX— Résultat du code figure 3.2

Soient deux processus A et B, B a été créé à partir de A par l'appel système `fork` (voir code source en figure 3.2). On qualifie alors A de processus *père* ou *parent* de B et B de processus *fils* ou *enfant* de A. Soit un troisième processus C, celui-ci est un processus fils du processus B. C est un *descendant* de B. La relation de descendance est transitive, C est un descendant de A (Par exemple, sur la Fig. 3.1 (a), le processus 10 a un processus fils 11 qui lui-même a plusieurs processus fils dont le processus 14. Le processus 14 est un descendant du processus 10).

Le processus *racine principale* désigne par la suite le premier processus créé de l'application (exemple, le processus 10 Fig. 3.1 (a)). Celui-ci se situe en haut de l'arbre de filiation des processus de l'application.

Une structure de données `task_struct` est associée à chaque processus. Celle-ci comprend toutes les informations relatives au processus comme l'état du processus (en fonctionnement, en attente interruptible, en attente ininterruptible, zombie, ou arrêté), l'identifiant de processus, de groupe de processus, de session, les identifiants d'utilisateur, de groupe d'utilisateurs.

FIG. 3.2 – Code source d'une application utilisant l'appel système `fork`

```
int main ()
{
    int pid;
    if ( pid = fork() ){ // Code du processus père UNIQUEMENT
        ...
    } else { // Code du processus fils
        int i;
        for ( i=0; i < 3; i++){
            if ( (pid = fork()) == 0 ){ // Code des petits fils
                ...
                break;
            }
        }
    }
    // Code commun à tous les processus de l'application
    ...
}
```

Elle comporte également des pointeurs vers les structures des processus suivants (cf. figure 3.1 (b)) : le parent, les fils, les suivants et précédents dans la liste de tous les processus, les suivants et précédents dans la liste des processus susceptibles d'obtenir un quantum de temps processeur pour s'exécuter.

Dans la sémantique POSIX, la notion de fils plus jeune ou moins jeune n'existe pas. L'ordre des processus frère n'importe pas. En revanche, les relations père-fils sont nécessaires à la cohérence de certains appels systèmes (Par exemple, `wait()`).

### 3.1.2 Vision des processus dans KERRIGHED

Pour étendre les mécanismes classiques et les faire fonctionner à l'échelle d'une grappe de calculateurs, des extensions de la structure `task_struct` ont été nécessaires dans la mise en œuvre du système KERRIGHED. Pour des raisons de lisibilité et maintenance, l'ensemble de ces extensions se trouve dans la structure spécifique à KERRIGHED `krg_task` vers laquelle un pointeur a été ajouté dans la structure `task_struct`. Par la suite, pour simplifier le propos, nous assimilons les champs de la `krg_task` à des champs supplémentaires de la structure `task_struct`.

#### 3.1.2.1 Gestion globale des processus

Le système KERRIGHED permet la création distante et la migration de processus. Dès lors, un processus père A peut s'exécuter sur un nœud N1 et son processus fils B sur un nœud N2. Le processus B est alors un fils *distant* du processus A. Un processus fils distant est considéré comme une *racine locale* sur son nœud d'exécution.

Une structure `task_struct`, nommée *babysitter*, a été créée sur chaque nœud pour être le parent d'un processus orphelin localement. Lorsqu'un processus C est adopté par le babysitter, l'identifiant du processus (PID) père distant est stocké dans un champ supplémentaire de la structure `task_struct` du processus C.

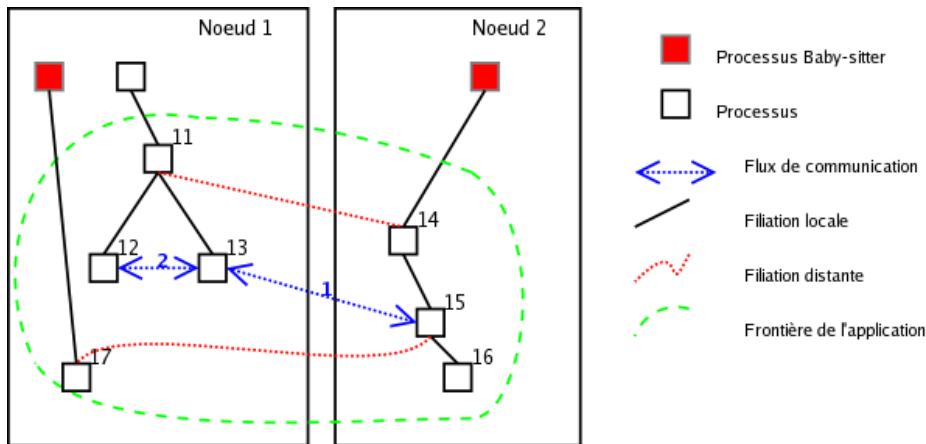


FIG. 3.3 – Schéma de répartition d’une application parallèle sur une grappe KERRIGHED

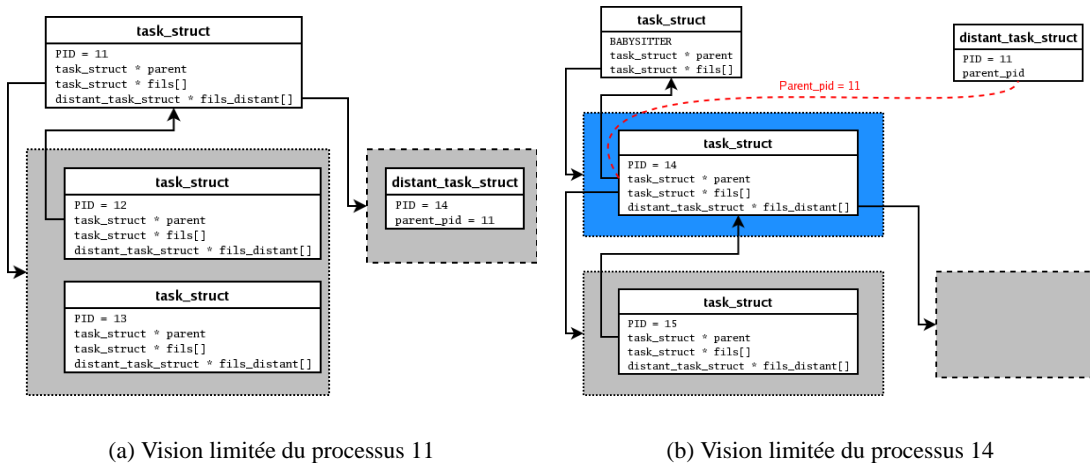


FIG. 3.4 – Exemple de filiation entre les structures de données de KERRIGHED (cf. Fig 3.3)

La figure 3.3 présente la répartition d’une application parallèle sur une grappe KERRIGHED. On distingue les structures `task_struct` *babysitter*, les liens de filiation locale et distante et les flux de communication.

Pour remplacer la structure `task_struct` d’un processus distant, il est possible d’obtenir une structure `distant_task_struct` qui contient les informations minimales nécessaires pour un processus père vis-à-vis de son fils distant ou pour un processus fils vis-à-vis de son père distant (cf. Fig 3.4). Lorsqu’un appel système nécessite d’effectuer des opérations sur le processus parent, il est vérifié que le processus parent n’est pas le *babysitter*. Si tel est le cas, l’appel système LINUX standard est appelé, sinon, une demande d’obtention de `distant_task_struct` est effectuée auprès du nœud hôte, la `distant_task_struct` est modifiée en conséquence puis renvoyée. Chaque processus KERRIGHED a une liste, éventuellement vide, de processus fils distants (sous forme de `distant_task_struct`) et l’identifiant de son véritable processus parent.

### 3.1.2.2 Les capacités dans KERRIGHED

Le concept de capacités provient de la norme POSIX. Les capacités sont des attributs associés à un processus qui sont hérités et peuvent être activés ou désactivés — en ligne de commande ou par programmation — alors même que le processus est en cours d'exécution.

Parmi les capacités spécifiques à KERRIGHED, on trouve `DISTANT_FORK`, qui permet ou non la création de processus fils sur un autre nœud, `CAN_MIGRATE`, `CHECKPOINTABLE` qui indiquent respectivement si un processus est autorisé à être migré ou sauvegardé.

Par défaut la capacité `CHECKPOINTABLE` est désactivée pour tous les processus. Il est possible de l'activer pour toutes les applications lancées à partir d'un terminal en tapant la commande `krig_capset -d +CHECKPOINTABLE` dans ce même terminal.

## 3.2 Frontière d'application parallèle

### 3.2.1 Cas général

Une application parallèle est comme nous l'avons vu précédemment composée de plusieurs processus s'exécutant sur différents nœuds de la grappe.

Pour réaliser un point de reprise, nous devons calculer la frontière de l'application. Il doit être possible à partir d'un processus d'une application d'identifier l'ensemble des processus de la même application.

Comme les différents processus d'une même application sont dépendants entre eux, que ce soit par filiation, ou par flux, une manière de distinguer la frontière de l'application se fonde sur l'examen des liens de filiation et les flux ouverts.

Le risque est alors de sauvegarder tous les processus s'exécutant sur le système d'exploitation de chacun des nœuds concernés par le point de reprise de l'application. Une des solutions possibles est la vérification de l'activation de la capacité `CHECKPOINTABLE`. Dès lors, les liens de filiation et de communication ne sont suivis que si le processus à l'autre extrémité possède la capacité `CHECKPOINTABLE`.

### 3.2.2 Cas des applications client/serveur

Dans le cas d'une application client/serveur (cf. Fig 3.5), il n'y a pas de lien de filiation entre l'arborescence des processus de la partie cliente de l'application et l'arborescence des processus de la partie serveur. Nous observons deux (ou plus s'il y a plusieurs clients) *racines principales* pour l'application, on parle de racines principales *multiples*. L'application se décrit comme un graphe de processus et non comme une arborescence. Ce graphe est constitué de plusieurs arborescences reliées entre elles par des flux de communication.

## 3.3 Organisation, désignation et stockage des points de reprise

À un instant donné, plusieurs applications sont en cours d'exécution sur une même grappe. Chacune d'elles utilise un ou plusieurs processus. Pour chaque application, plusieurs points de reprise peuvent être conservés. Chaque point de reprise d'application comporte les points de reprise des différents processus qui la composent.

