THÈSE DE DOCTORAT

École Doctorale
« Sciences et Technologies de l'Information et de la Communication »
de Nice - Sophia Antipolis

Discipline Informatique

UNIVERSITÉ DE NICE - SOPHIA ANTIPOLIS
FACULTÉ DES SCIENCES

TOLÉRANCE AUX PANNES POUR OBJETS ACTIFS ASYNCHRONES : PROTOCOLE, MODÈLE ET EXPÉRIMENTATIONS

---

par

Christian DELBÉ

Thèse dirigée par **Denis CAROMEL**

au sein de l'équipe OASIS,
équipe commune de l'INRIA Sophia Antipolis et du laboratoire I3S

présentée et soutenue publiquement le 24 Janvier 2007 devant le jury composé de

Président du Jury	Philippe LAHIRE	Professeur, UNSA
Rapporteurs	Franck CAPPELLO	Directeur de recherche, LRI-INRIA Futurs
	Michel RAYNAL	Professeur, Université de Rennes 1
	Pierre SENS	Professeur, Université Paris 6
Examinateur	Yann JOUANIQUE	Ingénieur, Amadeus
Directeur de thèse	Denis CAROMEL	Professeur, UNSA

A Tiffany,
— Christian

Table des matières

Table des figures

Remerciements

Mes premiers remerciements vont à Denis CAROMEL qui m'a encadré durant cette thèse. Je le remercie pour m'avoir fait confiance et m'avoir donner l'opportunité de me lancer dans cette aventure, pour son optimisme légendaire et pour avoir su m'encourager et me pousser dans les moments de doutes.

Je remercie tout particulièrement Ludovic pour sa patience, pour son amour de la recherche communicatif, pour m'avoir appris à parler Grec, et enfin pour m'avoir enseigner l'art de remplir les plus beaux tableaux qu'on ait jamais vu dans un bureau d'informaticien. Cette thèse est aussi la sienne.

Je remercie également les membres du jury : Philippe LAHYRE qui a accepter de présider ce jury, Yann JOUANIQUE, ainsi que Franck CAPPELLO, Michel RAYNAL et Pierre SENS qui m'ont fait l'honneur de rapporter ce manuscrit.

Un très grand merci toute l'équipe OASIS pour ces presque cinq années. Merci à tous ceux qui m'ont accueillis chaleureusement : Tonton Fabrice, Julien, Arnaud, Laurent, Olivier et tous les autres. Merci en particulier à Françoise pour son soutien et son attention. Merci à vous tous pour m'avoir tenu la main à "mes débuts".

Merci à Tomasito (chuuuut...) pour ces trois ans de voisinage, de collaboration et parfois de batailles franco-chiliennes. Maintenant, je suis un vrai tireur d'élite.

Merci à Romain, Matthieu (mon compagnon d'infortune pour ces samedis et ces dimanches de rédaction), Stéphane et Marc pour avoir su combiner amour du travail et amour de la fête.

Merci (et surtout bon courage...) aux prochains sur la liste : Alex, Guillaume (mon relecteur particulier mais aussi mon pire détracteur), Paul, Mario, Antonio et Marcella.

Merci à tous les "petits nouveaux", à Didier, Brian et Vladimir pour leur aide précieuse, et à Clément (fournisseur officiel de scripts Grid'5000 et de biscuits).

Merci à Claire, ainsi qu'à Christiane qui a eu le courage de se jeter dans le bain dès le début pour organiser la soutenance.

Je remercie mes parents pour avoir cru en moi et m'avoir permis d'arriver jusqu'ici. Merci pour les racines et les ailes...

Et bien sûr, merci de tout mon coeur à Tiffany, qui a su me supporter et m'encourager pendant tout ce temps, tout particulièrement pendant la rédaction. Sans toi, je n'aurais certainement pas tenu le coup.

Enfin, ma dernière pensée et mon dernier merci vont à Isabelle, pour la motivation et le courage qu'elle m'a insuffler au début de cette aventure, comme elle savait si bien le faire. Merci.

“Who controls the past controls the future ; who controls the present controls the past.”

1984, Georges Orwell

Chapitre 1

Introduction

L'objectif premier de cette thèse est de proposer un protocole de tolérance aux pannes par recouvrement arrière pour le modèle à objets actifs asynchrones communicants ASP (*Asynchronous Sequential Processes*) [CAR 04b] et son implémentation en Java ProActive [CAR 06c]. Cette thèse généralise la problématique soulevée par le développement de ce protocole : nous étudions le recouvrement d'une application répartie depuis un état global *non cohérent*.

1.1 Problématique

La tolérance aux pannes dans les systèmes répartis est un domaine qui a été très largement étudié depuis une trentaine d'années [RAN 75]. La littérature propose aujourd'hui un grand nombre de protocoles de tolérance aux pannes par recouvrement arrière, que l'on peut diviser en deux catégories : les protocoles par points de reprise [BRI 84, MAN 96, HéL 99a, BAL 99] et les protocoles par journalisation [STR 85, ELN 92b, BOS 02, BOU 03a]. Chacune de ces catégories présente des propriétés différentes en terme de performance, et n'est parfois pas applicable selon le système ou selon l'application considérée [SEN 95, ELN 02, CON 98, LEM 04]. Nous nous sommes intéressés dans cette thèse à l'approche par point de reprise. En effet, si les solutions par journalisation peuvent facilement être adaptées à notre contexte de travail, le modèle ASP et son implémentation ProActive, les solutions par points de reprise de la littérature ne sont pas applicables.

Les solutions par points de reprise classiques se basent sur la création d'états globaux cohérents, c'est-à-dire représentant un état global possible de l'exécution. Elle supposent pour cela qu'un point de reprise peut être déclenché n'importe quand durant l'exécution, par exemple sur réception d'un message. On parle de persistance *forte* des processus. Une telle persistance peut être obtenue au niveau du système d'exploitation à travers un noyau [GIO 05] ou une bibliothèque spécialisée [PLA 95a], ou en utilisant un compilateur introduisant la persistance au niveau du code source [TRU 00]. Ces solutions impliquent malheureusement une perte de la portabilité dans le cas d'une approche au niveau système, ou un impact souvent prohibitif sur les performances dans le cas d'une approche par compilation.

Pour respecter les objectifs d'efficacité et de portabilité que nous nous sommes fixés, nous ne pouvons pas supposer la persistance forte des processus. Par conséquent, nous ne pouvons plus, comme les approches classiques, nous baser sur la création d'états globaux *cohérents*. C'est la problématique que nous abordons dans cette thèse : permettre le recouvrement d'une application répartie depuis un état global non cohérent.

1.2 Contributions

Nous présentons ici les contributions de cette thèse. Elles se décomposent en trois axes principaux :

Un protocole par points de reprise pour le modèle ASP Nous proposons dans cette thèse un protocole par point de reprise *induits par messages* pour le modèle ASP et son implémentation ProActive. Ce protocole a été développé de manière à être :

- *efficace*, c'est-à-dire minimiser l'impact sur les performances durant une exécution sans pannes, mais aussi minimiser le temps d'exécution supplémentaire induit par une panne,
- *portable*, c'est-à-dire être totalement indépendant de l'architecture physique sous-jacente, et en particulier permettre la reprise après une panne sur n'importe quelle architecture,
- *transparent* pour l'utilisateur, c'est-à-dire ne pas nécessiter la prise en compte de la tolérance aux pannes dans le code source de l'application.

Ce sont ces contraintes qui nous ont conduit à l'étude du recouvrement d'application depuis des états globaux non cohérents. En effet, pour respecter ces contraintes, nous avons dû considérer une persistance *faible* des processus : les points de reprise ne peuvent être déclenchés que pendant certains moments de l'exécution. Par conséquent, nous ne pouvons pas assurer systématiquement la cohérence des états globaux formés durant l'exécution.

Nous proposons donc dans le cadre d'ASP un protocole par points de reprise qui permet à l'application de recouvrir depuis des états globaux non cohérents. Ce protocole a été implémenté dans l'intergiciel ProActive. Cette implémentation illustre tout d'abord la faisabilité de notre solution, et montre aussi à travers des expérimentations réalistes que les contraintes d'efficacité, de portabilité et de transparence ont été respectées.

Un formalisme général Notre étude du problème du recouvrement d'application depuis un état global non cohérent nous a amené à définir un formalisme général permettant la spécification d'une nouvelle condition de recouvrabilité d'un état global : la *cohérence promise*, ou $\mathcal{P}$-cohérence.

Ce formalisme permet de prendre en compte la sémantique de n'importe quel système dans un modèle de type évènementiel. Nous prouvons dans ce cadre général qu'une application peut toujours recouvrir depuis un état global $\mathcal{P}$-cohérent. Notre formalisme permet donc de prouver la correction de protocoles par points

de reprise dans un contexte où la persistance des processus est faible, et où la création d'état global cohérent est impossible. Nous avons appliqué ces résultats formels au modèle ASP, et ainsi pu prouver la correction de notre protocole et de son implémentation.

Une extension du protocole pour les grilles de calcul Nous proposons enfin une extension de notre protocole pour le contexte particulier des grilles de calcul. En effet, nous verrons que le modèle ASP et son implémentation ProActive représentent une solution adaptée à un tel contexte ; nous avons donc voulu prendre en compte les spécificités des grilles de calcul et proposer une extension adaptée.

Nous proposons une solution basée sur la notion de groupe de recouvrement et sur l'utilisation d'une journalisation pessimiste sélective des messages. Une telle approche est particulièrement adaptée aux grilles de calcul car elle permet de prendre en compte la structuration en groupes de machines ainsi que l'hétérogénéité des ressources de communication et de calcul propres à ce contexte.

Finalement, nous avons intégré cette extension dans l'implémentation du protocole existant, et réalisé plusieurs expérimentations sur la plateforme Grid5000 [CAP 05] afin d'évaluer ses performances dans un contexte réél.

1.3 Plan du manuscrit

Cette thèse s'ouvre sur ce chapitre introductif qui présente rapidement les notions fondamentales de tolérance aux pannes comme la sureté de fonctionnement. Nous nous focalisons ensuite sur les systèmes répartis, en présentant les problèmes et les différentes solutions qui permettent d'assurer la tolérance aux pannes dans de tels systèmes.

Le chapitre 2 présente plus particulièrement la solution qui nous intéresse dans cette thèse, la tolérance aux pannes par recouvrement arrière. Dans ce chapitre, nous exposons les principales problématiques liées à cette approche, la mémoire stable, la persistance et la recouvrabilité. Nous présentons ensuite les différentes solutions de tolérance aux pannes par recouvrement arrière proposées dans la littérature : les solutions par points de reprise et les solutions par journalisation. Nous comparons les propriétés de chacune des ces approches en fonction des caractéristiques du système et de l'application considerées. Nous concluons qu'il n'existe pas de solution de tolérance aux pannes unique idéale, adaptée à toutes les situtations.

Le chapitre 3 présente notre contexte de travail, le modèle à objets actifs communicants ASP, et son implémentation en Java ProActive. Ce chapitre est une présentation mais aussi une *analyse* de ce contexte dans le cadre du développement d'un protocole de tolérance aux pannes par recouvrement arrière. En effet, nous identifions toutes les propriétés et les contraintes, induites par le modèle ou par son implémentation, qui peuvent avoir un impact au niveau de la tolérance aux pannes par recouvrement arrière. Nous concluons ce chapitre en montrant que ces propriétés et ces contraintes ne permettent pas d'utiliser directement un protocole de tolérance aux pannes existant dans la littérature : il nous faut donc

proposer une solution adaptée.

Le chapitre 4 présente cette solution. Nous exposons tout d'abord notre protocole de tolérance aux pannes par points de reprise induits par message qui tient compte de toutes les contraintes et qui tire parti de toutes les propriétés de notre modèle. Nous présentons ensuite son implémentation dans l'intergiciel ProActive, ainsi que les différents éléments nécessaires à la configuration et à l'utilisation de notre solution en pratique. Enfin, nous évaluons les performances de notre solution et de son implémentation en présentant les résultats d'expérimentations menées sur une grappe de machines dédiées.

Le chapitre 5 présente une formalisation des concepts introduits par le protocole proposé dans le chapitre 4 ainsi qu'une preuve de sa correction. Nous définissons la $\mathcal{P}$-cohérence, une nouvelle caractérisation d'un état global, et prouvons qu'un état global $\mathcal{P}$-cohérent est recouvrable. Nous appliquons ce formalisme au modèle ASP et prouvons la correction de notre protocole et de son implémentation en montrant que les états globaux formés durant l'exécution sont toujours $\mathcal{P}$-cohérents.

Le chapitre 6 revient sur notre protocole et propose une extension qui permet de l'adapter au contexte particulier des grilles de calcul. Nous exposons cette extension, puis présentons son intégration à l'implémentation du protocole existant. Enfin, nous évaluons les performances de cette extension et de son implémentation en présentant les résultats d'expérimentations meneés sur la grille de calcul Grid5000.

Enfin, le chapitre 7 conclut en résumant les apports essentiels de ce travail, et en exposant quelques perspectives d'extension.

1.4 Notions de tolérance aux pannes

Nous introduisons dans cette section la notion de tolérance aux pannes, puis en présentons les différents aspects, ainsi que le vocabulaire relatif à cette notion. Nous nous focalisons ensuite sur le cas des systèmes répartis en présentant les aspects spécifiques à ce contexte, ainsi que les solutions possibles. Le cas des solutions par recouvrement arrière étant le cas qui nous intéresse dans cette thèse, il sera décrit plus précisément dans le deuxième chapitre de ce manuscrit.

1.4.1 Sûreté de fonctionnement

Laprie [AVI 04] définit la sûreté de fonctionnement comme la capacité de fournir un service dans lequel un utilisateur peut raisonnablement placer sa confiance. De façon plus quantitative, la sûreté de fonctionnement permet de décider si un système est capable d'assurer que la fréquence de défaillance du service et la gravité de ces défaillances restent inférieurs à un minimum considéré comme acceptable.

La tolérance aux pannes proprement dite n'est en réalité qu'une partie du concept de sûreté de fonctionnement, plus précisément *un moyen* de la sûreté de fonctionnement. La sûreté de fonctionnement est composée de différents aspects, ou attributs :

- *la disponibilité* : la capacité à rendre un service à tout instant ;
- *la fiabilité* : la continuité du service ;
- *la sécurité-innocuité* : l'absence de conséquences catastrophiques sur l'utilisateur ou son environnement ;
- *l'intégrité* : l'absence de donnée corrompue dans le système ;
- *la confidentialité* : l'absence de divulgation de données non-autorisée ;
- *la maintenabilité* : l'aptitude du système à être réparé ou amélioré.

La sûreté de fonctionnement peut-être entravée par une faute, ou *panne*. La première conséquence d'une panne est une *erreur* dans le système : le système se retrouve dans un état imprévu qui n'est pas considéré comme un état correct. Le système étant dans un état imprévu, le service ne peut pas être rendu : on dit que le système *défaille*. Cette chaîne *panne* $\rightarrow$ *erreur* $\rightarrow$ *défaillance* est la chaîne fondamentale des entraves définie par Laprie.

Pour assurer la sûreté de fonctionnement, il faut soit éviter que la source de la chaîne fondamentale des entraves - la panne - ne se produise, soit éviter que la panne ne se transforme en erreur, puis en défaillance.

La prévention ou l'élimination des pannes sont des approches qui tentent d'éviter les occurrences de pannes. Ces approches se basent sur des systèmes de vérification formelle des applications qui déterminent et minimisent la probabilité de panne, et sur des systèmes matériels qui sont supposés sûrs.

