Techniques de gestion des défaillances dans les grilles informatiques tolérantes aux fautes

Ndeye Massata Ndiaye

THÈSE

PRÉSENTÉE A

L'UNIVERSITÉ PIERRE ET MARIE CURIE

ÉCOLE DOCTORALE : EDITE

Par Ndeye Massata Ndiaye

POUR OBTENIR LE GRADE DE

DOCTEUR

SPÉCIALITÉ : Informatique

Techniques de gestion des défaillances dans les grilles informatiques tolérantes aux fautes

Directeur de recherche : Pierre SENS
Co-directeur de recherche (cotutelle) : Ousmane THIARE

Soutenue le : 17 Septembre 2013

Devant la commission d'examen formée de :

Pierre SENS	Professeur à l'université de Paris 6	Directeur de thèse
Ousmane THIARE	Maitre de conférences – HDR à l'université Gaston Berger	Co-Directeur de thèse
Jean-Fréderic MYOUPO	Professeur à l'université de Picardie	Rapporteur
Eddy CARON	Maitre de conférences – HDR à l'ENS de Lyon	Rapporteur
Franck PETIT	Professeur à l'université de Paris 6	Examinateur
Drissa HOUATRA	Ingénieur de recherche	Examinateur

TABLE DES MATIERES

Chapitre IV: Composition d'algorithmes de point de reprise dans les architectures hiérarchiques

Chapitre V: Un nouveau protocole hiérarchique adaptatif par points de reprise et journalisation

Table des figures

Tableaux

Algorithmes

Chapitre I : Introduction

I. Problématique

La biologie moléculaire, l'astrophysique, la physique des hautes énergies, qui ne sont que quelques exemples parmi les nombreux domaines de recherche, constituent des secteurs en pleine expansion. Ils nécessitent des puissances de calcul énormes pour stocker, traiter, ou analyser les données qu'ils génèrent. Il a fallu trouver des équipements spécifiques, des ordinateurs puissants, qui pourraient répondre aux besoins des applications.

L'évolution naturelle était de diviser le travail entre plusieurs unités de traitement. Le parallélisme a été introduit avec des machines parallèles monolithiques. C'est le début de l'avènement des ***supercalculateurs***. Ces derniers sont des superordinateurs conçus pour le Calcul Haute Performance (HPC). Leur principal objectif est de traiter un très grand volume de données sur une durée limitée. Même si les supercalculateurs ont eu un succès dans les industries, ils se heurtent à plusieurs obstacles liés à leur prix très élevé, à leur coût d'utilisation en termes d'électricité et de maintenance. Aussi l'augmentation de la puissance de calcul des machines pour faire face aux besoins applicatifs spécifiques a ses limites. L'amélioration de la qualité des réseaux a rendu possible le concept des ***grappes de machines*** (ou cluster). Il s'agit d'utiliser les PC standards reliés par un réseau haut débit et dont l'agrégation des ressources permet d'obtenir des performances aux supercalculateurs, à moindre coût. L'avantage des grappes de PC est leur extensibilité : il suffit de rajouter une machine pour augmenter la puissance. Les grappes de PC ont été étendues aux plates-formes distribuées à grande échelle interconnectant des clusters éloignés géographiquement, ce qui conduit à un nouveau champ en informatique, ***l'informatique en grille***.

Le principe fondamental des grilles informatiques est d'agréger un ensemble des ressources disponibles sur plusieurs sites pour obtenir une plus grande puissance de calcul et de stockage dont l'utilisation serait aussi transparente que la fourniture de l'électricité [1]. Ces ressources distribuées géographiquement visent à fournir un accès transparent, peu coûteux et fiable à de grandes capacités de données de calcul. Cette architecture est intrinsèquement fortement hiérarchique avec des communications à faible latence au sein de chaque cluster et des échanges plus soutenues entre les clusters.

Aujourd'hui les architectures en grilles sont très largement répandues que ce soit dans le monde de la recherche avec Grid'5000, TeraGrid, ou dans les entreprises. Elles sont à la base de la plupart des solutions proposées par les opérateurs de Cloud.

Cependant l'utilisation des grilles informatiques pose un certain nombre de problèmes liés à plusieurs facteurs imposés par la nature de l'environnement réparti. Les ressources peuvent être ***volatiles*** car les nœuds peuvent se déconnecter lors d'une perte de connexion ou une panne matérielle, ou de manière volontaire lors d'une utilisation propriétaire. De plus, la grille est exploitée par un ***grand nombre d'utilisateurs***, avec des ***applications de très grande taille*** avec de longue durée d'exécution.

Ainsi, les risques d'apparition de fautes deviennent très élevés. On estime dans les grandes grilles que le temps entre les fautes de machines (MTBF : Mean Time Between Fault) est de l'ordre de quelques heures. Ces fautes vont entrainer des défaillances qui empêchent l'exécution « correcte » des applications. Pour déployer des applications de calcul sur un grand nombre de nœuds, il est alors essentiel d'avoir des protocoles efficaces pour tolérer les fautes.

Dans les environnements répartis, les processus des applications communiquent essentiellement par **passage de messages**. Un message induit une dépendance causale entre un émetteur et un récepteur. La fermeture transitive des dépendances de message définit la dépendance globale de l'application. Dans ce cas, il n'est pas suffisant de simplement

redémarrer les processus défaillants pour le recouvrement des erreurs. De plus, les algorithmes de recouvrement doivent respecter les dépendances des applications et s'assurer que tous les processus soient cohérents entre eux après exécution de la procédure de recouvrement.

De nombreux protocoles de recouvrement arrière ont été développés dans la littérature. Ces protocoles peuvent être classés en deux catégories :

- Les techniques de sauvegarde par points de reprise [10]: chaque processus sauvegarde périodiquement son état et l'enregistre sur un support de stockage stable dans un point de reprise. En cas de panne, tous les processus effectuent un retour arrière vers un point de reprise appartenant à un état global cohérent, limitant ainsi la quantité de calcul perdu.
- Les mécanismes de journalisation des messages : les messages sont sauvegardés pour pouvoir être rejoué dans le même ordre en cas de défaillance.

II. Contributions de la thèse

Les contributions présentées dans cette thèse se déclinent en trois axes :

1) Une étude comparative des protocoles de recouvrement arrière pour sélectionner les meilleurs algorithmes adaptés aux grilles informatiques.

Pour choisir les protocoles pour notre composition hiérarchique, nous avons implémenté six protocoles de tolérances aux fautes les plus communément utilisés: la sauvegarde coordonnée par point de reprise, le protocole de point de reprise coordonné non bloquant de Chandy-Lamport [12], le protocole à synchronisation implicite induit par les communications [10] et les trois principaux mécanismes de journalisation pessimiste, optimiste et causale [16]. Ces algorithmes ont été testés dans deux architectures : une architecture plate qui représente un cluster, et une architecture hiérarchique qui représente la grille composée de nœuds dont le nombre total est égal au nombre de nœuds de la grille.

2) La conception d'algorithmes hiérarchiques.

Suite à l'étude comparative des performances, nous proposons des algorithmes hiérarchiques prenant en compte la topologie physique de grille. Ces algorithmes sont une composition d'algorithmes au niveau inter et intra-cluster.

3) Un nouveau protocole adaptatif hiérarchique.

Le nouveau protocole adaptatif est un algorithme hiérarchique résultant de l'étude comparative des différentes compositions d'algorithmes. Il s'agit de la technique de sauvegarde par point de reprise non-bloquant de Chandy-Lamport et de la journalisation pessimiste basée sur l'émetteur. Le cluster est la composante principale dans laquelle va s'exécuter un protocole : les messages inter-cluster vont êtres sauvegardés en utilisant soit la technique de Chandy- Lamport, soit la journalisation pessimiste ; de même pour les messages échangés à l'intérieur des clusters. Notre nouveau protocole s'adapte à la fréquence des messages échangés dans la grille. Il combine les deux compositions suivantes :

- Tous les messages inter et intra cluster sont sauvegardés en utilisant la journalisation pessimiste (configuration MLML : Message Logging Message Logging).

- On applique la solution non bloquante de Chandy-Lamport en intra-cluster et les messages inter-cluster sont sauvegardés avec la journalisation pessimiste basée sur l'émetteur (configuration MLCL : Message Logging Chandy-Lamport).

Pour un seuil de fréquence maximal calculé en fonction de la densité de la communication dans la grille, le protocole adaptatif exécute la composition « MLML », sinon c'est la combinaison « MLCL» si les applications ne sont pas fortement communicantes.

III. Organisation du manuscrit

Ce document est organisé en cinq chapitres. Un chapitre introductif expose la problématique de la thèse en abordant les problèmes liés à l'utilisation des environnements répartis, tels les grilles informatiques et les solutions existantes pour gérer la tolérance aux fautes.

Le chapitre 2 est un état de l'art sur la tolérance aux fautes dans les grilles informatiques. Les mécanismes de tolérance aux fautes utilisés dans les systèmes répartis sont décrits, ainsi que des protocoles hiérarchiques qui représentent le modèle de référence pour les nouveaux protocoles proposés.

Dans le chapitre 3, nous faisons une analyse comparative des protocoles de tolérance aux fautes présentés dans le chapitre 2. Ces protocoles seront implémentés avec le simulateur Omnet++. À l'issue des évaluations de performances, la journalisation pessimiste et le protocole de point de reprise coordonné non bloquant feront l'objet d'étude approfondie dans le chapitre 4.

Le chapitre 5 présente le nouveau protocole adaptatif hiérarchique pour la tolérance aux fautes dans les grilles informatiques. Une description détaillée de l'algorithme est faite, il est ensuite implémenté pour évaluer ses performances par rapport aux combinaisons statiques des deux protocoles qui le composent.

Le chapitre 6 conclut le mémoire en dressant un bilan des mécanismes de tolérance aux fautes et des travaux effectués au cours de la thèse, pour enfin présenter des perspectives.

Chapitre II : État de l'art

1. Introduction

Dans ce chapitre, nous allons d'abord présenter le vocabulaire relatif aux grilles informatiques. Ensuite nous allons expliquer les origines de la tolérance aux fautes dans les systèmes répartis, particulièrement les grilles. Puis nous aborderons les techniques utilisées pour assurer la tolérance aux fautes dans les grilles informatiques, et terminer avec des références de protocoles hiérarchiques basés sur ces techniques.

2. Tolérances aux fautes dans les grilles informatiques

2.1. Les grilles informatiques

Le terme « grid » (grille en français) a été utilisé pour la première fois par Ian Foster et Carl Kesselman [2]. Dans leur ouvrage intitulé « The Grid : Blueprint for a New Computing infrastructure » [2], ils comparent la disponibilité des ressources de la grille comme la disponibilité de l'électricité dans un réseau électrique. L'informatique en grille rend cela possible, en travaillant selon le même principe que le réseau électrique : on branche la fiche dans la prise, et l'électricité est censée passer. Dans le cas de l'informatique en grille, on se connecte au réseau et l'on est sûr de disposer de toutes les ressources suffisantes pour exécuter son travail. Ainsi des ordinateurs répartis dans le monde entier sont reliés entre eux au sein d'un réseau qui les rend accessibles. L'utilisateur n'a cependant pas besoin de savoir où ces ressources sont disponibles, ni où son application sera vraiment exécutée. Ainsi le partage de ressources, à l'échelle mondiale, devient l'objectif le plus essentiel à la réalisation d'une grille.

On pourra définir une grille informatique comme un ensemble de ressources matérielles et logicielles dans lequel des problèmes scientifiques peuvent être résolus. En effet, ce sont des systèmes répartis à haute performance dont certains des constituants sont des calculateurs parallèles. Elles permettent de mettre à la disposition des utilisateurs des ressources de manière transparente. Ces ressources peuvent être :

- Espace de stockage
- Capacité de calcul fournie par les processeurs
- Matériels informatiques spécifiques
- Applications ou logiciels couteux

Les ressources sont rassemblées souvent en fonction de l'espace géographique ou des groupes d'utilisateurs : on parle de ***clusters*** *(Figure 1)*.

Une grille informatique est constituée de plusieurs ***sites*** regroupés dans ce qu'on appelle une ***organisation virtuelle***, qui peut être une personne, un groupe de recherche, ou une entreprise. Cette subdivision est régie par un ensemble de règles de partage spécifiques qui sont les règles des entités qui le composent. Les organisations virtuelles, de même que les ressources informatiques qui les composent sont dynamiques. Elles peuvent quitter ou rejoindre la grille à tout moment, comme dans le cas d'une déconnexion ou d'une panne.

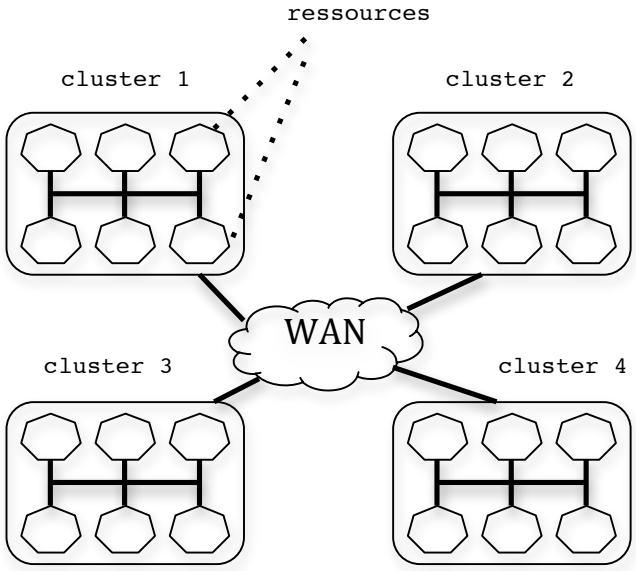

Figure 1 : Architecture physique d'une grille

Physiquement, la grille est constituée de **nœuds**. Ces derniers peuvent être des serveurs, des PCs, ou des matériaux spécifiques. Tous les constituants sont reliés en réseau à l'aide d'équipements tels que les routeurs, câbles, commutateurs, … Le côté logiciel est géré par ***l'intergiciel*** de la grille *(Figure 2)*. Il joue un rôle important dans l'allocation des ressources en fixant des règles et des priorités aux groupes d'utilisateurs. Parmi les composants de l'intergiciel, on peut citer :

- La base de données des ressources
- Le module d'authentification des utilisateurs
- Le répartiteur des tâches
- Le module de journalisation des tâches exécutées dans la grille

Le middleware le plus populaire est Globus Toolkit (GT). C'est un logiciel open source développé à Argonne par Ian Foster et son équipe. La première version est apparue en 1997, après une expérience réussie sur 80 sites à travers le monde [18]. Et depuis, l'intergiciel a évolué, la version 4 offre les services tels que la gestion des données, le service d'information, la sécurité et la gestion d'exécution. Un autre middleware beaucoup utilisé est l'intergiciel gLite du projet EGEE [19], déployé sur des centaines de nœuds dans le monde.

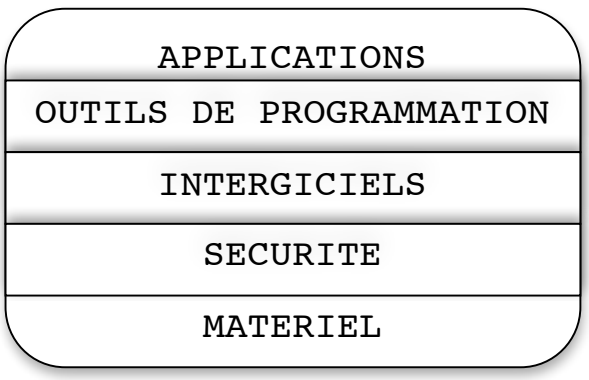

Figure 2 : Architecture en couche d'une grille

La couche matérielle comprend les constituants physiques cités plus haut notamment les stations de travail. Elle regroupe globalement tout ce qui est connectivité réseau et ressources.

La couche suivante assure la sécurité des ressources de la grille. Elle utilise les moyens communément utilisés dans les réseaux informatiques comme les pare-feu, et la détection d'intrusion. Mais le partage des ressources impose la mise en place de services de haute performance d'authentification, d'autorisation, et de comptabilisation. La sécurité constitue un aspect critique de la grille, parce qu'il doit exister un niveau de confiance très élevé entre les utilisateurs de la grille qui ignorent leur identités réciproques.

La quatrième couche regroupe tous les outils utilisés par les développeurs pour concevoir des applications adaptées à la grille. Elle est composée essentiellement de bibliothèques, de compilateurs et d'outils de développement.

La dernière couche la plus élevée regroupe les applications à l'origine de la création de la grille, par exemple des projets scientifiques des laboratoires de recherche. C'est la couche visible auprès des utilisateurs.

Une grille informatique contient essentiellement des éléments de calcul. Si l'on y ajoute également des éléments pour le stockage des données, on parle alors d'une ***grille de calcul/données***. Mais si leur architecture générale reste un dénominateur commun, les grilles présentent différentes topologies classées selon leur étendu. Il existe trois grandes topologies de grilles :

- **Intragrille** : Elle fait référence à un intranet avec une topologie simple composée de ressources appartenant uniquement à une seule organisation, comme par exemple une entreprise, une petite équipe de recherche. En général, les ressources de l'intragrille sont statiques et homogènes et elles sont reliées par un réseau haut-débit géré par les administrateurs de l'organisation.
- **Extragrille** : on peut définir comme une agrégation de plusieurs intragrilles. Donc, contrairement à une intragrille, son réseau ne se limite plus en local, mais sur une étendue pouvant atteindre un réseau hétérogène de type WAN. Dans ce cas, les ressources peuvent être dynamiques.
- **Intergrille** : Par analogie avec l'internet, l'intergrille est un modèle plus étendu et plus hétérogène que les deux premières catégories. En effet, elle regroupe en une seule grille, plusieurs grilles de plusieurs organisations sur une échelle mondiale.

La technologie de grille a éveillé l'intérêt du monde académique et scientifique. De nombreux projets de grille ont vu le jour comme Grid'5000 [3] et Eurogrid [4] en Europe, Teragrid [5] aux Etats Unis. Cependant, les grilles présentent plusieurs défis liés à l'utilisation. En effet, l'hétérogénéité des composants, la taille et le nombre accru d'utilisateurs représentent des facteurs déterminants pour l'évolution de l'environnement réparti. Le passage à l'échelle doit être pris en compte pour offrir une qualité de service quel que soit le nombre de nœuds. De plus, l'hétérogénéité des ressources, tant au niveau matériel qu'au niveau logiciel pourrait avoir des impacts sur l'utilisation optimale des ressources de la grille et sur les latences du réseau. Un des défis majeurs de l'informatique en grille est la tolérance aux fautes. Comme le nombre de nœuds augmente au fur et à mesure, la probabilité d'apparition de défaillances devient de plus en plus élevée à cause de la volatilité des ressources. Ces dernières utilisées par un grand nombre d'utilisateurs, qui exécutent un grand nombre d'applications est un défi de confidentialité et d'authentification dans la grille.

L'évolution des grilles a conduit vers un nouveau modèle appelé le Cloud Computing [21]. Comme la grille, le Cloud Computing met à la disposition de ses utilisateurs des services de stockage, de calcul, et des logiciels en ligne par abonnement, via Internet.

2.2. Notion de tolérance aux fautes

De manière générale, une grille informatique est un ensemble de nœuds de calcul reliés par un réseau de communication et communiquant entre eux par messages. Dans ce contexte, la tolérance aux fautes devient une obligation pour assurer le bon fonctionnement du système réparti. Pour comprendre la notion de tolérance aux fautes, nous allons présenter les aspects de la sûreté de fonctionnement d'un système pour comprendre les origines des défaillances dans les systèmes répartis.

La **sûreté de fonctionnement** (dependability) d'un système peut se définir selon Jean-Claude Laprie dans son guide de la sûreté de fonctionnement, « la propriété qui permet aux utilisateurs du système de placer une confiance justifiée dans le service qu'il leur délivre » [6]. Un système sûr de fonctionnement doit assurer la continuité du service quel que soit les défaillances survenues au cours de son exécution. Cela implique la réparation des fautes ou pannes, qui sont les conséquences de la non-fiabilité du système. Cette fiabilité doit garantir la sécurité. Et toutes ces conditions vont assurer la disponibilité des ressources du système réparti, propriété de base des environnements distribués. Les fautes étant les causes des défaillances, la sûreté de fonctionnement cherche à les combattre, si possible en évitant qu'elles se produisent (prévention), ou en les éliminant (élimination).

Les **fautes**, les **erreurs** et les **défaillances** représentent les entraves à la sûreté de fonctionnement. Une erreur est un état susceptible d'entraîner une défaillance ; la faute est la cause supposée d'une erreur ; la défaillance est la manifestation d'une erreur. La Figure 3 montre la relation de causalité qui existe entre les fautes, les erreurs et les défaillances. Par propagation, plusieurs erreurs peuvent être générées avant qu'une défaillance ne se produise. Lorsque le système est défaillant, il n'est plus apte à assurer le service défini par l'utilisateur [6].

En empêchant ainsi une erreur de se propager, on améliore la sûreté de fonctionnement d'un système tout en tolérant les fautes [6].

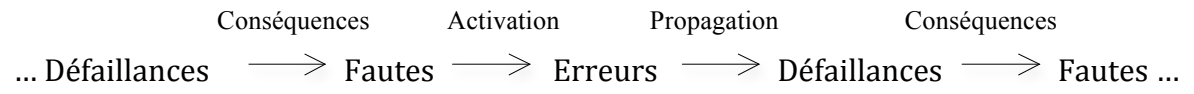

Figure 3 : Entraves à la sûreté de fonctionnement

Le seul moyen d'éviter que cette chaine ne se produise est d'empêcher son activation déclenchée par les fautes survenues dans le système. On distingue deux catégories de fautes selon leur origine:

- **Fautes accidentelles** : elles sont en général d'origine matérielle, comme par exemple un court-circuit ou une défectuosité d'équipements.
- **Fautes intentionnelles** : elles sont liées à la sécurité ou bien une mauvaise action sur les logiciels. Elles peuvent aussi concerner les déconnexions de machines.

Dans les environnements répartis, il existe plusieurs types de fautes classés selon le degré de défaillances:

- **Fautes franches ou par arrêt total** : c'est le modèle de panne le plus simple ; le système arrête subitement de fonctionner et se retrouve dans un état « incorrect ».
- **Fautes par omission** : ce type de faute survient lorsque des messages sont perdus.
- **Fautes par valeur** : elles concernent essentiellement les résultats produits par les composants qui peuvent avoir des valeurs incorrectes.
- **Fautes byzantines** : elles sont très complexes parce que le système continue en présence de messages qui ne suivent pas sa spécification.
- **Fautes par temporisation** : le comportement erroné du système est dû au temps.

Pour évaluer la sûreté de fonctionnement d'un système, J.-C. Laprie a défini des attributs (*Figure 4*). Ces six propriétés de base doivent être vérifiées pour garantir la sûreté :

- **Fiabilité** : elle évalue la continuité du service délivré par le système et défini par l'utilisateur.
- **Disponibilité** : pour qu'un système soit sûr de fonctionnement, il faut que les ressources soient disponibles à tout moment.
- **Maintenabilité** : c'est la capacité du système à subir des réparations et des modifications.
- **Sécurité** : elle doit permettre la résistance aux pannes catastrophiques sur l'utilisateur et l'environnement.
- **Confidentialité** : la notion de partage dans les systèmes impose la confidentialité sur toutes les informations en mettant en place des mécanismes d'authentification
- **Intégrité** : c'est un attribut qui permet d'évaluer l'aptitude du système à éviter les altérations de son état.

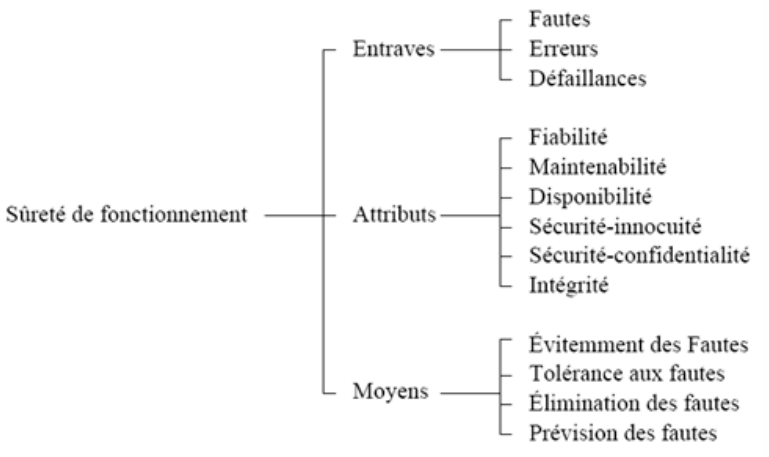

Figure 4 : Arbre de la sûreté de fonctionnement [6]

En fonction de la demande des applications, certains attributs peuvent être mis en valeur plus que d'autres. Par exemple, la disponibilité est toujours nécessaire alors que la fiabilité, la confidentialité ou la sécurité peuvent ne pas être primordiales. Donc les attributs de la sûreté de fonctionnement doivent être interprétés au sens relatif et probabiliste plutôt que dans un sens absolu dans la mesure où les systèmes ne sont jamais totalement disponibles, fiables ou sécurisés.

Il existe d'autres attributs secondaires, liés à la sécurité des informations comme l'authenticité et la non-répudiation. Ces deux propriétés permettent respectivement de garantir l'intégrité du contenu des messages et de connaître l'identité de l'émetteur et du récepteur du message.

Pour que le système vérifie les attributs nécessaires à la sûreté de fonctionnement, des moyens ont été définis dans [6] comme le montre l'arbre de la sûreté de fonctionnement:

- **L'élimination des fautes**
- **La prévision des fautes**
- **L'évitement des fautes**
- **La tolérance aux fautes** ou **tolérance aux pannes**

Dans le cas de notre étude, nous nous intéressons à la tolérance aux fautes. L'objectif de la tolérance aux fautes est la préservation de la continuité du service malgré l'apparition de fautes. Du fait de leur nature hétérogène, l'occurrence de fautes dans les grilles est inévitable. Cette nature fait référence à la dynamicité des ressources, de l'hétérogénéité des équipements, des systèmes d'exploitation et applications, de l'absence de mémoire et d'horloge commune, et l'asynchronisme des communications entre les processus. Il est donc primordial de mettre en place des mécanismes permettant au système de continuer à fonctionner, même en présence de fautes.

Deux méthodes permettent de tolérer les fautes : la détection d'erreur et la récupération du système. La détection d'erreur est possible grâce aux messages d'erreur émis dans le système. La récupération ramènera ce dernier dans un état antérieur correct, sans erreur. Nous détaillerons cette dernière méthode dans la suite de ce document.

2.3. Techniques de tolérance aux fautes dans les grilles

On distingue deux principales techniques de tolérance aux fautes dans les grilles informatiques :

- **La réplication**
- **Le recouvrement arrière**

Nous allons présenter ces deux mécanismes en insistant sur la technique de réplication. Les mécanismes de recouvrement arrière seront étudiés de manière plus précise dans le chapitre suivant.

2.3.1. La réplication

La tolérance aux fautes basée sur la réplication consiste à créer des copies multiples des

processus sur des machines différentes. Lors de la panne d'un processus, il est remplacé par une de ses copies. La réplication dans un système réparti présente plusieurs avantages. Elle apporte une meilleure sûreté de fonctionnement en augmentant la disponibilité et la fiabilité des données. En effet, elle permet d'éviter la perte de cycle d'exécution en cas de panne en masquant les fautes. La réplication améliore aussi les performances car la répartition des données permet d'éviter les accès multiples à un seul site pouvant réduire les latences du réseau.

Ils existent trois techniques de réplication proposées dans la littérature : la réplication active, passive et semi-active. Les deux premières stratégies de réplication sont les bases et elles constituent les références des autres mécanismes de réplication.

2.3.1.1. La réplication active

C'est une technique dans laquelle toutes les répliques traitent les mêmes messages d'entrée, mais en gardant leur état étroitement synchronisé pour assurer qu'elles reçoivent les messages dans le même ordre. Chaque réplique joue un rôle identique à celui des autres. Les mêmes messages de sortie sont donc générés et une technique de vote permet de choisir le message à renvoyer.

En cas de panne, les erreurs sont masquées par les copies non défaillantes qui assurent le relais. La défaillance d'une copie est masquée par les autres répliques, donc la défaillance de l'une d'entre elles ne perturbe pas le fonctionnement du système. Il est à noter que grâce au vote sur les messages en sortie des comportements byzantins peuvent être tolérer.

2.3.1.2. La réplication passive

On distingue deux types de copies de processus dans cette technique : une copie primaire et des copies secondaires. Seule la copie primaire traite les messages d'entrée et fournit les messages de sortie. Pour maintenir la cohérence, la copie primaire transmet par intervalle de temps son nouvel état aux copies secondaires.