Il est nécessaire de définir une politique de conservation afin qu'il n'y ait pas de confusion entre les différents points de reprise de processus.

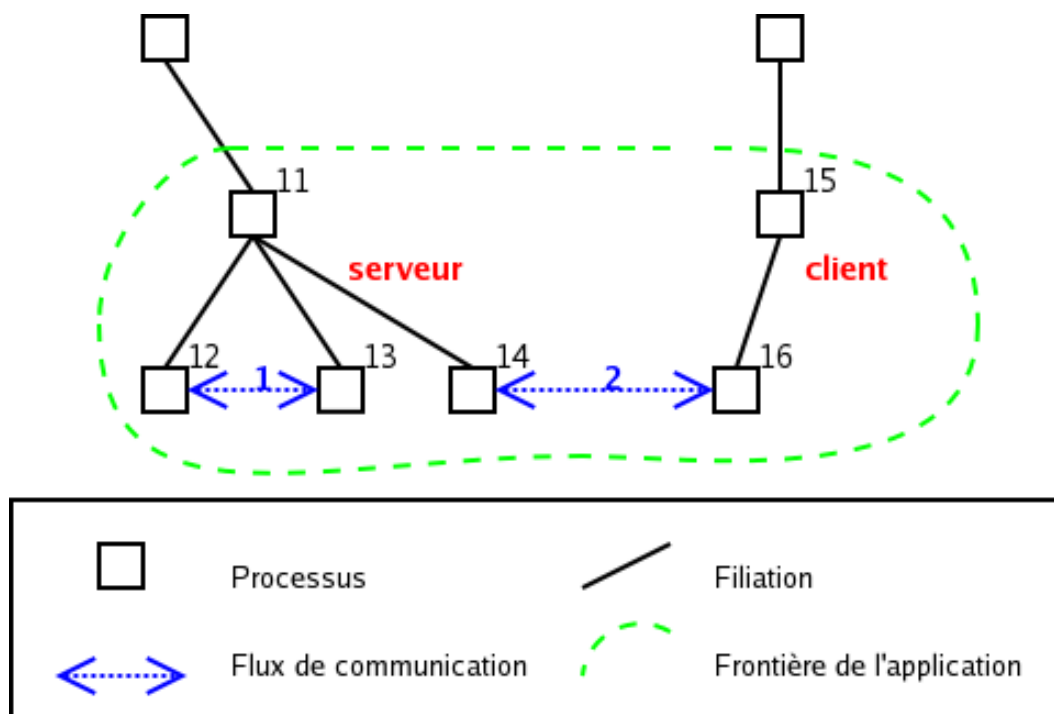


FIG. 3.5 – Application client/serveur

### 3.3.1 Stockage physique des points de reprise

Lors de la sauvegarde sur disque, les fichiers de points de reprise doivent être stockés dans un répertoire accessible de tous les nœuds de la grappe pour la restauration. Grâce au montage de système de fichiers, ce répertoire peut être n'importe quel système de gestion de fichiers du moment qu'il respecte cette condition.

Le système de gestion de fichiers KERFS proposé par KERRIGHED permet d'obtenir une arborescence unique pour la grappe. Les fichiers sont alors répartis et/ou répliqués partiellement sur les différents disques physiques des nœuds de la grappe, tout en étant accessibles dans un même répertoire. Ce système permet une distribution des données et offre des performances supérieures à l'utilisation d'un serveur centralisé.

Une autre possibilité est l'utilisation d'un serveur centralisé externe à la grappe utilisant par exemple le système de gestion de fichiers NFS. Ce serveur peut être accessible par une autre grappe et dispose généralement d'un espace de stockage bien supérieur à celui proposé par la mise en commun des disques physiques des nœuds de la grappe.

### 3.3.2 Organisation et désignation des points de reprise

Le répertoire de stockage choisi est `/var/chkpt` pour être conforme à l'organisation des fichiers définie dans *Linux Standard Base* [3].

Un point de reprise d'une application est constitué de plusieurs fichiers. Pour chaque point de reprise, ses fichiers sont conservés dans un même sous-répertoire dont le nom est issu de la concaténation d'un identifiant d'application — généralement l'identifiant du processus racine de l'application (en cas de racines principales multiples, l'identifiant de l'une d'entre elles est arbitrairement choisie) — et d'un numéro de version de point de reprise. Le numéro de version utilisé est le premier numéro de version

disponible lors de la création du point de reprise. Par exemple, lors de la réalisation d'un deuxième point de reprise pour l'application, visible en figure 3.3, le répertoire créé est `/var/chkpt/11_v2/`.

Pour chaque processus, un fichier permettant sa restauration est créé sous le nom `task_PID.bin`. Le numéro de nœud et la liste des processus descendants locaux sont enregistrés dans le fichier `filiation_PID-RACINE.bin` où `PID-RACINE` représente un processus racine principale ou locale. Enfin, le fichier `main.bin` conserve la liste des identifiants (PID) des processus racines *principales*.

Lors de la sauvegarde de l'application présentée en figure 3.3, les fichiers créés sont les suivants :

- `main.bin`, qui ne contient que le nombre 11 ;
- `filiation_11.bin`, `filiation_14.bin`, `filiation_17.bin`;
- `task_11.bin`, `task_12.bin`, `task_13.bin`, `task_14.bin`, `task_15.bin`, `task_16.bin`, `task_17.bin`;

## 3.4 Interface utilisateur

### 3.4.1 Types d'interfaces

Pour initier un point de reprise d'application, deux types d'interfaces complémentaires sont envisageables : en ligne de commande ou interface de programmation exportée en mode utilisateur.

La première solution permet la sauvegarde d'applications n'ayant pas prévu la nécessité d'être sauvegardées et déjà en cours lors de la demande de création de point de reprise. Cette technique est transparente pour le programmeur puisqu'il n'est pas nécessaire de modifier le code source de l'application.

La deuxième solution nécessite de modifier le code source de l'application et n'est pas transparente pour le programmeur. L'avantage est ici que le programmeur choisit les endroits les plus judicieux pour établir un point de reprise. Il peut ainsi minimiser la taille du point de reprise en insérant les appels de création à des périodes où l'application a une faible utilisation mémoire.

Ces deux types d'interface ont leurs avantages et doivent être offertes dans KERRIGHED pour la souplesse.

Un troisième type, le *system initiated checkpoint*, réalise la sauvegarde de points de reprise d'application sans aucune intervention de l'utilisateur.

### 3.4.2 Interface existante dans KERRIGHED

Pour permettre à l'utilisateur de réaliser un point de reprise d'un processus séquentiel, KERRIGHED ne propose pas d'interface de programmation mais offre l'interface suivante en ligne de commande :

- `checkpoint pid`
- `restart pid`

Avec `pid` l'identifiant du processus pour lequel on souhaite un point de reprise.

Actuellement, le type de mémoire stable utilisé pour le stockage du point de reprise est indiqué par les capacités. En effet, la capacité `CHECKPOINTABLE` n'existe pas mais est découpée en deux capacités : `CHECKPOINT_ON_DISK` et `CHECKPOINT_ON_MEMORY` qui utilisent respectivement le disque ou la mémoire.

Par ailleurs, la capacité `EXPORTABLE_CHECKPOINT` indique que l'application peut être restaurée sur une autre grappe. Le point de reprise doit être plus complet. Cette grappe doit utiliser un système KERRIGHED mais les versions des bibliothèques partagées et autres fichiers disponibles ne sont pas systématiquement les mêmes.

Cette interface pose problème sur plusieurs points :

- aucune interface n'est définie pour le traitement des applications parallèles ;

- il est impossible de spécifier une périodicité à laquelle les points de reprise doivent être créés ;
- l'utilisation des capacités n'est pas pertinente dans le cas des points de reprise. Ainsi, il est délicat de réaliser, pour une même application, des points de reprise sur disque et en mémoire puisqu'il faut réaffecter à chaque processus la bonne capacité au moment de la sauvegarde. De même, il est possible de vouloir réaliser des points de reprise exportables et d'autres non exportables pour une même application, là encore l'utilisation des capacités rend l'opération complexe pour l'utilisateur.

### 3.4.3 Proposition d'une nouvelle interface pour les points de reprise dans KERRIGHED

La première nécessité est la prise en compte des applications parallèles.

Notre choix est d'étendre l'interface initiale en ligne de commande de création de point de reprise pour qu'elle soit commune aux applications parallèles et séquentielles.

L'option `-u` (ou `-unique`) permet de limiter la sauvegarde à l'unique processus dont le PID est fourni par l'utilisateur. Dans le cas contraire, la frontière de l'application est calculée comme indiqué dans le paragraphe 3.2 (à partir du PID de n'importe quel processus de l'application).

Les capacités `CHECKPOINT_ON_DISK` et `CHECKPOINT_ON_MEMORY` sont fusionnées en `CHECKPOINTABLE` et remplacées par une option `-media=disk|memory` qui permet de choisir si l'on stocke le point de reprise sur disque ou en mémoire.

L'option `-exportable` remplace la capacité `EXPORTABLE_CHECKPOINT` et permet de préciser pour chaque instance de points de reprise s'il doit être plus complet (intégration des bibliothèques partagées, des fichiers réguliers ouverts, etc.).

L'option `-id` doit permettre à l'utilisateur de choisir l'identifiant du point de reprise global. Par défaut il s'agit de l'identifiant du processus racine principale de l'application. Dans le cas d'une application client-serveur, l'identifiant est choisi arbitrairement parmi ceux des différentes racines principales.

L'option `-d "description"` indique une description à associer au point de reprise. Celle-ci est affichée à l'utilisateur dans la commande de restauration d'application.

Une autre interface en ligne de commande doit permettre la réalisation de points de reprise de manière périodique sans intervention supplémentaire de la part de l'utilisateur (*system initiated checkpoint*).

Il est intéressant de disposer d'une interface de restauration de point de reprise proposant la liste des points de reprise disponibles à un instant donné. Cette interface doit afficher les identifiants de points de reprise, les dates de création, et les descriptions associées.

### 3.4.4 Exemples

Ce paragraphe présente des exemples de sauvegarde et restauration de points de reprise réalisés à l'aide d'une interface conforme à celle décrite en 3.4.3.