La tolérance aux pannes est une approche différente qui ne cherche pas à éviter les pannes, mais qui évite la transition de la panne vers l'erreur dans la chaîne fondamentale des entraves. La panne ne doit pas engendrer un état erroné du système ; dans ce cas, le système ne défaille pas. C'est cette approche qui nous intéresse dans ce manuscrit, plus particulièrement dans le contexte des systèmes repartis.

1.4.2 Tolérance aux pannes dans les systèmes répartis

Nous considérons ici un système réparti comme un ensemble de processus communicants par messages. On peut distinguer deux types de système réparti, selon les hypothèses faites sur le mode de communication : les systèmes asynchrones, pour lesquels le délai entre l'envoi et la réception d'un message n'est pas borné, et les systèmes synchrones, pour lesquels ce délai est borné et connu.

Avant de présenter les différentes méthodes pour rendre un système réparti tolérant aux pannes, il convient de comprendre ce qu'est exactement une *panne*, et comment elle peut être détectée durant l'exécution, selon les hypothèses faites sur le mode de communication, i.e. synchrone ou asynchrone.

1.4.2.1 Types de pannes

Les pannes qui peuvent survenir durant une exécution répartie peuvent être classées en quatre catégories :

- *les pannes franches* (*crash, fail-stop*), que l'on appelle aussi arrêt sur défaillance. C'est le cas le plus simple : on considère qu'un processus peut être dans deux états, soit il fonctionne et donne le résultat correct, soit il ne fait plus rien. Dans le second cas, le processus est considéré comme *définitivement* défaillant.
- *les pannes par omission* (*transient, omission failures*). Dans ce cas, on considère que le système peut perdre des messages. Ce modèle peut servir à représenter des défaillances du réseau plutôt que des processus.
- *les pannes de temporisation* (*timing, performance failures*). Ce sont les comportements anormaux par rapport à un temps, comme par exemple l'expiration d'un délai de garde.
- *les pannes arbitraires*, ou byzantines (*malicious, byzantine failures*). Cette classe représente toutes les autres pannes : le processus peut alors faire "n'importe quoi", y compris avoir un comportement malveillant.

Le cas le plus simple est bien sûr le cas des pannes franches, et on essaie toujours de s'y ramener, par exemple en tuant un processus en cas de comportement imprévu. La plupart des protocoles de tolérance aux pannes pour les systèmes répartis ne considèrent que ce type de pannes, et c'est le cas que nous considérons par la suite.

1.4.2.2 Détection des pannes

Le problème de la détection des pannes est résolu différemment selon le modèle de communication du système. Considérons d'abord le cas des modèles synchrones, dans lesquels le temps de transmission d'un message est borné. Supposons qu'un envoi de message provoque la mise en attente de l'émetteur d'une confirmation de réception de la part du récepteur. La détection des pannes franches et de temporisation peut alors être réalisée à l'aide de délais de garde lors des communications : lorsqu'un processus communique avec un autre et ne reçoit pas de confirmation après ce délai de garde, il peut considérer que le processus cible est défaillant.

Le problème devient plus complexe dans le cas des modèles asynchrones : le temps de transmission d'un message n'est pas borné. Les communications entre les nœuds ne peuvent donc plus être utilisées pour détecter les pannes car il est impossible de prédire le moment où le message est effectivement reçu par le récepteur. On a alors recours à un détecteur (ou suspecteur) de pannes [CHA 96a], qui informe les processus sur l'état du système. Ces détecteurs utilisent eux aussi des délais de garde. On distingue deux modèles différents :

- Le modèle *push*, dans lequel chaque processus du système doit régulièrement informer les détecteurs de pannes[1] de son état. Si ce détecteur n'a pas

[1] Il peut y en avoir plusieurs, jusqu'à un par processus.

reçu de message de type “je suis vivant” de la part d'un processus depuis un temps donné, ce processus est suspecté d'être défaillant.
– Le modèle *pull*, dans lequel ce sont les détecteurs de pannes qui envoient régulièrement des requêtes de type “es-tu vivant ?” aux processus du système. Un processus qui ne répond pas dans un temps donné est suspecté d'être défaillant.

On note que l'on parle de *soupçons* parce que le processus incriminé n'est pas forcement en panne : le message peut être encore en transit dans le système à la fin du délai. Chandra et al. ont défini dans [CHA 96a] les notions de *justesse*, qui indique si un détecteur de pannes peut considérer comme défaillant un processus correct, et de *complétude*, qui indique si il est possible qu'un processus en panne ne soit jamais détecté.

1.4.3 Solutions de tolérance aux pannes pour les systèmes répartis

On peut distinguer deux approches principales dans le domaine de la tolérance aux pannes dans les systèmes répartis : l'approche par réplication et l'approche par recouvrement arrière. Nous présentons rapidement dans cette section l'approche par réplication. L'approche par recouvrement arrière est présentée de façon plus précise dans le chapitre suivant.

La tolérance aux pannes par réplication [MAR 03] se base sur la redondance des processus composants l'application. Cette approche permet de masquer les pannes éventuelles : lorsqu'un processus tombe en panne, une des copies de ce processus prend sa place dans le système. On distingue trois approches différentes pour réaliser la redondance des processus, selon que les copies s'exécutent systématiquement en parallèle ou que l'exécution des copies soit démarrée en cas de panne. Plus précisément, on distingue :
– *la réplication active* : toutes les copies de processus s'exécutent en même temps. En particulier, tous les messages à destination d'un processus sont envoyés à toutes les copies. Les différentes copies gardent donc un état étroitement synchronisé durant l'exécution. Ce type de réplication permet de gérer tout type de faute, en particulier les pannes byzantines en mettant en oeuvre un consensus sur un résultat entre toutes les copies [LAM 04]. Notons que la réplication active impose que tous les processus soient totalement déterministes ; on suppose que les exécutions des différentes copies ne peuvent pas diverger.
– *la réplication passive* : une seule des copies, la copie primaire, s'exécute à la fois. En cas de panne, une copie secondaire est démarrée et doit rattraper l'exécution jusqu'à l'état de la copie primaire avant la panne. L'état des copies secondaires doit donc être régulièrement mis à jour par la copie primaire. Cette technique est en fait similaire aux approches par recouvrement arrière détaillées dans le chapitre suivant.
– *la réplication semi-active* : toutes les copies s'exécutent en même temps, mais une seule d'entre elle, la copie maîtresse, émet les messages résultants de l'exécution. Ainsi, les autres copies mettent à jour leur état interne

et sont donc étroitement synchronisées avec la copie maîtresse. L'intérêt de cette approche est de pouvoir prendre en compte les processus non déterministes : seule la copie maîtresse est responsable des choix non déterministes, et en informe les autres copies par des messages spécifiques.

L'approche par réplication active est souvent utilisée dans un contexte où le temps d'exécution est primordial, par exemple s'il est borné. En effet, les pannes sont très rapidement masquées et n'influent que très peu sur le temps d'exécution total. Cependant, cette approche implique la disponibilité d'un nombre important de ressources. L'approche par réplication semi-active présente les mêmes propriétés, avec la capacité de gérer les processus non déterministes.

Le chapitre suivant présente la solution alternative à la réplication active et semi-active, la tolérance aux pannes par recouvrement arrière.

Chapitre 2

Tolérance aux pannes par recouvrement arrière

Le but d'un protocole de tolérance aux pannes par recouvrement arrière est de **capturer** suffisamment de données pendant une exécution distribuée sans panne, ou *exécution de référence*, et de stocker ces données en **mémoire stable**. Les données collectées doivent pouvoir être utilisées pour redémarrer tout ou partie de l'application si une panne est détectée ; ces données doivent former un état **recouvrable**. La détection de panne, discutée dans la section 1.4.2.2 ne sera plus investiguée plus loin dans ce thèse. Nous supposerons par la suite qu'un mécanisme de détection de panne adéquat est disponible.

On peut donc distinguer trois éléments clés dans la tolérance aux pannes par recouvrement arrière :

- l'accès à une mémoire stable,
- la capacité de capturer l'état des processus,
- la caractérisation et la création d'états recouvrables.

Nous commençons ce chapitre en détaillant chacune de ces trois notions, puis proposons un panorama des différentes techniques de tolérance aux pannes par recouvrement arrière. Des comparaisons plus précises entre les travaux présentés ici et les résultats de cette thèse seront présentées dans les chapitres suivants.

2.1 Mémoire stable

La tolérance aux pannes par recouvrement arrière repose sur l'existence d'une *mémoire stable*. Les informations qui sont nécessaires en cas de panne et de reprise d'un ou plusieurs éléments du système doivent être elles-mêmes stockées de façon *tolérante aux pannes*. Lorsqu'une donnée est stockée sur la mémoire stable, on dit que cette donnée est *stable*. Une mémoire stable doit avoir les propriétés suivantes :

- **protection contre les pannes** : la mémoire stable doit survivre aux pannes qui sont tolérées par le mécanisme de tolérance aux pannes ;
- **accessibilité aux données** : la mémoire stable doit être accessible par tous les éléments de l'application.

La notion de mémoire stable est un concept abstrait [ELN 02], par opposition à la mémoire *volatile*, la mémoire des processus. En particulier, une mémoire stable ne suppose pas l'existence d'un *processus* stable. Elle est par exemple représentée dans le système de traitement par lots Legion [LEW 03] sous la forme d'objets nommés *vault objects* qui sont des supports de stockage stable, mais dont l'implémentation n'est pas spécifiée. Ils peuvent être en pratique approximés par les disques locaux des ressources utilisées par l'application.

La mémoire stable est souvent représentée comme un serveur stable, c'est-à-dire tolérant aux pannes. Cette représentation est trop restrictive. En effet, la mémoire stable nécessaire est dépendante des propriétés du mécanisme de tolérance aux pannes considéré. Par exemple, dans le cas d'une approche ne supportant qu'un nombre fini k de pannes simultanées, un ensemble de $k+1$ disques locaux peut être vu comme une mémoire stable. Les données doivent être répliquées sur au moins $k+1$ ressources pour supporter k pannes simultanées. Le nombre de ressources nécessaires peut encore être réduit en utilisant par exemple un codage de données redondant [PLA 98].

Nakamura et al. proposent dans [NAK 04] un protocole par recouvrement arrière qui est capable de supporter $log_2 n$ pannes simultanées (ou n est le nombre de processus de l'application) *sans nécessiter de réplication des données*. Les données nécessaires à la reprise ne sont stockées sur le disque local que d'une seule ressource ; l'originalité de l'approche réside dans le choix de cette ressource pour chaque processus.

De manière plus générale, il est possible d'obtenir une mémoire stable à partir d'un ensemble de ressources non fiables. Par exemple, en utilisant un système de fichiers répliqués tolérant aux pannes, tel que le système SFS [SEN 00], ou encore sur des systèmes de fichiers de type pair-à-pair [BUS 05a, MUT 02], dans lesquels aucun élément du système n'est supposé fiable à 100%.

Dans le cas d'une grappe de machines dédiées (*cluster*) ou d'un réseau local, l'existence d'un serveur NFS est une bonne approximation d'une mémoire stable. En effet, le serveur NFS est lui-même souvent un élément de confiance, basé sur des techniques de fiabilisation physique, telles que les disques de stockage redondants type RAID. L'accessibilité est dans ce cas le point faible : dans le cas d'une application distribuée sur différents réseaux qui ne partagent pas de NFS, cette approximation n'est bien sûr plus valable.

2.2 Capture de l'état d'un processus : persistance

La tolérance aux pannes par recouvrement arrière suppose aussi qu'un processus peut capturer une image de son état, ou point de reprise (*checkpoint*), au cours de son exécution ; cette image peut éventuellement être stockée sur la mémoire stable. Elle doit pouvoir être utilisée directement pour redémarrer le processus ; l'image produite doit donc contenir tout le contexte du processus comme les registres courants, les segments de données ou la pile d'exécution.

Nous donnons ici les différentes méthodes pour obtenir la persistance des processus, en les séparant en deux catégories :

– les méthodes offrant une persistance *forte*, c'est-à-dire que la capture d'état peut être déclenchée à n'importe quel moment de l'exécution par un autre processus indépendant,
– et les méthodes offrant une persistance *faible*, c'est-à-dire que la capture d'état ne peut être réalisée que lorsque l'exécution atteint un point particulier, ou point de capture potentielle.

De plus, nous les comparons selon les critères suivants [BRO 06] :

– l'impact sur les performances : le surcoût global sur le temps d'exécution induit par le mécanisme de persistance ;
– l'efficacité du mécanisme : le surcoût sur le temps d'exécution induit par la capture d'une image ;
– la portabilité du mécanisme : si le mécanisme de persistance peut être utilisé sur différentes architectures ;
– la portabilité des images : si les images produites peuvent être utilisées pour redémarrer les processus sur des architectures différentes de celle dans laquelle la capture a été faite ;
– la transparence : si le mécanisme de persistance fait intervenir l'utilisateur.

Deux techniques permettent de réduire le surcoût induit par la capture d'une image. La première consiste à créer les images de manière incrémentale : lors de la capture, seules les parties de l'état qui ont été modifiées depuis la capture précédente sont stockées. Cette technique permet en particulier de réduire la taille des images produites. Elle nécessite de pouvoir tracer les modifications des zones mémoire utilisées par le processus, et requiert donc des accès de bas niveau. Plank et al. proposent dans [PLA 95b] une méthode incrémentale qui ne nécessite pas de possibilité d'accès bas niveau, basée sur la comparaison de l'image avec la précédente ; la capture est dans ce cas totale, mais la taille des données à stocker effectivement sur la mémoire stable est réduite.

La seconde technique, dite capture non-bloquante, consiste à laisser l'exécution continuer pendant la capture de l'image. Pour éviter les problèmes d'incohérence dans l'image capturée, on utilise un mécanisme de copie déclenchée par l'écriture (*copy on write*) : les différentes zones mémoire utilisées par le processus sont protégées en écriture, puis le mécanisme de persistance copie sur la mémoire stable puis déverrouille une à une ces zones mémoire. Si le processus tente d'accéder à une zone qui n'a pas encore été copiée, cette zone est répliquée en mémoire volatile ; le processus peut alors accéder à la zone concernée, et le mécanisme de persistance copie le replica sur la mémoire stable.

Cette optimisation nécessite aussi un accès bas niveau au système de gestion de la mémoire. Si ces accès sont impossibles, cette technique peut être approximée : une copie de la totalité de l'image du processus est d'abord stockée en mémoire volatile, puis cette copie intermédiaire est copiée sur la mémoire stable. Pendant la copie sur mémoire stable, le processus peut continuer son exécution. Le coût de la copie intermédiaire en terme de temps mais aussi d'occupation de la mémoire volatile peut cependant être prohibitif [SEN 95].

2.2.1 Persistance forte

La persistance forte peut être implémentée soit au niveau de l'environnement d'exécution, soit au niveau du compilateur. Dans le premier cas, la persistance peut être implémentée directement au niveau du système d'exploitation, soit au niveau du noyau lui-même [GIO 05], soit dans une bibliothèque dédiée qui est liée à la compilation avec l'application [PLA 95a]. Avec cette approche, les optimisations de capture incrémentale et de capture non-bloquante sont réalisables puisque le système d'exploitation peut accéder au contexte d'un processus et peut manipuler directement la mémoire ; cette approche est donc souvent très efficace. Elle est cependant très intrusive dans le cas d'une modification du noyau, et de manière plus générale peu portable. En effet, l'implémentation du mécanisme est totalement dépendante de l'architecture, et les images générées ne peuvent être utilisées pour redémarrer le processus que sur la même architecture. Par exemple, la persistance dans système de traitement par lots Condor [THA 05] repose sur une bibliothèque dédiée [TAN 95], mais ne fonctionne que sur le système UNIX.