En l'absence de fautes, les copies secondaires ne traitent pas les messages d'entrée, mais leur état est régulièrement mis à jour au moyen des points de reprise transmis par la copie primaire. Lors d'une panne de la copie primaire, une des copies secondaires la remplace et le traitement des messages reçus depuis la dernière sauvegarde est perdu.

2.3.1.3. La réplication semi-active

Cette technique permet un recouvrement de fautes plus rapide que la réplication passive. Les messages d'entrée sont traités par toutes les copies, mais une seule, le leader, fournit les messages de sortie. Pour mettre à jour leur état interne en l'absence de fautes, les autres répliques traitent directement les messages d'entrée, ou utilisent les « notifications » du leader. En cas de panne franche du leader une des copies peut ainsi prendre le relais sans perte.

La réplication passive est présentée par le projet Delta 4 [7] comme la technique la plus performante en cas d'absence de faute. La réplication active, exigeant la création de plusieurs processus, pourrait dégrader les communications réseau. La troisième catégorie intermédiaire, trouve un compromis pour gérer les événements non-déterministes. En effet, dans la réplication semi-active, le leader joue le rôle de synchronisation des messages internes. Les

autres copies appelées suiveurs ne traitent les messages qu'après avoir reçu une notification de la part du leader.

D'autres types de réplication ont définis dans la littérature, comme la réplication **semi-passive** et la réplication **coordinateur-cohorte** [17].

2.3.2. Le recouvrement d'erreur

Le recouvrement d'erreur consiste à remplacer un état erroné par un état stable garantissant un système fonctionnel. Il existe deux techniques de recouvrement d'erreur:

- **Le recouvrement par reprise** : C'est une technique générale qui ramène le système à un « état correct sans erreur » qu'il occupait (*Retour arrière*). Elle nécessite la sauvegarde périodique de l'état du système sur support stable.

- **Le recouvrement par poursuite** : le système est ramené à un nouvel état « reconstitué », sans effectuer de retour arrière. La reconstitution n'est souvent que partielle. C'est une technique spécifique parce que la reconstitution d'état correct dépend de l'application et exige une analyse préalable des différents types d'erreurs possibles.

La *Figure 5* illustre le recouvrement arrière. Un état stable, sauvegardé au cours de l'exécution des applications est appelé **point de reprise** (*checkpoint*). Périodiquement, un point de reprise est effectué et stocké sur un support stable, qui peut être un disque ou un serveur. En cas de défaillance, le système redémarre au point de reprise le plus récent. Il effectue ainsi un **retour arrière** depuis un point de reprise antérieur à la panne : ce qui constitue un gain en terme de temps d'exécution, puisse que les applications ne reprennent pas leur calcul à l'instant zéro.

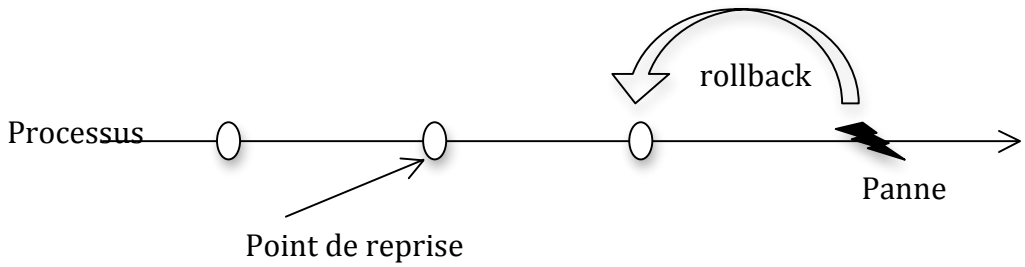

Figure 5 : Recouvrement arrière après panne (rollback-recovery)

Les deux mécanismes de recouvrement présentent des avantages et des inconvénients. La reprise, même si elle est la plus utilisée, peut être couteuse en temps et en espace de stockage. En effet, les phases de sauvegarde des points de reprise augmentent le temps de réponse des applications. En plus, les points de reprise occupent de la place selon le protocole exécuté dans le système. Son avantage est sa propriété générique, parce qu'elle s'adapte en générale à la plupart des architectures informatiques. La poursuite, quant à elle, présente un champ d'application limite. Étant spécifique, elle peut être efficace si les types d'erreurs possibles

ont été listés antérieurement.

Dans la suite de ce document, nous ne traiterons que les mécanismes de tolérances aux fautes utilisant la technique du recouvrement arrière par reprise.

3. Problèmes liés au recouvrement arrière

Deux problèmes interviennent pour effectuer un recouvrement arrière: la cohérence d'état global et le déterminisme d'exécution.
L'état global d'un système réparti est l'ensemble des états locaux de tous les processus participants au fonctionnement de ce système. Il doit être cohérent pour garantir une exécution correspondant à un état possible du système.
Ensuite, le déterminisme d'exécution est une propriété des systèmes répartis qui indique la capacité de reproduire une exécution.
Nous allons voir dans les chapitres suivants les caractéristiques de ces problèmes liés à la reprise.

3.1. État global cohérent

Dans [8], l'état global d'un système est défini comme la collection des états locaux des processus et des canaux de communication qui les relient. L'état local d'un processus *p* est défini par son état initial et l'ensemble des événements ayant eu lieu sur *p*. Un état global cohérent est un état global intervenu lors d'une exécution correcte.
Cet état doit garantir la causalité des événements. L'ensemble des processus de la grille communiquant par passage de messages, l'état global cohérent doit assurer que l'événement d'émission d'un message *m* précède sa réception.
Un ensemble de points de reprise représentant un état global cohérent est appelé **une ligne de recouvrement**.
Un message orphelin est un message qui a été reçu sans être envoyé. Autrement dit, lors du calcul de la ligne de recouvrement, si la réception d'un message appartient à un des états locaux et son émission n'appartient pas à un de ces états, l'état global devient incohérent.

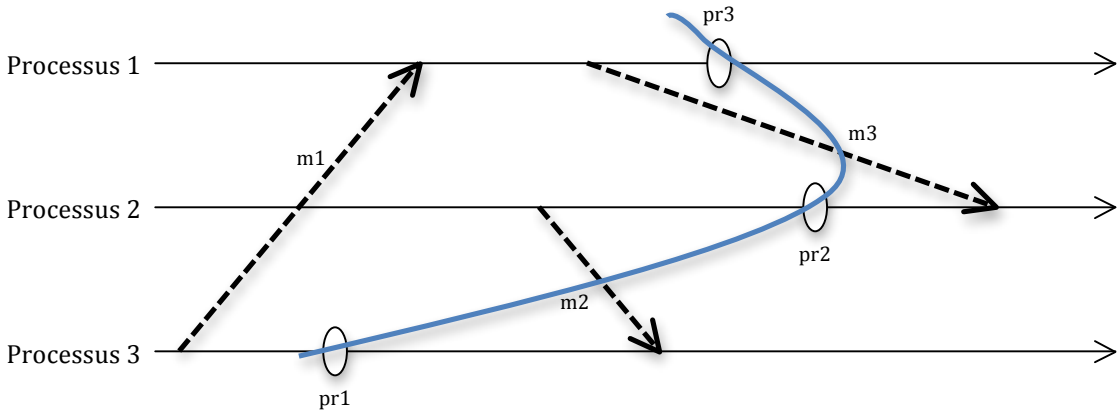

Figure 6.1 : État global cohérent

Sur la Figure 6.1, l'état global représenté par l'ensemble des points de reprise pr1, pr2 et pr3 est cohérent car aucun état local ne contient de message orphelin. Les messages m2 et m3 sont des messages en transit, ils sont envoyés, mais pas encore reçus.
La Figure 6.2 illustre un état incohérent. Si le processus 1 tombe en panne après envoi du message. L'état global formé par les états locaux *pr1* et *Z* est incohérent car l'émission du message *m* n'appartient pas à l'état global. Le processus 2 est orphelin et doit faire un retour vers le point de reprise pr2 pour supprimer le message orphelin et assurer la cohérence du système. La suppression des messages orphelins peut donc entraîner **un effet domino** avec des reprises en cascade.

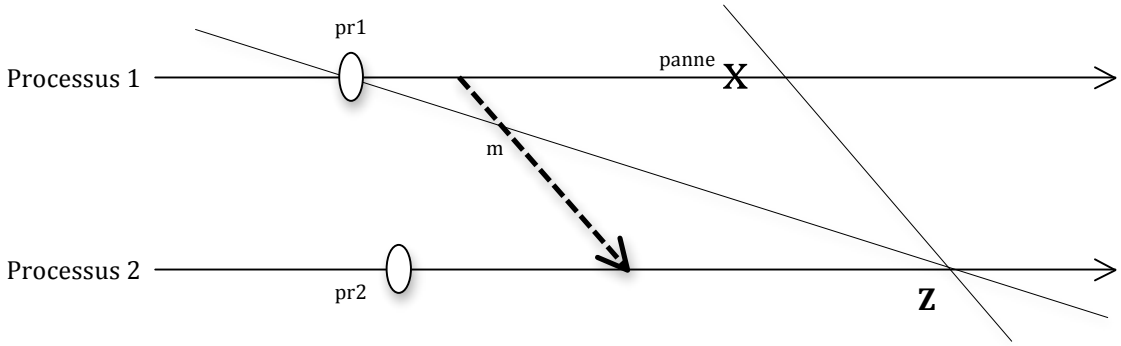

Figure 6.2 : État global incohérent

3.2. Déterminisme d'exécution

Un événement est déterministe si et seulement si plusieurs exécutions de cet événement à partir du même état initial donnent le même état final.
Le déterminisme d'exécution est basé sur l'hypothèse PWD (Piecewise Deterministic Assumption) définie par Strom et Yemini [9] et qui dit :

- un processus peut être modélisé comme un intervalle d'état
- chaque intervalle d'état débute par un événement non-déterministe
- l'exécution durant chaque intervalle d'états est déterministe

En effet, durant l'exécution sans panne, les processus sauvegardent dans leur journal les événements non-déterministes. Ils correspondent aux réceptions de messages ou aux événements internes au processus.

En cas de panne, le processus défaillant fait un retour arrière sur le dernier point de reprise et utilise son journal pour rejouer les événements non-déterministes. L'hypothèse PWD garantit que la réexécution aboutira au même état final.

Un problème survient lorsqu'un événement non-déterministe apparaît dans un intervalle état, par exemple si l'on ne dispose pas de ses caractéristiques. Dans ce cas les protocoles de journalisation des messages ne seront pas utilisés.

4. Les protocoles de points de reprise

Par définition, un point de reprise (*checkpoint*) est la sauvegarde de l'état d'un processus à un instant donné, sur un support de stockage stable. Un support stable est un espace de stockage accessible par toutes les applications utilisées dans la grille disposant des droits suffisants, et protégé contre toute sorte de pannes. Il est souvent représenté par un serveur de stockage.
Les protocoles basés sur les points de reprise permettent de faire des sauvegardes périodiques de l'ensemble des états des processus. Pendant le recouvrement, les points de reprises les plus récents sont restaurés pour reproduire un état global cohérent du système avant la panne (*Figure 4*).
La tolérance aux fautes par recouvrement arrière se base sur les protocoles de points de reprise pour éviter la perte de temps de calcul dans les environnements de calcul haute performance comme les grilles. Il existe différentes catégories de protocoles de points de reprise classés selon la méthode utilisée pour effectuer la sauvegarde :

- Les protocoles de points de reprise coordonnés
- Les protocoles de points de reprise non-coordonnés
- Les protocoles de points de reprise induits par les communications.

4.1. Points de reprise coordonnés

Ce protocole exige aux processus de coordonner l'établissement de leurs points de reprise pour former un état global cohérent. L'algorithme bloquant se déroule ainsi [10] (*Figure 7*):

- Les communications sont bloquées pendant l'établissement des points de reprise.
- Un coordinateur fait un point de reprise et diffuse un message '*checkpoint-request*' à tous les processus pour initier une phase de sauvegarde.
- Quand un processus reçoit ce message, il arrête son exécution et vide ses canaux de communication, fait une tentative de point de reprise et envoie un accusé de réception au coordinateur.
- Après avoir reçu l'accusé de réception de tous les processus, le coordinateur diffuse un message '*commit*' à tous les processus pour leur ordonner de faire un point de reprise.
- Chaque processus supprime ainsi l'ancien point de reprise, et sauvegarde son état. Le processus est maintenant libre de terminer son exécution et d'échanger des messages avec les autres processus du système.

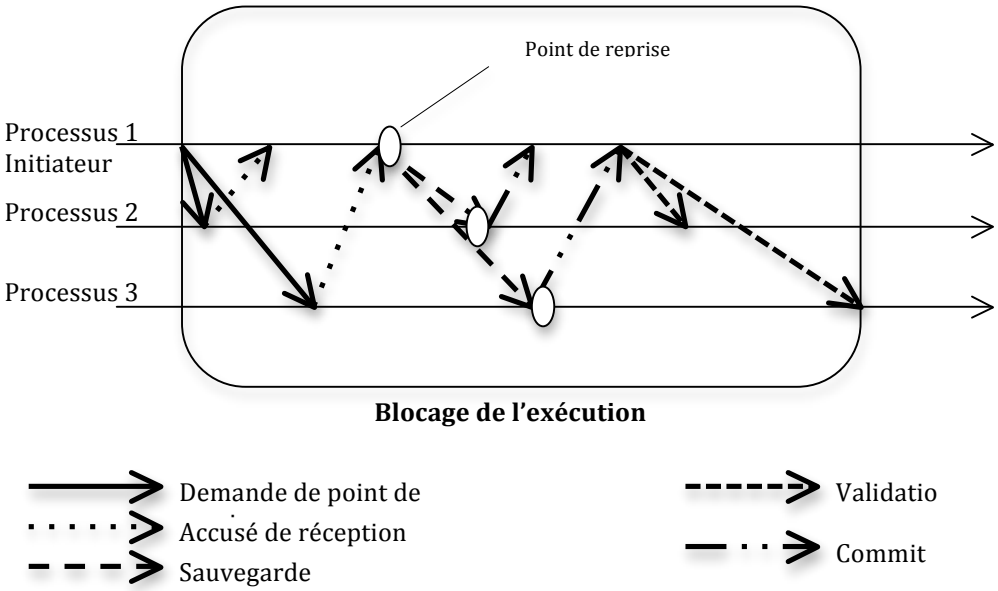

Figure 7 : Point de reprise coordonné

Cette approche simplifie le recouvrement et évite l'effet domino, puisque chaque processus redémarre toujours au point de reprise le plus récent. Aussi, le protocole exige à chaque processus de maintenir un seul point de reprise permanent en mémoire stable, ce qui réduit le surcoût dû au stockage ainsi que la nécessité d'avoir un algorithme de récupération des points de reprise (*garbage collection*) [10]. En pratique, le protocole coordonné est facile à implémenter. Par exemple, CoCheck [23], le mécanisme assurant la cohérence des états dans Condor [24] est basé sur le protocole de Chandy & Lamport [12].

Son principal inconvénient est cependant la large latence introduite par la coordination des processus participants. Pour améliorer les performances de la sauvegarde coordonnée, plusieurs techniques ont été proposées :

- ***Point de reprise coordonné non-bloquant*** : le plus connu est celui de Chandy et Lamport [12]. Il fonctionne sous l'hypothèse des canaux FIFO (First In First Out ; premier arrivé, premier servi), en utilisant des marqueurs. En effet, lors de la sauvegarde des points de reprise, chaque processus enregistre son état local et envoie sur tous ses canaux de sortie un marqueur pour informer les autres processus voisins qu'il a effectué un point de reprise. À la réception de ce marqueur, un processus recevant ce message pour la première fois sauvegarde à son tour son état et diffuse le marqueur, et ainsi de suite.

- ***Point de reprise coordonné minimale*** : Koo et Toueg [13] propose un protocole coordonné minimal, qui ne fait intervenir que les processus ayant participé à la dernière sauvegarde. Seuls les processus susceptibles de créer une incohérence en cas de reprise sauvegardent leur état. L'algorithme se déroule en deux phases. Durant la première phase, l'initiateur identifie tous les processus dont il a reçu un message depuis le dernier point de reprise, et leur envoie une demande, le message étant un orphelin potentiel. Sur réception de la demande, chaque processus identifie à son tour tous les processus pouvant créer des messages orphelins et leur envoie aussi une demande, jusqu'à ce que tous les processus soient identifiés. Après l'identification des processus, la deuxième phase correspondant à la sauvegarde des états locaux proprement dite.

- ***Point de reprise coordonné avec horloge de synchronisation*** : Dans [14] [15], les auteurs utilisent des horloges de synchronisation pour effectuer des points de reprise locaux au niveau de tous les processus participants, et approximativement au même moment sans un initiateur. Le protocole assure qu'aucun message n'est échangé durant la sauvegarde.

4.2. Points de reprise indépendants

Ce protocole évite la phase de synchronisation en laissant à chaque processus l'autonomie de sauvegarder son état local (*Figure 8*). L'avantage de cette autonomie est que chaque processus pourra sauvegarder son état au moment opportun, par exemple si la quantité d'information est minimale. Ce qui réduit le surcoût du stockage en termes de quantité d'information à sauvegarder.

Dans [10], les étapes du recouvrement sont [10]:

- Le processus de récupération entame un retour arrière par la diffusion d'un message de '*dependancy-request'* pour rassembler les 'informations de dépendance' maintenues par chaque processus
- Quand un processus reçoit ce message, il arrête son exécution et envoie les informations de dépendances enregistrées sur la mémoire stable
- L'initiateur calcule la ligne de recouvrement basée sur les 'informations de dépendance' globales et diffuse un message de '*rollback request*' contenant la ligne de recouvrement.
- Sur réception de ce message, chaque processus appartenant à la ligne de recouvrement reprend son exécution, sinon il revient au point de reprise précédent, celui indiqué par la ligne de recouvrement (*Figure 8*).

La dernière étape de l'algorithme peut causer **l'effet domino**, qui peut engendrer une perte du travail réalisé avant la panne et la sauvegarde inutile de points de reprise qui ne feront jamais partie d'un état global cohérent. Le protocole non coordonné fait revenir en arrière le processus concerné par la panne, ainsi que ceux qui en dépendent causalement. Il oblige aussi chaque processus à maintenir plusieurs points de reprise.

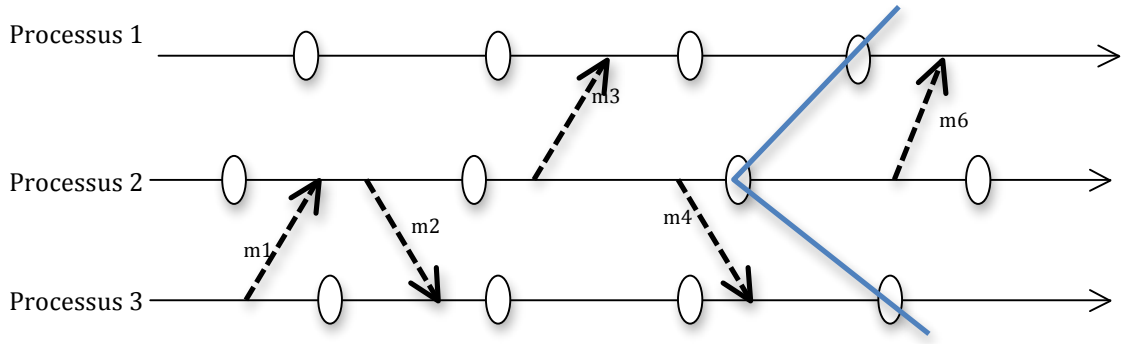

Figure 8 : Point de reprise indépendant

4.3. Points de reprise induits par les communications

Ce protocole (CIC: Communication Induced Checkpointing) définit deux types de points de reprise [41][42]:

- *Des points de reprise locaux* pris par les processus de manière indépendante, pour éviter la synchronisation de la sauvegarde coordonnée
- *Des points de reprise forcés* en fonction des messages reçus et envoyés et des informations de dépendance estampillées sur ces messages, pour ainsi éviter l'effet domino de la sauvegarde non coordonnée, garantir l'avancement de la ligne de recouvrement.

À l'opposé des protocoles de point de reprise coordonné, le surcoût dû à l'accès au support stable disparaît parce que le protocole CIC ne nécessite aucun échange de message pour forcer un point de reprise. En effet, il utilise la technique utilisée par la journalisation causale en insérant la signalisation sur les messages échangés.

Le protocole de Xu et Netzer [42] repose sur les notions de Z-chemin et de Z-cycle. Ces notions permettent de déterminer si une sauvegarde sera utile ou non en cas de panne et de reprise. Un Z-chemin est une séquence de messages qui connecte deux points de reprise et un Z-cycle est un Z-chemin qui boucle c'est à dire il commence et se termine par le même point de reprise. Pour forcer un point de reprise, le protocole CIC vérifie que le point de reprise n'appartient pas à un Z-cycle.

Les techniques de sauvegarde de point de reprise induit par les communications peuvent être classées en deux catégories :

- **Model-based coordination** : elle repose sur la prévention des modes de communication et de points de reprise qui pourraient entrainer des Z-cycles. Ces modèles sont conçus pour détecter la formation de ces Z-cycles qui pourraient se former au cours de l'exécution des applications [42].

- **Index-based coordination** : cette catégorie utilise les horloges logiques. En effet, ces horloges sont ajoutées aux messages afin de savoir à quel moment il faut sauvegarder un point de reprise [42].

Même si les protocoles CIC trouvent un compromis entre la sauvegarde coordonnée et la sauvegarde indépendante, elles présentent des inconvénients liés à la fréquence de sauvegarde des points de reprise forcés. Comme les points de reprise augmentent le temps d'exécution des applications dans les grilles, il serait mieux de réduire leurs nombres pour éviter la dégradation des performances du système. Malheureusement, une évaluation de ce type de protocoles a montré que le nombre de points reprise forcé était souvent élevé pour certains modèles d'application [41].

5. Les protocoles de journalisation

Depuis plusieurs décennies, la journalisation des messages a été une technique très utilisée pour assurer la tolérance aux pannes dans les systèmes répartis. Les protocoles de journalisation nécessitent la sauvegarde périodique des états locaux des processus, et l'enregistrement sur support stable de tous les messages reçus après l'établissement des points de reprise locaux.

En cas de panne d'un processus, il est relancé à partir du dernier point de reprise, et tous les messages reçus après ce dernier lui sont renvoyés dans l'ordre où ils ont été initialement reçus. Tous les protocoles de journalisation des messages exigent que l'état d'un processus récupérable soit toujours compatible avec l'état des autres processus. Ils doivent ainsi garantir l'absence du processus orphelin, qui rendent le calcul de l'état global incohérent.

Il existe trois catégories de journalisation de messages : optimiste, pessimiste, causale. Elles différent de par leur technique de sauvegarde, de recouvrement, de récupération de point de reprise (garbage collection) et de leur surcoût induit durant l'exécution sans faute [16].

5.1. Journalisation optimiste

Ce protocole utilise l'hypothèse selon laquelle la journalisation d'un message sur support

fiable sera complète avant qu'une panne ne se produise. En effet, au cours de l'exécution des processus, les déterminants des messages, à savoir leur contenu et identifiants, sont conservés en mémoire volatile avant d'être périodiquement vidés sur support stable. Le stockage sur mémoire stable se fait de manière asynchrone : le protocole n'exige pas le blocage de l'application pendant la sauvegarde sur mémoire stable. La latence induite est alors très faible.

Cependant une panne peut survenir avant que les messages ne soient enregistrés sur mémoire stable. Dans ce cas, les informations sauvegardées en mémoire volatile du processus en panne sont alors perdues. Le recouvrement devient compliqué puisque tous les processus ayant reçu un message provenant du processus défaillant deviennent orphelins (*Figure 9*). Ce qui peut produire l'effet domino. En plus, le protocole est obligé de maintenir plusieurs points de reprise en mémoire. La récupération des points de reprise devient plus complexe. Le calcul pour obtenir un état global cohérent peut avoir un coût important parce que les processus peuvent être amenés à redémarrer au point de reprise qui n'est pas le plus récent.

Pour éviter l'effet domino, le protocole enregistre les informations de dépendances entre les messages durant l'exécution sans faute. Une solution consiste à utiliser des vecteurs de dépendances attaché aux messages et dont la taille est égale au nombre total de processus. Mais cela est très coûteuse pour les grands systèmes comportant un nombre important de processus.

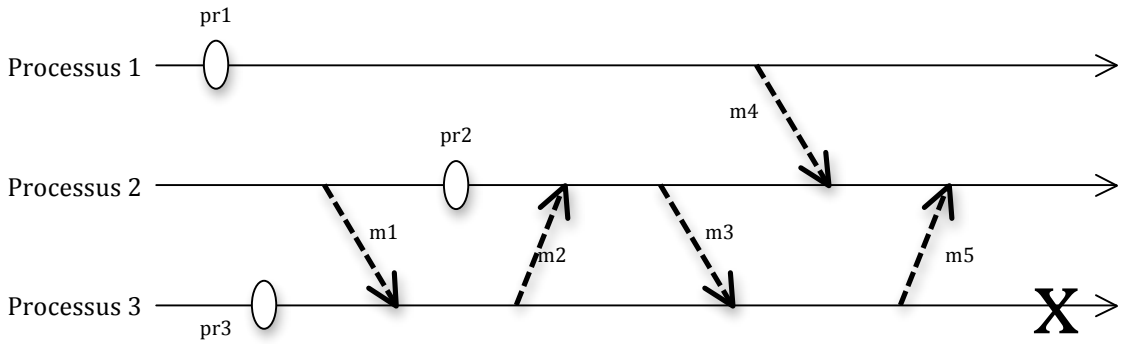

Figure 9 : Enregistrement de messages optimiste : si le processus 3 tombe en panne avant la sauvegarde des messages m3 et m5, il va reprendre à partir du point de reprise pr3. Les messages m2 et m5 devenant orphelin, le processus 2 va reprendre à partir du point de reprise pr2

Dans [31], les auteurs présentent un protocole optimiste qui élimine la latence introduite par les vecteurs de dépendances estampillés sur les messages. En effet, le nouveau protocole appelé O2P, utilise des listes de dépendances qui sont maintenues par les processus. Les listes contiennent les déterminants des messages. Un processus qui délivre un message, ajoute le déterminant correspondant à sa liste de dépendances et l'enlève une fois que celui-ci est stocké sur support stable. Pour cela, quand un processus émet un message, il attache sa liste de dépendances sur le message émis. Le destinataire met à jour sa liste en ajoutant la liste de l'émetteur à sa propre liste de dépendance. En revanche, si un message devient orphelin alors tous les processus ayant le déterminant de ce message dans leur liste de dépendances deviennent des processus orphelins. Ainsi la quantité de données attachées sur chaque

message ne dépend pas du nombre de processus dans l'application.

D'autres travaux [9] ont utilisés le protocole de journalisation optimiste, mais le problème de la taille du vecteur de dépendance persiste. Une autre solution consiste à utiliser des vecteurs d'horloges [32]. La taille du vecteur est toujours égale au nombre de processus dans le système, mais la solution des vecteurs permet de tolérer des fautes multiples [33].

5.2. Journalisation pessimiste

Ce protocole a été conçu dans l'hypothèse qu'une panne peut se produire après n'importe quel événement non déterministe. Il enregistre en mémoire stable le déterminant de chaque message avant que ce dernier ne soit autorisé à interagir avec le système (*Figure 10*). Ces protocoles font souvent référence à la journalisation synchronisée parce que quand un processus enregistre un événement non déterministe sur mémoire stable, il attend de recevoir un acquittement pour continuer son exécution.