- Sauvegarde en mémoire du processus ayant pour PID 1059.  
`checkpoint -u -media=memory 1059`
- Sauvegarde sur disque de l'application dont l'un des processus a pour PID 1056. La description associée au point de reprise de l'application est "benchmark n°1".  
`checkpoint -media=disk -d "benchmark n°1" 1056`
- Sauvegarde avec possibilité de restauration sur une grappe différente.  
`checkpoint -d "benchmark n°2" -exportable 1092`
- Lancement de l'application `./mon_application` avec sauvegarde toutes les 2 heures sur disque.  
`reliable_run -media=disk -periodicity=2h -d "Application X" ./mon_application`
- Restauration de l'application, dont le processus racine principale a pour PID 2538, à partir de la deuxième version du point de reprise de l'application.  
`restart 2538 2`

- Interface de restauration d’applications lancée à l’aide de la commande `restart`

Identifiant	Description	Date	Version
0001	Application X	Jun 2 11:35	12
1056	benchmark n°1	Jun 1 17:36	1
1059	Processus 1059	May 29 12:09	3
1092	benchmark n°2	Jun 2 10:04	1
2538	test checkpoint	May 28 15:42	1

---

Which application to restart : 0001  
version : 8

---

### 3.5 Sauvegarde d’un point de reprise d’application parallèle

Nous décrivons ici la sauvegarde d’un point de reprise d’une application parallèle selon l’approche de la coordination globale.

Cette création du point de reprise a lieu en plusieurs étapes :

1. déterminer quels sont les processus de l’application,
2. endormir l’ensemble des processus de l’application pour qu’ils stoppent toute communication,
3. marquer l’ensemble des processus de l’application comme devant être sauvegardé, puis les réveiller,
4. laisser les processus effectuer la sauvegarde de leur état lorsque l’ordonnanceur des tâches leur donne la main,
5. valider ou annuler le point de reprise de l’application.

De la première à la dernière étape, sur chaque nœud concerné, un verrou système est pris s’assurant qu’aucun processus de l’application ne disparaisse ou ne soit créé pendant la réalisation du point de reprise de l’application.

À chaque étape, aucun processus (même le processus racine principale) n’a une connaissance complète des processus pour lesquels un point de reprise doit être réalisé. Une racine n’a conscience que de sa descendance locale, et chacun de ses descendants a conscience de ses fils distants. L’information est décentralisée et la coordination a lieu de manière hiérarchique.

#### 3.5.1 Déterminer quels sont les processus de l’application et les endormir

Au cours de la première étape, la frontière de l’application est calculée en parcourant les liens de filiation et de communication entre processus. Tous les processus de l’application sont endormis et les fichiers de sauvegarde de filiation sont construits.

Cette étape se déroule selon l’algorithme 1. Il apparaît ici dans une version simplifiée — il en est de même pour l’ensemble des algorithmes présentés dans ce paragraphe — qui ne tient pas compte des flux de communication. La version complète est décrite dans le paragraphe 3.7.1.

---

**Algorithme 1** : Algorithme d'endormissement de l'ensemble des processus d'une application

---

**Coordination globale hiérarchique par endormissement (*IdRacine*) :**

```
{
  Racine = RecupereProcessusParIdentifiant(IdRacine) // fonction spécifique KERRIGHED
  pour tout processus P appartenant aux descendants locaux du processus Racine ou étant Racine faire
    si P a capacité CHECKPOINTABLE alors // consultation de la krq_task
      endormir(P)
      ajouter Processus.Identifiant au fichier filiation_IdRacine.bin
      pour tout fils distant F du processus P faire
        // effectuer un appel bloquant de procédure à distance.
        // c'est ce même algorithme (1) qui est appelé.
        RPC_Sync_EndormirProcessus (F.identifiant)
      fin pour
    fin si
  fin pour
}
```

---

FIG. 3.6 – Code de la fonction *HandlerAckSauvegarderProcessus*

```
fonction HANDLERACKSAUVEGARDERPROCESSUS (IdProcessus, resultat)
{
  si resultat != OK alors
    AnnulationPointReprise();
  sinon
    Mise à jour de la liste des acquittements
    si Active == FAUX alors // l'ensemble des points de reprise locaux ont été créés
      ValidationPointReprise()
    fin si
  fin si
}
```

### 3.5.2 Sauvegarder les processus de l'application

Pour obtenir une bonne rapidité de réalisation du point de reprise global de l'application, les descendants locaux sont sauvegardés en parallèle avec les fils distants et leurs descendances respectives.

L'initiateur du protocole est le nœud à partir duquel la commande de sauvegarde de point de reprise est lancée. Lorsque un nœud reçoit une requête de demande de sauvegarde d'un point de reprise (appel de fonction `RPC_ASynch_SauvegarderEnsProcessus`), il exécute l'algorithme 2 puis renvoie le résultat de la requête (*OK* ou un code d'erreur) au nœud solliciteur. Cette réponse est récupérée par un gestionnaire d'événements qui appelle la fonction présentée en figure 3.6. La réponse est alors mise à jour dans la liste des acquittements associée au processus *racine*. En cas de réception d'un code d'erreur, un message d'annulation est envoyé à tous les nœuds concernés par la sauvegarde et les fichiers déjà créés sont détruits. Lorsque tous les points de reprise locaux ont été pris et que plus aucun acquittement n'est attendu, si toutes les réponses ont été positives, le point de reprise est validé.

La fonction `AnnulationPointReprise()` n'est pas présentée ici puisqu'elle n'a que peu d'intérêt.

---

**Algorithme 2** : Algorithme post-coordination pour la réalisation d'un point de reprise d'une application

---

**Sauvegarde des processus d'une application (*IdRacine*) :**

```
{
  Racine = RecupereProcessusParIdentifiant(IdRacine) // fonction spécifique KERRIGHED
  Active = VRAI
  pour tout processus P appartenant aux descendants locaux du processus Racine ou étant Racine faire
    si P a capacité CHECKPOINTABLE alors // consultation de la krq_task
      pour tout fils distant F du processus P faire
        // effectuer un appel NON bloquant de procédure à distance.
        // c'est ce même algorithme (2) qui est appelé.
        RPC_ASync_SauvegarderEnsProcessus (F.identifiant)
      fin pour
      PointDeReprise(Processus);
    fin si
  fin pour
  Active = FAUX
  ValidationPointReprise()
}
```

---

L'ensemble des opérations de création de point de reprise sont interrompues et les fichiers déjà réalisés sont détruits. Cette méthode utilise elle aussi un gestionnaire d'évènements pour s'exécuter sur chacun des nœuds impliqués.

### 3.6 Restauration d'un point de reprise d'application parallèle

La restauration d'un point de reprise d'application parallèle se déroule de la façon suivante.

À partir de l'identifiant du processus racine *A*, l'identifiant *NI* du nœud responsable du processus *A* avant la sauvegarde est lu dans le fichier `filiation_A.bin`. Si cet identifiant correspond au nœud courant ou que le nœud *NI* n'est plus disponible, la restauration continue en local, sinon une requête bloquante est lancée pour réaliser la restauration sur le nœud *NI*.

Le processus *A* et ses descendants locaux — la liste des descendants locaux est lue dans le fichier `filiation_A.bin` — sont restaurés à partir de leurs fichiers de point de reprise respectifs. Dans le même temps les flux de communication nécessaires sont recréés et les processus y sont attachés. Les liens de filiation sont ensuite corrigés.

Une fois l'ensemble des descendants de *A* restaurés, une requête bloquante de restauration est lancée pour les fils distants de chacun de ces processus.

Cette méthode — qui apparaît ici en version simplifiée ne tenant pas compte des flux de communication pour la frontière de l'application — présentée par l'algorithme 3 se déroule de manière entièrement séquentielle et n'offre donc pas des performances intéressantes. Il doit être assez simple de le paralléliser en utilisant des requêtes non bloquantes comme cela est fait pour la sauvegarde. Il est cependant important de noter qu'elle s'attache à redémarrer les processus sur leurs nœuds d'origine tout en intégrant la gestion de nœuds indisponibles depuis la création du point de reprise.

FIG. 3.7 – Code de la fonction *ValidationPointReprise*

```
fonction VALIDATIONPOINTREPRISE ()
{
  global_OK = VRAI // pour la racine locale
  pour tous les acquittements Ack attendus faire
    si Ack.result != OK alors
      global_OK = FAUX
    fin si
  fin pour
  si global_OK == VRAI alors
    si nous sommes la racine principale alors
      // racine locale == racine principale
      // c'est la fin de la création du point de reprise
      suppression Point de Reprise ancienne version
    sinon
      // nous sommes une racine locale
      envoyer OK au processus distant qui a demandé le point de reprise
    fin si
  sinon
    mise en sommeil
  fin si
}
```

### 3.6.1 Gestion des PID

Dans le cadre des applications parallèles, l'utilisation de l'appel système `get_pid()` est courante. Cet appel renvoie la valeur de l'identifiant d'un processus (PID). Cet identifiant est ensuite généralement conservé dans une variable, un fichier, voire dans le nom d'un fichier verrou qui peut être créé. Il peut ensuite être utilisé pour permettre des communications entre processus. En cas de restauration de l'application, les identifiants affectés au processus doivent absolument être identiques à ceux assignés avant la sauvegarde, sous peine de provoquer un dysfonctionnement de l'application.

Comme tout identifiant, le PID se doit d'être unique à un instant  $t$  dans le système. Pour assurer cette unicité, plusieurs solutions sont envisageables en dehors de la chance et du hasard.