Si un environnement d'exécution virtuel est utilisé, comme c'est le cas avec le langage Java, cet environnement peut être modifié pour fournir la persistance forte des processus [BOU 99, AGB 00]. Cette approche permet d'appliquer des techniques d'optimisation de bas niveau, et permet donc une capture efficace sans surcoût global significatif sur le temps d'exécution.

La portabilité des images est intrinsèque à l'utilisation d'une machine virtuelle : une image peut être utilisée pour redémarrer un processus sur un autre environnement virtuel, quelque soit l'architecture réelle sous-jacente. Cependant, cette approche implique une perte de la portabilité du code applicatif, puisqu'elle repose sur une modification de l'environnement d'exécution ; l'application ne peut pas être exécutée sur la ou les versions standards de l'environnement d'exécution virtuel. De plus, le mécanisme lui-même n'est pas portable ; il faut une version de l'environnement d'exécution virtuel, donc une version du mécanisme pour chaque architecture.

Elle permet par contre une transparence totale pour l'utilisateur, puisque le processus peut être capturé dans son intégralité au niveau de la machine virtuelle. L'utilisateur n'a pas besoin de définir ce qui doit être capturé ni comment les données doivent être capturées.

Enfin, la persistance forte peut aussi être fournie au niveau du code source de l'application. Par exemple, les projets Brakes [TRU 00] ou JavaGo [SEK 99] proposent un préprocesseur respectivement de code machine Java (*bytecode*) et de code source Java qui instrumente le code de manière à permettre la capture des processus. Si la portabilité du mécanisme et des images est garantie par la portabilité de Java, l'inconvénient majeur de ce type d'approche est le surcoût sur le temps d'exécution qui est souvent prohibitif. Par exemple, Bouchenak et al. présentent dans [BOU 04] des expériences qui montrent dans le cas de [SEK 99] un surcoût de 100% sur le temps d'exécution normal pour une application de calcul de Fibonnaci sans déclencher de capture d'état.

2.2.2 Persistance faible

La persistance faible se base sur des *points de capture potentiels*, c'est-à-dire que la capture de l'état du processus ne peut être faîte que lorsque l'exécution du processus atteint un de ces points. Selon les approches choisies, ces points de capture peuvent être définis par le programmeur dans le code source de l'application, ou bien déterminés par le compilateur. Il est aussi possible que les points de capture potentiels soient découvert dynamiquement pendant l'exécution, en fonction de l'état courant du processus ; nous verrons que c'est le cas dans notre contexte.

La solution la plus simple pour fournir la persistance faible est de laisser faire le programmeur ; il doit définir lui-même les mécanismes de persistance dans le code source de son application. Bien que les aspects liés à la portabilité peuvent être gérés par le programmeur, cette approche n'est pas transparente et augmente de façon considérable la complexité du développement d'une application.

La persistance faible peut être fournie par un compilateur spécialisé tel que le compilateur source-vers-source Porch [STR 98], ou le compilateur C^3 [BRO 03]. Ces deux outils instrumentent le code source original de manière à être capable de capturer l'état du processus aux points de capture potentiels spécifiés par le programmeur dans le code source, par exemple par un pragma `potential_checkpoint` dans la cas du compilateur C^3. Des compilateurs ouverts comme OpenC++ [CHI 95] ou OpenJava [TAT 99] peuvent aussi être utilisés pour instrumenter le code source en utilisant la réflexivité à la compilation [KIL 99]. Des méthodes, qui peuvent être redéfinies dans un méta programme, sont appelées par le compilateur pendant la lecture du code ; elles permettent de définir les modifications à apporter au code pour assurer la persistance. Ces approches ne sont pas transparentes ; même si les modifications nécessaires sont minimes, le code doit être modifié et recompilé pour pouvoir profiter de la persistance.

Les points de capture potentiels peuvent aussi être spécifiés sans l'intervention de l'utilisateur. Un compilateur peut par exemple déterminer à la compilation les moments de l'exécution pendant lesquels la capture est optimale, particulièrement en terme de taille de donnée [CHO 02]. Il peut aussi devoir déterminer quand la capture est *possible*, en fonction de la disponibilité et de la validité des informations sur le processus à capturer [BUN 02]. Dans ces deux cas, la transparence est assurée puisque le code ne nécessite aucune modification.

Les solutions par compilation présentées ci-dessus assurent la portabilité des images des processus, même dans le cas d'un langage non portable tel que C. Par exemple, le compilateur Porch génère des fonctions de mises à plat, ou *sérialisation*, des structures de données utilisées par le programmeur, de manière à pouvoir les recréer sur différentes architectures. De la même manière, le compilateur ajoute au début des fonctions du code qui permet de restaurer l'état de la pile du processus de façon portable.

2.3 Recouvrabilité

La notion de recouvrabilité est le point clé de la tolérance aux pannes par recouvrement arrière ; un état recouvrable est un état stable, c'est-à-dire sauvegardé en mémoire stable, depuis lequel une exécution peut repartir correctement. En d'autres termes, un état recouvrable :

- soit représente directement un état *cohérent* de l'exécution,
- soit contient suffisamment d'informations pour contrôler l'exécution jusqu'à un état cohérent.

La notion de cohérence d'état a été introduite dans [RAN 75], puis formalisée par Chandy et Lamport dans [CHA 85]. Un état cohérent correspond à un état *possible* de l'exécution sans panne ; c'est donc un état depuis lequel une exécution peut repartir correctement. Informellement, un état cohérent est un état qui ne reflète pas les conséquences d'évènements qui n'ont pas encore eu lieu.

Nous caractérisons dans cette section cette notion de cohérence. Nous proposons dans un premier temps une formalisation d'une exécution distribuée, de manière à pouvoir proposer une formulation précise de la cohérence.

2.3.1 Caractérisation d'une exécution distribuée

Nous considérons comme système réparti un ensemble de processus $P_1, P_2, ..., P_n$ communiquant entre eux par envoi de message. La réception des messages répond à un ordonnancement. Les communications sont asynchrones et fiables : il n'y a ni perte, ni duplication ni altération des messages.

Une exécution distribuée peut être vue par un observateur extérieur comme la succession d'états globaux S, où un état global est un ensemble possible d'états locaux $S = \{s_1, s_2, ..., s_n\}$. Chaque changement d'un état global S à un état S' est la conséquence de l'exécution d'un évènement e ; on dit que l'état S est réduit par l'évènement e vers l'état S', noté $S \xrightarrow{e} S'$. Une exécution complète peut donc être vue comme une suite de réductions par des évènements d'un état initial S^0 vers un état final S^n ; on note alors l'exécution :

$$S^0 \xrightarrow{e^1 \ldots e^n} S^n \Leftrightarrow S^0 \xrightarrow{e^1} S^1 \ldots S^{n-1} \xrightarrow{e^n} S^n$$

Les états globaux considérés ici sont par définition des états possibles de l'exécution, donc des états cohérents. On pourra dire par la suite qu'un état *contient* un évènement e si la réduction $S \xrightarrow{e} S'$ a eu lieu ; en effet on peut voir un état comme la combinaison d'un état initial et de l'ensemble des évènements qui ont servi à réduire l'état initial vers cet état.

Lamport introduit dans [LAM 78] la notion de temps logique : il définit une relation *happened-before* qui permet de déterminer si un évènement e a lieu avant un autre évènement e'. Il définit un ordre *partiel* entre les évènements qui indique si un évènement a été exécuté avant un autre. Les évènements de communication, c'est-à-dire l'envoi et réception de message, sont identifiés par l'ensemble $\Gamma = \{(e, e') \in \{e^1 \ldots e^n\} * \{e^1 \ldots e^n\}\}$ tel que e est l'envoi d'un message donné, et e' la réception de ce message.

Définition 2.3.1 (Relation de Lamport $\prec^{hb}$) *Les évènements locaux exécutés sur un processus P_i sont totalement ordonnés par l'ordre de précédence locale $\prec_i^{hb}$. $\prec^{hb}$ est un ordre partiel entre les évènements tel que $e \prec^{hb} e'$ si et seulement si*

$$e \prec_i^{hb} e' \quad \vee \quad (e, e') \in \Gamma \quad \vee \quad (\exists e'', e \prec^{hb} e'' \prec^{hb} e')$$

Cette relation modélise la précédence entre les évènements observables localement au moment de l'exécution ; les évènements sont localement totalement ordonnés par l'ordre $\prec_i^{hb}$. La synchronisation due aux communications implique que les évènements qui ont lieu avant l'envoi d'un message ont aussi lieu avant la réception de ce message. Si deux évènements e et e' ne sont pas ordonnés par $\prec^{hb}$, on note $e \parallel e'$.

Mattern propose d'utiliser la précédence de Lamport comme relation de causalité entre les évènements : si e précède e', alors e affecte *potentiellement* e'. Il caractérise dans [MAT 89] une exécution distribuée sous la forme d'un ensemble partiellement ordonné d'évènements par la relation de Lamport :

Définition 2.3.2 (Caractérisation d'une exécution répartie $(E, \prec^{hb})$) *Soit une exécution $S \xrightarrow{e^1 \ldots e^n} S'$, $E = \{e^1 \ldots e^n\}$ l'ensemble des évènements exécutés. L'ensemble partiellement ordonné d'évènements $(E, \prec^{hb})$ caractérise l'exécution $S \xrightarrow{e^1 \ldots e^n} S'$.*

Une exécution est donc caractérisée par l'ensemble des évènements qui ont servi à réduire l'état initial vers l'état final, ordonnés par la relation de Lamport.

2.3.2 Caractérisation de la cohérence

La relation de causalité entre les évènements a permis de formaliser le concept de cohérence d'un état global. Un état global est un ensemble qui contient un point de reprise par processus de l'application. Un état global forme une *coupe* de l'exécution, c'est-à-dire l'ensemble des évènements qui ont servi à la réduction depuis les différents états initiaux vers chacun des états représentés par les points de reprise. Une coupe est définie par Mattern dans [MAT 92], comme un ensemble partiellement ordonné d'évènements tel que :

Définition 2.3.3 (Coupe) *Une coupe $\mathcal{C}$ dans une exécution $(E, \prec^{hb})$ est un sous-ensemble fini $\mathcal{C} \subseteq E$ tel que*

$$\forall e, e' \in E, e \in \mathcal{C} \wedge e' \prec_i^{hb} e \Rightarrow e' \in \mathcal{C}$$

Un état global est possible, donc cohérent, si il ne contient pas d'évènement qui sont la conséquence d'autres évènements qui ne sont pas contenu dans l'état. Mattern a proposé la définition de la coupe cohérente, qui étend la définition d'une coupe quelconque :

Définition 2.3.4 (Coupe cohérente) *Une coupe $\mathcal{C}$ est cohérente si et seulement si*

$$e \in \mathcal{C} \wedge e' \prec^{hb} e \Rightarrow e' \in \mathcal{C}$$

Une coupe cohérente représente un état possible d'une exécution. Soit $\mathcal{C}$ une coupe cohérente dans une exécution $(E, \prec^{hb})$ et S l'état global caractérisé par $\mathcal{C}$. Si $S \xrightarrow{e} S'$ est une réduction de $(E, \prec^{hb})$ alors $\mathcal{C}' = \mathcal{C} \cup \{e\}$ est aussi une coupe cohérente dans $(E, \prec^{hb})$. Nous pouvons donc étendre la notion de réduction sur les coupes cohérentes, notée $\mathcal{C} \xrightarrow{e} \mathcal{C}'$.

Comme on peut le voir dans la définition 2.3.3, un état global est *localement* cohérent, c'est à dire qu'un point de reprise contient forcement les causes locales de tous les évènements qu'il contient. L'incohérence d'un état global ne peut donc être causée que par des évènements de communication. Dans la figure 2.1, nous montrons trois processus ayant chacun pris un point de reprise, notés C_P^1,C_Q^1 et C_R^1 ; pourtant, l'ensemble de ces trois points ne représente pas un état global cohérent. En effet le point de reprise C_R^1 contient la réception du message m_3, alors que le point de reprise C_Q^1 ne contient pas son envoi. La réception étant une conséquence de l'envoi, on a donc un état global qui contient un évènements sans contenir toutes les causes. En particulier, le message m_3 est un *message orphelin*.

Définition 2.3.5 (Message orphelin) *Un message orphelin dans un état global donné est un message qui a été envoyé après un point de reprise appartenant à cet état global et reçu avant un point de reprise appartenant à cet état global.*

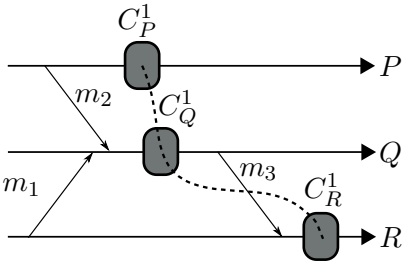

FIG. 2.1 – État global incohérent : m3 est orphelin

Un message orphelin introduit une dépendance causale entre deux points de reprise de l'état global : un point de reprise devient conséquence d'un autre. L'état ne peut donc plus être cohérent. On peut en déduire une condition nécessaire pour qu'un état global soit cohérent :

Propriété 2.3.1 *Si deux points de reprise C_P et C_Q appartiennent tous les deux à un état global cohérent, alors il n'existe pas de dépendance causale entre les deux événements "prise de C_P" et "prise de C_Q".*

Cependant, la réciproque n'est pas vraie : on ne peut pas dire que si deux points de reprise ne sont pas causalement liés, alors ils peuvent faire partie d'un état global cohérent. En effet, des relations indirectes entre les points de reprises peuvent empêcher la cohérence de l'état global. Comme on peut le voir dans la figure 2.2, C_P^1 n'est pas en relation causale avec C_R^1, pourtant ils ne peuvent pas appartenir tous deux à un même état global cohérent, puisque soit le message $m1$

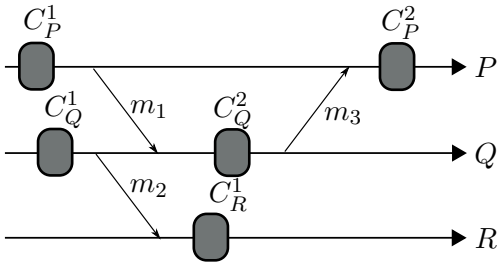

FIG. 2.2 – Chemin "zigzag" entre C_P^1 et C_R^1

dans l'état global $[C_P^1, C_Q^2, C_R^1]$, soit le message $m2$ dans l'état global $[C_P^1, C_Q^1, C_R^1]$ serait orphelin.

Cette relation qui existe entre C_P^1 et C_R^1 se nomme un *chemin "zigzag"*. Cette généralisation de la relation entre deux points de reprise a été proposée par Netzer et al. dans [NET 95]. Un chemin "zigzag" peut se définir de la façon suivante :

Définition 2.3.6 *Il existe un chemin "zigzag" entre deux points de reprise C_P et C_R pris sur deux processus P et R si et seulement s'il existe des messages m_1, m_2, ..., m_n tels que :*

- *m_1 est émis par P après la prise de C_P*
- *si m_k ($1 \leq k \leq n$) est reçu par un processus Q_k, alors le message m_{k+1} est envoyé par Q_k dans le même intervalle de points de reprise (ou dans un intervalle suivant). m_{k+1} peut être envoyé aussi bien avant que après la réception de m_k*
- *m_n est reçu par R avant la prise de C_R*

Lorsqu'il existe un chemin "zigzag" entre un point de reprise et lui-même, on parle alors d'un cycle "zigzag". De fait, un point de reprise pris dans un cycle "zigzag" ne peut bien sûr appartenir à aucun état global cohérent, et est donc inutile. Par exemple, dans la figure 2.2, le point C_Q^2 est pris dans un cycle "zigzag" à cause des messages m_1 et m_3 et du point de reprise C_P^2.