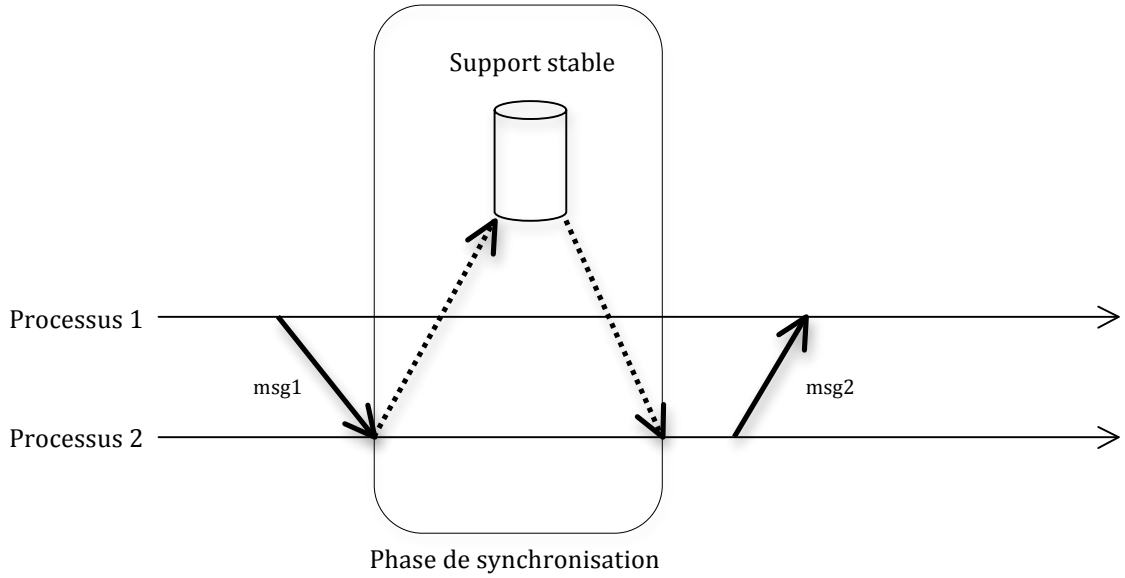

Figure 10 : Processus de sauvegarde avec la journalisation pessimiste

Dans un système à journalisation pessimiste, l'état de chaque processus est toujours récupérable. Cette propriété a quatre avantages :

- Les processus peuvent envoyer des messages à l'extérieur sans utiliser un protocole spécial.
- Lors d'une défaillance, les processus redémarrent au point de reprise le plus récent, et rejoue les messages stockés dans le journal.
- Le recouvrement est simple parce que les effets d'une panne se limitent seulement aux processus en panne.
- La procédure de récupération des points de reprise et des messages est simple, puisque les messages reçus avant le dernier point de reprise ne seront jamais utilisés car les processus redémarrent au processus le plus récent.

Le principal inconvénient reste la synchronisation qui entraine la dégradation des

performances du système. Plusieurs approches ont été définies afin de minimiser les synchronisations, comme la technique utilisée par Johnson et al. [10][11].

L'algorithme **Sender Based Message Logging** (**SBML**) de Johnson et al. [11]: Le contenu des messages avec leur date d'émission sont sauvegardés dans la mémoire volatile de l'émetteur. Le processus récepteur envoie un accusé de réception à l'émetteur qui mémorise la date de réception des messages. Ce nouveau protocole évite les accès fréquents à la mémoire stable et réduit ainsi le coût de la sauvegarde synchronisée.
Des extensions du mécanisme de journalisation pessimiste ont été décrites dans la littérature. Au lieu de créer de nouvelles solutions logicielles, elles s'appuient sur un matériel spécifique : des mémoires semi-conducteurs non volatiles [10][34] ; ou un bus spécial qui garantit la journalisation atomique de tous les messages échangés dans le système [35]. Une telle prise en charge matérielle garantit que le journal d'une machine est automatiquement stocké sans bloquer l'exécution des applications [10].

5.3. Journalisation causale

Ce protocole combine les avantages des deux précédents mécanismes de journalisation. Comme la journalisation optimiste, elle réduit le surcoût à l'exécution dû à l'accès synchronisé au support stable. Comme la journalisation pessimiste, il ne crée jamais de processus orphelin, l'état de chaque processus défaillant est recouvrable à partir de son dernier point de reprise, et il n'y a pas de risque d'effet domino. La journalisation causale permet aux processus d'effectuer des interactions avec l'extérieur de manière indépendante.

La procédure de journalisation s'effectue de la manière suivante:

- les messages sont enregistrés sur support stable, mais le processus n'attend pas un accusé réception pour continuer son exécution.
- Chaque processus détient une table de dépendances causales de messages. Tant que l'accusé de réception des messages n'est pas arrivé, les déterminants des messages sont ajoutés aux informations de dépendance de ces messages, en sachant que la table de dépendance causale de chaque message est attachée aux messages échangés.

En cas de panne, le processus défaillant effectue un retour arrière au point de reprise le plus récent et rejoue les messages enregistrés sur support stable ou utilisent les tables de dépendances des processus.

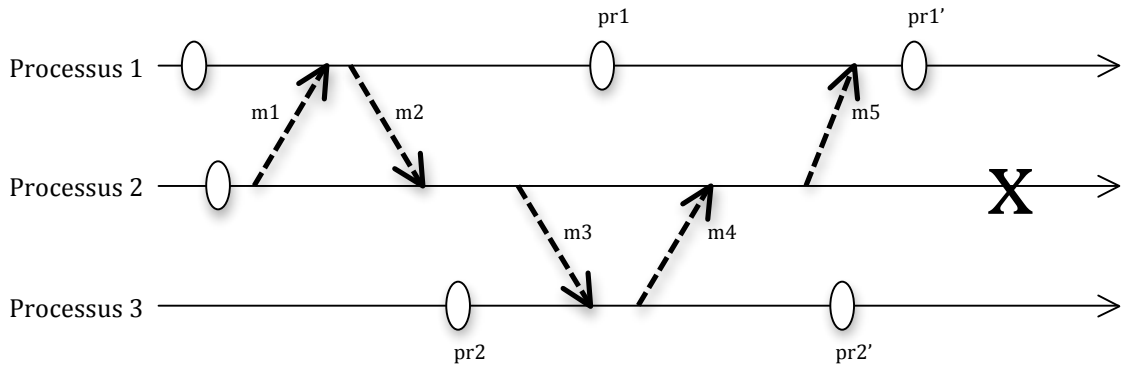

Figure 11 : Recouvrement journalisation causale

La Figure 11 illustre un recouvrement en utilisant la journalisation causale. Le processus 2 subit une défaillance. Il va redémarrer à partir de son dernier point de reprise et rejouer tous les messages reçus, comme par exemple le message m4 dont le déterminant est contenu dans le message m5 grâce au protocole de journalisation causale qui attache les déterminants des messages sur les autres messages circulant dans le système.

L'inconvénient de la journalisation causale est le surcoût sur la latence du réseau introduit par les informations de dépendance attachées aux messages. Pour éviter la dégradation des performances causée par les données supplémentaires de causalité, d'autres protocoles sont nés de l'étude des relations de causalité entre les messages envoyés dans un système par passage de messages. Manetho [34], utilise un graphe d'antécédence contenant les informations de causalité entre les messages échangés. Le graphe n'est pas en totalité attaché aux messages, mais uniquement la portion contenant la différence entre les graphes de deux processus ayant échangés des messages.

Toujours pour réduire la quantité de données attachée aux messages, une solution [35] consiste à sauvegarder les messages en local au niveau du processus. Au lieu de faire circuler les messages avec les informations de causalité, ces dernières sont stockées sur un support distant par un élément appelé un « Event logger » ou « enregistreur d'événements » [35] [36]. En cas de panne, le processus fautif obtient les informations de causalité auprès de l'enregistreur d'événements.

6. Protocoles hiérarchiques basés sur les points de reprise et journalisation

Les grilles informatiques sont devenues des plateformes de référence pour exécuter les grandes applications scientifiques aussi bien dans le secteur commercial que dans le monde académique. Pour maximiser la disponibilité des ressources de l'environnement, plusieurs techniques de tolérance aux fautes ont été proposées. Cependant, ces dernières sont plus adaptées aux grappes de calcul ou cluster en reposant sur des mécanismes de tolérances aux fautes basés sur les protocoles classiques de point de reprise et de journalisation.

Comme les grilles informatiques peuvent être représentées comme une architecture hiérarchique, plusieurs auteurs ont proposé des protocoles hiérarchiques de tolérance aux fautes basés les mécanismes de recouvrement arrière décrits dans les paragraphes précédents. Ces nouveaux protocoles combinent soit deux techniques de journalisation [26] ou de points de reprise [25], soit une combinaison entre un protocole de journalisation et de points de reprise [27].

Point de reprise coordonné hiérarchique. Paul et al. [25] proposent un protocole hiérarchique basé sur les points de reprise coordonnés. Ce protocole est conçu pour des réseaux hiérarchiques comme internet. Les expériences ont été faites sur un réseau composé de quatre clusters de huit (8) nœuds. On suppose que les nœuds du cluster sont 'sûrs', et connectés via un réseau haut débit égal à 10Mbps. La vitesse du réseau entre les clusters est égale à 1Mbps. On suppose qu'une seule faute se produit à un instant donné et aucune faute ne se produit durant le recouvrement.

Durant une session d'établissement de point de reprise, on distingue trois rôles entre les différents processus :

- *Initiateur* : un processus initie une session de sauvegarde de point de reprise

- *Leader* : c'est un processus élu dans chaque cluster et qui coordonne les activités au sein de son cluster, en concordance avec les instructions de l'initiateur

- *Follower* : représente le reste des processus du système, ils suivent les instructions de leur leader.

Le protocole de point de reprise hiérarchique se déroule en deux phases :

- La première phase correspond à l'exécution de l'algorithme de point de reprise coordonné limité au cluster. Durant cette phase, les processus sont bloqués et sauvegardent un état global cohérent.

- La deuxième phase est aussi un point de reprise coordonné mais les leaders sont les seuls participants, avec l'initiateur qui joue le rôle de coordonnateur.

Les expériences ont montré que le surcoût des points de reprise dans l'approche hiérarchique est inférieur à celui du protocole coordonné standard. Cependant le protocole hiérarchique est sensible à la fréquence des messages entre clusters. En effet, le surcoût induit durant la sauvegarde hiérarchique des points de reprise augmente au fur et à mesure que la fréquence des messages augmente, et tend vers celui du protocole de point de reprise coordonné standard pour les applications fortement communicantes.

Journalisation causale hiérarchique. Les auteurs de [26] proposent un protocole de journalisation causale hiérarchique pour un meilleur passage à l'échelle. En effet, les algorithmes de journalisation sont utilisés avec succès dans les systèmes à petite échelle mais ils ne conviennent pas aux environnements à large échelle parce que :

- Le protocole maintient à tout moment les informations de dépendances des processus dans une structure de données qui peut quadrupler ***n***, ***n*** étant le nombre processus dans le système. Ainsi si le nombre de processus augmente, la mémoire nécessaire pour maintenir ces données devient importante.

- Les données estampillées au niveau des messages réduisent la bande passante et la latence de la grille.

L'approche hiérarchique utilise un réseau de proxies qui stocke dans la mémoire cache les informations de recouvrement. En plus des avantages du protocole standard de journalisation causale, l'avantage des proxies est qu'ils réduisent exponentiellement la taille des données nécessaires pour suivre les traces de la causalité. Les auteurs ont également trouvé que l'utilisation des proxies réduit considérablement la surcharge de la bande passante induite par les informations de causalité.

Protocole hybride. Dans [27], il est proposé un protocole hybride de points de reprise hiérarchique conçu pour les applications par couplage de code. Il combine une technique de sauvegarde coordonnée de points de reprise au sein d'une grappe et une technique de points de reprise induits par les communications entre les grappes. Contrairement aux protocoles hiérarchiques décrits précédemment, le nouveau protocole prend en charge les fautes simultanées. Son évaluation a montré que :

- Dès que les communications inter-grappe sont limitées, le nombre de points de reprise forcés l'est également, d'où la nécessité de régler le délai de garde entre les points de reprise non forcés pour une application dont les communications sont intenses.

- Il est nécessaire de bien régler la fréquence des déclenchements du ramasse-miette (qui coûte cher en encombrement réseau) et le coût en espace de stockage.

Groupe de processus. D'autres protocoles hiérarchiques utilisent les mécanismes de journalisation de message pour faciliter le recouvrement arrière. C'est le cas de la technique utilisée dans [28]. En effet, les protocoles de points de reprise imposent à tous les processus de revenir en arrière contrairement aux mécanismes de journalisation qui évite la restauration globale rejouant les messages du processus fautif uniquement. Cependant, la sauvegarde par journalisation introduit un surcoût sur la mémoire puisque chaque message échangé doit être enregistré sur support stable. C'est dans ce contexte, que les auteurs ont proposé une nouvelle méthode basée sur une organisation des processus en groupe, et qui réduit l'utilisation de la mémoire pendant l'exécution sans faute. Chaque groupe représente une unité lors du recouvrement et de la sauvegarde des messages.

L'idée principale est que seuls les messages inter-groupes sont stockés dans le journal, alors que pour les messages intra-groupes seuls leurs déterminants sont conservés pour connaître leur ordre de réception en cas de panne. En cas de défaillance, seul le processus fautif, ainsi que tous les autres processus appartenant au même groupe exécute la procédure de recouvrement [29]. Les messages inter-groupes sont tout simplement rejoués puisqu'ils sont stockés en mémoire stable. Ainsi, ce système représente une amélioration significative par rapport aux accès fréquents au support stable pour la sauvegarde des messages dans le journal.

Dans [30], les auteurs utilisent le même principe de groupes de processus et proposent un nouveau protocole de recouvrement arrière combinant un point de reprise coordonné au sein des groupes et un protocole de journalisation de messages fondé sur l'émetteur entre les groupes. Hormis l'avantage de limiter le recouvrement dans le groupe du processus fautif, les expériences ont montré que le protocole introduit un faible surcoût sur les performances des communications sur les petits messages et aucun surcoût sur les messages de grande taille.

7. Conclusion

Dans ce chapitre, nous avons présenté essentiellement les techniques utilisées pour assurer la tolérance aux fautes dans les grilles informatiques. Pour comprendre l'origine des pannes dans les grilles, nous avons exposé le vocabulaire relatif à la sûreté de fonctionnement, pour ensuite centrer notre étude sur les mécanismes de recouvrement arrière par reprise. Ces mécanismes se heurtent à plusieurs problèmes liés à la nature répartie des grilles qui rend complexe le calcul de l'état global cohérent du système, et au déterminisme de l'exécution des applications.

Ces problèmes liés à la reprise permettent de classifier les protocoles de recouvrement arrière en deux groupes : les protocoles de sauvegarde par point de reprise et les protocoles de journalisation des messages. Ces protocoles se différencient particulièrement de par leur

manière de sauvegarder des points de reprise, d'effectuer le recouvrement en cas de panne. Cependant, ces techniques ne sont pas totalement adaptées à tous les types de grilles. En effet, selon les applications et l'environnement, une technique présenterait plus d'avantages qu'une autre. C'est pour cela que nous avons orienté notre analyse en fin de chapitre sur les protocoles hiérarchiques, en adéquation avec l'architecture hiérarchique des grilles informatiques, et basés sur ces différents protocoles de recouvrement arrière.

Jusque là, il n'existe pas de solutions génériques de tolérances aux fautes dans les grilles informatiques. Dans le chapitre suivant, nous allons faire une étude comparative des protocoles de tolérance aux fautes, pour choisir la combinaison hiérarchique la mieux adaptée aux architectures de grilles.

Chapitre III : Étude comparative des mécanismes de recouvrement arrière

1. Introduction

Dans le chapitre précédent, nous avons présenté les techniques fondamentales de tolérances aux fautes utilisées dans les systèmes répartis. Il s'agit des techniques de réplication et de recouvrement arrière. Notre étude sera axée essentiellement sur les mécanismes de recouvrement arrière.

De nombreux algorithmes ont été proposés dans la littérature. Dans ce chapitre, nous analysons ces algorithmes dans le cadre de configurations de grille. Nous proposons des versions hiérarchiques de ces algorithmes et étudions leurs performances. Cette étude comparative s'appuie sur les principaux protocoles à base de point de reprise et de journalisation qui ont été décrits dans le chapitre précédent :

- Protocole de points de reprise coordonné bloquant [10].
- Protocole de points de reprise coordonné non- bloquant : solution de Chandy-Lamport[12].
- Protocole de points de reprise induit par les communications [10].
- Journalisation pessimiste [11].
- Journalisation optimiste [16].
- Journalisation causale [26] [16].

Les mécanismes de points de reprise indépendants ne feront pas partie des protocoles implémentés car ils sont sujets à un effet domino lors des reprises qui rend leur utilisation délicate pour des applications réelles. L'ensemble de ces techniques de sauvegarde a été implémenté dans OMNET++ dans deux types d'architecture : une architecture plate à savoir un cluster et une architecture hiérarchique, à savoir une grille informatique. Cette comparaison permet de déterminer quels sont les protocoles les mieux adaptés aux architectures de grille.

La section 2 fait une analyse qualitative des protocoles. Dans la section 3, nous présentons les versions hiérarchiques de ces protocoles. Enfin la section 4 est l'étude des performances.

2. Analyse comparative

Les mécanismes de recouvrement arrière sont les techniques de base pour assurer la tolérance aux fautes dans les grilles informatiques. Les protocoles à base de point de reprise et de journalisation permettent de sauvegarder les états locaux du système qui seront utilisés durant le recouvrement pour éviter que les applications ne recommencent leur exécution à partir du début.

2.1. Comparaison des protocoles de points de reprise

Les mécanismes à base de points de reprise permettent de sauvegarder les états globaux cohérents de la grille à un instant donné.
Ces protocoles se différencient par les critères suivants :

- Le surcoût engendré durant l'exécution sans faute.
- Le surcoût du recouvrement.
- L'utilisation du support de stockage stable.
- La résistance aux effets domino et messages orphelins.

2.1.1. Le surcoût engendré durant l'exécution sans faute

L'approche indépendante est la technique la plus simple dans sa manière de sauvegarder les états des processus. En effet, les processus sauvegardent leur état sans aucune coordination, tandis que pour la sauvegarde coordonnée, l'exécution est bloquée durant toute la procédure d'établissement des points de reprise. La synchronisation des processus engendre un surcoût assez élevé si la latence du réseau est importante, surtout quand il s'agit d'une grille informatique où les clusters sont reliés par des liens à longue distance.

Les protocoles induits par les communications permettent d'éviter le coût des synchronisations tout en maintenant des ensembles de points de reprise représentant un état global cohérent. Cependant, leur principal surcoût est dû à la pose de nombreux points de reprise forcés. Ces derniers sont effectués en utilisant des données supplémentaires ajoutées aux messages. Ces données peuvent ralentir les communications du réseau. Des points de reprise sont forcés aussi après chaque interaction avec l'extérieur. Dans le cas d'applications fortement communicantes avec plusieurs interactions avec l'extérieur, les temps d'exécution vont s'allonger à cause de la sauvegarde de ces points de reprise forcés.

2.1.2. Le surcoût du recouvrement

Même si le protocole indépendant simplifie la manière d'établir des points de reprise, le recouvrement est complexe. Les derniers états locaux des processus ne font pas toujours partie de la ligne de recouvrement. Pour calculer un état global cohérent, le protocole est obligé, dans la plupart des cas, de faire plusieurs retours arrière.

Pour la technique coordonnée, tous les points de reprise sont utiles. Le système redémarre à partir du dernier point de reprise. Le protocole induit par les communications peut aussi sauvegarder des points de reprise qui ne sont pas utiles. Ces derniers peuvent faire partie de chemin zigzag [42] qui entraine des cycles qui rendent la procédure de recouvrement assez longue.

2.1.3. L'utilisation du support de stockage stable

Les points de reprise sauvegardés par les protocoles sont enregistrés sur un support de stockage stable qui peut être par exemple un serveur NFS répliqué ou un disque de stockage redondant. Les protocoles indépendants effectuent plusieurs points de reprise au cours de l'exécution des applications. Comme les processus sauvegardent leurs états de manière totalement indépendante, le protocole maintient plusieurs points de reprise en support stable pour garantir le calcul d'un état global cohérent lors de la réexécution. De même, le protocole induit par les communications préserve plusieurs points de reprise. Le maintien de plusieurs points de reprise sur support stable engendre un large surcoût durant la procédure de libération (*garbage collection*).

À la différence de la technique indépendante, un protocole coordonné ne garde que le dernier point de reprise sur le support de stockage car elle garantit la ligne de recouvrement. Dès qu'un point de reprise est sauvegardé, le précédent est supprimé.

2.1.4. La résistance aux effets domino et messages orphelins

Un état global cohérent est un état ne contenant pas de messages orphelins. Il faut que le protocole garantisse une ligne de recouvrement ne contenant aucun processus orphelin. La sauvegarde de points de reprise coordonnée garantit l'absence d'orphelin. Cependant, en cas de défaillance, le protocole indépendant a tendance à produire des messages orphelins qui vont entrainer un effet domino. Ce dernier peut faire redémarrer l'exécution à partir du début.

2.2. Comparaison des protocoles de journalisation

L'objectif principal des protocoles de journalisation est l'enregistrement des messages qui transitent entre les processus sur un support stable, afin d'être utilisé ultérieurement en cas de défaillance. Durant le recouvrement, l'exécution redémarre au dernier point de reprise, et les messages sauvegardés sont rejoués. Nous allons comparer les mécanismes de journalisation selon les critères suivants :

- Le surcoût durant l'exécution.
- Les performances durant le recouvrement.
- L'utilisation du support de stockage stable.

2.2.1. Le surcoût durant l'exécution

La journalisation pessimiste a un impact significatif sur le temps de communication. En effet, pour chaque envoi, le protocole enregistre le message sur mémoire stable avant que le processus ne soit autorisé à interagir avec le système. Cette synchronisation introduit un large surcoût sur le temps d'exécution des applications.

À la différence du protocole pessimiste, le surcoût de l'approche optimiste est réduit en diminuant la fréquence d'enregistrement des messages sur le support stable. Dans la journalisation optimiste les messages sont sauvegardés par vague. Chaque message envoyé est enregistré en mémoire volatile ; à un instant donné les messages en mémoire sont déversés sur le support stable. Ainsi pour éviter le ralentissement des communications, il suffit de réduire les fréquences d'accès au support stable. Cependant, ce réglage peut allonger le temps de recouvrement, si la fréquence d'enregistrement est trop longue.

Si les mécanismes d'enregistrement pessimiste et optimiste ont un surcoût lié aux sauvegardes fréquentes sur le support stable, la journalisation causale quant à elle induit d'importantes quantités de données transportées à travers le réseau. Comme décrit dans le chapitre précédent, l'approche causale ajoute des déterminants dont la taille peut être très grande dans les messages échangés entre les processus. Les communications à travers le réseau peuvent être alors surchargées, ce qui augmente le temps d'exécution des applications.

2.2.2. L'utilisation du support de stockage stable

En comparant des temps d'exécution des protocoles de journalisation, nous avons observé que les accès fréquents au support stable introduisent un surcoût important. En effet, la journalisation est basée sur la sauvegarde des messages. Le support de stockage joue donc un rôle prépondérant dans la mesure où il doit pouvoir contenir tous les messages transitant dans le réseau.

Le protocole d'enregistrement causal augmente la taille des messages à cause des données

supplémentaires ajoutées aux messages. Mais il existe plusieurs variantes dans ce protocole. L'utilisation du support de stockage dépend de la variante utilisée. En effet, les auteurs de [35] [36] [37] utilisent la mémoire stable pour garder les informations causales. Ces dernières sont enregistrées régulièrement sur le support. Le protocole de journalisation causale utilise ainsi partiellement les supports de stockage. Mais tel n'est pas le cas pour les deux autres mécanismes de journalisation.

Les protocoles à enregistrement pessimiste et optimiste utilisent le support stable pour sauvegarder les messages. La première approche ne modifie pas la taille des messages, alors que la deuxième enregistre les informations de dépendance dans un vecteur ou graphe de dépendance dont la taille peut atteindre le nombre total de processus du système.

2.2.3. Les performances durant le recouvrement

Les performances durant le recouvrement dépendent du nombre de processus à reprendre. Quel que soit le protocole de journalisation utilisé, le processus fautif redémarre au point de reprise le plus récent et rejoue les messages du journal. Dans le cas de la journalisation optimiste, plusieurs processus peuvent être amenés à reprendre leur exécution. En effet si une panne survient avant l'enregistrement des messages sur support stable, une partie de l'exécution sera perdue, il faut alors régénérer les messages en ré-exécutant les processus émetteurs à partir des données antérieures sauvegardées. Ceci rend le recouvrement long et complexe à cause du nombre potentiellement important de processus qui reprennent et ainsi que le nombre de retour arrière.

2.3. Synthèse

Le tableau 1 est un récapitulatif des différences entre les protocoles de tolérances aux fautes. À travers les comparaisons faites précédemment, les deux classes de protocoles de recouvrement arrière se différencient par les critères comme le surcoût durant l'exécution sans recouvrement, les performances durant le recouvrement et les accès au support de stockage.

Une des principales différences entre les protocoles à base de point de reprise et de journalisation est la complexité du recouvrement. En effet, pour la journalisation, seul le processus fautif reprend tandis que pour les techniques de point de reprise tous les processus reprennent leur exécution.

Dans la section suivant, nous proposons des versions hiérarchiques de ces algorithmes, puis nous comparons leurs performances en les implémentant sous deux architectures différentes, dans un cluster et dans une grille informatique, en utilisant un simulateur.

	Journalisation Pessimiste	Journalisation Optimiste	Journalisation Causale	Sauvegarde Coordonnée	Sauvegarde Non Coordonnée	Sauvegarde Induite par les Com.
Hypothèse PWD	Oui	Oui	Oui	Non	Non	Non
Effet Domino	Non	Non	Non	Non	Possible	Non
Point de reprise par processus	Un	Plusieurs	Un	Un	Plusieurs	Plusieurs
Processus Orphelin	Non	Possible	Non	Non	Possible	Possible
Propagation de reprise	Dernier point de reprise	Plusieurs points de reprise	Dernier point de reprise	Dernier point de reprise	Non bornée	Plusieurs points de reprise

Tableau 1 : Tableau comparatif des mécanismes de recouvrement arrière [38]

3. Algorithmes hiérarchiques

Les environnements à grande échelle telles que les grilles informatiques permettent l'exécution de grandes applications scientifiques qui nécessitent des temps d'exécution qui peuvent aller sur plusieurs heures voire plusieurs jours. Le principal objectif de ces infrastructures est d'utiliser au mieux les puissances de calcul et de stockage disponibles au sein des ressources connectées à la grille informatique. Pour assurer la sûreté de fonctionnement au sein de la grille, il faudra mettre en place des mécanismes de tolérance aux fautes adaptés à leur architecture. C'est dans ce contexte que nous avons adapté les protocoles de points de reprise et de journalisation aux grilles informatiques.

3.1. Modèle du système

Nous considérons qu'une grille informatique est un ensemble de ressources regroupé dans différents clusters. On pourra définir ainsi une grille informatique comme une fédération de clusters reliés par un réseau de type WAN (Wide Area Network).

Nous nous intéressons à la partie logicielle, c'est-à-dire à la manière dont les applications s'exécutent. Les applications installées dans les grilles, en général dans les systèmes répartis, communiquent **par passage de messages**. Dans un système de communication par passage de messages, l'acteur principal est le **processus**.

Les processus communiquent en s'échangeant des messages en utilisant au moins deux procédures :

- Une procédure d'envoi de messages
- Une procédure de réception de messages

Les communications par messages ont pour but l'échange de données ou la synchronisation entre les processus. Chaque envoi ou réception de messages est un événement. Les échanges entre les processus produisent des dépendances entre eux. Lamport [12] a défini une relation de précédence entre les événements survenus entre les différents processus : l'émission d'un message précède causalement la réception du message.

En résumé, notre modèle de système utilise la communication par passage de messages, en respectant la relation causale de Lamport. Pour la gestion des types défaillances, nous ne considérons que les *pannes franches* de processus. Il s'agit du modèle de pannes le plus simple à gérer. En cas de défaillance, le processus arrête immédiatement de fonctionner, il n'émet plus de messages et n'en reçoit plus.

Les solutions de calcul d'état global cohérent sont basées sur la relation de précédence de Lamport [8] qui permettra d'ordonner les événements inter-processus. La tolérance aux fautes devient ainsi un besoin imposé par les communications inter-processus et la nature hétérogène des grilles informatiques. Cependant les protocoles de tolérance aux fautes utilisés dans les systèmes répartis sont en général utilisés dans les réseaux de type LAN ou limité uniquement au niveau d'un cluster.

Notre approche inspirée des solutions des auteurs de [25] [27], est basée sur la structure hiérarchique des grilles informatiques. Nous avons adapté les protocoles de point de reprise

et de journalisation à l'architecture hiérarchique de la grille. Le cluster est l'unité représentative dans laquelle va s'exécuter un des protocoles de sauvegarde d'états. Au sein de chaque cluster, un **processus leader** assure le rôle de coordonnateur (*Figure 12*). Il est relié aux autres processus appartenant au même cluster, mais aussi aux leaders des autres clusters appartenant à la même grille informatique. Quel que soit le protocole utilisé, le leader assure toujours le rôle d'intermédiaire dans les communications inter-cluster. Nous décrivons dans la section suivante la procédure d'établissement des points de reprise et de journalisation des messages dans les grilles.