1. Une première solution consiste à refuser la restauration d'une application si l'un des identifiants de processus de l'application est déjà utilisé.
2. Il est possible d'arrêter (tuer) l'ensemble des processus occupant l'espace de nommage des PID de l'application et de restaurer ensuite l'application. Cette solution n'est évidemment pas souhaitable.
3. L'application peut fonctionner dans une boîte noire [24], auquel cas les PID vus par l'application elle-même sont locaux à cette boîte et indépendants du reste du système. L'inconvénient majeur de ce système est l'isolement entraîné pour l'application, puisque celle-ci ne doit pas effectuer de communication vers l'extérieur.
4. Un niveau d'indirection peut-être ajouté lors de la restauration, une table de correspondance assurant le détournement de tous les appels systèmes nécessaires. Cette solution, si elle permet une pseudo-unicité, risque de poser des problèmes entre une application utilisant un PID A réel et une application restaurée utilisant un PID A simulé. En effet, en cas de fichier verrou comportant l'identifiant

---

**Algorithme 3** : Algorithme de restauration d'une application parallèle (*IdRacine*)

---

**Restauration d'un processus et de ses descendants(*IdRacine*) :**

```
{
  ouverture du fichier filiation_IdRacine.bin
  lecture du numéro de noeud dans le fichier
  si noeud != noeud_courant et noeud existe alors
    // redémarrage distant
    fermeture du fichier filiation_IdRacine.bin
    RPC_Sync_RestaurationProcessus(IdRacine, nœud )
  sinon
    chargement de la liste des descendants locaux de IdRacine à partir de filiation_IdRacine.bin
    fermeture du fichier filiation_IdRacine.bin
    restauration du processus IdRacine et des descendants locaux à partir des fichiers task_<PID>.bin
    création des flux nécessaires au fur et à mesure et attachement des processus aux flux
    // assurance que les processus ne vont pas avoir la main
    pour tout processus P, P appartient à la liste des processus redémarrés faire
      pour tout fils distant F du processus P faire
        // Lance ce même algorithme (3 sur le nœud responsable du processus F avant
        // la sauvegarde du point de reprise
        RPC_Sync_Restauration(F)
      fin pour
    fin pour
  fin si
}
```

---

de processus, il risque d'y avoir conflit. De plus, de l'extérieur de l'application restaurée, celle-ci n'apparaît pas avec les PID d'origine, ce qui est perturbant pour l'utilisateur s'il consulte la sortie des commandes UNIX `top` ou `ps`.

5. Un système de réservation de PID peut être mis en place. Ainsi, lorsqu'une application est conservée sous forme de point de reprise, les PID qui lui étaient affectés ne peuvent être réassignés à une autre application. Cela entraîne la nécessité pour l'utilisateur de faire le tri dans ces points de reprises. Dans le cadre de cette solution, un système de quota par utilisateur est souhaitable. Pour permettre d'augmenter l'espace de nommage de PID accessible à l'utilisateur, cette mesure peut s'accompagner, d'un passage sur 64 bits (au lieu de 32) pour stocker les identifiants de processus, ou d'une organisation différente du système identifiant un processus par le couple identifiant de processus (PID), identifiant de l'utilisateur (UID).

La dernière solution paraît la meilleure bien que contraignante pour l'utilisateur. C'est en effet la seule qui assure réellement l'unicité tout en permettant la restauration d'une application à tout moment, en ayant un résultat des commandes UNIX `top` ou `ps` cohérent et sans limiter les interactions de l'application avec l'extérieur.

## 3.7 Traitement des entrées-sorties

### 3.7.1 Les flux dynamiques

#### 3.7.1.1 Description des flux

Un flux de données est un mécanisme permettant de transmettre une suite d'octets entre deux entités distinctes. Plusieurs types de flux sont traditionnellement offerts par les systèmes UNIX. Les `pipe` et `fifo` correspondent à des mécanismes d'échange de données à l'intérieur d'une même machine. Les `sockets` se subdivisent en diverses familles, notamment `INET` pour les communications entre machines et `UNIX` pour les communications internes à une machine.

#### 3.7.1.2 Flux et migration

Dans un système à image unique, les extrémités des `pipe`, `fifo` et `socket` UNIX peuvent se situer sur des machines distinctes.

Une implémentation des flux différente de l'implémentation LINUX est disponible dans le code de KERRIGHED [16] pour gérer cette contrainte. Par ailleurs, l'implémentation offre la possibilité aux processus de changer de nœuds d'exécution, tout en gardant les flux ouverts.

Lors de la migration d'un processus, les flux de communication ouverts par ce processus sont vidés et suspendus à l'aide de la fonction `kernet_stream_suspend`. Le processus est ensuite déplacé sur un autre nœud puis réattaché au flux en utilisant la fonction `kernet_stream_activate` avec comme paramètre l'identifiant du flux dynamique.

#### 3.7.1.3 Flux et création et restauration de point de reprise

Lors de la création du point de reprise d'une application parallèle, les flux de communication qu'elle a établis doivent être suspendus au début de la création, et réactivés lorsque la sauvegarde est terminée. Les fonctions utilisées pour ces actions sont les mêmes que pour une migration.

À la restauration, les flux n'ont plus d'existence logique et il est nécessaire de les recréer. Un conteneur est alloué et partagé pour stocker pour chaque restauration une table de correspondance entre les nouveaux identifiants de flux et les anciens. Le premier processus restauré qui doit être attaché à un flux crée un nouveau flux et indique dans la table l'ancienne valeur de l'identifiant du flux, et la nouvelle valeur. En effet, les numéros de flux peuvent avoir été réutilisés par une autre application. Chaque processus concerné se rattache alors au flux par son nouvel identifiant en appelant la fonction `kernet_stream_activate`. Un changement de l'identifiant de flux ne provoque pas de dysfonctionnement des applications puisque cette donnée n'est pas exportée en dehors du noyau.

Pour gérer les applications client-serveur, il est nécessaire comme nous l'avons vu en partie 3.2 de suivre les flux de communication. Nous allons mettre en évidence les ajouts nécessaires en comparaison des algorithmes présentés en parties 3.5 et 3.6.

#### **Déterminer quels sont les processus de l'application et les endormir**

L'algorithme 4 est une évolution de l'algorithme 1. Pour chaque descendant local d'une racine, on examine les flux et effectue une demande d'endormissement sur les racines principales respectives des processus situés à l'autre extrémité d'un flux. Si une racine principale est déjà référencée — dans la liste des racines principales conservée en mémoire partagée grâce à un conteneur —, rien n'est fait sinon l'algorithme 4 est de nouveau appelé sur cette racine.

À la fin de cette phase d'endormissement de l'ensemble des processus de l'application, la liste des identifiants (PID) des processus racines principales de l'application est enregistrée dans le fichier

`main.bin`. Le conteneur mémoire contenant la liste des flux parcourus (cf. algorithme 4) est détruit tandis que celui contenant la liste des racines principales reste alloué jusqu'à la fin de la sauvegarde du point de reprise de l'application.

Par exemple, en figure 3.5, si cette phase démarre par l'endormissement du processus 11, le processus 12 est ensuite endormi. En suivant le flux de communication n°1, une demande d'arrêt de la racine principale de l'arborescence d'application incluant le processus 13 est effectuée. L'examen de cette demande permet de constater que la racine principale en question est le processus 11. Celui-ci est d'ores et déjà dans la liste des racines principales, rien n'est fait. Les processus 13 et 14 sont ensuite stoppés. Suite au parcours du lien de communication n°2, la racine principale de l'arborescence d'application incluant le processus 16 est retrouvée. Il s'agit du processus 15. L'algorithme 4 est alors lancé en prenant comme racine le processus 15. Le processus 15 est stoppé, suivi du processus 16. Le flux de communication n°2 est alors suivi, le processus à l'extrémité a pour identifiant 14. Sa racine est le processus 11, cet identifiant est déjà dans la liste. La phase d'endormissement se termine par l'enregistrement dans le fichier `main.bin` des identifiants des processus racines principales.

### **Sauvegarder les processus de l'application**

Le nœud coordinateur appelle à distance de manière asynchrone l'algorithme 2 pour chacune des racines principales — la liste des racines principales est conservée en mémoire partagée grâce à un conteneur, elle a été remplie lors de la phase d'endormissement — et attend les acquittements pour valider ou annuler le point de reprise de l'application.

### **Restauration du point de reprise de l'application**

La restauration débute par le chargement de la liste des racines principales de l'application sauvegardée dans le fichier `main.bin`. Pour chaque racine, le nœud coordinateur appelle à distance de manière asynchrone l'algorithme 3 et attend les acquittements pour en cas d'erreur annuler l'opération de restauration.

## **3.7.2 Les fichiers réguliers**

Les processus d'une application peuvent écrire dans des fichiers durant leur exécution. L'écriture est d'abord réalisée dans un cache mémoire puis reportée en arrière plan, périodiquement, sur le disque par le système d'exploitation.

Actuellement, lors de la réalisation d'un point de reprise les fichiers ouverts ne sont pas bien pris en compte. Seuls les noms complets des fichiers sont mémorisés, ainsi que leurs pointeurs de fichier respectifs. Les fichiers sont alors simplement réouverts lors de la restauration du point de reprise. Le problème est que les fichiers ne sont pas forcément dans le même état que lorsque le point de reprise a été sauvegardé.

Différentes solutions existent. La première consiste à sauvegarder un point de reprise d'un fichier à chaque écriture dans ce fichier par un processus de l'application à sauvegarder. Cette solution ne traite pas le cas des modifications externes et nécessite de savoir que l'application sera sauvegardée.

La deuxième solution consiste à attendre la création d'un point de reprise pour reporter les données sur le disque (cela nécessite de savoir que l'application sera sauvegardée). Il s'agit donc d'un changement profond de la politique du système d'exploitation qui risque d'être pénalisant en terme de performance. En effet, une masse très importante de données est alors réellement écrite sur le disque à chaque point de reprise. De plus, il se peut que le cache mémoire soit plein, il faut dans ce cas forcer un point de reprise.

La dernière solution consiste à utiliser un système de gestion de fichiers capable de gérer les versions par journalisation [28]. Le numéro de version peut alors correspondre à celui du point de reprise de

l'application mais il est dans ce cas difficile de gérer deux applications accédant en écriture au même fichier.

Dans tous les cas, il est difficile de traiter les fichiers utiles à l'application mais modifiés à l'extérieur de celle-ci. De même, le problème des fichiers ouverts par l'application entre deux points de reprise mais fermés au moment des points de reprise n'est pas approfondi ici.