Un grand nombre de protocoles de tolérance aux pannes par points de reprise utilisent ces notions de chemin et de cycle "zigzag" pour déterminer des états globaux cohérents, ou pour savoir si il est utile à un instant donné de prendre un point de reprise [MAN 99, HéL 00].

2.3.3 Cohérence forte

Un message peut avoir été envoyé mais ne pas avoir été reçu (message m_3 sur la figure 2.3) ; l'état global reste cohérent, au sens de [CHA 85]. En effet, c'est un état possible du système, le message m_3 est *en cours d'envoi*. Ce message est appelé un message *en transit* :

Définition 2.3.7 (Message en transit) *Un message en transit par rapport à un état global est un message qui a été envoyé avant un point de reprise appartenant à cet état global et reçu après un point de reprise appartenant à cet état global.*

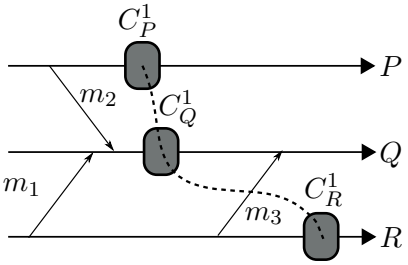

FIG. 2.3 – État global cohérent, mais m3 est en transit

La cohérence au sens de [CHA 85] n'interdit pas les messages en transit : elle suppose que l'état des canaux de communication fait partie de l'état global. Si on ne considère dans l'état global *que* l'état des processus, on parle alors de cohérence *simple*. Pour prendre en compte ces messages en transit, on introduit la notion de cohérence *forte*.

Définition 2.3.8 *La **cohérence forte** étend la cohérence simple, avec la condition supplémentaire qu'il n'existe pas de message en transit dans l'état global considéré.*

Hélary et al. montrent dans [HéL 99b] que si l'absence de chemin "zigzag" entre les points de reprise assure l'absence de message orphelins, alors l'absence de chemin "zigzag" *inverse* assure l'absence de message en transit. Un chemin "zigzag" inverse est un chemin "zigzag" que l'on trouverait dans l'exécution si on inversait tout les messages de sorte que les émetteurs deviennent les récepteurs, et réciproquement. En effet, on peut constater qu'un message en transit est le cas inverse, ou *dual*, d'un message orphelin : l'un n'a plus d'émetteur et l'autre n'a plus de récepteur.

Cette propriété permet d'assurer une cohérence forte à partir d'un protocole assurant la cohérence faible : si on est capable d'appliquer dans le même temps ce protocole *et* son dual, on empêche la présence de message orphelin *et* de message en transit : on assure alors la cohérence forte des états globaux formés [HéL 99a].

2.4 Tolérance aux pannes par points de reprise

Les protocoles de tolérance aux pannes par point de reprise se basent sur la création d'états directement recouvrables. Ce type de protocole créé pendant l'exécution ou recherche après une panne des états globaux cohérents qui sont utilisés pour redémarrer l'exécution. Un état global est alors appelé *ligne de recouvrement*. Il existe différentes approches pour assurer la cohérence de la ligne de recouvrement. Nous décrivons dans cette section les trois approches possibles : points de reprise *indépendants*, *synchronisés* et *induits par messages*.

2.4.1 Indépendants

Cette méthode consiste à faire prendre aux différents processus du système des points de reprise de manière totalement indépendante. La cohérence des états

globaux formés n'est donc pas assurée et c'est en cas de panne que le protocole recherche une ligne de recouvrement parmi tous les états globaux formés. Cette recherche s'effectue généralement par la création d'un graphe de dépendance entres les points de reprise locaux [WAN 97, BAL 95, HéL 99b].

L'inconvénient majeur de cette méthode est le risque d'effet domino [RAN 75], puisque rien n'assure la cohérence pendant l'exécution. L'effet domino est caractérisé par une cascade de retours arrière lors de la reprise du système après une panne. Considérons la figure 2.4 : lors de la panne du processus P (survenant au point $\times$), le protocole va rechercher une ligne de recouvrement en partant de l'état global le plus récent, c'est à dire celui formé par les points de reprise C_P^3,C_Q^3 et C_R^3. A cause du message orphelin m_6, cet état n'est pas cohérent : on va donc faire un retour arrière sur R, jusqu'au point C_R^2. Mais l'état global ainsi formé n'est toujours pas cohérent à cause du message m_5 qui est orphelin. De la même manière, le protocole va devoir reculer jusqu'à la ligne de recouvrement (formée de C_P^1,C_Q^1 et C_R^1) et donc perdre une grande quantité de travail.

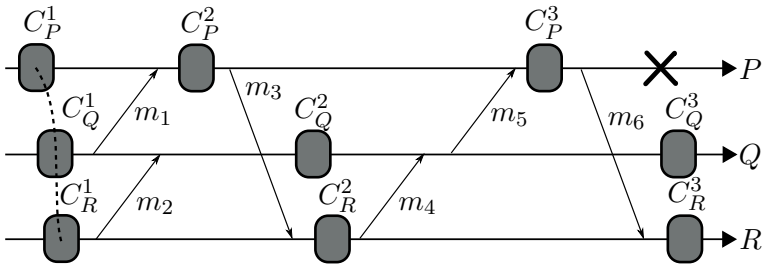

FIG. 2.4 – Effet domino

2.4.2 Synchronisés

Les points de reprises sont réalisés de manière *coordonnée* sur tous les processus de l'exécution, de façon à assurer que l'état global résultant soit cohérent. Cette synchronisation peut être :

- bloquante : la cohérence de l'état global est assurée par le fait que tous les processus sont stoppés lors de la prise de cet état global [SAN 05]. Cette approche est aussi nommée *sync-and-stop* [PLA 93] ;
- non bloquante : la cohérence est assurée par des messages "balais", qui vident les canaux de communication afin d'éviter les messages orphelins. Selon les approches, les canaux de communication sont supposés FIFO [CHA 85] ou non [ELN 92a] ;
- sur horloge : les horloges de chaque machine sont synchronisées, et les processus déclenchent les points de reprise en fonction de leur horloge locale [LIN 03a].

Dans les deux premiers cas, ce type de protocole nécessite l'utilisation de messages additionnels. L'approche synchronisée permet d'éviter l'effet domino puisque tous les états globaux crées sont forcément cohérents.

Une approche synchronisée est souvent choisie *en pratique* à cause de sa simplicité d'implémentation et d'intégration à un système existant. Par exemple, c'est la solution qui a été choisie dans [ZHE 04] pour Charm++, un intergiciel basé sur des objets C++ concurrents [KAL 93]. Le système CoCheck [STE 96], une extension du mécanisme de persistance de Condor assurant la cohérence des états globaux, se base aussi sur le protocole de Chandy et al. [CHA 85].

2.4.3 Induits par message

La synchronisation se fait ici de manière "paresseuse", en utilisant les messages de l'application. Chaque processus prend régulièrement des points de reprise de manière indépendante. Cependant, en fonction des messages reçus et envoyés et des informations qui sont estampillées sur ces messages (*piggybacking*), le processus devra peut-être prendre un point de reprise additionnel ; on dit que cette prise est *forcée*. L'algorithme de décision assure (ou maximise la probabilité) que le point pris fasse partie d'un état global cohérent, et donc qu'il existe une ligne de recouvrement toujours *assez* récente.
On distingue deux familles de protocole :

- les protocoles *model-based* [HéL 00, HéL 99a] : les communications et les prises de points de reprise doivent respecter un certain motif. Par exemple, si tout envoi et toute réception de message est précédé d'un point de reprise, alors tous les points appartiennent au moins à *un* état global cohérent, et le dernier point pris fait toujours partie de la ligne de recouvrement.
  Ces protocoles sont généralement basés sur les notions de chemins et de cycles "zigzag" définis dans 2.3.2 pour déterminer les états globaux cohérents [MAN 99].
- les protocoles *index-based* [BRI 84, MAN 96] : les points de reprise sont indexés, chaque message porte l'index du dernier point de l'émetteur. Sur réception d'un message indiquant un index supérieur au sien, le récepteur doit prendre un point de reprise *avant* la prise en compte du message : l'incohérence est évitée "au dernier moment".
  Ce type de protocole utilise donc une méthode basée sur la suppression des dépendances causales entre les points de reprise des processus qui ont des communications directes.

2.4.4 Analyse comparative des différentes méthodes de point de reprise

Nous allons comparer ici les approches présentées dans la partie précédente. Nous ne considérerons que les approches synchronisées et induites par messages ; en effet, l'approche indépendante n'implique pas réellement l'utilisation d'un protocole : la recherche d'un état global cohérent est faite *après* une panne, au moment de la reprise. Nous comparerons sur les points suivants [ELN 02, CON 98] :

1. le surcoût pendant l'exécution, en terme de CPU et de réseau,
2. le nombre de retours arrière moyen,
3. le surcoût de stockage, et la facilité du ramassage sur la mémoire stable des points de reprise devenus inutiles.

2.4.4.1 Surcoût pendant l'exécution

Les protocoles synchronisés bloquant ont en général un surcoût élevé durant l'exécution à cause de la phase de synchronisation qui stoppe l'exécution. De plus, le temps de blocage de l'exécution étant proportionnel au nombre de processus, ce type de protocole passe difficilement à l'échelle. Dans certains cas particuliers, l'approche bloquante peut cependant être intéressante : Coti et al. ont montré dans [COT 06] que l'approche bloquante était plus efficace que l'approche non bloquante dans le cas des petits systèmes avec des liens réseaux haute-performance.

Par rapport à une approche synchronisée, l'approche induite par messages ne rajoute pas de message additionnel durant l'exécution : le surcoût principal est dû aux points de reprise additionnels qui ont été forcés par le protocole durant l'exécution [ALV 99]. Il y a aussi un surcoût du à la *taille des messages* qui peut être très différent d'un protocole à l'autre, allant de sans surcoût pour les approches *model-based* à un vecteur d'entiers de la taille du nombre de processus dans le système [ZAM 98].

2.4.4.2 Nombre de retours arrière moyen

Les protocoles synchronisés ont ici un avantage certain : *tous* les points de reprise sont utiles, et lorsqu'un processus prend un point de reprise, il est assuré de ne pas devoir retourner plus loin que ce point en cas de panne.

Dans le cas des protocoles induits par messages, on trouve deux types de solution. La première est d'assurer que tous les points de reprise sont *utiles*, c'est à dire qu'ils font partie d'un état global cohérent. Ces protocoles sont basés sur l'approche *model-based*, et ont été classifiés par Manivannan dans [MAN 99], selon qu'ils assurent l'absence de chemin "zigzag" (*Strictly Z-Path Free (SZPF)*, *Z-Path Free (ZPF)*), ou l'absence de cycle "zigzag" (*Z-Cycle Free, ZCF*). La deuxième solution est de maximiser la probabilité que tous les points soient utiles, sans pour autant l'assurer. C'est le cas des protocoles *index-based*.

Cependant, avec ces deux solutions, le dernier point de reprise d'un processus ne fait pas toujours partie de la *dernière* ligne de recouvrement. Il est donc possible que le processus doive reprendre depuis un point de reprise plus ancien. A la différence d'une approche synchronisée, le travail potentiellement perdu par un processus en cas de panne n'est donc pas borné.

2.4.4.3 Surcoût de stockage et ramassage

Sur ce point aussi, les protocoles synchronisés sont avantageux : puisque tout état global nouvellement créé est forcément un état cohérent, alors l'état global précédent peut être effacé de la mémoire. Le surcoût est donc minimal puisqu'il n'est nécessaire de conserver qu'un seul état global (plus bien sûr celui en construction), et le ramassage des états devenus inutiles devient trivial [SIL 99a].

Le ramassage des points de reprise dans le cas des protocoles induits par messages est dépendant de la politique choisie, et nécessite le plus souvent un mécanisme complexe pour déterminer les points devenus inutiles.

2.5 Tolérance aux pannes par journalisation

Les états recouvrables créés par les protocoles de tolérance aux pannes par journalisation ne sont pas directement des états globaux cohérents de l'application. Ce type de protocole sauvegarde suffisamment d'informations pour pouvoir contrôler la réexécution de manière à amener cette réexécution dans un état cohérent. De manière générale, l'histoire de l'exécution de chaque processus est journalisée, c'est-à-dire sauvegardée, durant l'exécution sans panne. En cas de panne d'un processus, son exécution est relancée et est contrôlée jusqu'à ce que le processus atteigne l'état dans lequel il était juste avant la panne. Le système retrouve alors un état cohérent.

Comme l'histoire du processus doit pouvoir être journalisée, les protocoles par journalisation reposent sur l'hypothèse de *déterminisme par morceaux* (*piecewise determinism*) [STR 85]. Sous cette hypothèse, chaque exécution locale peut être vue comme séquence d'intervalles d'états, c'est-à-dire une séquence de réductions d'états par des évènements déterministes séparés par des réductions par des évènements non déterministes. En pratique, cette hypothèse signifie que tous les évènements non déterministes sont *observables* durant l'exécution, et peuvent donc être journalisés par un mécanisme extérieur.

Dans une exécution distribuée, les réceptions de messages sont vus comme des évènements non déterministes ; c'est pour cela que l'approche par journalisation s'appelle traditionnellement *journalisation des messages*. Les protocoles considèrent généralement les réceptions de messages comme les seuls évènements non déterministes de l'exécution. Les solutions proposées peuvent être étendues si l'on considère l'existence d'un mécanisme qui peut observer et journaliser le contenu des évènements non déterministes d'autre type comme les tirages aléatoires. On décrira par la suite les approches possibles en terme de réceptions de message, mais chacune de ces approches peut s'appliquer pour tout évènement non déterministe.

Le principe de base des approches par journalisation est de sauvegarder en mémoire stable les messages qui transitent entre les processus, et, occasionnellement et de façon totalement indépendante, prendre un point de reprise et le sauvegarder en mémoire stable. L'ensemble point de reprise et journal des messages reçus représente donc un état recouvrable pour le processus concerné. En cas de panne de ce processus, il sera relancé depuis son point de reprise le plus récent, puis tous les messages entrant enregistrés depuis la prise de ce point lui seront renvoyés. Si les messages sont enregistrés par le récepteur, on parle de *journalisation basée sur la réception*. L'alternative est la *journalisation basée sur l'émission* [JOH 87] : dans ce cas, c'est l'émetteur qui enregistre tous les messages sortant. En cas de panne, un processus devra demander à tous les processus avec qui il a communiqué la réémission de ces messages.

On peut distinguer trois approches différentes dans la tolérance aux pannes par journalisation [ALV 98] : *pessimiste*, *optimiste* et *causale*.

2.5.1 Journalisation pessimiste

La journalisation pessimiste [BOS 02] est une approche qui enregistre sur une mémoire stable de façon synchrone tous les messages en transit dans le système. Tout message, une fois arrivé dans le contexte du récepteur est forcément enregistré, et pourra donc être rejoué en cas de panne : ainsi, tous les points de reprise du processus sont utilisables pour une reprise, puisque *tous* les événements non déterministes ont été enregistrés depuis la prise de ce point. L'ordre de ces messages doit lui aussi être conservé, de manière à recréer lors de la reprise la même histoire de message.