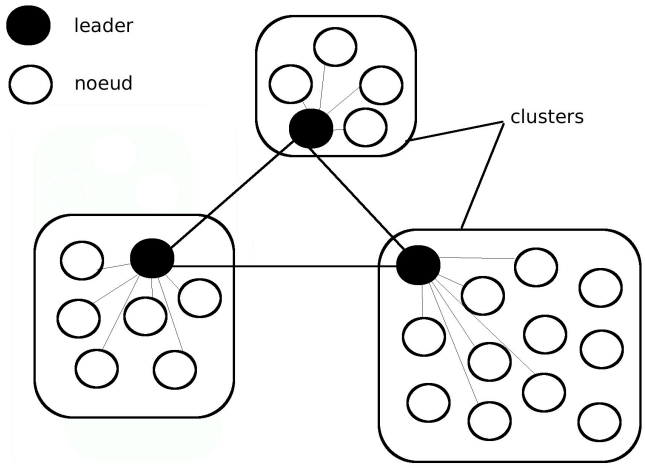

Figure 12 : Architecture hiérarchique d'une grille

3.2. Protocoles hiérarchiques

Dans cette section, nous décrivons deux protocoles hiérarchiques représentatifs qui illustrent notre approche à base de leader.

3.2.1. Point de reprise coordonné

Sauvegarde coordonnée :

La procédure de sauvegarde de l'état global cohérent de la grille peut être divisée en quatre grandes phases:

- *Phase d'initialisation* : un processus initiateur envoie une demande d'établissement de point de reprise à son leader.
- *Phase de coordination des leaders* : Durant cette phase les leaders constituent les uniques acteurs. Après la réception de la demande de sauvegarde, le leader de l'initiateur envoie la même demande de point de reprise aux autres leaders de la grille.
- *Phase de sauvegarde de point de reprise local* : après la réception de la demande, chaque cluster sauvegarde son état local en utilisant le mécanisme de point de reprise coordonné décrit dans le chapitre précédent.
- *Phase de clôture* : c'est la dernière étape où chaque leader envoie un accusé de réception au leader de l'initiateur pour lui signifier la fin de sauvegarde de l'état global de son cluster.

L'algorithme 1 présente le traitement des messages pour le protocole coordonné hiérarchique. Cinq types de messages peuvent circuler dans le réseau :

- Les messages de l'application
- Les messages de demande de point de reprise 'ckptRqtMsg'
- Les accusés de réception 'ackRMsg', 'ackFMsg'
- Les messages de validation 'commitMsg'
- Les fautes 'FaultMsg'

Initialement, chaque processus exécute la fonction *initialize* qui permet de fixer la fréquence des points de reprise. Cette fonction, comme son nom l'indique, permet aussi d'initialiser les canaux entrants et sortants des processus (ligne 15), ainsi que les différents des états des processus pris au cours de l'exécution des applications (lignes 6 et 10). Ensuite le processus récepteur fait appel à la fonction *handleMessage*. Cette fonction exécutée à chaque réception de messages permet d'enregistrer les messages reçus par chaque processus dans une structure appelée '*canalMsg'* (lignes 2 et 13). Pour démarrer une session de sauvegarde, un processus initiateur envoie la demande à son leader (ligne 12). À la réception de la demande, le processus leader transfère la demande aux autres leaders (ligne 26) et la diffuse le message dans son cluster (ligne 27). Un processus qui reçoit une demande de sauvegarde effectue un point de reprise (ligne 34) si ce dernier ne l'a pas encore fait et envoie un accusé de réception à son leader (ligne 35). Dès que tous les accusés de réception sont reçus, le leader du cluster envoie un message de validation au leader initiateur de la sauvegarde pour clôturer la procédure de sauvegarde de l'état de la grille (ligne 41). En cas de faute, un message '*FaultMsg'* est diffusé à tous les nœuds dans la grille pour exécuter la procédure de recouvrement (lignes 50 et 52) détaillée dans l'algorithme 2.

```
     Variables
1.   etatNoeud.chkpt     (* indique si le processus a effectué un point de reprise *)
2.   canalMsg            (* messages contenus dans les canaux d'entrée et de sortie *)
3.   inFault             (* indique si le processus fait un recouvrement *)
4.   senderMsg           (* émetteur du message *)
5.   freqCkpt            (* indique la fréquence de sauvegarde *)

     initialize ()
6.        etatNoeud.chkpt=false
10.       inFault=false
11.       canalMsg=vide
12.       envoyer ckptRqtMsg toutes les freqCkpt unités de temps

     handleMessage (cMessage *msg)
13.       canalMsg=canalMsg+msg
14.       si ((ckptRqtMsg ∈ canalMsg) ou (FaultMsg ∈ canalMsg))
15.       alors
16.            bloquer les communications
17.       sinon
18.            délivrer msg à l'application
       ** A la réception d'une demande d'un point de reprise **
19.       si (msg= ckptRqtMsg)
20.       alors si (processus=leader)
```

```
21.                 alors
22.                         si (senderMsg=initiateur)
25.                            alors
26.                                  envoyer ckptRqtMsg aux leaders
27.                                  diffuser ckptRqtMsg dans le cluster
28.                            sinon
29.                                 diffuser ckptRqtMsg dans le cluster
30.                 sinon
31.                         si etatNoeud.chkpt=false
32.                         alors
33.                                 etatNoeud.chkpt=true
34.                                 makeCheckpoint()
35.                                 envoyer (ackRMsg)

36.         si (msg= ackRMsg)
37.         alors si (nombre(ackRMsg)=(nombre(processus du cluster)))
40.                 alors
41.                         envoyer commitMsg à leader(initiateur)
42.                 sinon
43.                         nombre(ackRMsg)= nombre(ackRMsg)+1
    ** A la réception d'une demande de recouvrement **
44.         si (msg= FaultMsg)
45.         alors si (processus=leader)
46.                 alors
47.                         si (senderMsg=initiateur)
48.                            alors
49.                                  envoyer FaultMsg aux leaders
50.                                  diffuser FaultMsg dans le cluster
51.                            sinon
52.                                 diffuser FaultMsg dans le cluster
53.                 sinon
54.                         inFault= true
55.                         recovery()
56.                         envoyer (ackFMsg)

57.         si (msg= ackFMsg)
58.         alors si (nombre(ackFMsg)=(nombre(processus du cluster)))
59.                 alors
60.                         envoyer commitMsg à leader(initiateur)
61.                 sinon
62.                         nombre(ackFMsg)= nombre(ackFMsg)+1
```

Algorithme 1: Protocole coordonné hiérarchique

L'algorithme 2 représente les procédures exécutées respectivement lors de l'établissement d'un point de reprise d'un processus et durant le recouvrement.

```
   Variables
1. etatProci
2. inFault  (* indique si le processus a fait un recouvrement *)

   makeCheckpoint ()
3.       vider canaux d'entrée et canaux de sortie
4.       etatProci = etat processus i
5.       débloquer communication
6.       etatNoeud.chkpt=false

   recovery()
7.       etatProci = copie etat processus i
8.       inFault =false
9.       débloquer communication
```

Algorithme 2 : Établissement de point de reprise et de recouvrement

Recouvrement hiérarchique :

Le recouvrement obéit aux mêmes règles que le calcul de l'état global avec les mêmes quatre phases:

- *Initialisation du recouvrement* : Le processus fautif envoie un message de demande d'exécution de la procédure de recouvrement à son leader
- *Coordinations des leaders* : le leader du cluster du processus fautif transfère la demande de recouvrement aux leaders (Algorithme 1 : ligne 49).
- *Recouvrement au sein du cluster* : tous les processus exécutent la procédure de recouvrement après avoir reçu la demande de leur leader (Algorithme 1 : ligne 55).
- *Clôture du recouvrement du système* : chaque leader envoie un message au leader du processus fautif (Algorithme 1 : ligne 56).

3.2.2. Journalisation optimiste

Comme décrit dans le chapitre précédent, le protocole établit des points de reprise indépendants et sauvegarde les messages sur support stable en utilisant le mécanisme de journalisation optimiste.

Nous avons ajouté un nœud supplémentaire au réseau. Ce nœud représente le serveur de stockage stable.

Au cours de l'exécution d'une application, le protocole enregistre les messages dans la mémoire du processus (Algorithme 3). Un compteur (ligne 6) permet d'enregistrer le nombre de messages reçus par le processus. Dès qu'un processus reçoit un nombre de messages égal à un certain seuil (ligne 7), la procédure de sauvegarde en mémoire stable est enclenchée. Elle se traduit par l'envoi du message 'stockMsg' à tous les processus de la grille. À la réception du message (ligne 12), tous les messages stockés dans les mémoires des processus commencent à être sauvegardés vers le serveur de stockage.

En respectant la structure de la grille informatique, tous les messages transitent par le leader du cluster avant d'être transférés vers le serveur de stockage.

```
      Variables
1.    booléen inFault    (* indique si le processus est en cours de recouvrement *)
2.    nbMsg=0            (* nombre de messages reçus par un processus *)
3.    etatNoeud.chkpt    (* indique si le processus a effectué un point de reprise *)
4.    logThreshold       (* nombre de message à recevoir avant la journalisation *)

5.    handleMessage (cMessage *msg)
6.          nbMsg=nbMsg+1 ;
7.          si (nbMsgVolatilMem = logThreshold)
8.          alors
9.                envoyer (stockMsg) au leader
10.         sinon
11.               saveOnVolatilMem ()
12.
13.         si ((processus=leader) et (msg=stockMsg))
14.         alors
15.               diffuser stockMsg dans le cluster
16.         sinon
17.               saveOnServer ()
18.
19.   ** A la réception d'une demande d'un point de reprise **
20.         si (msg= ckptRqtMsg)
21.         alors si (processus=leader)
22.               alors
23.                     si (senderMsg=initiateur)
24.                        alors
25.                              envoyer ckptRqtMsg aux leaders
26.                              diffuser ckptRqtMsg dans le cluster
27.                        sinon
28.                              diffuser ckptRqtMsg dans le cluster
29.               sinon
30.                     si etatNoeud.chkpt=false
31.                     alors
32.                              etatNoeud.chkpt=true
33.                              makeCheckpoint()
34.                              envoyer (ackRMsg)

35.         si (msg= ackRMsg)
36.         alors si (nombre(ackRMsg)=(nombre(processus du cluster)))
37.               alors
38.                     envoyer commitMsg à leader(initiateur)
39.               sinon
40.                     nombre(ackRMsg)= nombre(ackRMsg)+1

41.   ** A la réception d'une demande de recouvrement **
42.         si (msg= FaultMsg)
```

```
43.   alors si (processus=leader)
44.         alors
45.               si (senderMsg=initiateur)
46.                  alors
47.                        envoyer FaultMsg aux leaders
48.                        diffuser FaultMsg dans le cluster
49.                  sinon
50.                        diffuser FaultMsg dans le cluster
51.         sinon
52.               inFault= true
53.               recovery()
54.               envoyer (ackFMsg)
55.
56.   si (msg= ackFMsg)
57.   alors si (nombre(ackFMsg)=(nombre(processus du cluster)))
58.         alors
59.               envoyer commitMsg à leader(initiateur)
60.         sinon
61.               nombre(ackFMsg)= nombre(ackFMsg)+1
```

Algorithme 3: Journalisation optimiste 'hiérarchique'

4. Évaluation de performance

La plupart des implémentations sur les techniques de tolérance aux fautes ont été effectuées sur des clusters.

L'objectif de notre étude est de savoir quel est le protocole ou la combinaison de protocole la mieux adapté aux grilles informatiques. C'est dans ce contexte que nous avons implémenté les six protocoles décrits dans l'état de l'art : les trois techniques de journalisation de message, la sauvegarde par point de reprise non bloquant (solution de Chandy et Lamport [12]), bloquant et induit par les communications. Chacune de ces techniques a été implémentée dans un cluster et dans une architecture de grille avec dans ce cas une version hiérarchique des protocoles.

Cependant il est très coûteux d'estimer les performances des protocoles dans des environnements différents pour une application bien déterminée. C'est pour cette raison que les expérimentations ont été effectuées dans le simulateur OMNeT ++ [39]. OMNet++ nous permet d'évaluer les métriques pour différentes configurations de grilles et plusieurs types d'applications.

La performance d'un protocole recouvrement arrière dépend du coût dû aux points de reprise et au recouvrement. En cas de faute, le coût du recouvrement dépend de l'efficacité du protocole de restauration de l'état et surtout de la quantité de calcul perdue à cause des retours arrières.

4.1.Architectures simulées

Nous avons implémenté les six protocoles de recouvrement arrière dans ces deux types d'architecture en utilisant le même nombre de nœuds (*Figure 13*).

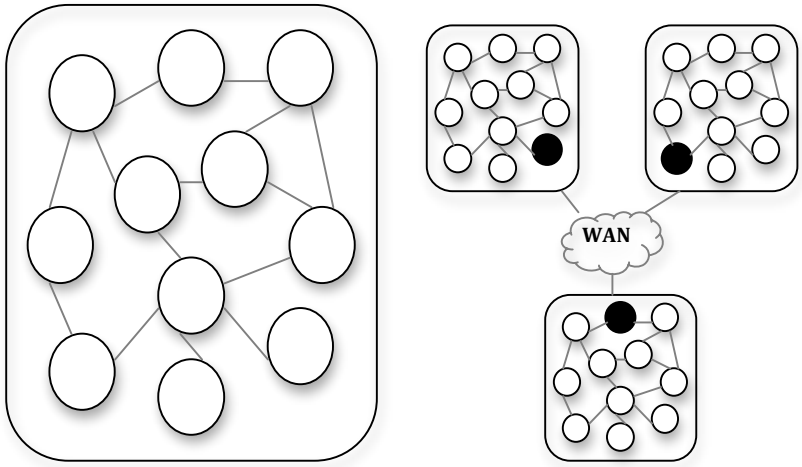

Figure 13 : Plat vs Hiérarchique

4.1.1. Modèles d'application

Pour évaluer les six protocoles, nous avons implémenté deux modèles d'applications :

- Une application de diffusion de messages.
- Une application d'envoi de messages en jeton.

Ces modèles d'application ont deux modes de communications extrêmement différents. En effet pour les applications de diffusion de messages, un message est diffusé vers tous les nœuds du réseau. Chaque message est transmis simultanément à tous les processus de la grille. Toutefois les communications sont quasi nulles entre deux diffusions. Par contre pour les applications à jeton, c'est un seul message qui circule en permanence dans la grille. Ces applications permettent de mesurer l'impact des protocoles implémentés durant des envois simultanés de messages et durant les communications point à point.

La simulation se déroule pendant 1000 secondes. La configuration à plat, est composée de 50 processus regroupés dans un même cluster composé de 25 nœuds. Pour la grille, le même nombre de processus a été réparti uniformément dans 5 clusters chacun possédant 5 nœuds. Pour chaque nœud, il y a 2 processus que s'y exécutent.

4.1.1.1. Diffusion de messages

Chaque processus possède un identifiant unique. L'application de diffusion s'exécute de la manière suivante : un processus initiateur, diffuse un message toutes les 30 secondes à tous les processus. Au niveau de l'architecture composée uniquement d'un cluster, les processus reçoivent les messages directement. En mode grille, les leaders servent d'intermédiaires entre le processus initiateur et les autres processus (Figure 14) :

1) Le processus initiateur envoie un message à son leader.
2) Le leader transmet le message aux autres leaders.
3) Chaque leader diffuse le message dans son cluster.

Techniquement, les messages diffusés sont des messages spécifiques de par leur destination. Ils sont identifiés par la valeur de la destination qui est nulle. Dès qu'un processus envoie ce type de message à son leader, le système identifie un début de diffusion de messages. Il active la programmation d'une autre diffusion après 30 secondes. Ainsi l'application continue à diffuser des messages toutes les 30 secondes jusqu'à la fin de la simulation.

Ce modèle d'application a été implémenté pour évaluer la capacité des protocoles à supporter des envois simultanés de messages.

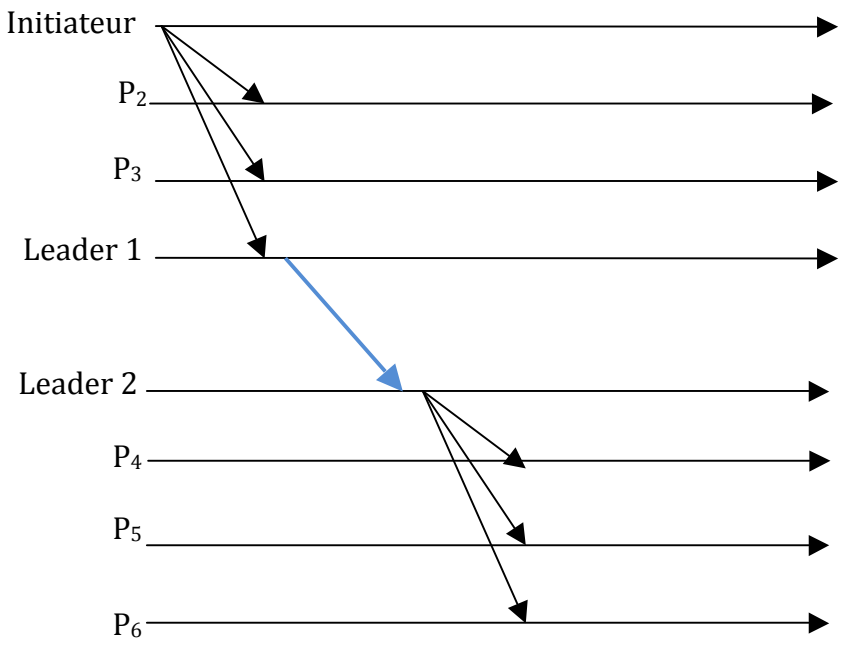

Figure 14 : Diffusion de message

4.1.1.2. Application en Jeton

En début de simulation, un processus aléatoire envoie un message à un autre processus choisi aussi aléatoirement. Le message circule entre les processus jusqu'à la fin de la simulation (Figure 15). Les messages inter-clusters passent par l'intermédiaire des leaders pour atteindre leur destination dans le cas de l'architecture en grille. Durant toute la durée de la simulation, un message circule en permanence, le réseau est toujours en activité ; alors que dans le cas de la diffusion de messages, entre deux diffusions les communications sont presque inexistantes.

Ces deux modèles d'application nous permettront d'évaluer les protocoles de tolérances aux fautes durant leurs exécutions et en cas de défaillance.

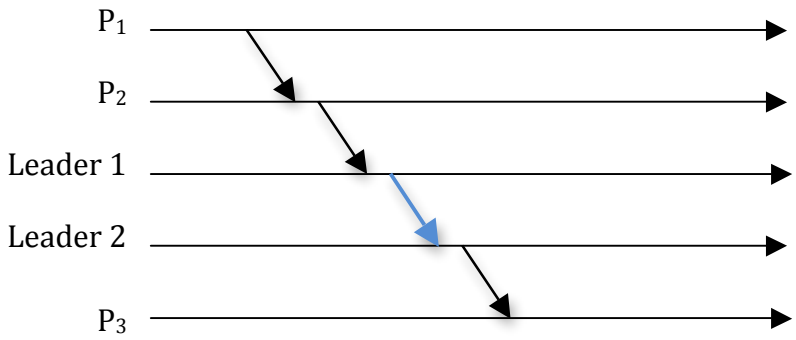

Figure 15 : Jeton: un message circule dans la grille de nœud à nœud

4.2. Environnement de simulation

OMNeT++ est un environnement de simulation à évènements discrets basé sur la plateforme Eclipse. Son domaine d'application principal est la simulation de réseaux de communication. De par son architecture générique, il est aussi utilisé avec succès dans d'autres domaines comme la simulation de systèmes informatiques complexes, des réseaux de files d'attente, les réseaux sans fil, ou la validation des architectures matérielles. Le simulateur OMNeT++ est développé sur C++. Il est constitué essentiellement des composants suivants [39]:

- le noyau de simulation,
- les classes utilitaires,
- les éléments constitutifs de la simulation : les modules, les canaux, les messages,
- les fichiers de configuration,
- les environnements de simulation,
- l'interface de conception basée sur Eclipse,
- des extensions pour la simulation parallèle, de connexion base de données, etc.

4.2.1. Configuration des réseaux

Pour implémenter les protocoles de tolérances aux fautes, nous avons mis en place un réseau de 25 nœuds, avec 2 processus par noeud. Ces nœuds sont répartis uniformément sur cinq clusters pour constituer une grille informatique (Figure 16). Dans le cas des mécanismes de journalisation, un nœud supplémentaire est ajouté à la configuration pour jouer le rôle de serveur de stockage stable. Ce nœud est résistant aux fautes et est accessible par tous les leaders des clusters de la grille informatique. Dans les deux configurations, la latence d'accès au serveur de stockage est de 100ms.

Une simulation OMNeT++ se compose de modules. Principal composant du réseau, ils sont organisés de manière hiérarchique et permettent une grande flexibilité dans la création des nœuds. Ces modules peuvent être imbriqués; on distinguera dans ce cas des modules simples et des modules composés. Les modules simples contiennent les algorithmes du modèle. Leur création constitue la première étape de la configuration. En effet, ils permettent de décrire la structure des nœuds. Les modules communiquent via des ports par passage de message, dont la structure est définie par le concepteur de la simulation. Les messages peuvent être envoyés directement au module.

Dans le cas de notre simulateur, nous avons configuré un module générique qui peut être soit un nœud simple ou un serveur de stockage. Nous avons utilisé des modules avec deux ports de communication : un port d'entrée et un port de sortie. Le nœud envoie un message via le port de sortie et reçoit un message via le port d'entrée.

Après avoir créé les nœuds avec leurs ports, l'étape suivante consiste à mettre en place le réseau. Des connexions entre les nœuds sont créées, ainsi que les liens avec le serveur. Pour la grille, chaque cluster possède un routeur. La latence pour les communications intra-cluster est égale à 0,1 ms et celle des messages inter-cluster est égale à 100 ms.

Les modules, les ports, les connexions, ainsi que leurs paramètres sont définis dans un fichier appelé Fichier NED (NEtwork Description) écrit en utilisant le langage NED propre au simulateur (Annexe 1).

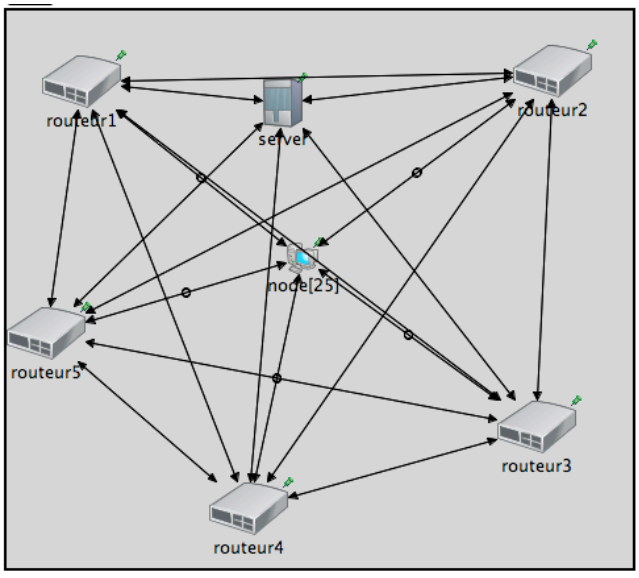

Figure 16 : Configuration de la grille sur OMNeT++

4.2.2. Implémentation des protocoles de tolérance aux fautes

Une simulation OMNeT++ est composée de quatre types de fichiers :

- Les fichiers NED (.ned)
- Les fichiers de description de messages (.msg)
- Les fichiers sources des modules écrits en C++
- Le fichier de configuration (.ini)

Les programmes de simulation sont construits à partir de ces fichiers en utilisant le noyau de simulation et les interfaces fournies par OMNeT++. Dans un premier temps, les fichiers de description de messages sont traduits en C++ ; ils sont ensuite compilés avec les autres fichiers pour obtenir le programme exécutable qui représente la simulation finale. Le résultat de cette dernière est écrit dans des fichiers de sorties vectoriels et scalaires qui peuvent être interprétés par d'autres outils de programmation tels que Matlab, Python ou des tableurs (Figure 17).

La mise en place d'une simulation commence par la construction du réseau, dans notre cas le cluster et la grille, en utilisant des fichiers NED (Annexe 1). Cette étape est suivie par la création des fichiers sources décrivant toutes les interactions entre les éléments de la simulation.

Tous les composants du réseau, à savoir les modules, les ports, les connexions et les messages sont représentés par des classes C++ [39]:

- cMessage
- cSimpleModule
- cGate

Ces classes peuvent être reprogrammées en redéfinissant leurs fonctions membres. Nous avons défini un projet de simulation pour chacun des six protocoles en utilisant les deux architectures décrites plus haut.

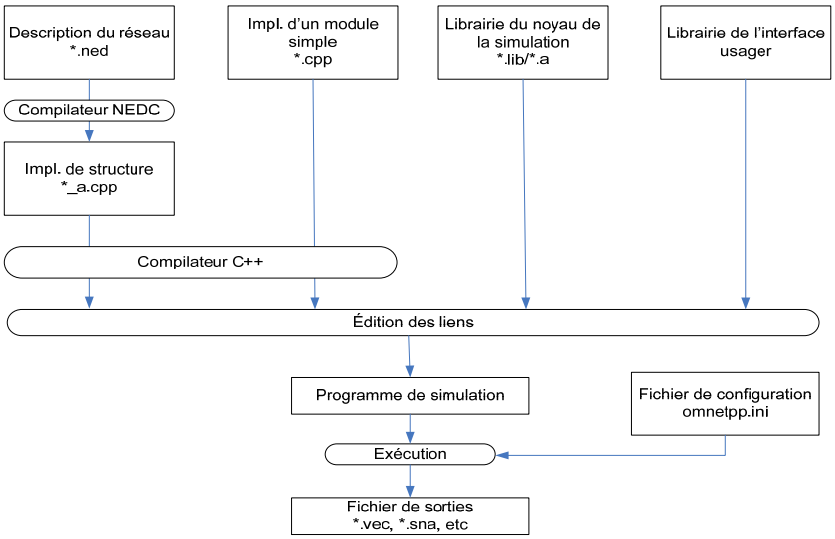

Figure 17 : Processus de programmation dans OMNeT++

À côté des classes de base fournies par OMNeT++, chaque module défini dans le projet de simulation doit être décrit dans une classe contenant toutes les fonctionnalités du module et qui hérite de la classe ***cSimpleModule*** propre à OMNeT++. Elle comporte trois fonctions fondamentales qui doivent être redéfinies par le programmeur:

- ***initialize ()*** : c'est la première méthode invoquée par le simulateur après la mise en place du réseau, c'est à dire après la création des modules et des liens entre eux. Elle permet de lancer le premier message qui marque le début de la simulation ; mais aussi d'initialiser les variables des classes et les paramètres des modules.
- ***handleMessage (cMessage *msg)*** : cette méthode est exécutée par le module à chaque réception de message. Toutes les fonctionnalités du module sont décrites dans cette fonction. Par exemple, pour la sauvegarde coordonnée, c'est la même fonction qui est exécutée à la réception de la demande de sauvegarde ou à la fin de la coordination des processus.
- ***finish ()*** exécutée à la fin d'une simulation normale. Elle permet d'enregistrer les statistiques sur les modules, comme par exemple le nombre de messages envoyés ou reçus.

4.2.3. Évaluation

Nous allons présenter les performances de l'application d'envoie de messages en jeton [43]. Les mêmes phénomènes ont été observés sur l'application de diffusion. Une étude plus approfondie sera faite dans les chapitres suivants.

4.2.4. Exécution sans faute

La Figure 18 présente le temps d'exécution de l'application d'envoi de messages en jeton ('AppJeton') dans les deux types d'architecture. En exécution normale, les processus s'envoient des messages. Les points de reprise sont effectués toutes les 180 secondes.

Il est évident que le temps mis par une application pour s'exécuter sans effectuer de points de reprise est inférieur au temps d'exécution avec sauvegarde d'état local.

Les protocoles de points de reprise induits par les communications (CIC: Communication-Induced Checkpointing) introduisent un surcoût important sur le temps d'exécution de l'application par rapport aux autres protocoles. Il est dû aux points de reprise forcés. En effet dans notre cas, le protocole effectue une sauvegarde après chaque réception de message inter-cluster. Notre application envoie des messages inter-clusters en moyenne toutes les 15 secondes.

La technique de Chandy-Lamport présente de meilleures performances parce que même si la sauvegarde est coordonnée, l'exécution de l'application n'est pas bloquée durant la procédure. Cependant, le temps d'exécution est plus élevé en mode cluster qu'en mode grille. En effet, comme le protocole utilise des marqueurs pour coordonner la sauvegarde, ces derniers sont envoyés à tous les processus, qui à leur tour envoient des acquittements à tous les autres processus pour leur signifier que le point de reprise est déjà effectué. Dans ce cas, c'est 50 messages supplémentaires qui seront envoyés par chaque processus, tandis qu'en mode grille les processus envoient des acquittements uniquement à ceux appartenant au même cluster.