---

**Algorithme 4** : Algorithme d'endormissement de l'ensemble des processus d'une application avec support des applications client-serveur

---

**Coordination globale hiérarchique par endormissement (*IdRacine*, *IdPR*) :**

```
{
    // IdRacine est l'identifiant d'un processus racine principale
    // IdPR est l'identifiant du point de reprise de l'application

    Ajouter IdRacine à la liste des racines principales de l'application
    // liste stockée dans un conteneur mémoire spécifique à la sauvegarde

    Racine = RecupereProcessusParIdentifiant(IdRacine) // fonction spécifique KERRIGHED

    pour tout processus P appartenant aux descendants locaux du processus Racine ou étant Racine faire
        si P a capacité CHECKPOINTABLE alors // consultation de la krg_task
            endormir(P)
            ajouter Processus.Identifiant au fichier filiation_IdRacine.bin

            pour tout fils distant F du processus P faire
                // effectuer un appel bloquant de procédure à distance.
                // c'est ce même algorithme qui est appelé.
                RPC_Sync_PreparerProcessus (F.identifiant)
            fin pour

            pour tout flux de communication C auquel P est attaché faire
                si C n'est pas dans la liste des flux parcourus alors
                    // cette liste est dans un conteneur en mémoire dédié à cette sauvegarde

                    pour tout processus Pr, différent de P, attaché à C faire
                        // effectuer un appel bloquant de procédure à distance.
                        // c'est ce même algorithme qui est appelé sauf que l'on
                        // recherche la racine principale à l'autre extrémité
                        RPC_Sync_PreparerRacineProcessus (Pr.identifiant, IdPR)
                    fin pour
                fin si
            fin pour
        fin si
    fin pour
}
```

---



## Chapitre 4

# Mise en œuvre et expérimentation

Ce chapitre présente la mise en œuvre actuelle et les tests d'expérimentations qui ont été réalisés afin de la valider et d'évaluer les temps de sauvegarde et restauration de point de reprise d'applications.

### 4.1 Mise en œuvre

Les contributions présentées dans le chapitre 3 ont été mises en œuvre sur la dernière version stable de KERRIGHED, la version 1.0.2 utilisant le noyau LINUX 2.4.29.

Nous nous sommes attachés particulièrement à la sauvegarde efficace du point de reprise d'une application parallèle communiquant par messages tout en permettant la restauration de l'application.

Dans le cadre du stage, la méthode de coordination globale a été implémentée en utilisant les algorithmes simplifiés présentés dans les paragraphes 3.5 et 3.6. L'interface utilisateur a été étendue permettant la réalisation de multiples sauvegardes d'une application — en indiquant le processus racine principale — et la restauration de la version du point de reprise que l'utilisateur souhaite. Enfin, nous nous sommes appliqués au bon fonctionnement de ce mécanisme pour les applications dont les processus communiquent par un flux de communication ouvert par une interface `socket`. La mise en œuvre du prototype consiste en une extension des modules (cf. paragraphe 2.3) EPM et KERSTREAM et fait intervenir les modules KERRPC, GHOST, KERMM, CAPABILITIES et KERPROC.

Il subsiste quelques limitations dues à la complexité de la mise en œuvre et au manque de temps. Ainsi, le calcul de la frontière n'utilise pas les algorithmes présentés en paragraphe 3.7.1 et ne prend donc pas en compte les flux de communication. Par ailleurs, la restauration a lieu de manière totalement séquentielle. La limitation majeure est liée au mécanisme d'export de l'état d'un processus [33]. En effet, il est impossible de réaliser le point de reprise d'un processus lorsque celui-ci est bloqué dans un appel système. Dans le cadre de la méthode de la coordination globale, cela revient à interdire le point de reprise de l'application. Ce problème devrait être corrigé dans une future version de KERRIGHED.

### 4.2 Conditions d'expérimentations

Les expérimentations ont été réalisées sur une grappe d'ordinateurs de 2 à 4 nœuds, équipés d'un processeur *Pentium III* 1 GHz et 512 Mo de mémoire vive, et reliés par un réseau *Fast Ethernet*.

Quatre applications *A*, *B*, *C* et *D* différentes ont été utilisées. L'application *A* génère une arborescence de 16 processus (cf. Fig. 4.1) répartis sur l'ensemble des nœuds de la grappe, chaque processus réalisant un calcul indépendant sur une matrice d'entiers. Ces processus sont créés par l'appel système `fork`. L'application *B* est similaire à l'application *A* mis à part que la matrice d'entiers est de taille plus

```

0  \_ Processus 1
1      \_ Processus 2
2      |  \_ Processus 3
3      |  |  \_ Processus 4
0      |  |  |  \_ Processus 5
3      |  |  \_ Processus 6
2      |  \_ Processus 7
3      |  |  \_ Processus 8
2      |  \_ Processus 9
1      \_ Processus 10
2      |  \_ Processus 11
3      |  |  \_ Processus 12
2      |  \_ Processus 13
1      \_ Processus 14
2      |  \_ Processus 15
1      \_ Processus 16

```

FIG. 4.1 – Répartition des processus de l’application *A* ou *B* sur 4 nœuds numérotés de 0 à 3 (chiffre de la première colonne)

importante permettant de comparer l’impact de la taille d’un processus sur la rapidité de création du point de reprise. Le programme *C* est semblable à *B* à cela près qu’il génère une arborescence de 64 processus. Enfin l’application *D* est une application simple qui met en relation 2 processus communiquant par un flux de type `socket_INET`. Le code source de ces applications est disponible en annexe.

L’ordonnanceur global de tâche est désactivé pendant les tests. Ainsi la répartition des processus est toujours la même et suit la règle suivante : un processus fils s’exécute sur le nœud  $n + 1$  modulo le nombre total de nœuds avec  $n$  l’identifiant du nœud parent.

Les mesures de sauvegarde de points de reprise sont le résultat d’une moyenne sur 5 mesures différentes consécutives réalisées juste après redémarrage de l’ensemble des nœuds de la grappe. Le temps de restauration d’une application a été mesuré en une fois juste après redémarrage, les mesures suivantes étant faussées par la mise en cache mémoire des fichiers de point de reprise.

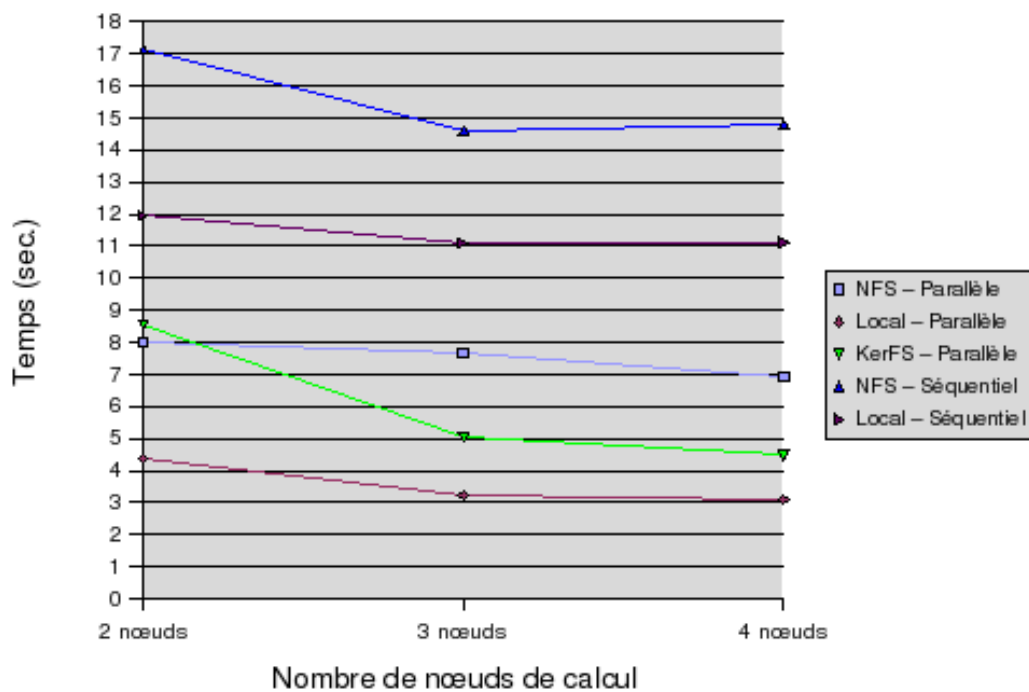
Lorsque le système de fichiers utilisé pour conserver les points de reprise est *NFS*, le serveur est le même pour l’ensemble des nœuds et ne fait pas partie des nœuds de calcul. Celui-ci est relié sur le même switch *Fast Ethernet* que les nœuds de calcul. Les sauvegardes sur disque local s’effectue sur un système de gestion de fichiers (SGF) de type *Ext2*. Les fichiers sont sauvegardés sur le disque local du nœud qui les a créés. Il faut noter que l’utilisation des disques locaux ne permet pas de restaurer l’application directement. L’utilisateur doit au préalable réaliser lui-même les copies de fichiers nécessaires.

### 4.3 Analyse des résultats

La première expérience a pour objectif d’étudier l’impact de la taille du point de reprise sur le temps de sauvegarde du point de reprise. Le temps de sauvegarde, en utilisant le système de fichier *KERFS*, a été mesuré pour les applications *A* (2,59 s) et *B* (4,47 s) s’exécutant sur 4 nœuds avec une répartition identique (cf. Fig. 4.1). L’état d’un processus de l’application *A* nécessite 2,2 Mo pour être conservé sur disque, celui d’un processus de l’application *B* en demande 4 Mo. Cette expérience montre que le temps de sauvegarde d’un point de reprise global est directement proportionnel à la taille du point de reprise.

	Parallèle			Séquentiel	
	NFS	KerFS	SGF Local	NFS	SGF Local
2 nœuds	7,98	8,52	4,38	17,13	11,95
3 nœuds	7,64	5,04	3,24	14,61	11,09
4 nœuds	6,94	4,47	3,08	14,77	11,12

(a)

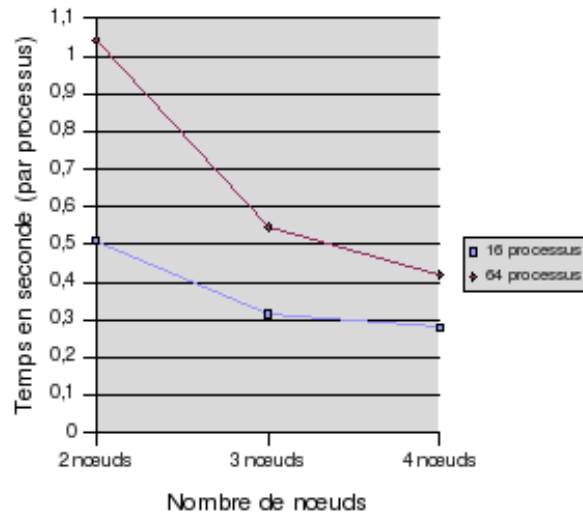


(b)

FIG. 4.2 – Sauvegarde séquentielle ou parallèle de points de reprise de l'application *B* — Comparaison de temps entre stockage sur disques locaux, sur NFS et sur KerFS.