2.5.2 Journalisation optimiste

Dans le cas de la journalisation optimiste [STR 88], les messages reçus ne sont pas forcément enregistrés immédiatement sur une mémoire stable afin d'améliorer les performances. L'enregistrement des messages en transit dans le système se fait de façon asynchrone, de manière à grouper les enregistrements sur mémoire stable qui représentent un surcoût considérable. Les messages sont conservés en mémoire volatile avant un enregistrement sur mémoire stable dont le moment dépend du protocole. Ces protocoles font donc l'hypothèse optimiste que ces enregistrements seront terminés avant qu'une panne n'arrive, de façon à ce que tous les événements puissent être rejoués en cas de panne d'un processus.

Ils doivent cependant gérer le cas où une panne survient avant que tous les messages ne soient enregistrés : les informations enregistrées en mémoire volatile du processus tombé en panne sont alors perdues, et le processus fautif ne peut plus rejouer une exécution équivalente après sa reprise. Dans ce cas, les messages qu'il a envoyé après la dernière sauvegarde sur mémoire stable deviennent des messages orphelins, puisqu'ils ne seront peut être jamais envoyés dans la nouvelle exécution. On dit que les processus qui ont reçus ces messages deviennent à leur tour des processus *orphelins* ; une fois orphelin, un processus rend orphelin tous processus à qui il envoie un message.

Ces processus orphelins ne sont plus dans un état cohérent avec le reste du système, et doivent donc eux aussi reprendre. Un état recouvrable est donc constitué de l'ensemble des points de reprise et journaux de messages de tous les processus orphelins de l'exécution.

Regardons par exemple la figure 2.5, et supposons que l'on utilise un protocole optimiste basé sur la réception. Si R tombe en panne avant que $m4$ soit enregistré (mais après émission du message $m5$) puis reprend depuis le point de reprise C_R, Q devient alors orphelin et doit reprendre en C_Q pour défaire la réception de $m5$.

Les processus doivent alors maintenir des informations sur les relations causales entre eux, pour pouvoir déterminer, *au moment de la reprise*, quels sont les processus qui doivent reprendre pour que l'état global reste cohérent.

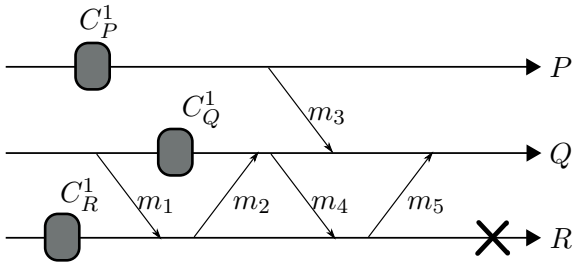

FIG. 2.5 – Le processus Q doit lui aussi reprendre si $m4$ n'a pas été journalisé

2.5.3 Journalisation causale

La troisième méthode de journalisation est la journalisation causale ou répartie [ELN 92b]. Comme dans la journalisation optimiste, les messages ne sont pas forcément journalisés immédiatement sur une mémoire stable. Pourtant, la panne d'un processus ne peut pas rendre orphelin un autre processus du système : Alvisi et Marzullo ont montré dans [ALV 98] que si tous les processus dont l'état dépend causalement de la réception d'un message m possèdent une copie de m, alors il ne peut pas y avoir de processus orphelin en cas de panne. En fait, ils montrent que seul le *déterminant* du message, c'est à dire l'ensemble <*source, destination, numéro de séquence d'émission, numéro de séquence de réception*>, est primordial pour la correction du protocole ; le contenu d'un message est supposé disponible à partir de son déterminant.

Le principe est le suivant : tous les processus délivrant un message dont l'envoi dépend causalement de la remise de m doivent journaliser une copie du déterminant de m, jusqu'à ce que un nombre f de processus possèdent une copie de ce déterminant ou jusqu'à qu'il soit journalisé sur un support stable. Pour ce faire, les déterminants des messages précédant causalement m, c'est à dire son *histoire*, sont estampillés sur m. Le nombre f est un paramètre du protocole et représente le nombre maximal de pannes simultanées supporté par ce protocole.

La difficulté majeure de ce type de protocole est le calcul par un processus de l'ensemble des processus ayant journalisé m à un instant donné ; cet ensemble est toujours *estimé* par un processus [ALV 02]. Il permet aux processus de savoir si le déterminant de m est *stable*, c'est à dire journalisé dans au moins f processus du système : dans ce cas, il n'est plus nécessaire de le journaliser et de l'ajouter sur les prochains messages.

2.5.4 Analyse et comparaison des différentes méthodes de journalisation

Nous allons comparer ici les trois approches de la journalisation selon les critères suivants :

1. le nombre de processus devant reprendre en cas de panne,
2. le surcoût en terme de temps d'exécution,
3. le surcoût en terme de taille des messages.

2.5.4.1 Nombre de processus à reprendre

C'est là le point fort des protocoles de journalisation pessimiste et causale : lors de la panne d'un processus, ces protocoles assurent que seul le processus fautif devra reprendre depuis son dernier état enregistré. Les messages étant réémis, soit par un processus extérieur dans le cas de l'approche pessimiste basée sur la réception, soit par les autres processus de l'application dans le cas de celle basée sur l'émission, les autres processus n'ont pas à recouvrir un état antérieur.

Dans le cas des protocoles optimistes, le nombre de processus qui reprennent après une panne peut être égal dans le pire des cas au nombre de processus du système. En fait, ce nombre dépend du maillage et du taux de communication de l'application : dans le cas d'une application fortement maillée où les processus communiquent très souvent, la panne d'un processus peut entraîner la reprise de *tout* le système.

2.5.4.2 Surcoût

Les protocoles pessimistes ont un coût élevé à l'exécution. Tout message envoyé doit être journalisé de façon synchrone avant d'être reçu, ce qui en moyenne va doubler les temps de communication [RAO 00].

Les protocoles optimistes ont eux un surcoût à l'exécution "réglable". En effet, il est possible d'adapter la fréquence des enregistrements sur support stable en fonction de la fréquence des pannes et du surcoût maximal accepté durant une exécution sans panne. Bien sûr, des enregistrements peu fréquents entraînent un temps de reprise en cas de panne important, puisque plus les intervalles entre enregistrements sont grands, plus les processus ont le temps de communiquer. Dans ce cas, un processus orphelin peut rendre orphelin un grand nombre d'autres processus.

Les protocoles de journalisation causale ont un surcoût à l'exécution généralement plus faible que les deux autres approches, en particulier ceux qui n'utilisent pas de mémoire stable et qui répartissent les informations de recouvrement dans le système lui-même. Mais pour obtenir ce faible coût, un compromis est fait au niveau de la reprise du système, qui est d'abord très complexe, et qui nécessite parfois des communications entre le processus en cours de reprise et *tous* les autres processus du système [ELN 92b]. Un compromis est proposé dans [BOU 03a] avec l'utilisation d'une mémoire stable : les déterminants des messages sont journalisés de manière asynchrone sur une mémoire stable plutôt que d'être répartis sur les processus de l'application. Les auteurs ont montrés dans [BOU 05] que cette approche permet de diminuer le temps de reprise mais aussi le surcoût à l'exécution.

2.5.4.3 Taille des messages

Les protocoles pessimistes n'ajoutent aucune information supplémentaire sur les messages, et n'ont donc aucun impact sur la taille des messages.

Les protocoles optimistes doivent conserver assez d'informations pour pouvoir identifier les processus qui ont communiqué avec le processus fautif lors d'une panne : la plupart des propositions utilisent des vecteurs dont la taille est égale

au nombre de processus du système (dans le meilleur des cas, ces vecteurs sont des vecteurs d'entiers). Ils doivent être ajoutés à tous les messages en transit dans le système, ce qui peut augmenter considérablement le temps d'émission mais aussi de traitement des messages.

Enfin, les protocoles de journalisation causale ajoutent eux aussi des informations sur les messages puisque chaque message porte avec lui son histoire causale. La taille de cet historique est dépendante de la technique utilisée dans le protocole : si on utilise une journalisation sur un support stable, elle dépend de la fréquence d'enregistrement. Si il n'y a pas de support stable, cette taille est proportionelle à f, le nombre de fautes simultanées supporté par le protocole.

2.6 Analyse comparative globale

Nous proposons ici une analyse comparative des protocoles de points de reprise et des protocoles de journalisation en fonction de certaines caractéristiques du système et de l'application. Nous détaillons ici quatre critères qui nous semblent déterminants et qui sont, hormis la fréquence des pannes, dépendants de l'application.

2.6.1 Fréquence des pannes

La valeur caractéristique de la fréquence des pannes dans un système distribué est le MTBF (*Mean Time Between Failures*), c'est à dire le temps moyen entre deux pannes consécutives de n'importe quelle ressource physique du système. Cette valeur varie d'une minute pour les systèmes constitués de stations de travail [BOL 00], à deux semaines pour une grappe de 1024 machines dédiée au calcul haute-performance [LIN 03b].

Dans le cas d'un système où le MTBF est très petit, de l'ordre de quelques minutes, il faut privilégier la rapidité de la reprise. On choisira alors une approche par journalisation de message, dans laquelle généralement[1] seul le processus fautif doit reprendre. Lemarinier et al. ont montré dans [LEM 04] que le surcoût plus élevé des protocoles par journalisation de messages par rapport à une approche par points de reprise synchronisés était compensé par le temps de recouvrement lorsque le MTBF devient petit. En effet, les protocoles de points de reprise nécessitent une reprise de tout le système. Dans un cas extrême, si le temps moyen entre deux pannes est inférieur au temps de reprise globale, le système pourrait alors rester indéfiniment dans l'état de reprise.

Il faut exclure les protocoles ne supportant pas les pannes simultanées (ou seulement k pannes simultanées, où k est un paramètre du protocole), comme certaines approches par journalisation optimiste et causale.

[1] Dans le cas des protocoles optimistes, on a vu que la panne d'un processus pouvait entraîner la reprise de plusieurs processus.

2.6.2 Nombre de processus

Les protocoles qui nécessitent la reprise de tous les processus en cas de panne de l'un d'entre eux sont mal adaptés aux systèmes comportant un grand nombre de processus. Les protocoles de journalisation des messages sont donc plus adaptés aux applications de grande taille que les protocoles par points de reprise, à l'exception de certaines approches par journalisation causale, qui induisent des communications d'un processus vers tous les autres lors de la reprise.

2.6.3 Taux de communication, maillage

Un taux de communication élevé va rendre l'utilisation de protocoles de journalisation très coûteuse pendant une exécution sans panne, surtout dans le cas d'une approche pessimiste. Ces protocoles nécessitent des accès fréquents à la mémoire stable, ce qui a pour effet, en plus du coût de la communication, de créer de la contention au niveau de cette mémoire.

De plus, l'espace de stockage nécessaire pour les points de reprise et les journaux de messages est important, en particulier pour les approches optimistes, pour lesquels le ramassage des points et des journaux devenus inutiles est complexe et nécessite souvent de conserver plusieurs points consécutifs par processus.

Une approche par journalisation causale, qui permet pourtant de minimiser l'utilisation de la mémoire stable, peut aussi induire un surcoût important puisque la taille et la quantité des informations à maintenir durant l'exécution sont proportionnelles au taux de communication.

Les protocoles par points de reprise synchronisés sont insensible au taux de communication, puisqu'ils ne reposent que sur l'utilisation de messages spécifiques. Ils représentent donc une solution intéressante pour les applications fortement communicantes.

Les protocoles par points de reprises induits par messages sont eux influencés par le taux de communication. Le comportement de ce type de protocole diffère selon les approches choisis. Dans [ALV 99], Alvisi et al. montrent en particulier que les solutions *index-based* génèrent un grand nombre de points de reprise forcés lorsque le taux de communication est important ; le surcoût induit par l'utilisation de tels protocoles devient alors prohibitif.

2.6.4 Indéterminisme

Si le système contient des processus indéterministes, alors les protocoles de journalisation de messages ne peuvent pas être utilisés. En effet, si le comportement de certains processus est la conséquence d'actions indéterministes internes qui ne sont pas observables, ils ne sont alors plus déterministes par morceaux, et deux exécutions avec la même histoire de messages peuvent être différentes. L'état global du système après la reprise d'un processus peut donc être incohérent. On devra alors choisir dans ce cas un protocole par points de reprise.

2.7 Causalité potentielle

Les notions de recouvrabilité et de cohérence d'état global sont donc des aspects essentiels de la tolérance aux pannes par recouvrement arrière. Elles reposent sur les relations de causalité entre les évènements, qui sont généralement exprimées par la relation de Lamport. Or, lorsque Lamport a défini sa relation dans [LAM 78], son but était de modéliser un temps logique dans une exécution distribuée, faute de pouvoir distinguer le temps réel. Ce n'est que par la suite que Mattern a observé que cette relation temporelle était aussi une approximation des relations de causalité qui lient les évènements pendant l'exécution : $e \prec^{hb} e'$ signifie aussi que l'exécution de e peut *potentiellement* affecter l'exécution de e'.

2.7.1 Existence d'une causalité potentielle

La relation de causalité de Lamport suppose que tous les évènements exécutés consécutivement sur le même processus sont causalement liés ; en effet, les évènements locaux sont totalement ordonnés par $\prec^{hb}$. Dans la pratique, ce lien de causalité n'existe pas toujours. C'est ce que Tarafdar et al. ont appelé la *fausse causalité* dans [TAR 98b] : un évènement qui est exécuté avant un autre n'en est pas nécessairement la cause. Dans [TAR 98b], les auteurs font la distinction entre le temps logique introduit par Lamport, la causalité *potentielle* et la causalité *véritable*.

Deux évènements sont véritablement causalement liés, noté $e \prec^{true} e'$, si ils sont consécutifs dans le temps réel, et si l'effet de leurs exécutions dépend de l'ordre dans lequel ils sont exécutés. En pratique, la causalité véritable n'est pas observable. Une relation de causalité potentielle, notée $\prec$, est une approximation de la causalité véritable. Une relation de causalité potentielle $\prec$ correcte pour une exécution $(E, \prec)$ doit vérifier les relations suivantes pour tout évènement e et e' :

$$e \prec^{true} e' \Rightarrow e \prec e'$$

$$e \prec e' \Rightarrow (e \prec^{true} e' \vee e \parallel e')$$

La relation de Lamport est la relation de causalité potentielle la plus générale possible puisque c'est une approximation correcte de la causalité véritable pour tous les systèmes possibles. D'abord, la relation causale entre l'envoi et la réception d'un message est évidente pour tout système physiquement réaliste. De plus, elle lie causalement tout couple d'évènements consécutifs dans le temps, et qui sont donc susceptibles d'être véritablement causalement liés. Pour cela, Lamport définit l'ordre entre les évènements consécutifs sur un même processus comme *total*. C'est donc aussi la relation la plus stricte possible et la moins précise : si l'ordre local total permet d'assurer $e \prec^{true} e' \Rightarrow e \prec e'$, il implique aussi que toutes les exécutions locales sont vus comme une suite purement séquentielle d'évènements dépendants.

Si la relation entre l'envoi et la réception d'un message semble immuable, l'ordre local peut lui être relâché et devenir un ordre *partiel* pour minimiser le nombre de relations de fausse causalité, et donc rendre la relation plus précise.

La figure 2.6 montre que la précision d'une causalité potentielle varie entre deux extrême, selon la rigueur de l'ordre locale qu'elle définit.