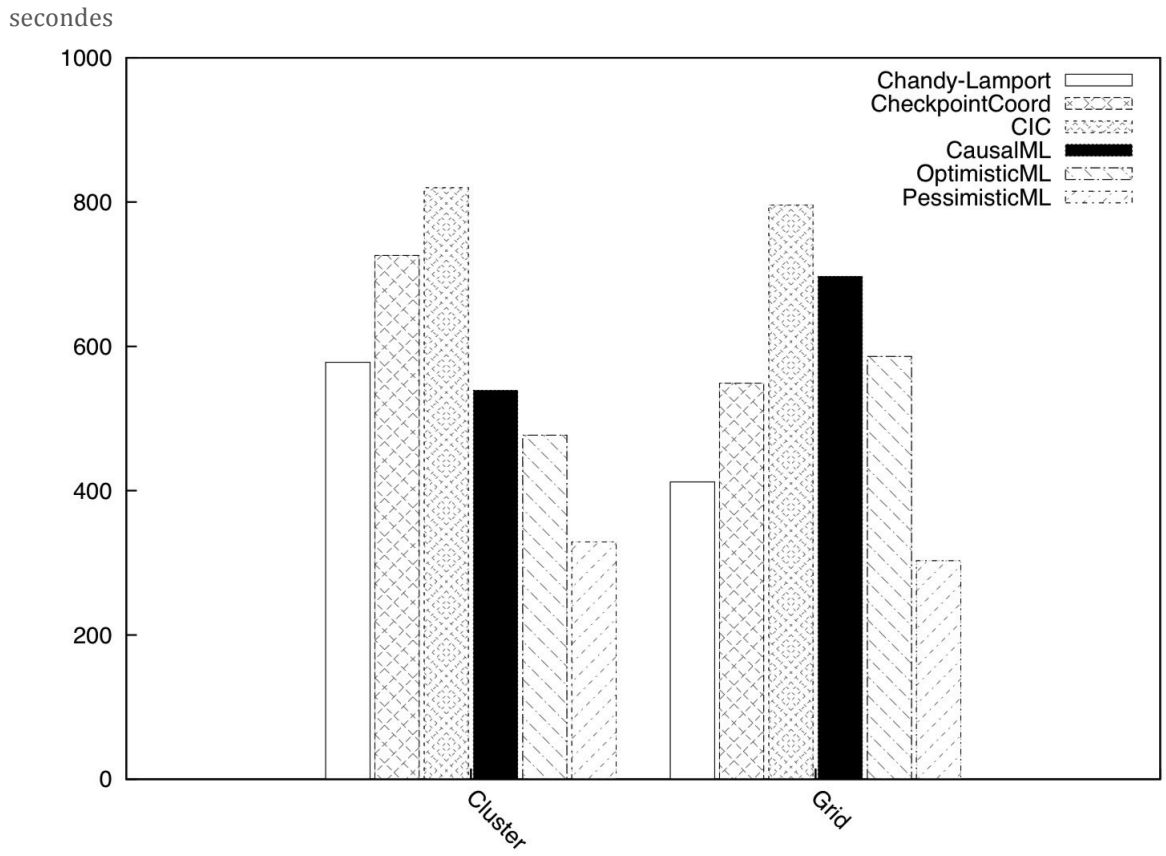

Figure 18 : Exécution sans faute de l'application à Jeton

Pour les techniques de journalisation, même si l'approche pessimiste souffre de la synchronisation durant le stockage en support stable, elle présente de meilleures performances par rapport aux deux autres techniques. En effet, la journalisation causale attache une quantité importante de données supplémentaires aux messages. De même, la technique optimiste ajoute les données supplémentaires enregistrées au niveau du processus émetteur (2) qui doivent être sauvegardées sur le support de stockage.

D'après les résultats obtenus sur les deux architectures, l'utilisation des protocoles coordonnés présente un avantage certain dans les réseaux en grille, car la coordination se limite aux processus appartenant à un même cluster, ceci avant que les leaders finalisent la procédure. Tandis que dans l'architecture à plat, si le nombre de processus augmente, les messages supplémentaires envoyés pour assurer la coordination globale augmentent considérablement et ralentissent l'application.

4.2.5. Nombre de processus à redémarrer

Dans le cas des mécanismes de points de reprise coordonnés, tous les processus reprennent leur exécution lors du recouvrement. Tandis que pour les techniques de journalisation, en général seul le processus fautif reprend.

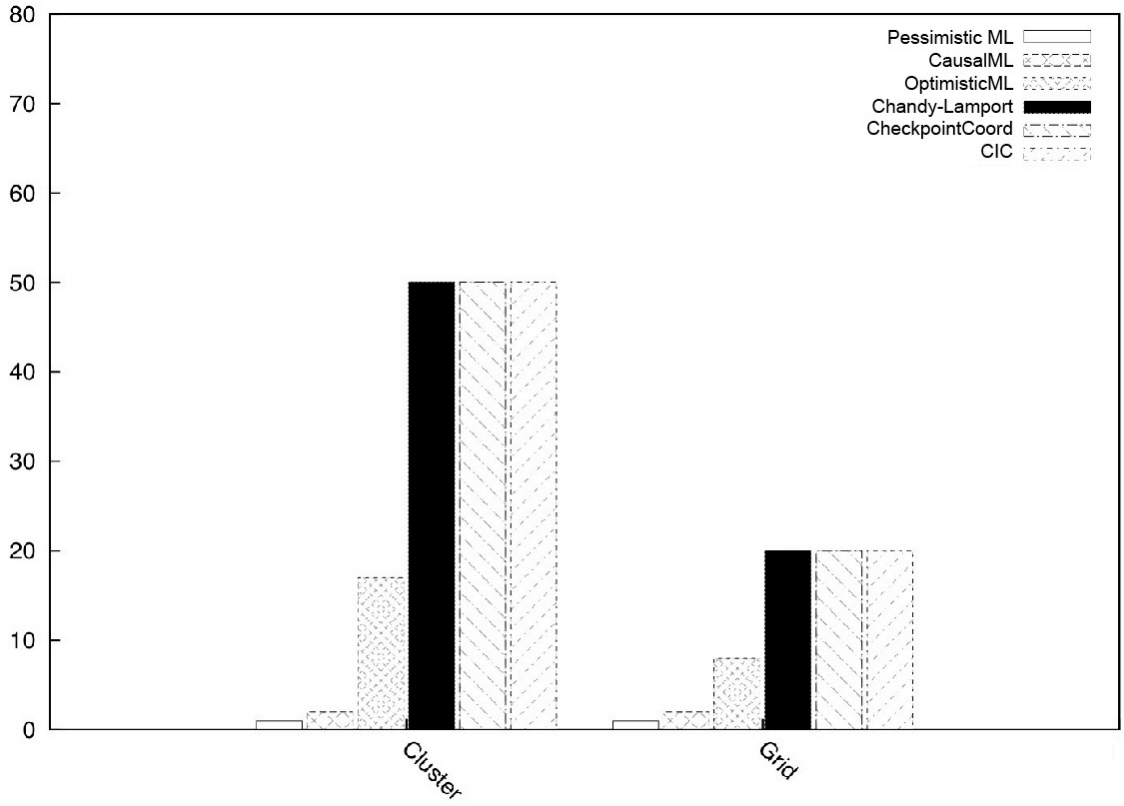

Figure 19 : Nombre de processus à redémarrer

Sur la Figure 19, nous avons injecté deux fautes après 300s et 600s de simulation, dans le cluster et dans la grille. Les 50 processus reprennent alors leur exécution pour les cas des protocoles coordonnés. Les versions hiérarchiques présentent un avantage certain au niveau de la grille, puisque seuls les processus du cluster contenant le processus reprennent leur exécution.

Les mécanismes de journalisation offrent un atout durant le recouvrement. En effet, l'approche pessimiste garantit la reprise du processus fautif uniquement. Pour les techniques de sauvegarde optimiste et causale, d'autres processus différents du processus fautif peuvent reprendre leur exécution à cause des dépendances durant l'enregistrement de leurs états locaux.

4.2.6. Performance durant le recouvrement

La durée du recouvrement dépend du nombre de processus à reprendre et du nombre de retours arrière. Dans les cas de la sauvegarde à base de points reprise coordonnés non bloquant ('Chandy-Lamport') et de la journalisation pessimiste ('PessimisticML'), le recouvrement est simplifié parce que le système effectue un retour arrière uniquement depuis le dernier point de reprise maintenu par le protocole.

En mode grille, le surcoût du recouvrement baisse relativement par rapport au mode cluster (Figure 20). En effet, si le processus fautif n'a pas de dépendance avec ceux des autres clusters, la propagation de la faute va se limiter au cluster contenant le processus fautif. Ainsi tous les nœuds de la grille n'exécuteront pas la procédure de recouvrement. Cependant, si les

communications inter-cluster sont intensives, ce surcoût augmentera dans le cas des approches de journalisation causale et optimiste, à cause de la latence inter-cluster élevé (100ms).

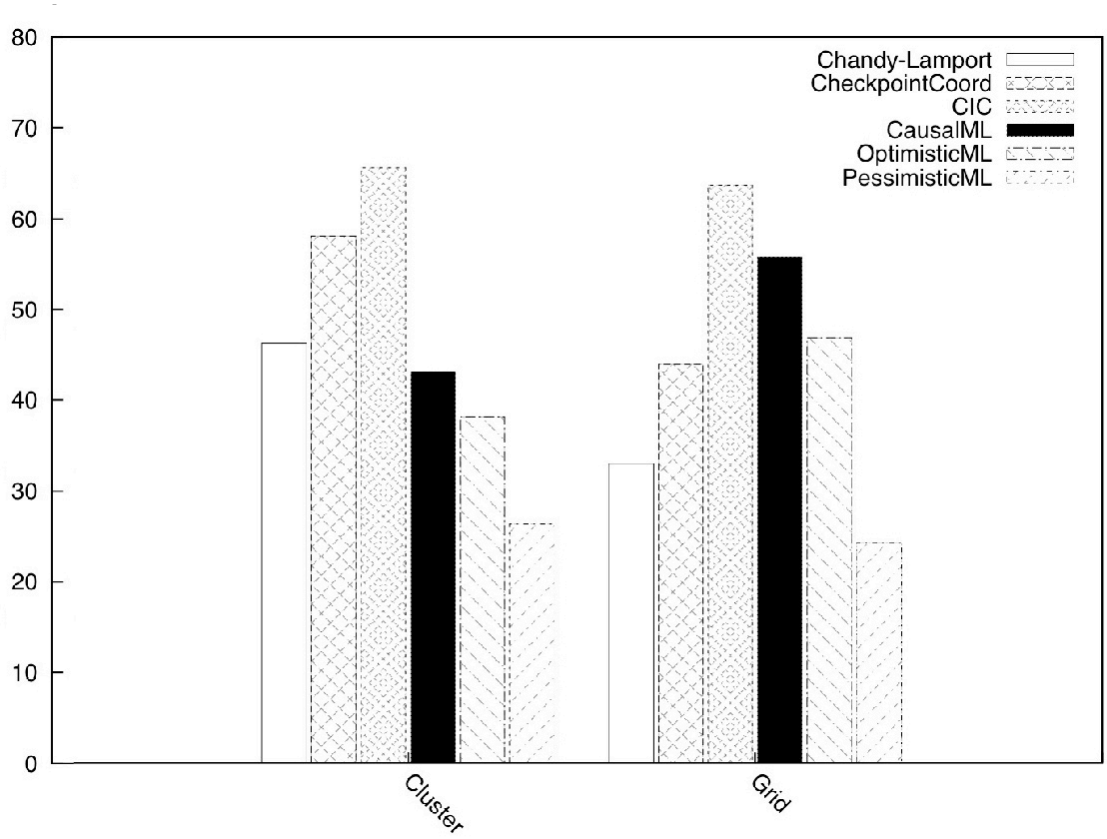

Figure 20 : Performance durant le recouvrement

4.2.7. Impact de l'hétérogénéité des clusters

Les évaluations de performance effectuées précédemment ont été faites dans une grille informatique dont les nœuds sont répartis uniformément au niveau des clusters, c'est à dire 5 nœuds par cluster. Nous avons essayé de changer les répartitions du nombre de processus pour voir leur impact sur l'exécution de l'application d'envoi de message en jeton (Figures 21 et 22).

Quatre modèles d'architecture ont été utilisés :

- *grid1G2P* : un gros cluster de 30 nœuds et deux petits clusters de 10 nœuds
- *grid1G3P* : un gros cluster de 35 nœuds et trois petits clusters de 5 nœuds
- *grid1G4P* : un gros cluster de 30 nœuds et quatre petits clusters de 5 nœuds
- *grid1G5P* : un gros cluster de 25 nœuds et cinq petits clusters de 5 nœuds

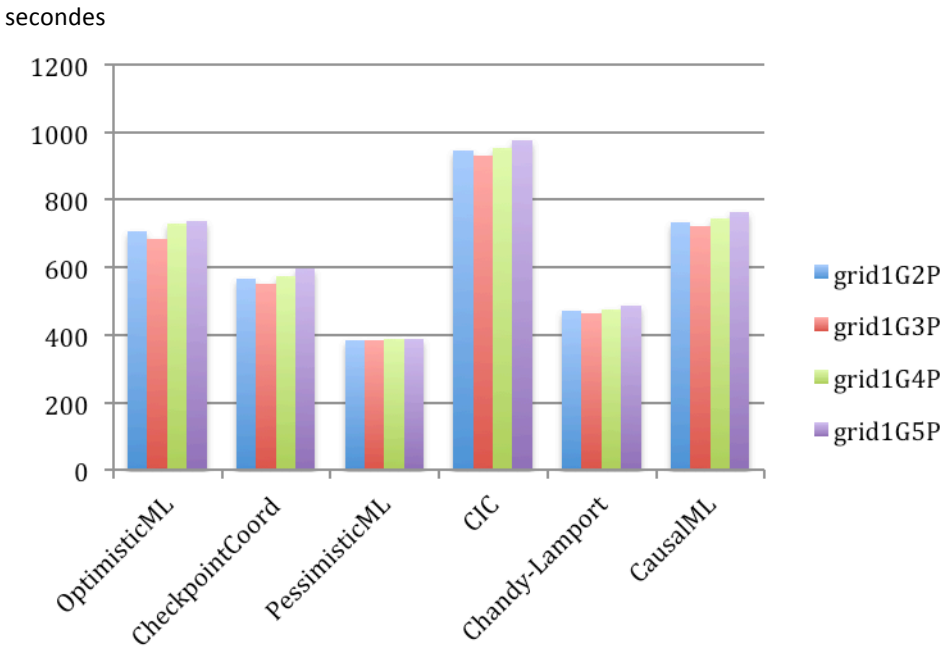

Figure 21 : Exécution sans faute

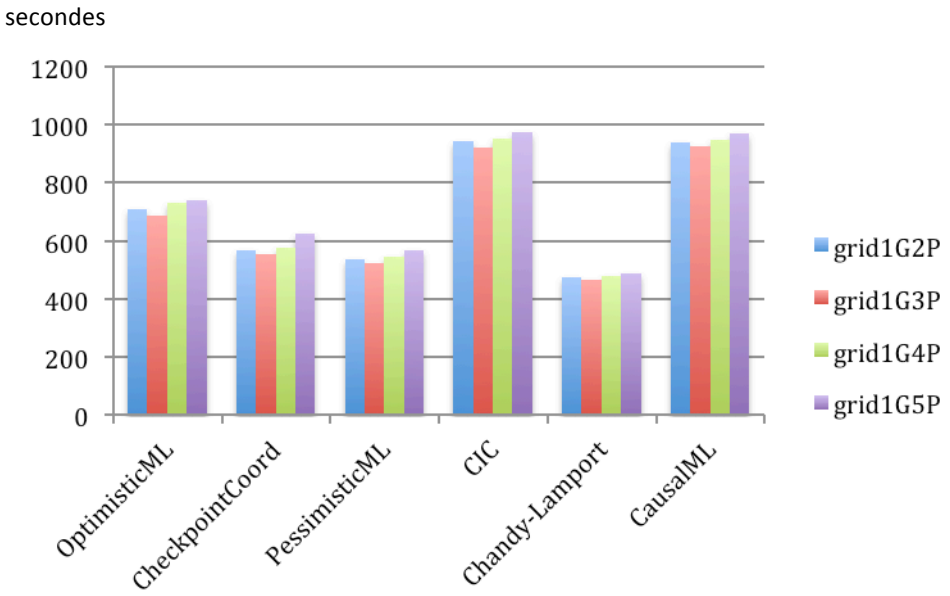

Figure 22 : Exécution avec une faute

Dans chaque cluster, nous avons configuré un processus par nœud. Les expériences ont montré que le temps de réponse augmente quand le nombre de clusters augmente en exécution normale et avec injection de fautes, sauf pour la configuration 1G2P. Nous remarquons aussi que la configuration 1G2P a un temps de réponse plus élevé que pour les autres configurations. Cela pourrait s'expliquer par la taille de ses clusters qui est le double de la taille des petits clusters des autres réseaux.

5. Conclusion

Nous avons comparé deux catégories des protocoles de sauvegarde : ceux à base de points de reprise synchronisés et ceux reposant sur la journalisation des messages. Les protocoles ont été évalués sur deux architectures différentes : une architecture plate modélisant un cluster et une architecture hiérarchique représentant une grille. Dans l'architecture en grille nous avons proposé des versions hiérarchiques d'algorithmes.

Les techniques de journalisation effectuent à la fois des points de reprise indépendants et enregistrent en même temps les messages sur support stable tandis que les points de reprise synchronisés font des sauvegardes de manière coordonnée sans traitement spécifique sur les messages applicatifs. Les expériences ont montré que les techniques de journalisation engendrent moins de surcoût durant l'exécution sans faute.

Les mécanismes de journalisation présentent aussi un avantage lors du recouvrement car seul le processus fautif reprend son exécution. Ils sont donc intrinsèquement mieux adaptés aux environnements à large échelle telles que les grilles informatiques. La version hiérarchique des protocoles de sauvegarde coordonnés présente de meilleures performances par rapport aux versions à plat car les fautes restent généralement confinées au sein des clusters.

Même si la solution dépend de l'environnement physique et des applications qui s'y exécutent, notre étude comparative nous a permis de sélectionner deux protocoles adaptés à notre architecture de grille :

- En intra-cluster : la technique de sauvegarde coordonnée non bloquante
- En inter-cluster : la journalisation pessimiste

Nous allons étudier plus en détails la composition ces deux protocoles dans une architecture en grille dans le chapitre suivant.

Chapitre IV: Composition d'algorithmes de point de reprise dans les architectures hiérarchiques

1. Introduction

Les résultats obtenus après l'étude comparative des protocoles de recouvrement arrière nous ont montré que la solution non bloquante de Chandy-Lamport exécutée dans les grilles donne de meilleures performances par rapport aux autres protocoles de sauvegarde à base de points de reprise. De même, l'approche pessimiste de la journalisation des messages est mieux adaptée aux architectures hiérarchiques car elle ne produit pas de messages orphelins.
Nous allons faire une étude approfondie de ces deux protocoles dans les grilles informatiques avec les différentes combinaisons possibles.

Dans un premier temps, nous allons faire une description détaillée des deux protocoles qui composent nos combinaisons. Ensuite, une présentation des compositions hiérarchiques sera effectuée. Enfin, nous allons faire une étude comparative de ces combinaisons de protocoles avec le simulateur OMNeT++.

2. Protocoles de sauvegarde

2.1. Point de reprise coordonné non bloquant de Chandy et Lamport

Dans [12], Chandy et Lamport explique leur algorithme par analogie à un groupe de photographes prenant une scène panoramique dynamique composée d'oiseaux migrateurs. Deux problèmes se posent pour effectuer cette prise de vue. Le premier problème est que la scène est tellement vaste qu'elle ne peut pas être capturée par un seul photographe. Et même si plusieurs photographes décident de prendre des clichés (***snapshot***), ils ne pourront pas être effectués au même instant. Le deuxième problème est lié à la nature : il est impossible d'arrêter le vol des oiseaux à un instant donné pour prendre la vue globale.

Cette scène dynamique d'oiseaux migrants s'apparente à l'exécution des processus dans un système réparti : les clichés ou snapshot représentent les états locaux des processus et la vue globale de la scène correspond à l'état global du système réparti. C'est ainsi qu'ils ont élaboré *l'algorithme du snapshot* communément appelé *protocole de points de reprise non bloquant de Chandy et Lamport*. L'objectif du protocole est de sauvegarder (capter) un état global passé du système qui soit cohérent (instantané). Pour ce faire, les auteurs définissent l'état global cohérent du système (cliché global instantané) comme étant la somme des états locaux des processus (clichés locaux) et des canaux de communication (Figure 23).

La solution de sauvegarde non bloquante de Chandy et Lamport est une variante des protocoles coordonnés à base de points de reprise. Le principal problème des systèmes répartis est l'absence d'horloge commune. Pour résoudre ce problème, il faudra concevoir un algorithme qui puisse enregistrer les états de processus et de leurs canaux de communication, de telle sorte que l'ensemble de ces états appartient à un état global cohérent [12]. En même temps, le calcul de l'état global du système ne doit pas empêcher l'exécution des processus et vice et versa l'exécution continue du système ne doit pas modifier le calcul de l'état global.

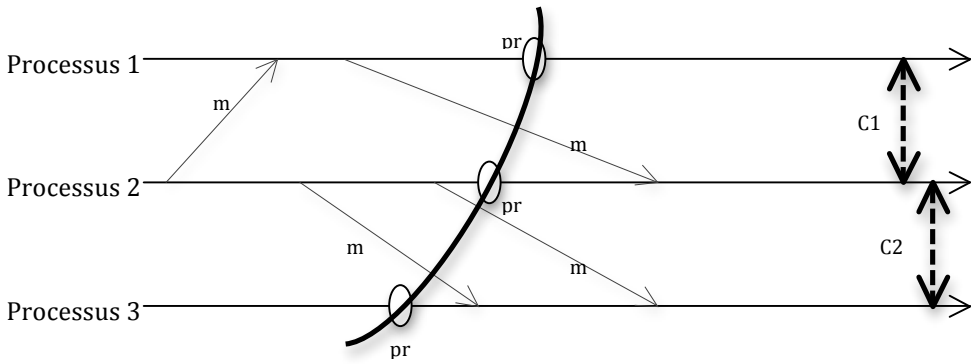

Figure 23 : pr1, pr2 et p3 sont les états locaux respectifs des processus P1, P2, P3
C1 et C2 représentent les états des canaux de communication entre [P1, P2] et [P2, P3]
C1= m3 ; C2=m2+m4

Le protocole fonctionne sous l'hypothèse des canaux FIFO, c'est-à-dire que sur un même canal tous les messages sont reçus dans l'ordre où ils ont été émis. Pour coordonner la capture des états locaux des processus, l'algorithme utilise des messages de contrôle ou **marqueurs**. Au cours de l'exécution du système, n'importe quel processus peut initier le calcul de l'état global du système.
L'objectif est d'enregistrer les états locaux de tous les messages en transit. Le principe est le suivant :
Soit *P1* l'initiateur du point de reprise :

a) *P1* envoie le marqueur « CtrlMsg » à tous les processus et sauvegarde son état local. (Figure 24)
b) À la réception de « CtrlMsg », deux cas peuvent se présenter :
   b.1) le processus reçoit le marqueur pour la première fois, alors il enregistre son état local et transfère le marqueur sur tous ses canaux sortants. L'état du canal relatif au processus émetteur est vide car, comme les canaux sont FIFO, le processus a reçu tous les messages émis avant la réception du marqueur, et les messages reçus après le marqueur ne font pas partie de l'état global (Figure 24).

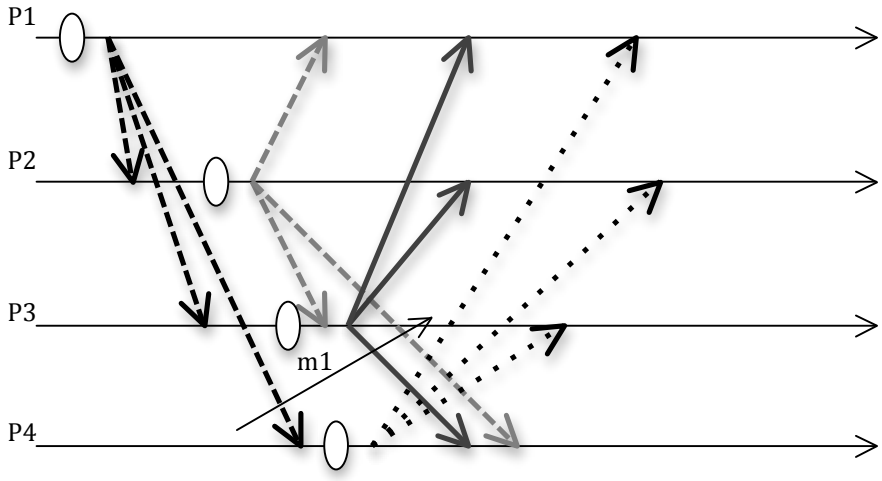

Figure 24 : Chaque processus transfère le marqueur vers tous les processus

b.2) le processus a déjà enregistré son état local, suite à une réception antérieure de marqueur, soit de sa propre initiative. L'état du canal entrant relatif au processus émetteur de ce marqueur est égal aux messages reçus entre l'enregistrement du dernier état local et la réception du marqueur. Par exemple, dans la Figure 25, le message m1 sera enregistré par P3 car il est reçu avant le marqueur envoyé par P4.

2.2.Journalisation pessimiste basée sur l'émetteur

Le protocole de journalisation pessimiste souffre de la synchronisation durant la sauvegarde des messages sur support stable. Pour éviter le surcoût induit par cette synchronisation, Johnson et al. [11] ont mis en œuvre un nouveau mécanisme de journalisation, dérivée de l'approche pessimiste. En effet, les auteurs voulaient réduire les surcoûts des mécanismes de tolérance aux fautes intégrés dans les systèmes répartis. Ces surcoûts sont causés par les trois éléments suivants :

- la procédure d'établissement des points de reprise,
- la journalisation des messages,
- le recouvrement en cas de défaillance.

Dans le cas de la journalisation pessimiste, les coûts supplémentaires du recouvrement et des points de reprise, sont faibles par rapport à ceux de la journalisation des messages. Si les pannes sont rares, le recouvrement n'aura pas un impact significatif sur les performances du système. De même la sauvegarde des points de reprise ne nécessite pas de coordination entre les processus. Compte tenu de ces faibles coûts, Johnson et al. se sont concentrés sur la réduction du coût de la journalisation synchronisée. En effet, quand un processus envoie un message à un autre, aussi bien l'émetteur que le récepteur possède une copie du message. Le moyen le plus rapide de conserver les messages envoyés dans le système, est de les sauvegarder au niveau l'émetteur, pour éviter la synchronisation avec le récepteur.
La technique de Johnson et al. [11] est basée sur l'émetteur du message. La plupart des techniques de journalisation présentée dans la littérature utilise un support de stockage stable comme journal pour sauvegarder les messages. Contrairement au '*Sender Based message logging* ' de Johnson et al., qui sauvegarde les messages dans la mémoire volatile de l'émetteur (Figure 25).

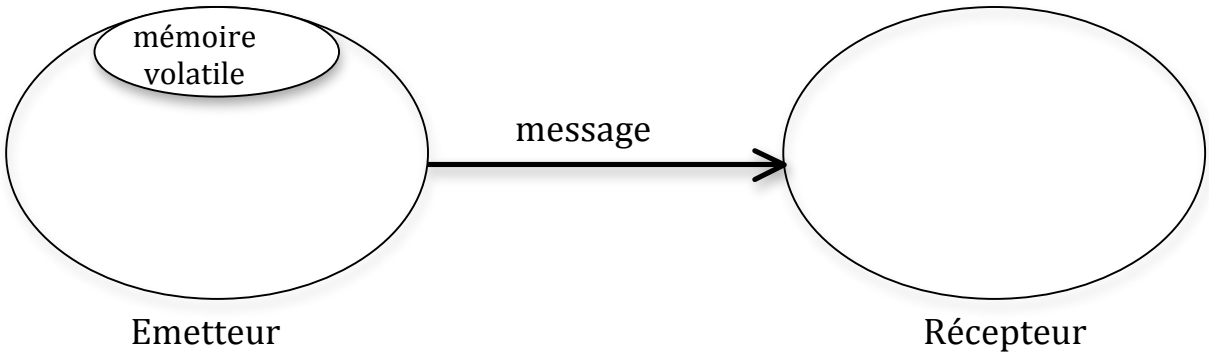

Figure 25 : Journalisation pessimiste basée sur l'émetteur [11]

Pour sauvegarder un message M envoyé par un processus E (comme émetteur) vers le

processus R (comme récepteur), l'algorithme déroule les étapes suivantes (Figure 26):

1) Le processus E envoie M au processus R et garde M dans sa mémoire volatile. Chaque processus possède un numéro de séquence (RSN: Receive Sequence Number) incrémenté à chaque réception de message.
2) À la réception de M, le processus R envoie un accusé de réception (ack) à E, après avoir mis à jour le numéro de séquence. Ce dernier est attaché à l'accusé de réception.
3) Le processus E ajoute le numéro de séquence (RSN: Receive Sequence Number) à M, et envoie un acquittement (act) au processus récepteur R.