	16 processus	64 processus
2 nœuds	0,51	1,04
3 nœuds	0,31	0,55
4 nœuds	0,28	0,42

(a)

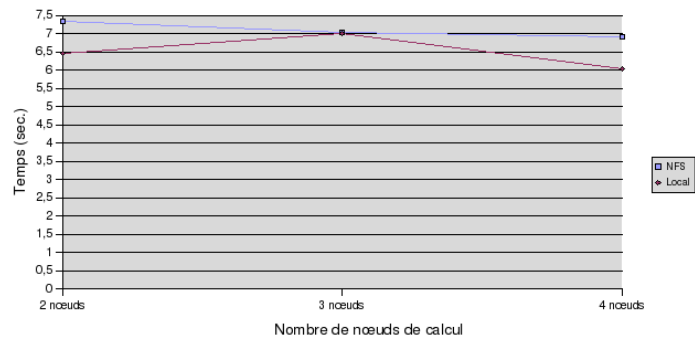


(b)

FIG. 4.3 – Sauvegarde de points de reprise des applications *B* et *C* — Les temps indiqués (en sec.) correspondent aux temps de sauvegarde du point de reprise global divisés par le nombre de processus.

	NFS	SGF Local
2 nœuds	7,34	6,46
3 nœuds	7,02	7,01
4 nœuds	6,92	6,02

(a)



(b)

FIG. 4.4 – Restauration de l'application *B* à partir de son point de reprise — Comparaison de temps (en secondes) entre stockage sur disques locaux, et sur NFS

Les résultats de la deuxième expérience sont visibles en figure 4.2. L'objectif de cette expérience est de comparer l'utilisation d'un algorithme parallèle par rapport à une version séquentielle pour la sauvegarde de point de reprise d'application et d'apprécier l'impact de la méthode de stockage utilisée. Le temps de création d'un point de reprise de l'application *B* est mesuré d'une part en utilisant l'algorithme parallèle avec les systèmes de gestion de fichiers NFS, KERFS et locaux et d'autre part en utilisant un algorithme séquentiel avec NFS et sur disques locaux. Dans le cadre de l'algorithme séquentiel, le temps de sauvegarde du point de reprise mesuré est relativement constant. La diminution importante du temps sur NFS en passant de 2 à 3 nœuds s'explique par une saturation moins rapide du cache disque lorsque le nombre de processus par nœuds est moins important. La sauvegarde en parallèle des processus locaux de l'application et des processus distants apporte un gain de performances substantiel par rapport à la version séquentielle. On observe un ralentissement sur NFS comparativement à la sauvegarde sur les disques locaux, ceci est dû à l'écriture concurrente des points de reprise de chaque processus à travers le réseau. L'utilisation de KERFS apporte de meilleures performances tout en permettant une visibilité des fichiers sur l'ensemble de la grappe. Nous constatons la nécessité de bénéficier d'un système de gestion de fichiers distribué pour avoir les meilleures performances.

La troisième expérience cherche à démontrer que plus l'application s'exécute sur un grand nombre de nœuds, plus le temps de sauvegarde du point de reprise est faible. Cette expérience indique le temps de sauvegarde du point de reprise des applications *B* et *C* s'exécutant sur 2, 3 ou 4 nœuds. L'état des processus de l'application *B* et de l'application *C* est de taille similaire. La sauvegarde a été réalisée en utilisant l'algorithme parallèle. La figure 4.3 montre que l'hypothèse est vérifiée.

La quatrième expérience a pour but de valider la restauration de point de reprise d'une application sur un nombre inférieur de nœuds à celui présent lors de la sauvegarde du point de reprise. Le point de reprise a été sauvegardé alors que l'application *B* s'exécutait sur 4 nœuds. La restauration a lieu sur 2, 3 ou 4 nœuds, le point de reprise étant stocké sur NFS ou recopié sur chacun des disques locaux des nœuds. La faible différence de temps mesurée (cf. Figure 4.4) entre l'utilisation des différents systèmes de gestion de fichiers s'explique par le fait que la restauration se fait de manière totalement séquentielle.

L'objectif de la dernière expérience est la validation de la sauvegarde et la restauration d'une application dont les processus communiquent par flux de communication. L'application *D* est constituée de 2 processus. Ces 2 processus sont père et fils. Ils communiquent par un flux de communication de type `socket_INET`. La sauvegarde du point de reprise d'une taille de 352 Ko a pris 0,156 secondes en moyenne (en utilisant NFS) et 0,093 secondes pour la restauration.



# Conclusion

Il existe un réel besoin de mécanismes de tolérance aux fautes pour pallier les défaillances lors de l'exécution des applications longue durée sur grappe comme en témoigne l'abondance des travaux sur les bibliothèques *MPI* et *PVM* tolérantes aux fautes. Mes travaux ont porté sur la mise en œuvre dans un système d'exploitation pour grappe. Nous avons ainsi étudié l'intégration dans le système à image unique (*SSI*) *KERRIGHED* d'un mécanisme de sauvegarde et restauration d'application parallèle communiquant par messages. L'idée est la transparence totale pour les applications et l'indépendance vis à vis des environnements de programmation *MPI*, *PVM*, etc. et de leurs multiples mises en œuvre.

L'étude des systèmes existants montre que s'il existe des mécanismes de sauvegarde de l'état d'un processus mis en œuvre dans le système d'exploitation, lorsqu'il s'agit de mécanismes de reprise d'applications parallèles, les solutions existantes sont mises en œuvre dans les environnements de programmation tels que *MPI* ou *PVM* ou directement dans le code des applications. Nos travaux proposent une approche originale qui vise à intégrer des stratégies de tolérance aux fautes dans un système d'exploitation pour grappe. Nous avons spécifié une interface système de sauvegarde et restauration de points de reprise d'application uniforme que l'application soit séquentielle ou parallèle et permet la sauvegarde de point de reprise à l'initiative du système. Des mécanismes, indispensables pour une transparence totale, ont été mis en œuvre pour la détermination pendant l'exécution d'une application de l'ensemble des processus de cette application à partir de l'identifiant (PID) de l'un d'entre eux en exploitant les informations système sur la filiation et les flux de communication. Nous avons mis en œuvre une stratégie de coordination globale de sauvegarde de points de reprise et de mécanisme de reprise associé qui nous a conduit en particulier à définir le format des points de reprise de l'application. Suite aux expérimentations, nous constatons le gain de performance substantiel amené par le parallélisme de la sauvegarde des points de reprise des processus de l'application.

Par la suite, il convient d'implémenter une interface de programmation équivalente à l'interface en ligne de commande et de compléter cette dernière avec la gestion du ramasse-miettes.

Au niveau de la mise en œuvre, plusieurs axes sont à poursuivre. Il est intéressant de paralléliser la restauration de point de reprise et d'implémenter les algorithmes présentés précédemment qui inclue la gestion des flux pour le calcul de la frontière de l'application. Par ailleurs, il convient de valider la solution avec des applications réelles — utilisant une bibliothèque *MPI* (*Lam-MPI* et *MPICH*) — sur un plus grand nombre de nœuds. Cela nécessite de régler le problème des appels systèmes bloquant évoqué dans le paragraphe 4.1. Enfin, il est important de suivre l'évolution de *KERRIGHED*.

D'un point de vue recherche, le système de gestion de fichiers *KERFS* doit évoluer pour gérer différentes versions d'un même fichier. Le mécanisme de points de reprise coordonnés doit être étendu afin de supporter les applications parallèles à mémoire partagée. Il apparaît également intéressant de proposer d'autres stratégies, de les comparer sur une même plateforme et d'offrir le choix au programmeur. Cela nécessite de mettre en place des mécanismes de journalisation, d'estampille et de gestion de multiples points de reprise pour une application. Enfin, il convient d'étudier le passage à l'échelle de la solution sur une fédération de grappes.



# Bibliographie

- [1] Top 500 Supercomputer Sites. <http://www.top500.org/>.
- [2] Message Passing Interface Forum. <http://www.mpi-forum.org/>.
- [3] Linux Standard Base. <http://www.linuxbase.org/>.
- [4] Adnan Agbaria and Roy Friedman. Starfish : Fault-tolerant dynamic MPI programs on clusters of workstations. In *8th IEEE International Symposium on High Performance Distributed Computing*, Los Alamitos, 1999.
- [5] Lorenzo Alvisi, Elmootazbellah Elnozahy, Sriram S. Rao, Syed A. Husain, and Asanka Del Mel. An analysis of communication-induced checkpointing. Technical report, 1999.
- [6] George Bosilca, Aurelien Bouteiller, Franck Cappello, Samir Djilali, Gilles Fedak, Cecile Germain, Thomas Herault, Pierre Lemarinier, Oleg Lodygensky, Frederic Magniette, Vincent Neri, and Anton Selikhov. MPICH-V : toward a scalable fault tolerant MPI for volatile nodes. In *Supercomputing '02 : Proceedings of the 2002 ACM/IEEE conference on Supercomputing*, pages 1–18. IEEE Computer Society Press, 2002.
- [7] Aurélien Bouteiller, Hinde-Lilia Bouziane, Pierre Lemarinier, Thomas Héroult, and Franck Cappello. Hybrid preemptive scheduling for mpi applications on the grids. Presentation in 5th IEEE/ACM International Workshop on Grid Computing, Novembre 2004.
- [8] Aurélien Bouteiller, Franck Cappello, Thomas Héroult, Géraud Krawezik, Pierre Lemarinier, and Frédéric Magniette. MPICH-V2 : a fault tolerant MPI for volatile nodes based on pessimistic sender based message logging. In *High Performance Networking and Computing (SC2003)*, Phoenix USA, November 2003. IEEE/ACM.
- [9] Aurélien Bouteiller, Pierre Lemarinier, Thomas Héroult, Géraud Krawezik, and Franck Cappello. Improved message logging versus improved coordinated checkpointing for fault tolerant MPI. In *The 2004 IEEE International Conference on Cluster Computing*, San Diego USA, September 2004.
- [10] Tushar Deepak Chandra and Sam Toueg. Unreliable failure detectors for reliable distributed systems. *J. ACM*, 43(2) :225–267, 1996.
- [11] K. Mani Chandy and Leslie Lamport. Distributed snapshots : determining global states of distributed systems. *ACM Trans. Comput. Syst.*, 3(1) :63–75, 1985.
- [12] J. Duell, P. Hargrove, and E. Roman. The design and implementation of Berkeley Lab’s Linux Checkpoint/Restart. Technical report, Future Technologies Group white paper, 2003.
- [13] E. N. (Mootaz) Elnozahy, Lorenzo Alvisi, Yi-Min Wang, and David B. Johnson. A survey of rollback-recovery protocols in message-passing systems. *ACM Comput. Surv.*, 34(3) :375–408, 2002.
- [14] Graham E. Fagg and Jack Dongarra. FT-MPI : Fault tolerant MPI, supporting dynamic applications in a dynamic world. In *Proceedings of the 7th European PVM/MPI Users’ Group Meeting on Recent*