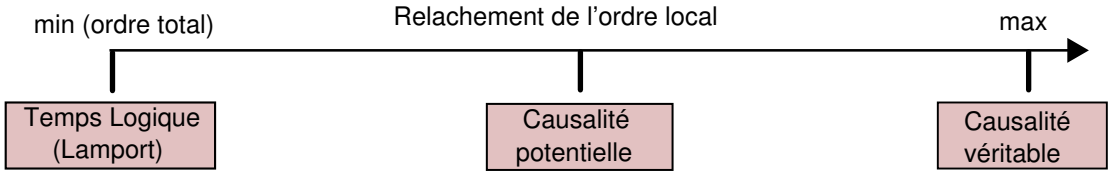

FIG. 2.6 – Relâchement de l'ordre local

En effet, certains évènements bien que locaux ne sont pas causalement liés. Prenons l'exemple d'un processus "mémoire", qui ne sert qu'à stocker une valeur ; il ne dispose que de deux types d'opérations `lireValeur()` et `ecrireValeur()`. Ces opérations peuvent être appelées par les autres processus. La réception d'un message de type `lireValeur()` déclenche l'envoi d'un message réponse contenant la valeur actuelle. La réception d'un message de type `ecrireValeur()` déclenche la modification de la valeur.

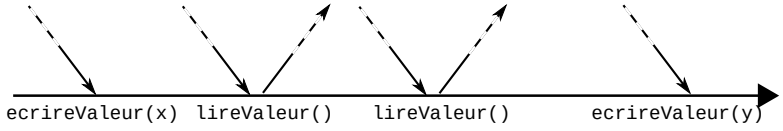

FIG. 2.7 – Exemple d'exécution d'un processus "mémoire"

Considérons l'exécution décrite par la figure 2.7 : la valeur renvoyée par les deux messages de type `lireValeur()` est la même puisqu'il n'y a pas eu d'exécution d'un `ecrireValeur()` entre temps. Ces deux couples d'évènements réception-envoi ne sont donc *pas causalement liés entre eux*.

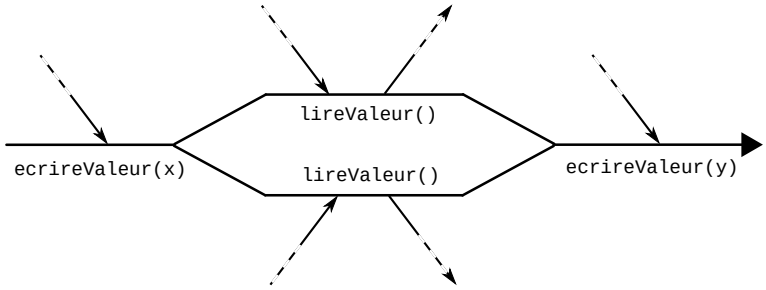

FIG. 2.8 – Branches causalement indépendantes d'un processus "mémoire"

En fait, on peut voir cette même exécution comme l'exécution de deux branches causalement indépendantes sur le même processus, comme représenté par la figure 2.8. Considérer une causalité potentielle plus précise que la relation de Lamport, et donc un ordre local de causalité partiel revient à considérer l'existence de plusieurs branches d'exécution causalement indépendantes dans un même processus. L'exécution locale peut donc être vue comme un graphe orienté acyclique

(DAG) d'évènements.

L'existence de relation de fausse causalité est plus évidente dans les systèmes formés de processus à fils d'exécution multiples (*multi-thread*) [DAM 99]. Les multiples fils d'exécution qui constituent un processus peuvent alors être vus comme plusieurs "sous-processus" communiquant entre eux par messages. L'existence d'un ordre de causalité locale partiel sur ces processus à fils d'exécution multiples devient alors évidente ; l'exécution de chaque processus est un ensemble d'évènements partiellement ordonnés par la relation de Lamport.

Si la représentation en plusieurs sous-processus peut être une solution intéressante dans le cas des processus à fils d'exécution multiples, elle ne l'est pas dans les autres cas. En effet, il est difficile dans la pratique manipuler le processus comme plusieurs sous-processus, et par exemple être capable de faire des images d'état indépendantes pour chacun de ces sous-processus.

2.7.2 Pourquoi une causalité potentielle ?

Éviter les liens de fausse causalité peut être utile dans différents domaines tels que le *debugging* distribué [MIT 00], la détection dynamique d'erreur de synchronisation [TAR 98a], mais aussi dans le domaine qui nous intéresse ici : la tolérance aux pannes par recouvrement arrière.

En effet, considérer une causalité potentielle plus précise augmente le nombre d'exécutions équivalentes, puisque certains évènements locaux qui ne sont pas causalement liés peuvent être exécutés dans n'importe quel ordre. De manière générale, le nombre d'évènements à ordonner de façon stricte pour assurer l'équivalence d'une réexécution diminue : dans [SIL 99b], les auteurs utilisent les spécificités de l'ordonnancement des messages dans une application pour réduire le nombre de réceptions à journaliser. Par exemple, dans la figure 2.7, les deux réceptions de `lireValeur()` n'ont pas à être journalisées de façon stricte entre elles. Grâce à cette réduction, les auteurs proposent un protocole de validation d'état global (*output-commit*) efficace.

L'utilisation de la sémantique des messages pour identifier les liens de fausse causalité entre les réceptions de messages est généralisée par Enokido et al. dans [ENO 98]. Les auteurs proposent un protocole qui assure l'ordonnancement causal des réceptions de message mais qui tient compte d'un ordre de *précédence significative des messages*. Ils montrent que certaines réceptions n'ont pas besoin d'être ordonnées sans pour autant violer l'ordonnancement causal ; ces réceptions sont liées par des liens de fausse causalité. Identifier ces liens leur permet de proposer un protocole plus efficace.

Dans le cadre de la tolérance aux pannes par points de reprise, un ordre de causalité potentielle plus précis permet de relâcher les conditions de cohérence d'un état global par rapport à une définition de la cohérence se basant sur la relation de Lamport. Par exemple, dans [LEO 94], les auteurs exploitent la sémantique des messages pour déterminer le dernier état recouvrable. Ils classifient les messages dans trois catégories : écriture, lecture et autres, et montrent que certains états,

bien qu'incohérents par rapport à la relation de Lamport, sont recouvrables selon la sémantique des messages orphelins. Nous verrons par la suite que le protocole proposé dans cette thèse tire parti de la même manière d'un ordre de causalité potentielle plus précis que la relation de Lamport.

2.8 Conclusion

Nous avons présenté dans ce chapitre les trois aspects essentiels de la tolérance aux pannes par recouvrement arrière : la mémoire stable, la persistance et la recouvrabilité d'état. Ce dernier point se reposant principalement sur les relations de causalité entre les évènements, nous avons présenté le concept de *causalité potentielle*. Une relation de causalité potentielle pour une exécution distribuée est une relation qui évite certains liens de fausse causalité entre les évènements en relâchant l'ordre de causalité local, considéré comme un ordre total dans la relation de Lamport. Nous avons vu que considérer une relation de causalité potentielle plus précise était utile dans le cadre de la tolérance aux pannes par recouvrement arrière ; cela permet par exemple de relâcher les conditions de cohérence donc de recouvrabilité d'un état global.

Nous avons présenté les deux grandes familles de protocoles de tolérance aux pannes par recouvrement arrière, i.e. par points de reprise et par journalisation des messages, et avons montré qu'il n'existe pas de solution unique qui soit adaptée à toutes situations. Selon les propriétés de l'infrastructure physique utilisée mais aussi selon les propriétés de l'application, une solution donnée peut engendrer de mauvaises performances, voir même être totalement inapplicable.

C'est cette constatation qui va motiver le développement d'un protocole par points de reprise adapté à notre contexte de travail présenté dans le chapitre suivant. Nous allons voir en effet que si les protocoles de journalisation de messages peuvent facilement s'appliquer dans ce contexte, les protocoles par points de reprise coordonnés ou induits par message classiques ne peuvent pas s'appliquer directement. Il nous faut donc proposer une solution par points de reprise adaptée ; cette solution est présentée dans le chapitre 4.

Chapitre 3

Contexte et Analyse

Nous décrivons dans ce chapitre le contexte de notre travail : le modèle à objets communicants ASP [CAR 04b], et son implémentation en Java ProActive [CAR 06c]. Le but de ce chapitre est d'identifier clairement les contraintes induites par le modèle et son implémentation, et leurs conséquences dans le cadre d'un protocole de tolérance aux pannes par point de reprise. Ces contraintes doivent être respectées par le protocole de tolérance aux pannes, mais aussi assurées en cas de panne et de reprise du système. Nous identifions de plus les propriétés qui pourront être exploitées dans ce cadre.

3.1 Le modèle : ASP

ASP pour *Asynchronous Sequential Processes* (Processus Séquentiels Asynchrones) est un calcul d'objet basé sur la notion d'activités communicantes. Les activités s'exécutent en parallèle, mais les opérations au sein d'une activité sont séquentielles.

3.1.1 Activités

De façon simplifiée, une *activité*[1] est un processus unique (*thread* d'exécution) ainsi qu'un ensemble d'objets. Parmi ces objets, un est dit *actif*, et les autres sont dits *passifs*. L'objet actif est le point d'entrée de l'activité : les autres activités ne peuvent référencer que cet objet actif. De manière générale, le modèle ASP ne permet pas le partage des objets passifs entre activités ; les seuls objets qui peuvent avoir une référence sur un objet passif d'une activité sont les objets appartenant à cette même activité. Un objet passif d'une activité ne peut avoir de référence sur un objet passif d'une autre activité, mais peut avoir une référence vers une autre activité.

Propriété 3.1.1.1 (Pas de partage de mémoire) *Les activités ne partagent pas de mémoire ; elles ne peuvent communiquer entre elles que par message.*

[1] Nous utiliserons à partir de maintenant le terme activité pour désigner l'unité d'exécution locale d'une exécution répartie

La Figure 3.1 donne un exemple de topologie d'objets dans une configuration formée de trois activités, représentées par les zones grises. Les objets sont représentés par des ellipses, les références par des flèches et les objets actifs par des ellipses en gras. Sur cette figure, l'absence de partage de mémoire est représenté par le fait que les seules flèches d'une activité à une autre sont des flèches en gras pointant vers des objets représentés par des ellipses en gras (référence vers un objet actif).

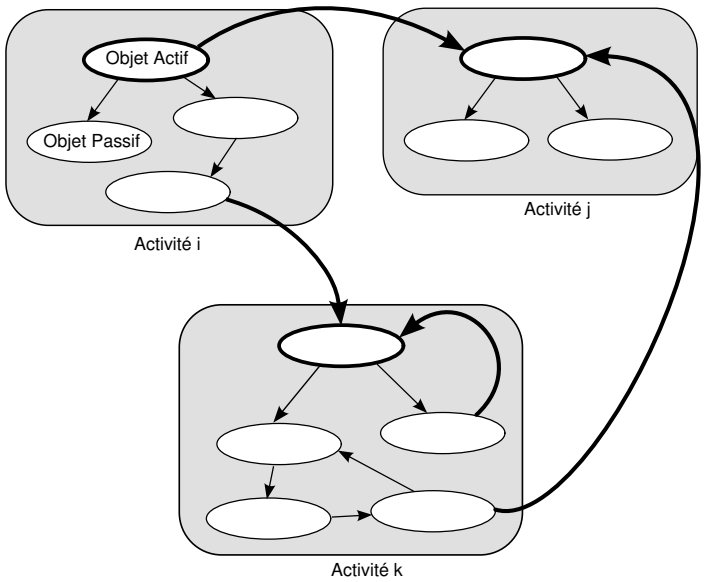

FIG. 3.1 – Exemple de topologie possible

3.1.2 Communications

Les activités communiquent entre elles par appels de méthode distante sur l'objet actif (*RPC, Remote Procedure Call*). Ces appels sont effectués sous la forme d'envois de messages de deux types uniquement : les *requêtes* et les *réponses*. Une requête correspond à un appel de méthode fait sur une activité, et une réponse correspond au retour de cet appel de méthode. Les appels de méthodes peuvent être de deux types :

- synchrones : l'appelant envoie la requête à l'activité cible et stoppe son exécution jusqu'à ce que le message réponse correspondant soit reçu ;
- asynchrones : l'appelant envoie la requête à l'activité cible et stoppe son exécution jusqu'à ce que cette requête soit effectivement *reçue* par l'activité cible, puis continue son exécution sans attendre le message réponse correspondant.

Même dans le cas d'un appel de méthode asynchrone, l'envoi de méthode lui même n'est pas asynchrone : un *rendez-vous* est conservé au niveau de l'envoi de

message. Une activité ne peut continuer son exécution que si le message envoyé a bien été reçu.

Cette attente systématique a pour conséquence l'ordonnancement causal des réceptions de messages : une réception de message ne peut avoir lieu sur une activité que si tous les messages qui le précèdent causalement ont déjà été reçus.

Contrainte 3.1.2.1 (Ordonnancement causal) *Les réceptions de messages sont causalement ordonnées.*

Cette propriété est très importante pour les applications distribuées car elle permet de garantir une certaine cohérence. En garantissant qu'une conséquence ne pourra jamais précéder sa cause, l'ordonnancement causal facilite le développement d'applications distribuées dans lesquelles il est souvent difficile de contrôler la cohérence de l'exécution.

De plus, le rendez-vous au moment d'une communication assure aussi à l'envoyeur que son message a bien été reçu. On peut déterminer à tout moment si un message a été ou n'a pas été reçu : l'envoi de message est donc atomique.

Propriété 3.1.2.1 (Envoi atomique) *Les envois de messages sont atomiques.*

Lorsqu'un appel de méthode est asynchrone, un *futur* est créé du côté de l'appelant ; ce futur est un substitut du résultat d'une méthode qui n'a pas encore été retourné. Un appel de méthode asynchrone de l'activité i sur l'activité j se déroule de la façon suivante :

- Une activité i envoie une requête à une activité j ; celle-ci est stockée dans la queue des requêtes en attente du côté de l'appelé j, et un futur lui est associé du coté de l'appelant i (étape 1 de la figure 3.2).
- Plus tard, cette requête est *servie* par l'activité j, c'est-à-dire prise dans la queue des requêtes pour être évaluée. Elle devient ainsi la *requête courante*. (étape 2 de la figure 3.2).
- Une fois que l'évaluation de cette requête est terminée, un message réponse est créé avec le résultat de l'évaluation, et est envoyé à l'activité i. Lorsque l'activité i reçoit cette réponse, le futur associé est mis à jour, c'est-à-dire que la référence au futur est remplacée par la valeur du résultat associé (étape 3 de la figure 3.2).

On voit ici qu'une activité doit donc contenir le futur correspondant pour être capable de recevoir une réponse. La conséquence est que de manière générale, une activité n'est pas toujours dans un état qui lui permet de recevoir une réponse : l'activité doit d'abord avoir créé le futur, donc avoir envoyé la requête correspondant à la réponse.