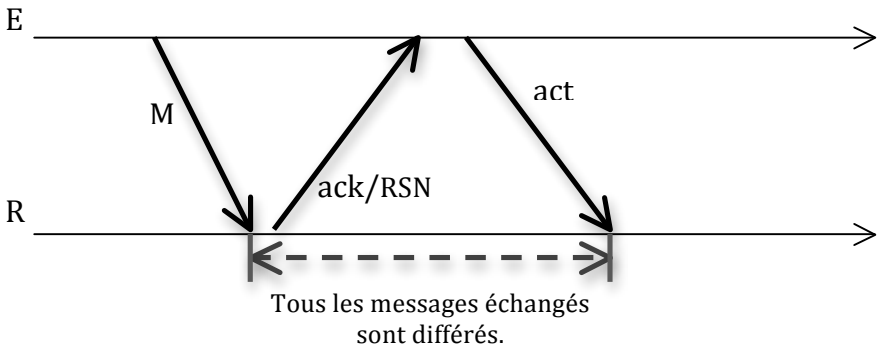

Figure 26 : Processus de sauvegarde par journalisation pessimiste [11]

Comme pour les autres techniques de journalisation, le recouvrement est simplifié : seul le processus fautif reprend son exécution. Il relance ainsi son exécution depuis son dernier point de reprise et rejoue les messages du journal. La journalisation pessimiste basée sur l'émetteur a plusieurs avantages :

a) Tous les messages reçus par le processus après le dernier point de reprise et avant la faute sont récupérés dans l'ordre à partir des journaux en mémoire des processus émetteur en utilisant les numéros de séquence. Le processus repris peut alors commencer son exécution et ne reçoit aucun nouveau message jusqu'à ce qu'il recouvre l'état précédant la faute.

b) L'effet domino est évité puisque seul le processus qui échoue est restauré au point de reprise le plus récent.

En revanche, les messages étant sauvegardés qu'en mémoire volatile, cette technique ne supporte pas des fautes multiples simultanées. Si plusieurs processus échouent, des messages à restaurer ne sont pas disponibles, le système averti alors l'utilisateur et interrompt le programme.

3. Description des combinaisons de protocoles

Notre approche s'inspire des solutions proposées par les auteurs de [25][26][27]. Nous allons faire une composition hiérarchique des deux protocoles de recouvrement arrière.

Nous considérons une architecture hiérarchique composée de clusters interconnectés par des liaisons longues distances. Nous distinguons deux types de messages :
- les messages inter-cluster qui circulent entre les clusters de la grille,
- les messages intra-cluster échangés entre les processus d'un même cluster.

Nous étudions, toutes les combinaisons des deux protocoles dont la configuration est résumée dans le Tableau 2.

Nom	**Protocole inter-cluster**	**Protocole intra-cluster**
« CLML : Chandy-Lamport Message Logging»	Chandy-Lamport	Journalisation pessimiste
« MLCL: Message Logging Chandy-Lamport »	Journalisation pessimiste	Chandy-Lamport
« MLML: Message Logging Message Logging »	Journalisation pessimiste	Journalisation pessimiste
« CLCL: Chandy-Lamport Chandy-Lamport »	Chandy-Lamport	Chandy-Lamport

Tableau 2 : Combinaisons entre la journalisation pessimiste et le protocole de Chandy-Lamport

3.1.« CLML : Chandy-Lamport Message Logging »

Dans cette composition hiérarchique [44], tous les messages échangés au sein de chaque cluster seront sauvegardés en utilisant le protocole de journalisation pessimiste basée sur l'émetteur [11] (Algorithme 4). Le protocole est identique à la méthode de Johnson décrite dans la section 2. Chaque processus est identifié par son identifiant unique, l'identifiant de son cluster, le numéro de séquence des messages (lignes 4-5-6).

Les résultats des expériences présentés dans le chapitre précédent nous ont montré que la technique de sauvegarde non bloquante, exécutée en mode grille, introduit un faible surcoût durant la sauvegarde des points de reprise et durant le recouvrement d'une faute. Pour sauvegarder l'état global cohérent de la grille, c'est le protocole à base de point de reprise non-bloquant de Chandy-Lamport est donc utilisé en respectant toujours l'architecture hiérarchique.

Les leaders jouent le rôle de coordonnateur au sein de chaque cluster. Ainsi pour sauvegarder l'état global cohérent de la grille, voici les différentes étapes (Figure 27):
Étape 1 : un processus initiateur envoie un marqueur à son leader et sauvegarde son état local
Étape 2 : cette étape ne fait intervenir que les leaders des clusters. À la réception du marqueur, le leader le transfère aux autres leaders de la grille informatique.
Étape 3 : cette étape est identique à celle de l'exécution du protocole de Chandy-Lamport dans une architecture plate. Chaque leader envoie le marqueur à tous les processus de son cluster. Les processus sauvegardent leur état selon les règles définies dans l'algorithme.
Étape 4 : chaque leader, après avoir sauvegardé son cluster envoie au leader un message de fin de sauvegarde. La fin du calcul de l'état global cohérent de la grille est marquée par la réception de tous les messages de fin par chaque leader.

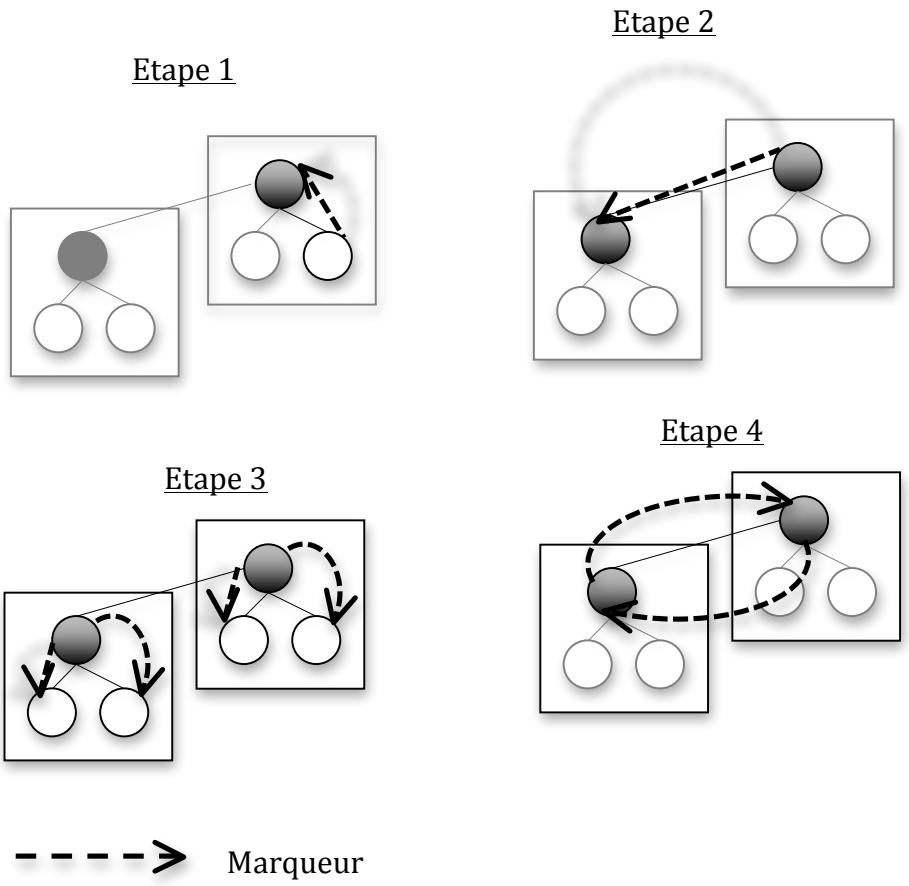

Figure 27: Protocole de point de reprise non bloquant hiérarchique

Durant l'exécution de l'algorithme, nous remarquons que le nombre de marqueurs diminue durant l'étape 3. Quand le protocole s'exécute dans une architecture plate, chaque processus envoie le marqueur à tous ses voisins dont l'étendue est celle du système entier. Alors que dans une architecture hiérarchique le nombre de marqueur envoyé est limité au cluster du processus. Dans la section sur l'évaluation de performance, nous analysons en détail l'impact de notre architecture hiérarchique sur le nombre de messages échangés au cours de l'exécution du protocole.

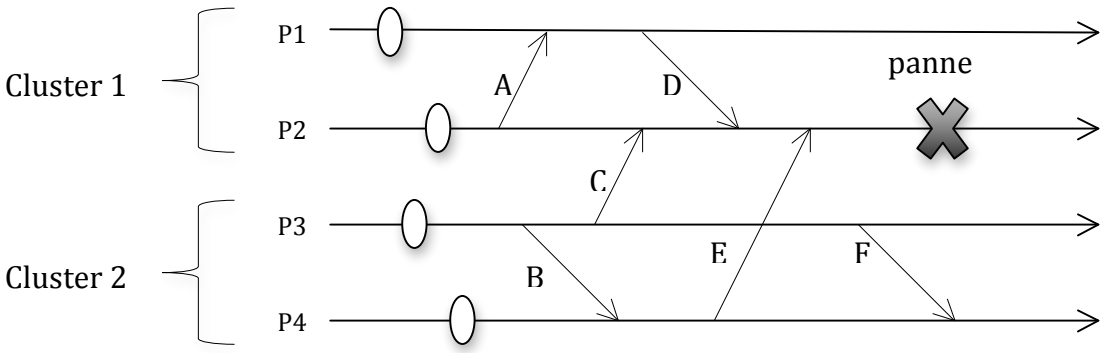

Figure 28 : Recouvrement avec le protocole « CLML »

En cas de panne, le processus fautif ainsi que tous les autres processus appartenant à ce cluster redémarrent à partir du dernier point de reprise et rejouent les messages reçus entre le dernier calcul d'état global et l'apparition de la faute.
Un problème se pose lors du recouvrement. Ce dernier doit garantir que tous les messages soient rejoués dans le même ordre où ils ont été reçus. La Figure 28 montre un exemple d'exécution dans un système comportant deux clusters (Cluster 1 et Cluster 2). Le Cluster 1 contient deux processus *P1* et *P2* et les processus *P3* et *P4* appartiennent au Cluster 2. *A, B, C*, *D, E* et *F* représentent les messages échangés par les processus et qui peuvent être classés selon les deux catégories précitées :

- *A, B, D* et *F* sont des messages intra-cluster
- *C* et *E* sont des messages inter-cluster

Selon les règles de la composition hiérarchique des deux protocoles, seuls les messages *A, B, D* et *F* sont sauvegardés par leurs émetteurs ; tandis que les messages *C* et *E* ne sont pas sauvegardés. En cas de panne du processus *P2, P1* et *P2* doivent reprendre leur exécution à partir de leur dernier point de reprise, et les messages *A, D, C* et *E* doivent être rejoués. Le recouvrement ne sera pas correct puisque les messages inter-cluster *C* et *E* seront perdus.
Pour garantir un recouvrement correct du système, nous sauvegardons au niveau des récepteurs les déterminants des messages inter-cluster. Nous allons voir dans la suite l'impact de la sauvegarde de ces déterminants sur les performances du système en le comparant aux autres combinaisons de protocoles.

```
    Variables
1.  etatProci          (* etat processus i*)
2.  ecIni              (* etat canal d'entrée processus i*)
3.  ecOuti             (* etat canal de sortie processus i*)
4.  rsn                (* numéro de séquence *)
5.  idProc             (* identifiant du processus *)
6.  idTeam             (* identifiant du cluster *)
7.  etatNoeud.chkpt    (* indique si le processus a effectué un point de reprise *)

8.  handleMessage (cMessage *msg)
9.          si (msg=CtrlMsg)
10.         alors
11.           si ((processus=initiateur) et (etatNoeud.chkpt=false))
12.             alors
13.                 makeCheckpoint()
14.                 envoyer msg au leader
15.             sinon
16.                 si ((processus<>leader) et (etatNoeud.chkpt=false))
17.                 alors
18.                   makeCheckpoint()
19.                   envoyer msg vers les canaux de sortie
20.           si ((processus=leader) et (etatNoeud.chkpt=false))
21.                 alors
22.                   makeCheckpoint()
23.                   diffuser msg dans le cluster
24.
25.         si ((msg<>CtrlMsg) et (idTeam(Sender)=idTeam (Receiver))
26.         alors
```

```
27.                 bloquer l'exécution
28.                 rsn=rsn+1
29.                 envoyer (ack, rsn, emetteur)
30.
31.         si ((msg=ack) et (processus=emetteur))
32.         alors
33.                 ajouter rsn au message
34.                 envoyer (acquittement, emetteur)
35.                 débloquer communication
36.         si ((msg<>CtrlMsg) et (idTeam(Sender)<>idTeam (Receiver))
37.         alors
38.                 enregistrer déterminant du message
```

Algorithme 4 : Protocole CLML

3.2.« MLCL : Message Logging Chandy-Lamport »

La combinaison précédente impose la journalisation de tous les messages intra-cluster, qui sont en général plus élevés que le nombre de messages échangés entre les clusters. Pour réduire le surcoût de la sauvegarde de ces messages, on se propose d'étudier une autre combinaison d'algorithme appelé « MLCL ». Il s'agit ici d'utiliser la journalisation pessimiste entre les clusters, et le protocole de point de reprise coordonné de Chandy-Lamport au sein des clusters (Figure 30) [28][30].
Chaque message échangé entre deux processus appartenant à deux clusters différents est sauvegardé dans la mémoire de l'émetteur en respectant la technique de sauvegarde pessimiste fondée sur l'émetteur. Pour garantir le déterminisme d'exécution, les déterminants des messages feront partie des éléments stockés au niveau du processus émetteur. Ce déterminant contient la date d'émission du message, sa date de réception, et son numéro de séquence. Ainsi durant le recouvrement, ces informations pourront être utilisées pour rejouer les messages dans l'ordre où ils ont été émis au cours de l'exécution normale.
On notera des différences entre la méthode classique de journalisation pessimiste basée sur l'émetteur et celle utilisée dans notre composition algorithmique. En effet, selon l'architecture hiérarchique définie dans le chapitre précédent, tous les échanges inter-cluster passent par l'intermédiaire des leaders qui représentent les coordonnateurs des clusters.

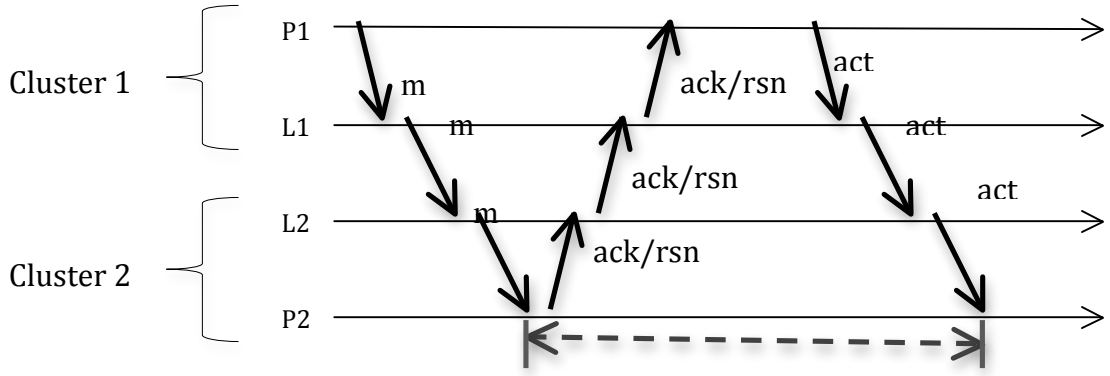

Figure 29: Journalisation pessimiste hiérarchique fondée sur l'émetteur

La Figure 29 montre la procédure de sauvegarde du message ***m*** échangé entre les processus *P1* et *P2* situés sur deux clusters différents. Les processus *P1* et *P2* appartiennent respectivement au Cluster 1 et Cluster 2, *L1* et *L2* étant leur leader. Voici les différentes étapes pour envoyer le message *m* de *P1* vers *P2*:

1) *P1* envoie m à son leader *L1*
2) *L1* transfère le message *m* à *L2*, leader du processus *P2*
3) *L2* envoie *m* au destinataire du message *P2*
4) À la réception de *m*, *P2* met à jour le numéro de séquence du message, l'attache à l'accusé de réception, et envoie ce dernier son leader *L2*
5) Le leader *L2* transfère l'accusé de réception à son homologue *L1*
6) Enfin *L1* envoie l'accusé de réception à l'émetteur *P1*
7) Le processus émetteur *P1* ajoute le numéro de séquence au message *m* sauvegardé dans la mémoire
8) *m* étant sauvegardé en mémoire avec son numéro, le *P1* peut maintenant envoyer un acquittement au récepteur pour qu'il continue à recevoir des messages. Durant toute la phase d'envoi du message *m*, les autres messages reçus par les processus *P1* et *P2* sont différés.

Au sein des clusters, les états des processus seront sauvegardés en utilisant la technique non bloquante de Chandy-Lamport. L'intérêt de l'utilisation de cette solution non bloquante en intra-cluster est que pendant son exécution le protocole de journalisation s'exécute en même temps sur les messages inter-cluster. Ces messages sauvegardés peuvent ne pas faire partie du calcul de l'état global du système, puisqu'ils seront rejoués en cas de panne.

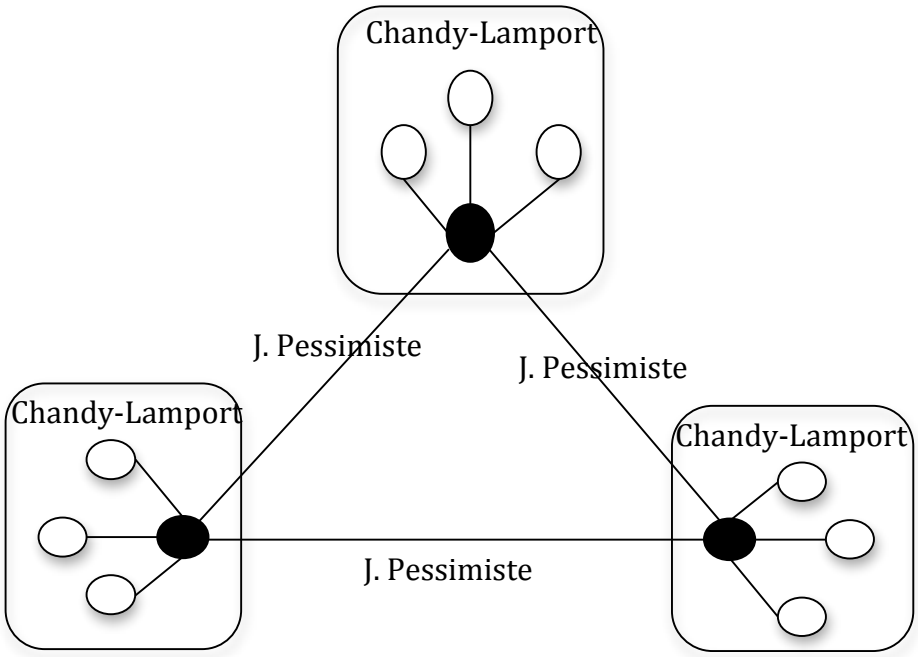

Figure 30 : Composition hiérarchique

En cas de défaillance, seuls les processus du cluster contenant le processus fautif redémarrent à partir du dernier point de reprise et rejouent les messages échangés après la dernière sauvegarde. Dans [28] les auteurs montrent que le recouvrement ne sera pas correct à cause des messages intra-cluster qui ne sont pas sauvegardés. En effet, si des messages sont échangés entre processus d'un même cluster avant une panne, ces messages seront tout simplement perdus. C'est pour cette raison, qu'il propose la sauvegarde de ces messages dans une structure appelée *TeamTable* stockée aussi dans la mémoire des processus et contenant les déterminants des messages. Les auteurs de [30] proposent aussi la sauvegarde des déterminants de tous les messages, c'est-à-dire aussi bien les messages inter-cluster et intra-cluster.

3.3.« MLML : Message Logging Message Logging »

À la différence des deux combinaisons décrites précédemment, celle-ci fait intervenir le même protocole sur les deux niveaux de hiérarchie de la grille. En effet, tous les messages circulant dans la grille, qu'ils soient échangés entre les clusters ou à l'intérieur d'un cluster, seront sauvegardés en utilisant la technique de journalisation pessimiste basée sur l'émetteur.
La seule différence notée sur l'exécution du protocole de journalisation est la présence des leaders des clusters qui sont les intermédiaires entre les processus appartenant à des clusters différents.
Pendant que les messages sont sauvés par leur émetteur, des points de reprise indépendants seront effectués par les processus. L'intérêt de ces points de reprise se trouve lors du

recouvrement : le nombre de messages rejoués est réduit. Ainsi lors d'une défaillance, le processus fautif redémarre au dernier point de reprise et rejoue les messages d'avant panne.
Les leaders jouent un rôle important durant le recouvrement. En effet durant l'exécution normale de l'application, comme les messages inter-cluster transitent via les leaders, leurs déterminants sont sauvegardés au niveau du leader. Ainsi en cas de défaillance d'un nœud, tous les messages sont recouvrables : chaque émetteur envoie au processus fautif les messages reçus par celui-ci, les leaders utilisent les déterminants mémorisés pour rejouer les messages inter-cluster.

3.4.« CLCL : Chandy-Lamport Chandy-Lamport »

Cette combinaison aussi s'apparente à la précédente de par sa composition : le même protocole de sauvegarde coordonnée non-bloquante est utilisé dans la grille. La procédure de sauvegarde se déroule en deux phases. Dans une première phase, le protocole s'exécute à l'intérieur des clusters. Durant la seconde phase, seuls les leaders participent à la coordination.
Si une panne se produit, tous les processus redémarrent depuis le dernier point de reprise le plus récent.

4. Évaluation de performance

Pour connaître la combinaison de protocole la mieux adaptée à notre architecture hiérarchique, nous avons implémenté les 4 configurations avec le simulateur OMNeT ++. La simulation se déroule pendant 1000s. Nous avons configuré 50 processus répartis uniformément dans la grille composée de cinq clusters possédant chacun 5 nœuds, dont la latence pour les communications intra-cluster est égale à 0,1 ms et celle des messages inter-cluster est égale à 100 ms.

4.1.Modèle d'application

Quatre modèles d'application sont utilisés pour évaluer les quatre combinaisons de protocoles: les deux applications décrites dans le Chapitre III, à savoir l'application de diffusion de messages ('broadcast') et l'application d'envoi de messages en jeton ('token'). Les deux autres modèles seront dérivés des deux premiers modèles. Il s'agit de 'nBroadcast' et 'nToken'.

L'application 'nBroadcast' fait plusieurs diffusions en même temps. En effet, à un instant donné, cinq processus aléatoires diffusent un message dans la grille. Ainsi au cours de l'exécution normale, une série de 5 diffusions de messages est effectuée tous les 30 secondes.
De même l'application 'nToken' lance cinq jetons en même temps. Durant toute la durée de la simulation, cinq jetons circulent en permanence dans la grille informatique. En effet, les jetons sont sélectionnés au hasard à l'aide d'une fonction aléatoire. Ces jetons peuvent traverser les clusters, mais aussi peuvent se limiter à un seul cluster.
Ces deux nouvelles applications, nous permettent d'évaluer les protocoles en présence d'une charge importante en termes de messages.

4.2. Exécution sans faute

La Figure 31 montre le temps d'exécution sans faute des quatre applications. Au cours de leur exécution, nous avons simulé les quatre combinaisons de protocoles décrites précédemment. Tous les protocoles implémentés comportent un mécanisme de sauvegarde par point de reprise. Les protocoles « CLCL », « CLML », « MLCL » comme leurs noms l'indiquent effectuent des points de reprise coordonnés non-bloquants, tandis que le protocole « MLML » établit des points de reprise indépendants au cours de son exécution. Ces points de reprise sont effectués toutes les 180 secondes au cours de l'exécution normale des applications.

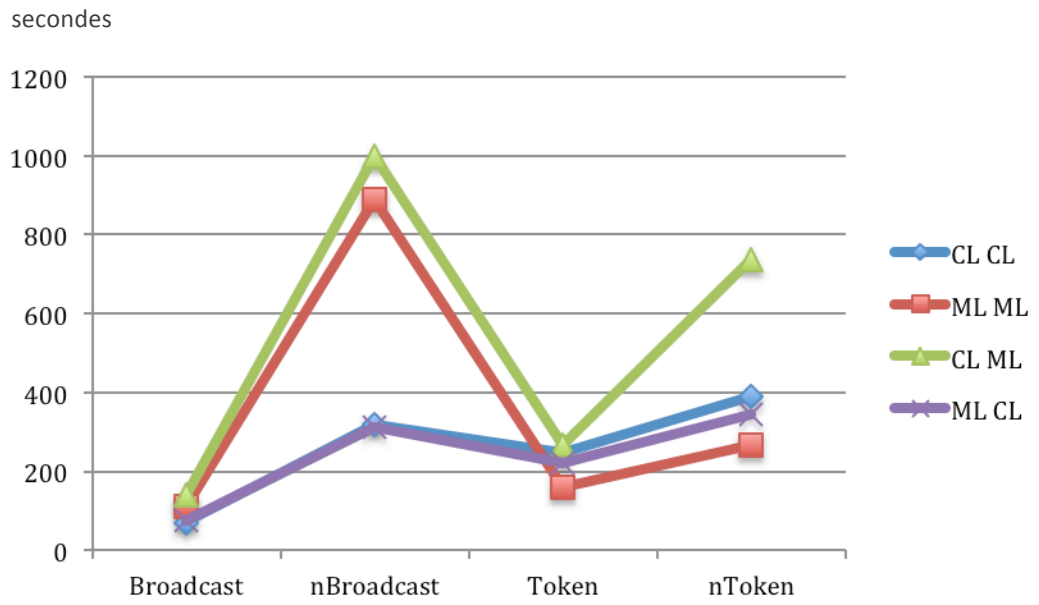

Figure 31 : Temps de réponse sans faute

Dans un premier temps, nous remarquons un temps de réponse assez élevé pour le protocole « CLML ». En effet, comme les messages intra-cluster sont plus nombreux que les messages inter-cluster, la sauvegarde synchronisée de tous les messages intra-cluster introduit un surcoût sur le temps de réponse, quelle que soit l'application en cours d'exécution.
Les applications « nBroadcast » et « nToken » nous ont permis d'avoir des variations significatives sur les temps de réponse.
Pour les types d'applications en diffusion, nous remarquons que le temps de réponse est faible pour la combinaison de protocoles « MLCL ». Pour ces applications, le facteur prédominant est le nombre important de message circulant au niveau de chaque cluster, le nombre de message inter-cluster se limitant uniquement aux messages échangés entre les leaders. Donc pour une diffusion, le nombre de messages inter cluster envoyés est égal à 5 ; ce qui explique le faible surcoût induit par la journalisation pessimiste qui dans le cas de la combinaison « MLCL » est appliquée uniquement aux messages inter-cluster. Nous remarquons aussi de bonnes performances pour la combinaison « CLCL ». Ceci est principalement dû à la parallélisation des instances de l'algorithme CL au niveau de chaque cluster.
Pour les applications à jetons, l'exécution de la combinaison de protocole « MLML » introduit le plus faible surcoût sur le temps de réponse. Ceci s'explique par le nombre réduit de messages qui circulent lors de l'exécution. En effet pour ce type d'application, un seul message circule à un instant donné dans le cas de « Token » et cinq messages dans le cas de

« nToken ». Donc la journalisation synchronisée n'aura pas un impact important sur le temps de réponse.

4.3. Exécution avec injection de fautes

Nous étudions dans cette section l'impact des fautes sur le temps de réponses des applications. Dans cette étude, nous avons injecté une faute toutes les 100 secondes soit 10 fautes en tout.
Les histogrammes de la Figure 32 confirment les résultats obtenus au niveau de la Figure 31. Le protocole « MLCL » est mieux adapté aux applications de diffusion, tandis que pour les applications en jeton la combinaison « MLML » est plus appropriée.