- Advances in Parallel Virtual Machine and Message Passing Interface*, pages 346–353. Springer-Verlag, 2000.
- [15] Pascal Gallard. *Conception d'un service de communication pour systèmes d'exploitation distribués pour grappes de calculateurs : mise en oeuvre dans le système à image unique Kerrighed*. Thèse de doctorat, IRISA, Université de Rennes 1, IRISA, Rennes, France, December 2004.
- [16] Pascal Gallard and Christine Morin. Dynamic streams for efficient communications between migrating processes in a cluster. In *Euro-Par 2003 : Parallel Processing*, volume 2790 of *Lect. Notes in Comp. Science*, pages 930–937, Klagenfurt, Austria, August 2003. Springer-Verlag.
- [17] Richard Koo and Sam Toueg. Checkpointing and rollback-recovery for distributed systems. *IEEE Trans. Software Eng.*, 13(1) :23–31, 1987.
- [18] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7) :558–565, 1978.
- [19] Juan Leon, Allan L. Fisher, and Peter Steenkiste. Fail-Safe PVM : A portable package for distributed programming with transparent recovery. Technical report, 1993.
- [20] Renaud Lottiaux. *Gestion globale de la mémoire physique d'une grappe pour un système à image unique : mise en oeuvre dans le système Gobelins*. Thèse de doctorat, IRISA, Université de Rennes 1, December 2001.
- [21] Renaud Lottiaux and Christine Morin. Containers : A sound basis for a true Single System Image. In *Proceeding of IEEE International Symposium on Cluster Computing and the Grid (CCGrid '01)*, pages 66–73, Brisbane, Australia, May 2001.
- [22] Soulla Louca, Neophytos Neophytou, Arianos Lachanas, and Paraskevas Evripidou. MPI-FT : Portable fault tolerance scheme for MPI. *Parallel Processing Letters*, 10(4) :371–382, 2000. World Scientific Publishing Company.
- [23] Christine Morin, Renaud Lottiaux, Geoffroy Vallée, Pascal Gallard, David Margery, Jean-Yves Berthou, and Isaac Scherson. Kerrighed and data parallelism : Cluster computing on Single System Image operating systems. In *Proc. of Cluster 2004*. IEEE, September 2004.
- [24] Steven Osman, Dinesh Subhraveti, Gong Su, and Jason Nieh. The design and implementation of zap : a system for migrating computing environments. *SIGOPS Oper. Syst. Rev.*, 36(SI) :361–376, 2002.
- [25] Eduardo Pinheiro. Truly-transparent checkpointing of parallel applications. [cite-seer.ist.psu.edu/434768.html](http://cite-seer.ist.psu.edu/434768.html).
- [26] Sriram Rao, Lorenzo Alvisi, and Harrick M. Vin. Egida : An extensible toolkit for low-overhead fault-tolerance. In *FTCS '99 : Proceedings of the Twenty-Ninth Annual International Symposium on Fault-Tolerant Computing*, page 48. IEEE Computer Society, 1999.
- [27] Sriram Sankaran, Jeffrey M. Squyres, Brian Barrett, Andrew Lumsdaine, Jason Duell, Paul Hargrave, and Eric Roman. The LAM/MPI checkpoint/restart framework : System-initiated checkpointing. In *Proceedings, LACSI Symposium*, Sante Fe, New Mexico, USA, October 2003.
- [28] Douglas S. Santry, Michael J. Feeley, Norman C. Hutchinson, Alistair C. Veitch, Ross W. Carton, and Jacob Ofir. Deciding when to forget in the elephant file system. In *SOSP '99 : Proceedings of the seventeenth ACM symposium on Operating systems principles*, pages 110–123, New York, NY, USA, 1999. ACM Press.
- [29] Georg Stellner. CoCheck : Checkpointing and process migration for MPI. In *IPPS '96 : Proceedings of the 10th International Parallel Processing Symposium*, pages 526–531. IEEE Computer Society, 1996.

- [30] V. S. Sunderam. PVM : a framework for parallel distributed computing. *Concurrency, Practice and Experience*, 2(4) :315–340, 1990.
- [31] Douglas Thain, Todd Tannenbaum, and Miron Livny. Distributed computing in practice : The Condor experience. *Concurrency and Computation : Practice and Experience*, 2004.
- [32] Geoffroy Vallée. *Conception d'un ordonnanceur de processus adaptable pour la gestion globale des ressources dans les grappes de calculateurs : mise en oeuvre dans le système d'exploitation Kerrighed*. Thèse de doctorat, IFSIC, Université de Rennes 1, France, March 2004.
- [33] Geoffroy Vallée, Renaud Lottiaux, David Margery, Christine Morin, and Jean-Yves Berthouand. Ghost process : a sound basis to implement new mechanisms for global process management in linux clusters. In *ISPDC 2005*, Lille, France, July 2005.
- [34] Gengbin Zheng, Lixia Shi, and Laxmikant V. Kalé. FTC-Charm++ : An in-memory checkpoint-based fault tolerant runtime for Charm++ and MPI. In *2004 IEEE International Conference on Cluster Computing*, San Diego, CA, September 2004.
- [35] Hua Zhong and Jason Nieh. CRAK : Linux checkpoint/restart as a kernel module. <http://www.ncl.cs.columbia.edu/research/migrate/crak.html>, November 2001.

# Annexes

<b>A</b>	<b>Code source de l'application de test <i>A</i></b>	<b>57</b>
<b>B</b>	<b>Code source de l'application de test <i>B</i></b>	<b>59</b>
<b>C</b>	<b>Code source de l'application de test <i>C</i></b>	<b>61</b>
<b>D</b>	<b>Code source de l'application de test <i>D</i></b>	<b>63</b>

# Annexes

## A Code source de l'application de test A

```
1 #include <stdio.h>
2 #include <sys/types.h>
3 #include <unistd.h>
4 #include <sys/wait.h>
5 #include <stdlib.h>
6
7 #define NB_SONS 4
8 #define TABSIZE 524288 // 1048576
9 int numloops = -1 ;
10
11 void parse_args(int argc , char *argv [])
12 {
13     int c;
14
15     while (1){
16         c = getopt(argc , argv , "l:h");
17         if (c == -1)
18             break;
19         switch (c) {
20             case 'l':
21                 numloops = atoi(optarg);
22                 break;
23             default:
24                 printf("** unknown option\n");
25             case 'h':
26                 printf("usage: %s [-h] [-l N]\n" , argv[0]);
27                 printf(" -h   : this help\n");
28                 printf(" -l N : number of loops\n");
29                 exit(1);
30         }
31     }
32 }
33
34
35 int main(int argc , char *argv [])
```

```

36 {
37     int i, j, k, pid, fpid ;// , status ; used for waitpid
38     int sonlist[NB_SONS] ;
39     int calcultab[TABSIZE];
40
41     parse_args(argc , argv);
42
43     fpid = getpid();
44
45     printf("Debut de l'application FORK\n");
46
47
48     //-----
49     // -- creation des differents processus --
50     //-----
51     if (fpid == getpid()) {
52         printf ("* Creation des processus...");
53         fflush(stdout) ;
54     }
55
56     for (i=0; i < NB_SONS; i++)
57     {
58         pid = fork() ;
59         if (pid)
60         {
61             sonlist[i] = pid ;
62         }
63         else
64         {
65             for (j = 0; j <=i; j++)
66                 sonlist[j] = 0 ;
67         }
68     }
69
70     if (fpid == getpid()) printf ("DONE\n");
71
72     //-----
73     sleep(10);
74
75     //-----
76     // -- seance de calcul intensif ;-)
77     //-----
78     for (i = 0; i < numloops; i++)
79     {
80         for (j = 0; j < 1000000; j++)
81         {
82             k = k + i * j ;

```

```

83         calcultab[j%TABSIZ] = k;
84     }
85 }
86
87 //-----
88 // -- attente des fils
89 //-----
90 /* for (i = 0; i < NB_SONS; i++) */
91 /*     { */
92 /*         if (sonlist[i] != 0) */
93 /*             waitpid(sonlist[i], &status, 0); */
94 /*     } */
95
96
97 if (fpid == getpid()) {
98     printf ("\n-- Fin du processus pere --\n\n");
99 }
100 else printf ("Fin du processus fils %d\n", pid);
101
102 return 0 ;
103 }

```

## B Code source de l'application de test B

```

1 #include <stdio.h>
2 #include <sys/types.h>
3 #include <unistd.h>
4 #include <sys/wait.h>
5 #include <stdlib.h>
6
7 #define NB_SONS 4
8 #define TABSIZE 1048576 // 4096
9 int numloops = -1 ;
10
11 void parse_args(int argc, char *argv[])
12 {
13     int c;
14
15     while (1){
16         c = getopt(argc, argv, "l:h");
17         if (c == -1)
18             break;
19         switch (c) {
20             case 'l':
21                 numloops = atoi(optarg);
22                 break;
23             default:

```

```

24         printf("** unknown option\n");
25     case 'h':
26         printf("usage: %s [-h] [-l N]\n", argv[0]);
27         printf("-h    : this help\n");
28         printf("-l N : number of loops\n");
29         exit(1);
30     }
31 }
32 }
33
34
35 int main(int argc, char *argv[])
36 {
37     int i, j, k, pid, fpid; // , status; used for waitpid
38     int sonlist[NB_SONS];
39     int calcultab[TABSIZE];
40
41     parse_args(argc, argv);
42
43     fpid = getpid();
44
45     printf("Debut de l'application FORK\n");
46
47
48     //-----
49     // -- creation des differents processus --
50     //-----
51     if (fpid == getpid()) {
52         printf ("* Creation des processus...");
53         fflush(stdout);
54     }
55
56     for (i=0; i < NB_SONS; i++)
57     {
58         pid = fork();
59         if (pid)
60         {
61             sonlist[i] = pid;
62         }
63         else
64         {
65             for (j = 0; j <=i; j++)
66                 sonlist[j] = 0;
67         }
68     }
69
70     if (fpid == getpid()) printf ("DONE\n");

```

```

71
72 //-----
73 sleep(10);
74
75 //-----
76 // -- seance de calcul intensif ;-)
77 //-----
78 for (i = 0; i < numloops; i++)
79 {
80     for (j = 0; j < 1000000; j++)
81     {
82         k = k + i * j ;
83         calcultab[j%TABSIZ] = k;
84     }
85 }
86
87 //-----
88 // -- attente des fils
89 //-----
90 /* for (i = 0; i < NB_SONS; i++) */
91 /*     { */
92 /*         if (sonlist[i] != 0) */
93 /*             waitpid(sonlist[i], &status , 0); */
94 /*     } */
95
96
97 if (fpid == getpid()) {
98     printf ("\n-- Fin du processus pere --\n\n");
99 }
100 else printf ("Fin du processus fils %d\n", pid);
101
102 return 0 ;
103 }

```

## C Code source de l'application de test C

```

1 #include <stdio.h>
2 #include <sys/types.h>
3 #include <unistd.h>
4 #include <sys/wait.h>
5 #include <stdlib.h>
6
7 #define NB_SONS 6
8 #define TABSIZE 1048576 // 4096
9 int numloops = -1 ;
10
11 void parse_args(int argc , char *argv[])

```

```

12 {
13     int c;
14
15     while (1){
16         c = getopt(argc , argv , "l:h");
17         if (c == -1)
18             break;
19         switch (c) {
20             case 'l':
21                 numloops = atoi(optarg);
22                 break;
23             default:
24                 printf("** unknown option\n");
25             case 'h':
26                 printf("usage: %s [-h] [-l N]\n" , argv[0]);
27                 printf(" -h      : this help\n");
28                 printf(" -l N   : number of loops\n");
29                 exit(1);
30         }
31     }
32 }
33
34
35 int main(int argc , char *argv [])
36 {
37     int i , j , k , pid , fpid ; // , status ; used for waitpid
38     int sonlist[NB_SONS] ;
39     int calcultab[TABSIZE];
40
41     parse_args(argc , argv);
42
43     fpid = getpid();
44
45     printf("Debut de l'application FORK\n");
46
47
48     //-----
49     // -- creation des differents processus --
50     //-----
51     if (fpid == getpid()) {
52         printf ("* Creation des processus...");
53         fflush(stdout) ;
54     }
55
56     for (i=0; i < NB_SONS; i++)
57     {
58         pid = fork() ;

```

```

59     if (pid)
60     {
61         sonlist[i] = pid ;
62     }
63     else
64     {
65         for (j = 0; j <=i; j++)
66             sonlist[j] = 0 ;
67     }
68 }
69
70 if (fpid == getpid()) printf ("DONE\n");
71
72 //-----
73 sleep(10);
74
75 //-----
76 // -- seance de calcul intensif ;-)
77 //-----
78 for (i = 0; i < numloops; i++)
79 {
80     for (j = 0; j < 1000000; j++)
81     {
82         k = k + i * j ;
83         calcultab[j%TABSIZ] = k;
84     }
85 }
86
87 //-----
88 // -- attente des fils
89 //-----
90 /* for (i = 0; i < NB_SONS; i++) */
91 /* { */
92 /*     if (sonlist[i] != 0) */
93 /*         waitpid(sonlist[i], &status , 0); */
94 /*     } */
95
96
97 if (fpid == getpid()) {
98     printf ("\n-- Fin du processus pere --\n\n\n");
99 }
100 else printf ("Fin du processus fils %d\n", pid);
101
102 return 0 ;
103 }

```

## D Code source de l'application de test D

```

1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <unistd.h>
4 #include <sys/types.h>
5 #include <sys/socket.h>
6 #include <netdb.h>
7 #include <netinet/in.h>
8 #include <string.h>
9
10 int main(int argc , char *argv [])
11 {
12     int     sockfd ,newsockfd ,clilen ;
13     char c ;
14     struct sockaddr_in cli_addr ,serv_addr ;
15     int pid ;
16
17     int j ,k ;
18
19     pid = fork() ;
20     if (pid) {
21         printf ("(server) starting [%d]\n" ,getpid ()) ;
22
23         /* ouverture du socket */
24         sockfd = socket (AF_INET , SOCK_STREAM , 0) ;
25         if (sockfd < 0) {
26             perror ("(server) socket") ;
27             exit (0) ;
28         }
29         printf ("(server) socket : OK\n") ;
30
31
32         /* initialisation des parametres */
33         bzero ((char *) &serv_addr , sizeof (serv_addr)) ;
34         serv_addr . sin_family      = AF_INET ;
35         serv_addr . sin_addr . s_addr = htonl (INADDR_ANY) ;
36         serv_addr . sin_port       = htons (8080) ;
37
38         /* effecture le bind */
39         if (bind (sockfd ,(struct sockaddr *)&serv_addr , sizeof (serv_addr))
40             < 0) {
41             perror ("(server) bind") ;
42             exit (0) ;
43         }
44         printf ("(server) bind : OK\n") ;
45
46         /* petit initialisation */

```

```

47     if (listen(sockfd,1) != 0) {
48         perror ("(server) listen");
49         exit(0);
50     }
51     printf ("(server) listen : OK\n");
52
53     /* attend la connection d'un client */
54     printf("(server) waiting for connection...\n");
55     fflush(stdout);
56     clilen = sizeof (cli_addr);
57     newsockfd = accept (sockfd ,(struct sockaddr*) &cli_addr , &clilen)
        ;
58
59     if (newsockfd < 0) {
60         perror ("(server) accept");
61         exit(0);
62     }
63
64     printf ("(server) connected\n");
65
66     // c'est moi qui fait les receive et affiche au client
67     while (1) {
68         int r = read(newsockfd,&c,1);
69         if (r != 1) {
70             printf ("(server) %d characters read\n" , r);
71             perror ("(server) read");
72             break;
73         }
74         else {
75             if (c == EOF) break;
76             printf ("(server) read \"%c\" \n" , c); fflush(stdout);
77             fflush(stdout);
78         }
79     }
80
81     // Ouh, le joli calcul pdt lequel on doit checkpointer !
82     printf ("(server) mega calcul start\n"); fflush(stdout);
83     j=0;
84     for (j=0; j<50;j++)
85         for (k=0;k<100000000;k++) ;
86
87     printf ("(server) mega calcul done\n"); fflush(stdout);
88
89     // c'est moi qui fait les receive et affiche au client
90     while (1) {
91         int r = read(newsockfd,&c,1);
92         if (r != 1) {

```

```

93     printf("(server) %d characters read\n", r);
94     perror("(server) read");
95     break;
96 }
97 else {
98     if (c == EOF) break;
99     printf("(server) read \"%c\" \n", c); fflush(stdout);
100    fflush(stdout);
101 }
102 }
103
104 printf("\n(server) connexion closing ... 1\n");
105 close(sockfd);
106 printf("\n(server) connexion closing ... 2\n");
107
108 close(newsockfd);
109 printf("\n(server) connexion closed\n");
110
111 } else {
112     int c=0, r=0;
113
114     /*
115      * partie client
116      */
117     printf("(client) starting [%d]\n", getpid());
118
119     sleep(10);
120
121     /* initialise la structure de donnee */
122     bzero((char*) &serv_addr, sizeof(serv_addr));
123     serv_addr.sin_family = AF_INET;
124     serv_addr.sin_addr.s_addr = htonl(INADDR_ANY);
125     serv_addr.sin_port = htons(8080);
126
127     /* ouvre le socket */
128     if ((sockfd = socket(AF_INET, SOCK_STREAM, 0)) < 0) {
129         perror("(client) socket");
130         exit(0);
131     }
132     printf("(client) socket : OK\n");
133
134
135     /* effectue la connexion */
136     if (connect(sockfd, (struct sockaddr*)&serv_addr, sizeof(serv_addr))
137         < 0) {
138         perror("(client) connect");
139         exit(0);

```

```

139     }
140
141
142     printf("(client) connected\n");
143
144     /* genere les lettres de l'alphabet... */
145     for (c=33; c<125 ;c++){
146         long int i;
147         printf("(client) send \"%d:%c\" \t",c,c);fflush(stdout);
148         r = write (sockfd, &c,1);
149
150         if (r!=1) {
151             perror("(client) write");
152             break;
153         }
154         // sleep(1);
155         for (i=0;i<1000000;i++) ;
156     }
157     c=EOF;
158     r=write (sockfd,&c,1);
159     if (r!=1) perror("(client) write");
160
161     printf("(client) mega calcul start\n");fflush(stdout);
162     j=0;
163     for (j=0; j<50;j++)
164         for (k=0;k<100000000;k++) ;
165
166     printf("(client) mega calcul done\n");fflush(stdout);
167
168
169     /* genere les lettres de l'alphabet... */
170     for (c=33; c<125 ;c++){
171         long int i;
172         printf("(client) send \"%d:%c\" \t",c,c);fflush(stdout);
173         r = write (sockfd, &c,1);
174
175         if (r!=1) {
176             perror("(client) write");
177             break;
178         }
179         // sleep(1);
180         for (i=0;i<1000000;i++) ;
181     }
182     c=EOF;
183     r=write (sockfd,&c,1);
184     if (r!=1) perror("(client) write");
185

```

```
186     printf ("\n(client) connexion closing ... \n");
187     close (sockfd);
188     printf ("\n(client) connexion closed \n");
189
190 }
191 return 0;
192 }
```