Contrainte 3.1.2.2 (Réception des réponses) *Une réponse ne peut être reçue par une activité qu'après avoir envoyé la requête correspondant à cette réponse.*

Un futur est soumis à un mécanisme de synchronisation locale simple, appelé *attente par nécessité*. Après un appel asynchrone et la création du futur correspondant, l'exécution du programme continue jusqu'à ce qu'une opération stricte sur

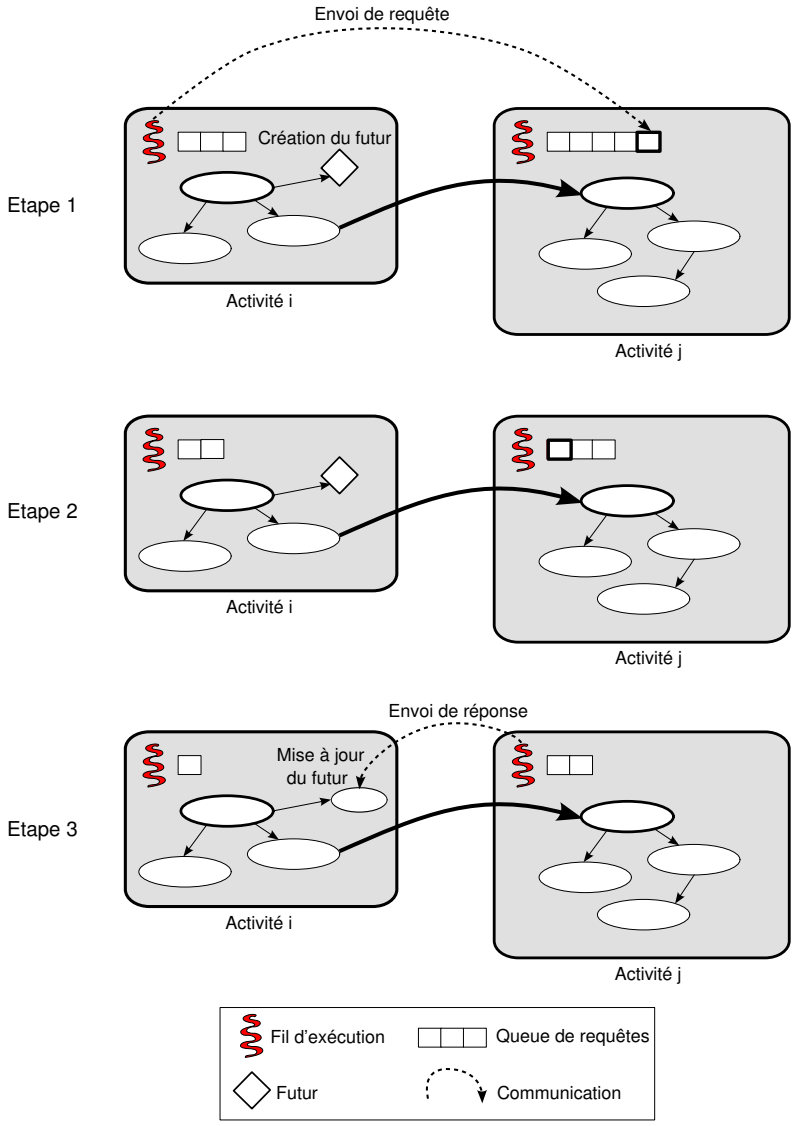

FIG. 3.2 – Appel asynchrone de i sur j

le futur, c'est-à-dire un accès à un champ de l'objet ou un appel de méthode sur l'objet, soit rencontrée. Dans ce cas, si le futur a déjà été mis à jour, l'opération est effectuée et l'exécution continue normalement, sinon l'exécution est bloquée en état d'attente par nécessité jusqu'à ce que le futur soit mis à jour. C'est cette attente par nécessité sur le résultat qui permet de synchroniser les différentes activités ; on parle de synchronisation orientée par les données.

Ce mécanisme de synchronisation particulier a permis à Caromel et Henrio dans [CAR 04b] de prouver que l'exécution d'un ensemble d'activités est déterminée de façon unique par l'ordre de réception des requêtes sur chaque activité.

Propriété 3.1.2.2 (Caractérisation d'une exécution) *Une exécution est* entièrement *caractérisée par l'ensemble des listes (une pour chaque activité) des identifiants d'activités ayant envoyé des requêtes.*

En particulier, Caromel et Henrio ont montré que les réceptions de réponses peuvent avoir lieu *n'importe quand* après l'envoi de la requête correspondante sans aucune conséquence observable sur le résultat de l'exécution.

3.1.3 Service des requêtes

La vie d'une activité consiste en une suite de service des requêtes qui sont stockées dans la queue des requêtes en attente. La politique de service de requêtes est par défaut FIFO (*First In First Out*, premier arrivé premier servi), donc les requêtes sont servies dans l'ordre dans lequel elles ont été reçues.

Lorsqu'une activité sert une requête, le fil d'exécution associé à cette activité évalue cette requête ; l'état de l'activité est alors potentiellement modifié par cette évaluation. De plus, l'avancement de l'évaluation est caractérisé par l'état courant du fil d'exécution. A ce moment, l'état de l'activité est donc caractérisé par trois éléments :

– l'état du fil d'exécution,
– l'objet actif et les objets passifs constituant l'activité,
– la queue des requêtes en attente.

Lorsqu'elle n'est pas en train de servir une requête, une activité est dite inactive. Dans ce cas, le fil d'exécution lié à cette activité est bloqué en attente d'une requête à servir ; son état n'est donc plus significatif. En effet, l'exécution passée est caractérisée par les modifications qui ont eu lieu sur l'activité, donc par l'état de l'activité, et l'exécution future par l'ensemble des requêtes qu'il reste à servir, donc par la queue des requêtes en attente. On dit que l'état de l'activité est *capturable*.

Propriété 3.1.3.1 (États capturables) *L'état d'une activité inactive est entièrement caractérisé par l'ensemble des objets constituant l'activité ainsi que par la queue des requêtes en attente.*

L'utilisateur peut modifier la politique de service d'une activité en faisant appel à une primitive de service explicite, $Serve(M)$, et déclencher un service *imbriqué*. M est suffisant pour identifier la requête qui doit être servie ; par exemple le nom d'une méthode. Lorsque l'utilisateur fait appel à cette primitive, le service de la requête en cours est stoppé, et la requête spécifiée par M est recherchée dans la queue des requêtes en attente, puis servie. Si la requête n'est pas présente, l'activité est mise en attente jusqu'à la réception d'une requête telle que spécifiée par M.

Donc, avant de déclencher un service de requête, une activité est inactive, sauf si ce service est un service imbriqué.

Propriété 3.1.3.2 (Occurence des états capturables) *Une activité est dans un état capturable avant chaque service de requête non imbriqué.*

3.1.4 Activités et déterminisme

L'évaluation du déterminisme d'une exécution distribuée est un point majeur du modèle ASP. La possibilité de modifier la politique de service de chaque activité peut permettre de contrôler l'exécution dans sa globalité. Par exemple, Caromel et Henrio ont défini dans [CAR 05] un sous-ensemble du calcul ASP, nommé DON pour *Deterministic Object Network*. Ce sous-calcul, en imposant des restrictions sur la topologie des activités ainsi que sur la politique de service des requêtes, définit une classe d'applications distribuées qui se comportent de manière *totalement* déterministe. Par exemple, une application communiquant uniquement à travers une topologie arborescente avec la politique de service FIFO se comporte de manière déterministe.

Le déterminisme des exécutions de DONs repose sur la propriété 3.1.2.2 sur les réceptions de réponse ; on considère dans ce cas que les seuls évènements non déterministes d'une exécution sont les réceptions de requête. Bien sûr, si une activité peut prendre localement des décisions non déterministes, la caractérisation d'une exécution doit tenir compte du résultat de ces décisions. La propriété 3.1.2.2 peut être modifiée pour tenir compte des évènements non déterministes locaux.

De manière générale, caractériser une exécution en fonction des évènements non déterministes suppose que tous les évènements non déterministes sont observables, donc que l'exécution d'une activité est déterministe par morceaux.

Propriété 3.1.4.1 (Déterminisme par morceaux) *L'exécution d'une activité est déterministe par morceaux.*

Si certaines topologies et politiques de service peuvent rendre une exécution distribuée déterministe, il va de soi qu'elles peuvent aussi accentuer l'indéterminisme de l'exécution. Par exemple, on pourrait définir pour chaque activité une politique de service *aléatoire* : l'activité servirait dans ce cas n'importe quelle requête se trouvant dans la queue des requêtes en attente. Le modèle ASP ne prend pas en compte ce type de comportement ; nous ne le considérons pas dans cette thèse.

3.1.5 Modélisation événementielle de ASP

La sémantique du modèle ASP, et en particulier l'utilisation d'un motif de communication requêtes/réponses avec service asynchrone des requêtes, induisent une relation de causalité locale *partielle*. En effet, certains évènements exécutés consécutivement sur une même activité peuvent être échangés sans modification observable du comportement de l'exécution. On parle d'évènements locaux compatibles. En fonction de ces relations de compatibilité, nous proposons donc dans cette section une relation de causalité potentielle pour ASP (section 2.7).

Il est important de noter que les relations de compatibilité entre les évènements sont définies entre les évènements locaux, et qu'elles découlent de la sémantique du modèle. Elles sont donc statiques, et permettent de définir une relation de causalité potentielle valable pour toutes les exécutions possibles en ASP, quelque soit l'application.

Propriété 3.1.5.1 (Causalité en ASP) *Une relation de causalité potentielle peut être définie statiquement pour toutes les exécutions possibles en ASP.*

Nous débutons cette section en définissant une classification des évènements ainsi qu'une notation adaptée à la définition d'une relation de causalité.

3.1.5.1 Notations

Nous distinguons dans une exécution ASP donnée $S \xrightarrow{e \dots e^n} S'$ sept types d'évènement :
- l'envoi ou la réception de requête,
- l'envoi ou la réception de réponse,
- le service de requête,
- le premier accès à un futur,
- enfin, l'évènement interne.

Dans l'ensemble des couples d'évènements de communication, noté Γ dans la relation de Lamport, les envois et réceptions de requête ou de réponse doivent être dissociés. Nous définissons donc deux ensembles distincts Γ_Q et Γ_R tels que :
- $\Gamma_Q \subseteq \{(e, e') \in \{e^1 \dots e^n\} * \{e^1 \dots e^n\}\}$ définit une bijection entre l'envoi et la réception correspondante de toutes les requêtes,
- $\Gamma_R \subseteq \{(e, e') \in \{e^1 \dots e^n\} * \{e^1 \dots e^n\}\}$ définit une bijection entre l'envoi et la réception correspondante de toutes les réponses,
- $\Gamma = \Gamma_Q \cup \Gamma_R$.

L'évènement de type service de requête caractérise le *démarrage* du service d'une requête. Ce type d'évènement est donc associé à la réception de la requête servie : $\Sigma \subseteq \{(e, e') \in \{e^1 \dots e^n\} * \{e^1 \dots e^n\}\}$ définit une bijection entre la réception d'une requête et le démarrage de son service.

L'évènement de type accès à un futur correspond au moment où une activité utilise *pour la première fois* le contenu de la réponse qui a mis à jour le futur. Ce type d'évènement est donc associé à la réception de la réponse correspondante au futur : $\varepsilon \subseteq \{(e, e') \in \{e^1 \dots e^n\} * \{e^1 \dots e^n\}\}$ définit une bijection entre la réception d'une réponse et le premier accès au futur correspondant à cette réponse.

Enfin, l'évènement interne correspond à une étape du service d'une requête, c'est à dire une étape élémentaire de l'évaluation d'une requête.

3.1.5.2 Causalité entre évènements dans ASP

Le service asynchrone des requêtes reçues, ainsi que le mécanisme d'attente par nécessité sur les futurs introduisent des relations de causalité particulières entre les évènements locaux sur une activité [DEL 04]. En effet, la réception d'une requête ne modifie que l'état de la queue des requêtes en attente tant qu'elle n'est pas servie. La réception d'une requête est donc compatible avec tous les évènements sauf les autres réceptions de requêtes tant qu'elle n'est pas servie. Bien sûr, elle n'est pas compatible avec son service.

Une réception de requête n'est donc pas causalement reliée aux évènements internes ou aux envois consécutifs tant qu'elle n'est pas servie. En revanche, l'ordre relatif des réceptions de requêtes est significatif : deux réceptions de requête consécutives sont causalement liées.

L'évènement de type premier accès à un futur est particulier : son exécution est gardée par le mécanisme d'attente par nécessité sur les futurs. Cet évènement ne peut avoir lieu que si la réception de la réponse correspondant au futur a déjà eu lieu. Donc la réception d'une réponse n'est pas compatible avec le premier accès au futur. De plus, l'exécution étant caractérisée uniquement par les réceptions de requêtes (propriété 3.1.2.2), une réception de réponse est compatible avec tous les autres évènements, sauf avec l'envoi de la requête correspondant à cette réponse (contrainte 3.1.2.2). En particulier, les réceptions de réponse sont compatibles entre elles.
Une réception de réponse n'est donc causalement liée qu'à l'envoi de sa requête correspondante, et au premier accès à son futur correspondant.

Enfin, on ne peut pas définir statiquement de relation de compatibilité entre les autres types d'évènements : ils sont donc tous causalement liés.

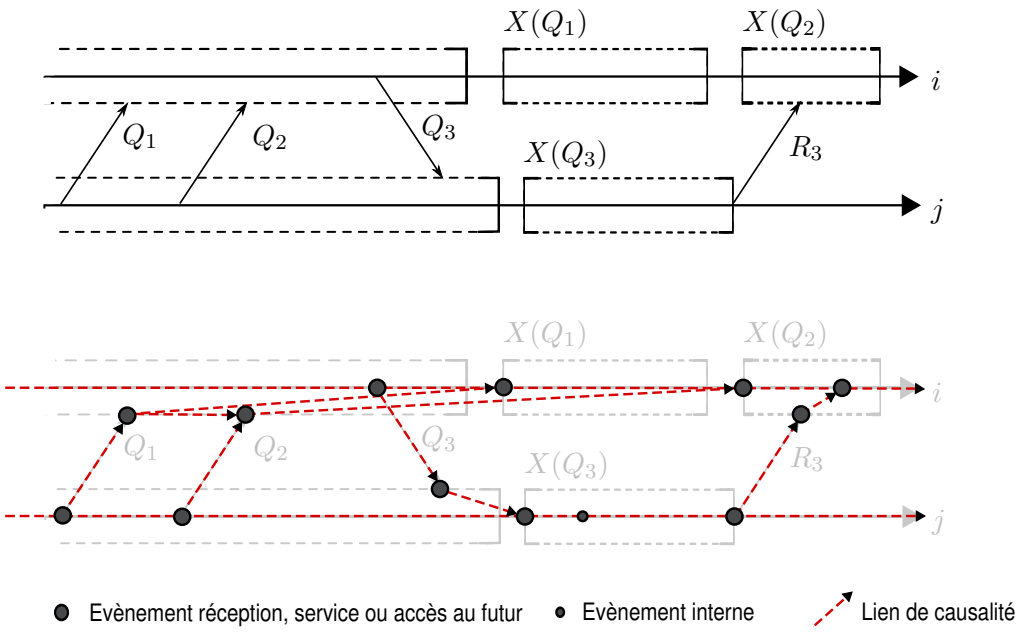

FIG. 3.3 – Exemple d'exécution distribuée ASP, et le graphe de causalité associé

Les relations causales partielles induites par le service de requêtes asynchrones sont illustrées par l'exemple de la figure 3.3. Cette figure présente une exécution avec deux activités i et j communicantes en haut, et le graphe de causalité associé à cette exécution en bas. On note Q_x une requête et R_x sa réponse correspondante, et $X(Q_x)$ le service de la requête Q_x. Le rectangle pointillé représente la durée du service d'une requête ; l'état de l'activité est donc capturable entre les rectangles

pointillés. Certains évènements internes peuvent être mis en valeur dans cette représentation si nécessaire.