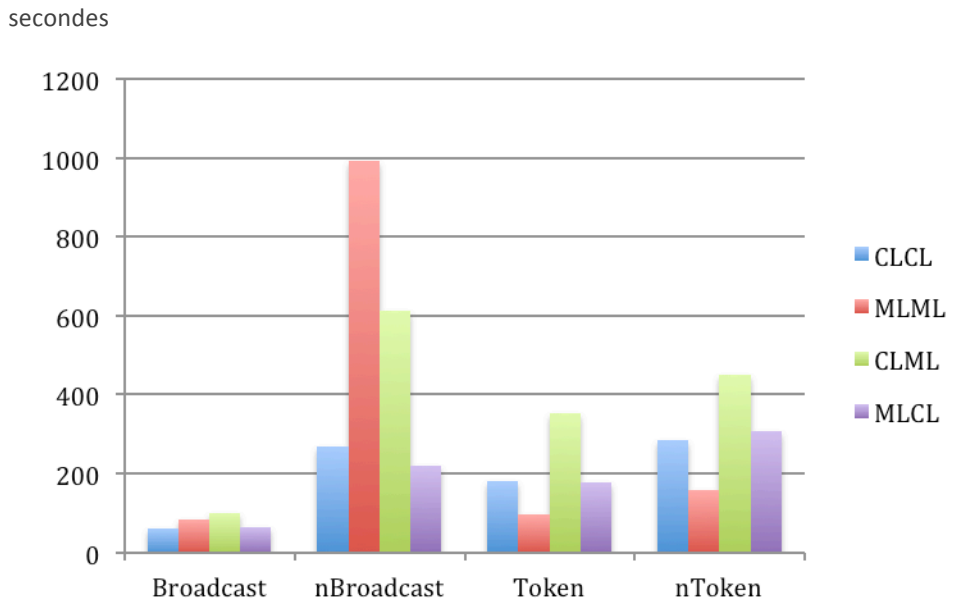

Figure 32 : Temps de réponse avec injection de 10 fautes

Le surcoût induit dans le cas des deux combinaisons de protocoles « MLML » et « MLCL » n'est pas élevé à cause de la clusterisation des techniques de recouvrement arrière. Dans le cas du mécanisme de journalisation pessimiste, seul le processus fautif reprend son exécution. Et pour la sauvegarde par points de reprise, seuls les processus du cluster contenant le processus fautif reprennent leur exécution.

4.4. Impact de la clusterisation sur le nombre de marqueurs

Comme nous l'avons vu dans les sections précédentes, le protocole de point de reprise coordonné de Chandy-Lamport utilise des marqueurs pour coordonner la sauvegarde des états locaux des processus du système. Ce marqueur doit atteindre l'ensemble des processus de la grille informatique pour que l'état global soit cohérent.
Notre grille est composée de cinq clusters. Dans chaque cluster, nous avons configuré dix processus. Les applications implémentées vont s'exécuter ainsi sur 50 processus.
Nous avons voulu savoir le nombre de marqueur envoyé par le protocole dans le cas des deux types d'architecture, à savoir, une grille informatique et une architecture plate composée uniquement d'un cluster de 50 machines.

	Architecture hiérarchique	Architecture plate
Nombre de marqueur	5 + 10*9*5 = **455**	50 * 49 = **2450**

Tableau 3 : Nombre de marqueurs

Le tableau 3 montre le nombre de marqueurs envoyés dans les deux architectures. En supposant que, dans chaque cluster, les processus sont tous reliés aux autres, le nombre de marqueur envoyé juste pour atteindre l'ensemble des processus du système est cinq fois plus élevé en mode hiérarchique qu'en mode à plat. Ceci explique les bonnes performances globales de la combinaison « CLCL ».

5. Conclusion

Dans ce chapitre, nous avons fait une étude approfondie du protocole de journalisation pessimiste fondée sur l'émetteur (ML) et de la sauvegarde à base de point de reprise coordonné non bloquant de Chandy et Lamport (CL).
Afin de les utiliser dans notre architecture hiérarchique en grille, nous avons implémenté les différentes combinaisons possibles de ces protocoles sur notre architecture hiérarchique à savoir: « CLML », « MLCL », « MLML » et « CLCL ». Lorsqu'il est utilisé au niveau de chaque cluster, le protocole de sauvegarde coordonné non bloquant introduit un faible surcoût car il profite alors pleinement de la clusterisation en diminuant le nombre de messages de contrôle et d'une parallélisation importante, chaque cluster effectuant sa sauvegarde en parallèle.
Les mécanismes de journalisation présentent un avantage certain lorsqu'ils sont utilisés avec des applications qui ne sont pas fortement communicantes, comme les applications d'envoi de message en jeton.
Notre étude a permis de sélectionner les deux combinaisons de protocoles suivantes :

- « MLCL » avec une journalisation pessimiste pour les messages inter-cluster et l'utilisation du protocole coordonné de Chandy-Lamport au niveau de chaque cluster.
- « MLML » où une journalisation pessimiste est utilisée pour sauvegarder les messages inter-cluster et intra-cluster

La première composition « MLCL » s'adapte aux applications de type diffusion de messages, tandis que le protocole « MLML » est le plus performant pour les applications en jeton.
Dans le chapitre suivant, nous allons proposer donc un algorithme adaptatif qui associe ces deux combinaisons de protocoles.

Chapitre V : Un nouveau protocole hiérarchique adaptatif par points de reprise et journalisation

1. Introduction

Nous allons présenter un nouveau protocole de recouvrement arrière adapté aux architectures hiérarchiques notamment les grilles informatiques.
Ce protocole est basé sur deux algorithmes très connus dans la littérature et utilisé dans les systèmes distribués : la journalisation pessimiste fondée sur l'émetteur de Johnson [11] et le protocole de sauvegarde par point de reprise coordonné de Chandy et Lamport [12].
Le nouveau protocole s'adapte au type d'application qui s'exécute dans le système. Notre étude a porté sur deux types d'application : les applications de diffusion de messages et d'envoi de messages en jeton.
Dans un premier temps, nous allons faire une description détaillée du nouveau protocole. Ensuite, une présentation des applications expérimentées sera effectuée. Enfin, nous présenterons les résultats des expériences effectuées dans le simulateur OMNeT++.

2. Description du protocole

Dans le chapitre précédent, nous avons fait une étude comparative de toutes les combinaisons possibles entre les deux mécanismes de recouvrement arrière, l'algorithme de Chandy-Lamport et la journalisation pessimiste. Finalement deux combinaisons sont sorties de cette étude. Une première composition « MLML » qui peut être qualifiée d'homogène puisqu'elle utilise uniquement la journalisation pessimiste pour la sauvegarde de tous les messages échangés dans la grille informatique. La seconde composition « MLCL » est hybride car elle est composée d'un protocole de journalisation appliquée sur les messages inter-cluster et le protocole de point de reprise utilisé au sein des clusters.

Les résultats des expériences menées dans les chapitres précédents nous ont montré que la combinaison « MLML » est adaptée aux applications peu communicantes. Nous avons implémenté deux applications de ce type reposant sur des envois de jetons :

- « token » : un jeton circule en permanence dans la grille
- « nToken » : cinq jetons circulent en permanence dans la grille

En effet, pour ces applications en jeton, le nombre de messages échangés par les processus à un instant donné n'est pas élevé (maximum cinq pour les tests effectués). Ainsi le surcoût induit par la journalisation des messages reste faible.

Pour ce qui est de la composition « MLCL », elle donne de meilleures performances pour les applications de diffusion échangeant un grand nombre de messages. Même si elles sont des applications fortement communicantes, seuls les messages inter-cluster seront sauvegardés avec le mécanisme de journalisation pessimiste.

Ainsi le nouveau protocole va combiner les deux compositions hiérarchiques en s'adaptant dynamiquement au modèle de communication des applications : au cours de l'exécution, si l'application fait des diffusions alors la combinaison « MLCL » va s'exécuter et si l'application est de type jeton c'est la combinaison « MLML » qui s'exécutera.

Quelle que soit la combinaison exécutée, tous les messages inter-cluster seront sauvegardés avec technique de Jonhson et al.[11].

3. Implémentation du protocole

3.1. Principe

L'objectif du nouveau protocole adaptatif est de réduire le surcoût de la journalisation pessimiste durant l'exécution sans faute, mais aussi d'éviter une procédure de recouvrement trop longue qui pourrait ralentir le fonctionnement du système en cas de défaillance. Pour ce faire, on suppose dans un premier temps que l'application en cours d'exécution génère peu de messages intra-cluster sous un seuil de fréquence maximale (LowComRate). Dans ce cas, la combinaison de protocole « MLML » est appliquée. Ainsi tous les messages qui circulent dans la grille et leurs déterminants sont sauvegardés en utilisant la technique de journalisation pessimiste fondée sur l'émetteur. Pour minimiser le nombre de messages à rejouer en cas de panne, des points de reprise indépendants sont effectués par les processus dés que le nombre de messages envoyés atteint un seuil (MaxLogMsg) fixé à 50.
Pour que le protocole adaptatif bascule vers la combinaison « MLCL », il faut que le nombre de messages échangés par les processus appartenant à un même cluster atteigne une valeur seuil suffisamment élevée pour dégrader les performances des applications. Les expériences ont permis de fixer le seuil à dix messages par seconde (10msg/s). Ainsi si la fréquence des communications intra-cluster dépasse ce seuil, la journalisation de ces messages est interrompue. Et dans ce cas, pour sauvegarder les états des processus, c'est le protocole de point de reprise coordonné non-bloquant qui sera utilisé. Ce dernier s'exécutera à une fréquence CkptFreq fixée à 120 secondes.

3.2. Détails de l'implémentation

L'algorithme 5 représente le protocole adaptatif en exécution sans faute. Chaque processus possède un identifiant (ligne 5) et l'identifiant de son cluster (ligne 6). Au début de l'exécution, tous les messages sont sauvegardés en appliquant la technique de journalisation pessimiste fondée sur l'émetteur. Ainsi le message est enregistré dans la mémoire de l'émetteur (ligne 8). De même, le déterminant de chaque message est enregistré au niveau du récepteur. Le déterminant est composé de : la date d'émission, la date de réception et le numéro de séquence du message. Pour effectuer un point de reprise (ligne 57), le protocole enregistre sur support stable les éléments suivants :

- L'état du processus (ligne 59)
- L'état des canaux entrants (ligne 60)
- L'état des canaux sortants (ligne 61)
- La date de sauvegarde du point de reprise (ligne 62)
- L'identifiant du dernier protocole utilisé pour sauvegarder les états des processus (ligne 63)

Pendant que le protocole de journalisation s'exécute, les messages échangés à l'intérieur des clusters sont comptabilisés (ligne 9). En effet, avant qu'un message ne soit sauvegardé, le protocole vérifie la fréquence d'envoi des messages à l'intérieur des clusters (ligne 20). Si elle est supérieure au seuil, la sauvegarde des messages intra-cluster est suspendue. Ce seuil de fréquence a été déterminé sur la base des expériences effectuées avec le simulateur OMNeT++. La Figure 33 montre le temps de réponse des combinaisons de protocoles « MLML » et « MLCL » en fonction de la fréquence des messages. La courbe du « MLML » peut être divisée en deux parties :

- La première partie concerne les fréquences de messages inférieures ou égales à 10msg/s
- La seconde partie concerne les temps de réponse pour les fréquences supérieures 10msg/s

À partir d'une fréquence égale à 10 msg/s, le temps de réponse augmente au fur et à mesure que la fréquence des messages intra-cluster augmente. Contrairement au protocole « MLCL », la fréquence des messages n'a pas d'impact sur les temps de réponse.

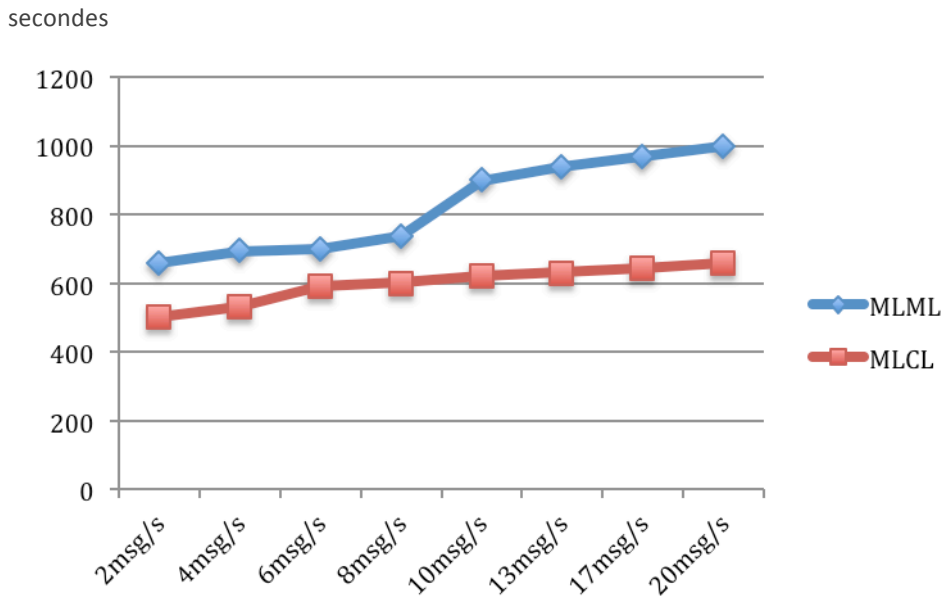

Figure 33: Variation de la fréquence des messages en fonction du temps de réponse

Le protocole de journalisation pessimiste n'est pas adapté aux environnements ayant une fréquence de messages élevée. Ainsi après suspension de la journalisation des messages intra-cluster, c'est le protocole à base de point de reprise coordonné non bloquant qui s'exécute à l'intérieur des clusters (ligne 35). Il sera lancé tous les cent vingt secondes (120s). Le protocole adaptatif continue de vérifier la fréquence des messages intra-cluster en temps réel.
En résumé, le protocole adaptatif est une composition hiérarchique entre d'une part la journalisation pessimiste appliquée aux messages inter-cluster et le protocole de points de reprise non-bloquant de Chandy-Lamport utilisé en alternance avec la journalisation pessimiste selon la fréquence des messages intra-cluster.

Nous allons voir dans la section suivante la procédure de recouvrement du protocole adaptatif.

```
      Variables
1.    ecIni                (* etat canal d'entrée processus i*)
2.    ecOuti               (* etat canal de sortie processus i*)
3.    rsn                  (* numéro de séquence *)
4.    idProc               (* identifiant du processus *)
5.    idTeam               (* identifiant du cluster *)
6.    etatNoeud.chkpt      (* indique si le processus a effectué un point de reprise *)
7.    ArrayMsg             (* contient les messages envoyés par le processus *)
8.    volatilMem           (* contient les messages non encore sauvegardés *)
9.    nbMsgTeam_i          (* nombre de messages échangés dans le cluster i *)
10.   dateEmission         (* date d'émission du message *)
11.   dateReception        (* date de réception du message *)
12.   frequence            (* fréquence des messages intra-cluster *)
13.   timing               (* temps écoulé après une sauvegarde d'état de processus*)

14.   Send (msg) (* envoi de msg *)
15.    ajouter msg à la mémoire volatile
16.    envoyer msg

17.   handleMessage (cMessage *msg) (* réception de msg *)
18.   si (idTeam(Sender)=idTeam (Receiver) et msg<> ack et msg<>CtrlMsg)
19.   alors
20.    nbMsgTeam_i= nbMsgTeam_i +1
21.    si (frequence < = LowComRate)
22.      alors
23.        bloquer l'exécution
24.        rsn=rsn+1
25.        envoyer (ack)
26.        enregistrer déterminant msg (idProc, idTeam, dateEmission, dateReception, rsn)

27.            (* réception de ack *)
28.   si (msg=ack)
29.    alors
30.      ajouter rsn au message
31.      enregistrer msg à la mémoire
32.      envoyer (acquittement)
33.      débloquer communication

34.          (* initier une sauvegarde coordonnée si la fréquence > 10msg/s *)
35.   si ((fréquence > LowComRate) et (timing=CkptFreq))
36.    alors
37.      choisir un processus aléatoire
38.      envoyer CtrlMsg au leader

39.                                  (* à la réception d'un marqueur *)
40.   si (msg= CtrlMsg)
41.   alors
42.    si ((processus=initiateur) et (etatNoeud.chkpt==false))
43.      alors
44.        makeCheckpoint()
```

```
45.      envoyer msg au leader
46.    sinon
47.      si ((processus<>leader) et (etatNoeud.chkpt=false)
48.        alors
49.        makeCheckpoint()
50.        envoyer msg vers les canaux de sortie
51.  si ((processus=leader) et (etatNoeud.chkpt=false))
52.   alors
53.      makeCheckpoint()
         diffuser msg dans le cluster

54.                          (* initier des points de reprise locaux *)
55.  si (frequence < = LowComRate) et taille (ArrayMsg)>MaxLogMsg alors
56.    makeCheckpoint()

57.  makeCheckpoint()
58.   Envoyer sur support stable
59.      * état processus
60.      * ecIni
61.      * ecOuti
62.      * dateChkpt
63.      * idPtc
64.   etatNoeud.chkpt=false
```

Algorithme 5 : Protocole adaptatif

3.3. Recouvrement

En cas de défaillance, deux cas peuvent se présenter. La panne peut survenir :

1) après une sauvegarde de points de reprise indépendants. C'est-à-dire le dernier protocole qui s'est exécuté au sein cluster est le mécanisme de journalisation pessimiste.
2) après exécution du protocole de point de reprise coordonné.

Pour connaître le dernier protocole exécuté au sein des clusters, l'algorithme vérifie la valeur de la variable ***idPtc*** détenue par le leader du cluster qui indique le protocole intra-cluster. Sa valeur par défaut est égale à « ML ». C'est-à-dire le dernier protocole exécuté au sein des clusters est la journalisation pessimiste. Une valeur égale à « CL » correspond au protocole de Chandy-Lamport.

Dans le cas « MLML », à la suite d'une panne, pour rétablir les états d'avant panne, le dernier point de reprise est exécuté, et les messages reçus après ce point de reprise sont rejoués.
Ici tous les messages reçus par les processus après leurs derniers points de reprise sont recouvrables. Pour ce qui est des messages inter-clusters, leurs contenus ainsi que leurs déterminants sont toujours disponibles au niveau de leurs émetteurs et récepteurs; de même pour les messages intra-cluster. Si on suppose que les applications sont déterministes, il faut garantir un recouvrement « correct » de l'exécution et délivrer à nouveau les messages dans le même ordre. Pour cela, le protocole utilise les numéros de séquences et les déterminants sauvegardés au cours de l'exécution. Ces derniers permettent de rejouer les messages dans l'ordre où ils ont été émis. Qu'il soit un message inter-cluster ou un message intra-cluster, un message dont la date d'émission est inférieure à un autre, ne sera jamais rejoué avant celui qui possède une date plus petite.

Dans notre architecture hiérarchique, toutes les activités au sein des clusters sont coordonnées par leurs leaders. Au cours du recouvrement (Algorithme 6) les leaders récupèrent les déterminants des messages envoyés au processus ou cluster fautif (ligne 24). Puis, ils envoient ces déterminants aux processus devant faire des retours arrière (ligne 25).
Pour le cas « MLCL », tous les processus du cluster contenant le processus fautif reprennent leur exécution au dernier point de reprise enregistré lors de la dernière sauvegarde coordonnée (lignes 18-28). Les messages reçus après le dernier point de reprise sont aussi rejoués. Leurs ordres d'émission et de réception sont sauvegardés dans les déterminants enregistrés lors de l'exécution sans faute.

L'intérêt de la sauvegarde coordonnée réside dans la manière de rétablir l'exécution d'avant panne. En effet, tous les processus ne seront pas obligés de faire plusieurs retours arrière. D'ailleurs le protocole ne maintient sur support que le point de reprise le plus récent.

```
     Variables
1.   etatProci          (* etat processus i*)
2.   ecIni              (* etat canal d'entrée processus i*)
3.   ecOuti             (* etat canal de sortie processus i*)
4.   rsn                (* numéro de séquence *)
5.   idProc             (* identifiant du processus *)
6.   idTeam             (* identifiant du cluster *)
7.   etatNoeud.chkpt    (* indique si le processus a effectué un point de reprise *)
8.   ArrayMsg           (* contient les messages envoyés par le processus *)
9.   nbMsgTeam_i        (* nombre de message échangés dans le cluster i *)
     idPtc              (* identifiant protocole*)

10.  handleMessage (cMessage *msg) (* réception de msg *)
     ** A la réception d'une demande de recouvrement **
           (* panne du processus_i *)
11.  si (msg = FaultMsg)
12.  alors
13.    si (processus=leader_i)
14.    alors
15.      si (idPtc = CL)
16.        alors
17.          envoyer FaultMsg aux autres leaders
18.          envoyer FaultMsg aux processus de son cluster
19.        sinon
20.      si (idPtc = ML)
21.          envoyer FaultMsg aux autres leaders
22.    sinon
           (* A la réception de FaultMsg par un leader *)
23.        si (processus≠leader_i)
24.            envoyer determinants à leader_i
25.            envoyer FaultMsg aux processus dont le determinant contient p_i

           (* A la réception de FaultMsg par un processus *)
26.        si (processus)
27.        alors
28.                exécuter etatProc_i
29.                rejouer les messages contenus dans sa mémoire
30.                rejouer les messages définis dans les déterminants
31.
32.  sinon
33.        si (msg=determinant)
34.        alors
35.                envoyer determinants à p_i
43.
44.
```

Algorithme 6 : Procédure de recouvrement

4. Évaluation des performances

Nous allons présenter les résultats des expériences effectuées avec le simulateur OMNeT++. Nous avons configuré une grille contenant 50 processus répartis uniformément sur 5 clusters. Pour l'évaluer, nous avons mis en place une application « DiffToken » qui alterne des envois de jetons et des diffusions de messages.

Au début de l'exécution, l'application envoie une succession de 10 jetons et fait 10 diffusions. Pour une diffusion donnée, le nombre de messages envoyé est égal à 50. Dès que les jetons ont atteint leur destination, une autre vague de 10 diffusions est lancée, et ainsi de suite.

4.1. Comparaison entre le protocole adaptif, « MLML » et « MLCL »

Pour mesurer les performances de notre protocole, nous l'avons comparé aux deux meilleures combinaisons statiques de protocoles. La Figure 34 montre le surcoût du recouvrement de l'application « DiffToken » en injectant des fautes sur une échelle de 1 à 10.

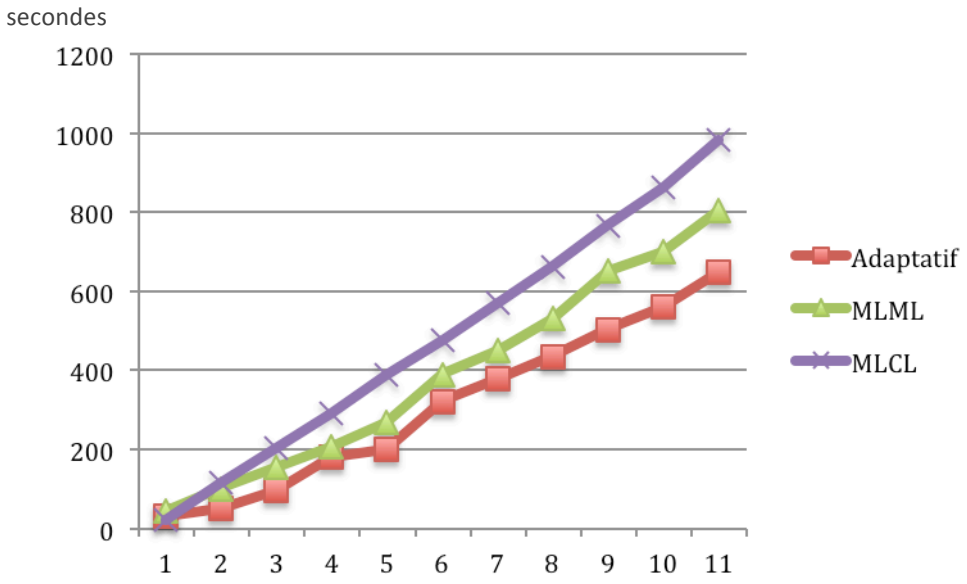

Figure 34: Temps de réponse avec injection de fautes

Nous remarquons que le protocole adaptatif engendre moins de surcoût que les deux autres protocoles. Ce qui est normal puisse qu'il combine les avantages des deux protocoles :

- Tous les états des processus sont recouvrables, puisque les messages sont disponibles au niveau des émetteurs et leurs déterminants sont sauvegardés au niveau de leurs récepteurs.
- Il n'y peu de risque d'effet domino pour le protocole adaptatif, grâce au protocole de point de reprise coordonné exécuté au sein des clusters. Même si les derniers états locaux sauvegardés sont indépendants, il existe un état global cohérent à partir duquel les processus peuvent reprendre leur exécution.

4.2. Taille des messages

La Figure 35 montre le surcoût du recouvrement avec une variation de la taille des messages (en octets). Le surcoût est la différence entre le temps de réponse sans faute et le temps de réponse avec faute. La courbe inférieure représente les valeurs durant l'exécution d'une faute et la deuxième courbe supérieure montre les valeurs pour une exécution de 10 fautes.

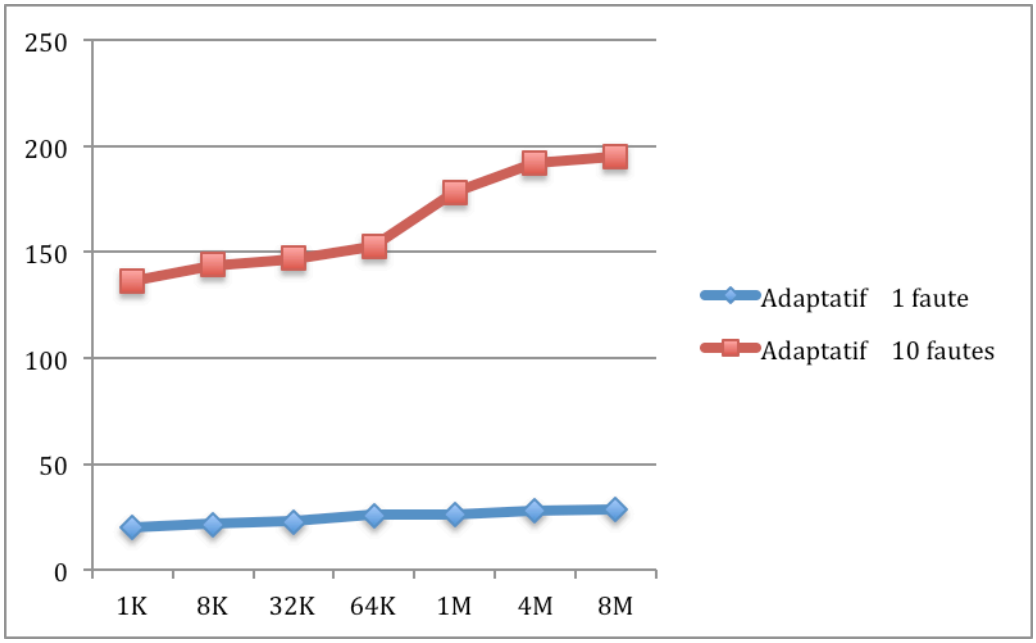

Figure 35: Surcoût du recouvrement avec variation de la taille des messages

Le protocole adaptatif est essentiellement basé sur la journalisation pessimiste. Cette dernière souvent comparée à la journalisation synchronisée, nécessite des échanges de messages tout au long de la procédure de sauvegarde. D'une part, les messages occupent de l'espace au niveau de leur émetteur. Et d'autre part, la taille de ces messages entraîne un surcoût sur les communications. Même compte tenu de tous ces points cités, la journalisation pessimiste fondée sur l'émetteur présente un avantage certain par rapport aux autres techniques de journalisation au niveau du recouvrement. En effet, les données nécessaires à la réémission des messages, sont directement disponibles au niveau de l'émetteur, ce dernier ne sollicitera pas en temps en réel le support stable pour récupérer les messages. Ces derniers sont disponibles en local. Ainsi les coûts d'accès au support ont tendance à disparaître dans l'approche pessimiste.

Au niveau de la Figure 35, dans un premier temps, nous remarquons qu'au fur et à mesure que la taille des messages augmente, le surcoût du recouvrement n'augmente pas de manière significative, pour une injection d'une faute. Nous observons aussi que la taille des messages n'a pas un impact significatif sur le temps de recouvrement.

En revanche, pour un nombre de fautes égal à 10, le surcoût du recouvrement augmente légèrement à partir d'une taille de messages égale à 1 mégaoctet. Cette augmentation pourrait s'expliquer par la latence des communications.

4.3.Nombre de processus à redémarrer

L'objectif principal d'un protocole de recouvrement arrière est de garantir un recouvrement correct en cas de panne. Mais aussi, il faut que la procédure de recouvrement ne dégrade pas les performances des applications qui s'exécutent dans la grille.
Notre protocole adaptatif étant une combinaison de deux catégories de mécanismes de recouvrement arrière, nous avons mesuré le nombre de processus ayant repris leur exécution après une défaillance. Ce nombre dépend :

- Des dépendances entre le processus fautif et les processus de la grille : en cas de défaillance, les processus qui avaient envoyé un message au processus fautif et reçu après le dernier point de reprise reprennent leur exécution dans le cas de la configuration MLCL
- De la fréquence des messages : le nouveau protocole s'adapte à la fréquence des messages intra-cluster (cf section 3.2).
- L'instant d'apparition de la panne : une panne peut survenir après une sauvegarde coordonnée ou après des points de reprise indépendants (cf section 3.2).

Ainsi pour connaître le nombre de processus repris après une défaillance, nous avons fait varier les fréquences d'envoi des jetons et des diffusions dans l'application DiffToken. intra-cluster et inter-cluster en injectant une faute en milieu de simulation, après 500s d'exécution.

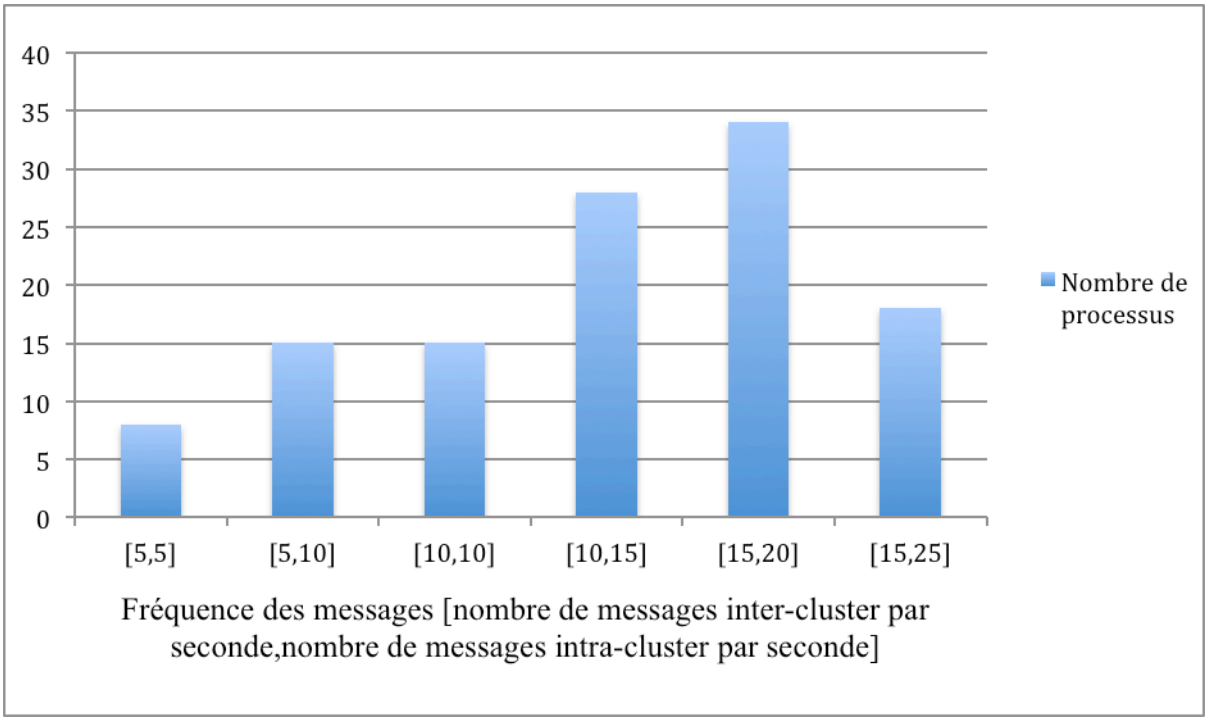

Figure 36: Nombre de processus repris avec variation de la taille des messages

Les résultats des expériences effectuées, nous avons remarqué que le nombre de processus à redémarrer est un paramètre non maitrisable. En effet, il est de difficile de connaître à l'avance les types d'applications qui vont s'exécuter dans la grille, et surtout les dépendances entre les processus. Sur la Figure 36, on peut voir que même pour une fréquence de 5 msg/s, le nombre de processus à redémarrer peut être supérieur au nombre total de processus contenus.

4.4. Impact de la journalisation

Le protocole adaptatif applique la journalisation pessimiste tantôt sur les messages inter-cluster, tantôt sur les messages intra-cluster. Pour connaître l'impact de la journalisation sur notre protocole de recouvrement, nous allons le comparer à la combinaison de protocole «CLCL ». Cette dernière utilise uniquement la technique de sauvegarde non-bloquant de Chandy-Lamport basée sur les points de reprise.

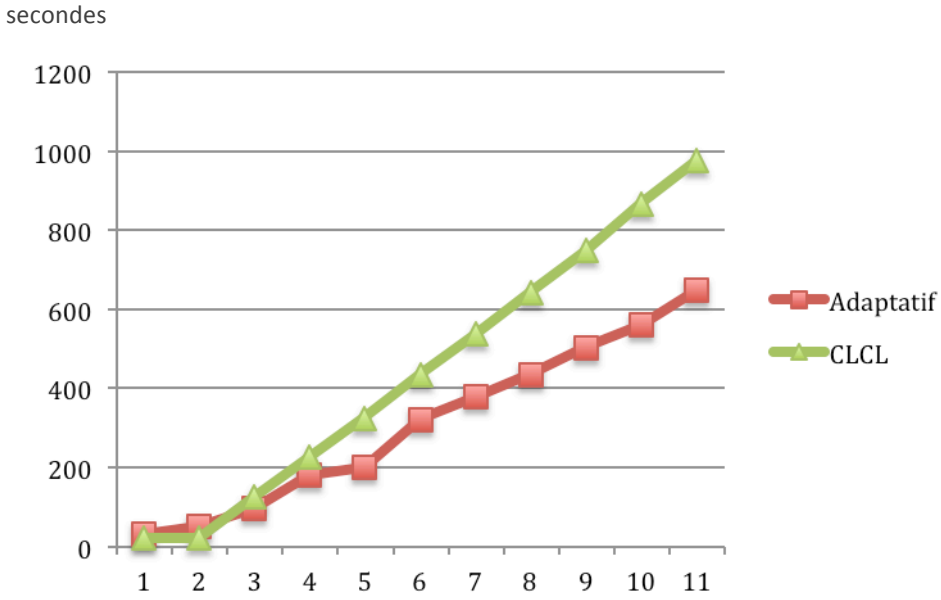

Figure 37: Temps de réponse de l'application « DiffToken » avec le protocole adaptatif et la composition «CLCL»

La Figure 37 montre le temps de réponse de l'application « DiffToken » durant l'exécution du protocole adaptatif et « CLCL » en injectant uniformément des fautes sur une échelle de 0 à 10 fautes.
En exécution normale, sans défaillance, le protocole « CLCL » donne de meilleures performances. En effet, par rapport au protocole adaptif, il n'y a pas de journalisation de messages sur support stable pour la combinaison « CLCL ». Donc le surcoût induit par le mécanisme de journalisation au niveau du protocole adaptatif n'existe pas pour le protocole « CLCL ».
En cas de panne, la procédure de recouvrement du protocole « CLCL » est relativement simple, puisque les processus du cluster contenant le processus fautif redémarrent au dernier point de reprise sauvegardé par le protocole. Le nombre de processus qui reprend leur exécution dans le cas du protocole adaptatif peut être inférieur à celui du protocole « CLCL ».
Pour le protocole adaptatif, le dernier algorithme exécuté en intra-cluster peut être des points de reprise indépendants si il s'agit de la combinaison MLCL. Dans ce cas, le nombre de processus repris peut être inférieur au nombre total de processus contenus dans le cluster. Ceci explique les meilleures performances du protocole adaptatif lorsque le nombre de fautes augmente.

Conclusion

Dans ce chapitre, nous avons présenté un nouveau protocole de recouvrement arrière adaptatif pour les architectures hiérarchiques comme les grilles informatiques. Ce nouveau protocole est basé sur deux mécanismes de base à savoir :

- la journalisation pessimiste fondée sur l'émetteur,
- la sauvegarde coordonnée non bloquant de Chandy-Lamport.

La première technique est appliquée aux messages inter-clusters. Et au sein les deux mécanismes sont alternés par rapport à la fréquence des messages intra-clusters. Nous allons présenter aussi les différentes phases d'exécution de notre protocole dans une architecture en grille dont le principal acteur est le leader de chaque cluster. Ce dernier coordonne tous les échanges au sein des clusters. Ces leaders jouent un rôle fondamental durant le recouvrement, car ils récupèrent tous les déterminants qui seront utilisés par les processus défaillants. Ceci garantit un recouvrement correct, limité aux processus défaillants.

CHAPITRE 6

Conclusion

Les grilles informatiques sont devenues des plateformes de partage de ressources très utilisées par l'industrie et le monde académique. On trouve des grilles opérationnelles de quelques dizaines à plusieurs centaines de milliers de serveurs, fonctionnant comme support privilégié en calcul intensif, dans le traitement de grandes masses de données distribuées ou en tant qu'infrastructures supports d'applications ASP Une grille est une infrastructure matérielle et logicielle qui fournit un accès normalisé et sécurisé à ses ressources. Elle est composée de grappes nœuds (des clusters) de calcul ou de stockage, reliées par des réseaux de type WAN.

Cependant, le nombre élevé de nœuds entraine une augmentation importante des défaillances. La tolérance aux fautes devient alors un élément indispensable pour assurer la continuité du service. Le recouvrement arrière et la réplication sont les deux solutions pour assurer la tolérance aux fautes dans les grilles informatiques. Dans le cadre d'applications de calcul intensif, la duplication des traitements est trop coûteuse. Dans cette thèse, nous nous sommes donc concentrés sur les techniques de recouvrement arrière.

Il existe deux grandes familles de mécanismes de recouvrement arrière : la sauvegarde synchronisée par point de reprise et les protocoles de journalisation. Généralement ces algorithmes sont utilisés dans des architectures plates, comme les clusters et ne prennent pas en compte la topologie hiérarchique de la grille. Dans un premier temps, nous avons implémenté et comparé les performances des principaux protocoles sur les deux types d'architectures (un cluster et une grille). Les résultats ont montré que les mécanismes de journalisation présentent un avantage pour les environnements à large échelle parce qu'en cas de faute seul le processus fautif reprend son exécution. Cependant, ils sont particulièrement coûteux lorsque les applications échangent beaucoup de messages. Les algorithmes de sauvegarde coordonnée supportent des applications fortement communicantes. En revanche, ils passent mal à l'échelle car en cas de défaillance, tous les processus peuvent amenées à redémarrer.

Suite à cette étude, nous avons sélectionné le meilleur algorithme dans chaque catégorie de protocole de recouvrement arrière : la journalisation pessimiste basé sur l'émetteur [11] et la solution non bloquante à base de point de reprise de Chandy-Lamport [12]. Nous avons proposé une nouvelle approche hiérarchique exploitant la topologie de la grille en combinant des algorithmes différents au niveau inter et intra-cluster. Pour trouver les meilleures combinaisons, nous avons mesuré les quatre configurations possibles. Nous avons utilisé deux types d'application : une application de communication intensive utilisant des diffusions de messages et une application moins communicante d'envoi de messages en jeton. Nous avons sélectionné les deux combinaisons suivantes :

- La combinaison « MLCL » applique une journalisation pessimiste pour les messages inter-cluster et le protocole coordonné de Chandy-Lamport au niveau de chaque cluster.
- La combinaison « MLML » utilise une journalisation pessimiste pour sauvegarder les messages inter-cluster et intra-cluster

La composition « MLCL » s'adapte aux applications de type diffusion de messages, tandis que le protocole « MLML » est le plus performant pour les applications en jeton.

À partir de cette étude, nous avons conçu un nouvel algorithme hiérarchique auto-adaptatif pour assurer la tolérance aux fautes dans les grilles informatiques. Ce protocole s'appuie sur l'architecture hiérarchique des grilles informatiques. Dans chaque cluster, nous avons défini un coordonnateur appelé processus leader, dont le rôle consiste à coordonner les échanges intra-cluster et à assurer le rôle d'intermédiaire entre les processus appartenant à des clusters différents.

Pour sauvegarder les états des processus inter-cluster, le protocole adaptatif utilise le mécanisme de journalisation pessimiste basé sur l'émetteur. A l'intérieur du cluster, le protocole exécuté dépend de la fréquence des messages. A partir d'un seuil de fréquence maximale déterminée en fonction de la densité des communications, c'est le protocole de point de reprise coordonné non bloquant qui sera utilisé, tandis que si le nombre de messages dans le cluster est faible, les messages sont sauvegardés avec la journalisation pessimiste.

En cas de défaillance d'un nœud, soit seul le processus fautif reprend sur exécution ; soit tous les processus du cluster contenant le processus fautif reprennent leur exécution au dernier point de reprise enregistré lors de la dernière sauvegarde coordonnée. Les messages reçus après le dernier point de reprise sont aussi rejoués.

Perspectives

Les précédents travaux ouvrent de nombreuses perspectives de recherche :

- Comme notre protocole s'appuie sur le regroupement des nœuds en cluster. Pour le nombre processus en cas de défaillance, on pourrait reproduire le même modèle à l'intérieur des clusters. Dans ce cas, il faudra un algorithme de clusterisation pour la formation des groupes pendant l'exécution des applications. Ainsi en cas de défaillance, l'impact de la faute resterait confinée aux nœuds du même cluster.
- Elimination des déterminants des messages : une meilleure solution consisterait à ne plus enregistrer les déterminants des messages afin de réduire le coût du stockage des déterminants au niveau de la mémoire de leur émetteur.

Références

[1] Magazine de la Politique scientifique fédérale • www.scienceconnection.be • décembre 2005, Bruxelles X / P409661 / ISSN 1780-8456

[2] Carl Kesselman and Ian Foster. « The Grid: Blueprint for a New Computing Infrastructure ». Morgan Kaufmann Publishers, November 1998.

[3] Grid'500 Project. http://www.grid5000.org/

[4] The Eurogrid project. http://www.eurogrid.org/.

[5] The TeraGrid project. http://www.teragrid.org/

[6] A. Avizienis and J. Laprie and B. Randell, "Fundamental Concepts of Dependability", Research Report N°1145, LAAS-CNRS, April 2001.

[7] D. Powell (Ed.), « Delta-4: a Generic Architecture for Dependable Distributed Computing », Research Reports ESPRIT, 484p. Springer-Verlag, Berlin, Germany, 1991.

[8] Jean-Michel Hélary, Achour Mostefaoui, Michel Raynal, « Déterminer un état global dans un système réparti ». Annales des Télécommunications, vol n° 49, Issue 7-8, pp 460-469.

[9] R. Strom and S. Yemini. « Optimistic recovery in distributed systems », ACM Trans. Comput. Syst., 3(3) :204–226, 1985.

[10] E. N. (Mootaz) Elnozahy, Lorenzo Alvisi, Yi-Min Wang, David B. Johnson, « A Survey of Rollback-Recovery Protocols in Message- Passing Systems », ACM Computing Surveys, Vol. 34, No. 3, September 2002, pp. 375–408.

[11] Johnson, D. B. and Zwaenepoel, W. 1987. « Sender based message logging », In Digest of Papers, FTCS-17, The Seventeenth Annual International Symposium on Fault-Tolerant Computing, 14–19.

[12] Chandy and Lamport, « Distributed snapshots : Determining global states of distributed systems », ACM Transactions on Computer Systems, 3(1) :63– 75, 1985.

[13] Koo and Toueg, « Checkpointing and Rollback-Recovery for Distributed Systems », In Proceedings of 1986 ACM Fall joint computer conference, ACM '86, pages 1150–1158, Los Alamitos, CA, USA, 1986. IEEE Computer Society Press.

[14] Cristian and Jahanian, « A timestamp based checkpointing protocol for long-lived distributed computations », In Proceedings, Tenth Symposium on Reliable Distributed Systems, 12–20, 1991.

[15] Tong, Kain, and Tsai : « Rollback-recovery in distributed systems using loosely synchronized clocks », IEEE Trans. Parallel and Distributed Syst. 3, 2, 246–251, 1992.

[16] Alvisi, L. and Marzullo, K. 1995. « Message logging: Pessimistic, optimistic and causal », In Proceedings of the IEEE International Conference on Distributed Computing Systems (Vancouver, Canada).

[17] S. Drapeau, and C. L. Roncancio. « Concepts et techniques de duplication Technical »

Report RR, 1050-I-LSR 17, Lab. LSR - IMAG, Juillet 2002.

[18] Globus Tollkit : http://www.globus.org/toolkit/about.html

[19] gLite : http://glite.cern.ch

[20] EGEE : http://www.eu-egee.org

[21] Cloud computing : http://cloudcomputing.fr

[22] Christian Delbé, « Tolérance aux pannes pour objets actifs asynchrones - protocole, modèle et expérimentations » PhD. Spécialité Informatique, Janvier 2007, Université de Nice – Sophia Antipolis, France

[23] G. Stellner, « CoCheck: Checkpointing and Process Migration for MPI », In Proceedings of the International Parallel Processing Symposium, pp 526–531, Honolulu, April 1996.

[24] Condor : http://research.cs.wisc.edu/htcondor

[25] Himadri S. Paul, Arobinda Gupta R. Badrinath, « Hierarchical Coordinated Checkpointing Protocol », In International Conference on Parallel and Distributed Computing Systems, pages 240-245, November 2002

[26] K. Bhatia, K. Marzullo, and L. Alvisi. « Scalable causal Message Logging for Wide-Area Environments ». Concurrency and Computation: Practice and Experience, 15(3), pp. 873-889, Aug. 2003.

[27] S. Monnet, C. Morin, R. Badrinath, « Hybrid Checkpointing for Parallel Applications in cluster Federations », Proc. 4th IEEE/ACM International Symposium on Cluster Computing and the Grid, Chicago, IL, USA, pp. 773-782, April 2004.

[28] Esteban Meneses, Celso L. Mendes, and Laxmikant V. Kale. « Team-based Message Logging : Preliminary Results ». In 3rd Workshop on Resiliency in High Perfor- mance Computing (Resilience) in Clusters, Clouds, and Grids (CCGRID 2010)., May 2010.

[29] Jin-Min Yang, Kin Fun Li, Wen-Wei Li, and Da-Fang Zhang. « Trading Off Logging Overhead and Coordinating Overhead to Achieve Efficient Rollback Recovery ». Concurrency and Computation : Practice and Experience, 21 :819–853, April 2009.

[30] Amina Guermouche, « Nouveaux protocoles de tolérance aux fautes pour les applications du calcul haute performance », PhD. Spécialité Informatique, Novembre 2011, Université Paris-Sud, France

[31] O2P T. Ropars and C. Morin, “O2P : un protocole à enregistrement de messages extrêmement optimiste,” in Rencontres Francophones du Parallèlisme (RenPar18), (Fribourg, Switzerland), 2008. In French.

[32] Peterson (S. L.) et Kearns (P.). « Rollback Based on Vector Time ». *In : Proceedings of the 12th IEEE Symposium on Reliable Distributed Systems*. pp. 68–77. – Los Alamitos, CA, USA, 1993.

[33] Smith (S. W.), Johnson (D. B.) et Tygar (J. D.). « Completely Asynchronous Optimistic Recovery with Minimal Rollbacks ». *In : FTCS-25 : 25th International Symposium on Fault Tolerant Computing Digest of Papers*, pp. 361–371. – Pasadena, California, 1995.

[34] Elnozahy, Elmootazbellah, and Zwaenepoel, « Manetho: Transparent rollback-recovery with low overhead, limited rollback and fast output », *IEEE Transactions on Computers*, vol. 41, no. 5, May 1992.

[35] Pierre Lemarinier, Aurelien Bouteiller, Thomas Herault, Geraud Krawezik, Franck Cappello. Impact of Event Logger on Causal Message Logging Protocols for Fault Tolerant MPI. Proceedings of the Int Parallel and Distributed Processing Symposium (IPDPS 05), Denver, USA April, 2005

[36] Thomas Ropars and Christine Morin. « Improving message logging protocols scalability through distributed event logging ». In Proceedings of the 16th international Euro-Par conference on Parallel processing : Part I, EuroPar'10, pages 511–522, Berlin, Heidelberg, 2010. Springer-Verlag.

[37] A. Bouteiller, F. Cappello, T. Herault, G.Krawezik, P. Lemarinier, and F. Magniette. « Mpich-v2 : a fault tolerant mpi for volatile nodes based on pessimistic sender based message logging ». In SC '03 : Proceedings of the 2003 ACM/IEEE conference on Supercom- puting, page 25, Washington, DC, USA, 2003. IEEE Computer Society.

[38] Samir Jafar, « Programmation des systèmes parallèles distribués : tolérance aux pannes, résilience et adaptabilité ». PhD. Spécialité Informatique : Systèmes et Logiciels, 30 Juin 2006, INP de Grenoble, France.

[39] OMNET++ : www.omnetpp.org

[40] MPI : www.open-mpi.org

[41] Briatico D., Ciuffoletti A., Simoncini L., « A distributed domino-effect free recovery algorithm », *IEEE International Symposium on Reliability, Distributed Software, and Data- bases*, 1984

[42] Robert H. B. Netzer and Jian Xu, « Necessary and sufficient conditions for consistent global snapshots », IEEE Transactions on Parallel and Distributed systems. VOL.6 NO. 2, February 1995

[43] N. M. Ndiaye, P. Sens and O. Thiare. "Performance comparison of hierarchical checkpoint protocols grid", IEEE/DCAI2012, International Journal of Interactive Multimedia and Artificial Intelligence (IJIMAI), vol, n°5, pp.46-53, June 2012

[44] N. M. Ndiaye, P. Sens and O. Thiare. "Hierarchical composition of coordinated checkpoint with pessimistic message logging", in Proceedings of IEEE International Conference on Parallel, Distributed and Grid Computing (IEEE/ PDGC 2012), September 2012.

Annexe 1 : Fichier de description de la grille dans le simulateur Omnet++ (grid.ned)

```
simple Nodes
{
    parameters:
       int route0;
       int route1;
       int route2;
       int route3;
       int route4;
       int route5;

    @display("i=misc/node_vs");
}

simple StarNode extends Nodes
{
    parameters:
        @display("i=device/pc2_s");
    gates:
        input in[];
        output out[];
}


simple HubNode extends Nodes
{
    parameters:
        @display("i=device/switch");
    gates:
        input in[];
        output out[];
}


simple Server extends Nodes
{
    parameters:
        @display("i=device/server");
    gates:
        input in[];
        output out[];
}


network grid
{
    parameters:
        int n @prompt("Number of stations") = default(11);
        @display("bgb=476,423");
    submodules:
        node[50]: StarNode {
            parameters:
                @display("p=225,188,ring,150,150");
        }
```

```
    hub1: HubNode {
        parameters:
            @display("p=57,54;is=l");
    }

    hub2: HubNode {
        parameters:
            @display("p=415,46;is=l");
    }

    hub3: HubNode {
        parameters:
            @display("p=403,316;is=l");
    }

    hub4: HubNode {
        parameters:
            @display("p=184,378;is=l");
    }

    hub5: HubNode {
        parameters:
            @display("p=30,245;is=l");
    }

         server: Server {
        parameters:
            @display("p=210,70");
    }
connections allowunconnected:

    // cluster 1
    for i=0..9 {
        hub1.out++ --> {  delay = 100ms; } --> node[i].in++;
        hub1.in++  <-- {  delay = 100ms; } <-- node[i].out++;
    }

         // cluster 2
    for i=10..19 {
        hub2.out++ --> {  delay = 100ms; } --> node[i].in++;
        hub2.in++  <-- {  delay = 100ms; } <-- node[i].out++;
    }

    // cluster 3
    for i=20..29 {
        hub3.out++ --> {  delay = 100ms; } --> node[i].in++;
        hub3.in++  <-- {  delay = 100ms; } <-- node[i].out++;
    }

    // cluster 4
    for i=30..39 {
        hub4.out++ --> {  delay = 100ms; } --> node[i].in++;
        hub4.in++  <-- {  delay = 100ms; } <-- node[i].out++;
    }

    // cluster 5
    for i=40..49 {
        hub5.out++ --> {  delay = 100ms; } --> node[i].in++;
```

```
        hub5.in++  <-- {  delay = 100ms; } <-- node[i].out++;
    }

    // connection des hubs
    hub1.out++ --> {  delay = 0.1ms; } --> hub2.in++;
    hub1.in++  <-- {  delay = 0.1ms; } <-- hub2.out++;
        hub1.out++ --> {  delay = 0.1ms; } --> hub3.in++;
    hub1.in++  <-- {  delay = 0.1ms; } <-- hub3.out++;
    hub1.out++ --> {  delay = 0.1ms; } --> hub4.in++;
    hub1.in++  <-- {  delay = 0.1ms; } <-- hub4.out++;
    hub1.out++ --> {  delay = 0.1ms; } --> hub5.in++;
    hub1.in++  <-- {  delay = 0.1ms; } <-- hub5.out++;

    hub2.out++ --> {  delay = 0.1ms; } --> hub3.in++;
    hub2.in++  <-- {  delay = 0.1ms; } <-- hub3.out++;
    hub2.out++ --> {  delay = 0.1ms; } --> hub4.in++;
    hub2.in++  <-- {  delay = 0.1ms; } <-- hub4.out++;
    hub2.out++ --> {  delay = 0.1ms; } --> hub5.in++;
    hub2.in++  <-- {  delay = 0.1ms; } <-- hub5.out++;

    hub3.out++ --> {  delay = 0.1ms; } --> hub4.in++;
    hub3.in++  <-- {  delay = 0.1ms; } <-- hub4.out++;
    hub3.out++ --> {  delay = 0.1ms; } --> hub5.in++;
    hub3.in++  <-- {  delay = 0.1ms; } <-- hub5.out++;

    hub4.out++ --> {  delay = 0.1ms; } --> hub5.in++;
    hub4.in++  <-- {  delay = 0.1ms; } <-- hub5.out++;

     // connection from hub to server
   hub1.out++ --> {  delay = 100ms; } --> server.in++;
   hub1.in++  <-- {  delay = 100ms; } <-- server.out++;

   hub2.out++ --> {  delay = 100ms; } --> server.in++;
   hub2.in++  <-- {  delay = 100ms; } <-- server.out++;

   hub3.out++ --> {  delay = 100ms; } --> server.in++;
   hub3.in++  <-- {  delay = 100ms; } <-- server.out++;

   hub4.out++ --> {  delay = 100ms; } --> server.in++;
   hub4.in++  <-- {  delay = 100ms; } <-- server.out++;

   hub5.out++ --> {  delay = 100ms; } --> server.in++;
   hub5.in++  <-- {  delay = 100ms; } <-- server.out++;
}
```

Ndeye Massata Ndiaye

Gestion des défaillances dans les grilles informatiques tolérantes aux fautes

Mots clés : grilles informatiques, tolérance aux fautes, points de reprise, journalisation, algorithmes hiérarchiques, défaillances

La construction des grilles informatiques est un des axes de recherche majeurs sur les systèmes informatiques en réseau. L'objectif principal de la construction d'une grille informatique, c'est de fournir les concepts et composant logiciels système adéquats pour agréger les ressources informatiques (processeurs, mémoires, et aussi réseau) au sein d'une grille de traitements informatiques, pour en faire (à terme) une infrastructure informatique globale de simulations, traitement de données ou contrôle de procédés industriels. Cette infrastructure est potentiellement utilisable dans tous les domaines de recherche scientifique, dans la recherche industrielle et les activités opérationnelles (nouveaux procédés et produits, instrumentation, etc.), dans l'évolution des systèmes d'information, du Web et du multimédia.
Les grilles de qualité production supposent une maitrise des problèmes de fiabilité, de sécurité renforcé par un meilleur contrôle des accès et une meilleur protection contre les attaques, de tolérance aux défaillances ou de prévention des défaillances, toutes ces propriétés devant conduire à des infrastructure de grille informatique sûres de fonctionnement. Dans cette thèse on propose de poursuivre des recherches sur les problèmes de gestion automatisée des défaillances, l'objectif principal étant de masquer le mieux possible ces défaillances, à la limite les rendre transparents aux applications, de façon à ce que, du point de vue des applications, l'infrastructure de grille fonctionne de façon quasi-continue.
Nous avons conçu un nouvel algorithme hiérarchique auto-adaptatif pour assurer la tolérance aux fautes dans les grilles informatiques. Ce protocole s'appuie sur l'architecture hiérarchique des grilles informatiques. Dans chaque cluster, nous avons défini un coordonnateur appelé processus leader, dont le rôle consiste à coordonner les échanges intra-cluster et à assurer le rôle d'intermédiaire entre les processus appartenant à des clusters différents.
Pour sauvegarder les états des processus inter-cluster, le protocole adaptatif utilise le mécanisme de journalisation pessimiste basé sur l'émetteur. A l'intérieur du cluster, le protocole exécuté dépend de la fréquence des messages. A partir d'un seuil de fréquence maximale déterminée en fonction de la densité des communications, c'est le protocole de point de reprise coordonné non bloquant qui sera utilisé, tandis que si le nombre de messages dans le cluster est faible, les messages sont sauvegardés avec la journalisation pessimiste.