On considère ici des activités avec une politique de service FIFO. L'activité j envoie les requêtes Q_1 et Q_2 à i, et i envoie la requête Q_3 à j. Le service des requêtes Q_1 et Q_2 ne nécessite pas de retour de résultat. Le résultat du service de Q_3 est renvoyé à i par la réponse R_3.

L'envoi de la requête Q_3 n'est pas causalement relié aux réceptions des requêtes Q_1 et Q_2 car il a lieu *avant* les services de Q_1 et Q_2, comme on le voit dans l'ordre causal correspondant à cette exécution. Le premier évènement sur i qui est une conséquence causale de la réception de Q_2 est le service de Q_2.

Enfin la réception de la réponse R_3 est une conséquence causale de l'envoi de la requête Q_3. Le premier évènement qui est une conséquence causale de cette réception est le premier accès au futur correspondant à R_3, qui dans cet exemple a lieu durant le service de la requête Q_2.

Ce graphique permet de constater visuellement que tous les évènements locaux à une même activité ne sont pas systématiquement causalement liés. Cette relation d'ordre partielle entre les évènements locaux est exprimée de manière formelle dans la définition d'un ordre de causalité local $\prec^{ASP}$ adapté au modèle ASP :

Définition 3.1.1 (Ordre causal local) *$\prec_i^{ASP}$ est un ordre* partiel *défini par $e \prec_i^{ASP} e'$ si et seulement si*

$$\begin{cases} (\nexists e^k, [(e^k, e) \in \Gamma \vee (e^k, e') \in \Gamma] \wedge e \prec_i^{hb} e') \vee & (a)\ \textit{évènements internes totalement ordonnés} \\ (\exists e^k, e^l, (e^k, e) \in \Gamma_Q \wedge (e^l, e') \in \Gamma_Q \wedge e \prec_i^{hb} e') \vee & (b)\ \textit{réceptions de requêtes totalement ordonnées} \\ (e, e') \in \Sigma \vee & (c)\ \textit{conséquence causale du service} \\ (e, e') \in \varepsilon \vee & (d)\ \textit{réception d'une réponse et accès au futur} \\ (\exists e^k, e \prec_i^{ASP} e^k \prec_i^{ASP} e') & (e)\ \textit{transitivité} \end{cases}$$

Enfin, la définition d'un ordre local permet de définir une relation de causalité potentielle qui lie tous les évènements d'une exécution distribuée en ASP :

Définition 3.1.2 (Causalité potentielle pour ASP) *$\prec^{ASP}$ est un ordre de causalité potentielle pour ASP : $e \prec^{ASP} e'$ si et seulement si*

$$\begin{cases} e \prec_i^{ASP} e' \vee & (a)\ \textit{ordre causal local} \\ ((e, e') \in \Gamma \vee & (c)\ \textit{causalité des communications} \\ (\exists e^k, e \prec^{ASP} e^k \prec^{ASP} e') & (c)\ \textit{transitivité} \end{cases}$$

3.2 L'implémentation : ProActive

ProActive est un intergiciel (*middleware*) Java pour la programmation concurrente, parallèle et distribuée. Il implémente le modèle d'activités communicantes défini par ASP. L'objectif de cette librairie est de proposer une interface de programmation (API) simple et compréhensible pour le développement et le déploiement d'applications distribuées sur différents environnements tels que les réseaux locaux (LAN), les grappes dédiées (*clusters*) ou encore les grilles de calcul.

De cet objectif général, on peut tirer deux axes majeurs dans le développement de ProActive :

- **transparence au niveau de la programmation** ; On privilégie la transparence pour l'utilisateur. Les mécanismes complexes liés à la distribution, à la concurrence ou à l'asynchronisme sont gérés au niveau de la librairie et ne font pas intervenir l'utilisateur. Par exemple, ProActive propose un mécanisme de communication collective qui permet à l'utilisateur de manipuler des références vers un *groupe* d'activités de la même manière qu'une référence vers une seule activité. La gestion du multiplexage lors d'un appel de méthode sur cette référence est entièrement gérée par la libraire [BAD 02].

- **visualisation des ressources** ; ProActive permet à l'utilisateur de considérer les ressources qui lui sont offertes pour déployer son application de manière uniforme, sans avoir à se soucier de leur nature, ni de la manière de les acquérir. Les ressources sont virtualisées au moment du déploiement, et l'hétérogénéité physique réelle est masquée par la libraire. Par exemple, la gestion des connexions entre les activités (ouverture et fermeture des connexions, choix du protocole adéquat,...) est entièrement réalisée au niveau de la libraire ; l'utilisateur ne voit au niveau de l'application qu'une référence vers une activité, quelle que soit l'infrastructure sous-jacente et le protocole d'accès.

La transparence des aspects non fonctionnels dans le développement d'applications distribuées est un sujet qui divise. Dans [WAL 94], les auteurs prônent une approche ouverte ; l'utilisateur doit par exemple savoir si un appel est distant ou non, ou encore il doit gérer lui-même les pannes dans l'application. L'inconvénient majeur couramment admis lié à la transparence est la performance : par exemple, la connaissance de la nature d'un appel (distant ou non) peut permettre de recouvrir au mieux la latence des appels.

Cependant, avec la complexification et la croissance des systèmes, en particulier dans le domaine du calcul sur grille, il n'est plus raisonnable de laisser l'utilisateur gérer tous les aspects non fonctionnels du système. On peut constater cette prise de conscience avec l'avènement de le programmation orientée *service*.

De plus, la tolérance aux pannes est un aspect non fonctionnel qui est particulièrement difficile à traiter au niveau de l'application. Silva et al. donnent dans [SIL 98] un exemple parlant. La transformation d'une application distribuée de *Scrabble* en application tolérante aux pannes a demandé un effort considérable

au programmeur : la gestion des pannes représente 50% du code source final, et a nécessité trois mois de correction de ce code. Un mécanisme de tolérance aux pannes totalement transparent a l'avantage considérable de pouvoir rendre tolérante aux pannes une application existante *développée sans tenir compte des pannes potentielles*, sans aucune modification du code ni recompilation.

Contrainte 3.2.0.1 (Transparence) *Les aspects liés à la tolérance aux pannes doivent être totalement transparents pour l'utilisateur.*

Enfin, la virtualisation des ressources qui est offerte par ProActive implique la portabilité totale du code ; si les ressources sont virtualisées en une seule ressource globale, le code applicatif doit pouvoir s'exécuter indifféremment sur n'importe laquelle des ressources physiques réelles constituant la ressource virtuelle globale.

Contrainte 3.2.0.2 (Portabilité) *Le code exécuté doit être totalement portable.*

3.2.1 Activités en Java

3.2.1.1 Choix du langage

La gestion de l'hétérogénéité étant un point majeur dans ProActive, le choix du langage d'implémentation s'est donc naturellement porté sur un langage s'exécutant sur une machine virtuelle. En effet, une machine virtuelle uniformise l'environnement d'exécution puisque l'application, compilée une seule fois, peut s'exécuter indifféremment sur diverses plates-formes. De plus, l'adéquation à la programmation réseau (RMI, téléchargement dynamique de classe,...), le support standardisé du *multi-threading* ainsi que sa disponibilité sur de nombreuses plates-formes ont fait de Java une solution intéressante pour le développement de la libraire ProActive.

Cette gestion de l'hétérogénéité offerte par Java doit donc être conservée dans ProActive. Par conséquent, les fonctionnalités offertes par ProActive ne doivent en aucun cas reposer sur des routines spécifiques à un système d'exploitation, ou encore sur une librairie ou sur une application compilée pour un système d'exploitation donné.

De plus, l'environnement d'exécution de Java devient disponible de façon quasi-standard sur les principales plates-formes et pour les principaux systèmes d'exploitation ; aucune fonctionnalité de ProActive ne doit nécessiter la modification de la machine virtuelle. En effet, une telle modification nécessiterait de déployer la machine virtuelle modifiée avant chaque déploiement ProActive, alors qu'il aurait été possible d'utiliser les environnements d'exécution potentiellement *déjà déployés* sur les ressources.

Contrainte 3.2.1.1 (Java standard) *L'implémentation doit se faire uniquement en Java standard, et ne doit reposer sur aucun code spécifique à un système d'exploitation, ni sur une modification de l'environnement d'exécution.*

3.2.1.2 Méthode d'implémentation

ProActive repose sur un protocole à méta-objet, qui utilise les possibilités de réflexivité offertes par Java [CAR 01], pour masquer les mécanismes complexes de gestion de la distribution, de la concurrence ou de l'asynchronisme. L'architecture de ce protocole à méta-objet est décrite par la figure 3.4.

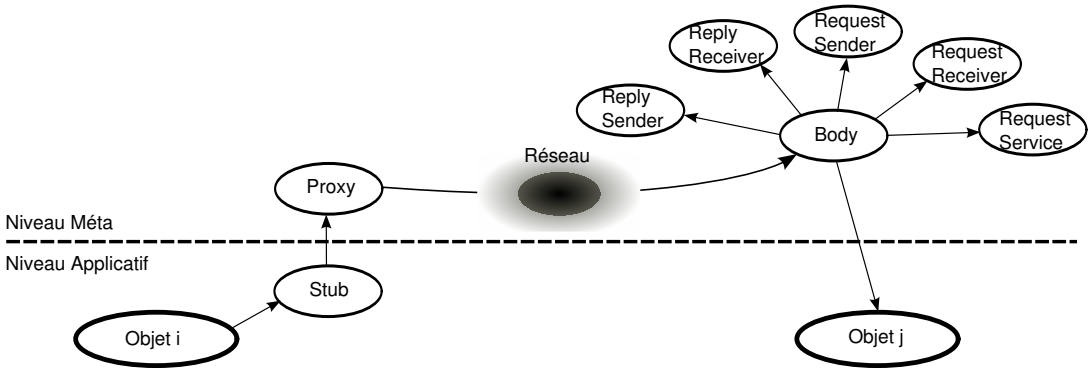

FIG. 3.4 – Architecture du protocole à méta-objet

Chaque objet défini au niveau méta gère un aspect de la vie d'une activité. À toute activité sont associés un *Proxy* et un *Body*. Le *Body* est le point d'entrée unique d'une activité ; toute communication vers ou en partance d'une activité passe par son *Body*. C'est la seule partie de l'activité visible et accessible à distance. Le *Body* sert de chef d'orchestre et gère le comportement de tous les autres méta-objets. Ainsi, à chaque action élémentaire de la vie d'une activité correspondant à la classification des évènements proposée dans la section 3.1.5 correspond un méta-objet spécifique :

- les réceptions de requête et de réponse sont gérées respectivement par le *RequestReceiver* et le *ReplyReceiver*,
- les envois de requête et de réponse sont gérés respectivement par le *RequestSender* et le *ReplySender*,
- et les services de requête sont gérés par le *RequestService*.

Propriété 3.2.1.1 (Interception des évènements) *Tous les évènements clés de la vie d'une activité sont interceptés au niveau méta et clairement identifiés dans l'implémentation.*

Les références vers une activité sont représentées par un couple *Stub-Proxy*. Le *stub* au niveau applicatif est un objet qui est polymorphiquement compatible avec le type de l'objet actif de l'activité référencée, et qui permet donc à une variable d'un type donné de référencer indifféremment un objet Java standard ou une activité. Le *stub* est créé dynamiquement lors de la création d'une activité et peut être passé par copie ; le polymorphisme est assuré par héritage du type de l'objet actif de l'activité référencée.

Le *proxy* au niveau méta permet de masquer la notion de référence locale ou distante, ainsi que le ou les protocoles permettant d'accéder à l'activité référencée. Le *proxy* contient la localisation réelle de l'activité ciblée ; il sert à relayer les

communications de manière générique. Par conséquent, le proxy est une référence qui reste valide quelque soit la localisation de l'activité qui utilise cette référence.

Propriété 3.2.1.2 (Références entre activités) *Les références entre activités sont représentées par des objets et sont universelles.*

Ce mécanisme de *proxy* permet aux applications développées en ProActive de communiquer à l'aide de différents protocoles de communication, tel que RMI, RMI sur SSH, Ibis [NIE 05] ou HTTP.

3.2.2 Déploiement des applications

Déployer une application consiste à créer et configurer toute l'infrastructure nécessaire à son exécution. Ce déploiement est habituellement fait manuellement, à l'aide de scripts et de sessions distantes sur les ressources ciblées. L'utilisateur désirant déployer son application doit :

- dans un premier temps *accéder* aux ressources si nécessaire, par exemple en les réservant via un ordonnanceur de tâches dans le cas d'une grappe dédiée,
- dans un second temps *configurer* l'environnement d'exécution, par exemple en démarrant une machine virtuelle sur les ressources acquises ;
- enfin, il doit *acquérir* ces ressources au niveau applicatif, c'est-à-dire que son application doit être capable de déployer des activités sur ces ressources, par exemple en passant les URLs des machines réservées et configurées.

Un des objectifs de la libraire ProActive est de virtualiser les ressources pour en simplifier l'accès et l'acquisition. C'est pour cette raison que le mécanisme de déploiement offert par ProActive permet d'offrir au niveau applicatif une vue abstraite uniforme des ressources disponibles, qui ne fait pas intervenir d'informations physiques telles que le protocole d'accès, le nom de machine ou encore le numéro de port. En plus de la portabilité au niveau de l'exécution, cette virtualisation permet d'obtenir un code *portable au niveau du déploiement* ; une application pourra être déployée consécutivement sur différentes infrastructures, par exemple sur un réseau local puis sur une grappe de machines dédiées avec un ordonnanceur de tâches, sans aucune modification du code source ni recompilation. C'est dans ce but que les nœuds d'exécution ont été définis.

3.2.2.1 Noeuds et nœuds virtuels

L'environnement d'exécution est virtualisé dans ProActive par la notion de *nœud* : du point de vue de l'utilisateur, une activité est localisée et s'exécute dans un nœud. Il abstrait le processus de connexion sur les ressources physiques et celui de création de l'environnement d'exécution réel ou d'acquisition d'un environnement existant.

Propriété 3.2.2.1 (Noeud d'exécution) *Les environnements d'exécution sont virtualisés en un environnement unique, le nœud d'exécution.*

Les nœuds sont représentés au niveau de l'application par des *nœuds virtuels*. Le nœud virtuel est le nom symbolique donné à un ou à un ensemble de nœuds d'exécution. Il est manipulé directement dans le code source de l'application ; il permet à l'utilisateur d'organiser de façon virtuelle l'ensemble des ressources disponibles en le regroupant en nœuds virtuels.

3.2.2.2 Descripteurs de déploiement

Les informations physiques du déploiement de l'application ne sont donc pas contenues dans le code de l'application elle même, mais dans un fichier externe, appelé *descripteur de déploiement*. C'est ce fichier qui définit les relations entre les ressources virtuelles (les nœuds virtuels) et les ressources physiques, et qui permet donc de créer concrètement l'environnement d'exécution. Plus précisément, les opérations pouvant être configurées dans un descripteur de déploiement sont les suivantes :

- définition du nom de chaque nœud virtuel ;
- association entre les nœuds virtuels et un ou plusieurs nœuds d'exécution ;
- association entre les nœuds d'exécutions et un processus ou une chaîne de processus permettant la création d'un environnement d'exécution, ou l'acquisition de l'environnement d'exécution existant.

La figure 3.5 synthétise le processus global de déploiement offert par ProActive. Un descripteur de déploiement peut être séparé en deux parties : définition des ressources virtuelles et définition des ressources physiques. Il définit les relations entre ces deux aspects associant à chaque ressource virtuelle une ou plusieurs ressources physiques.

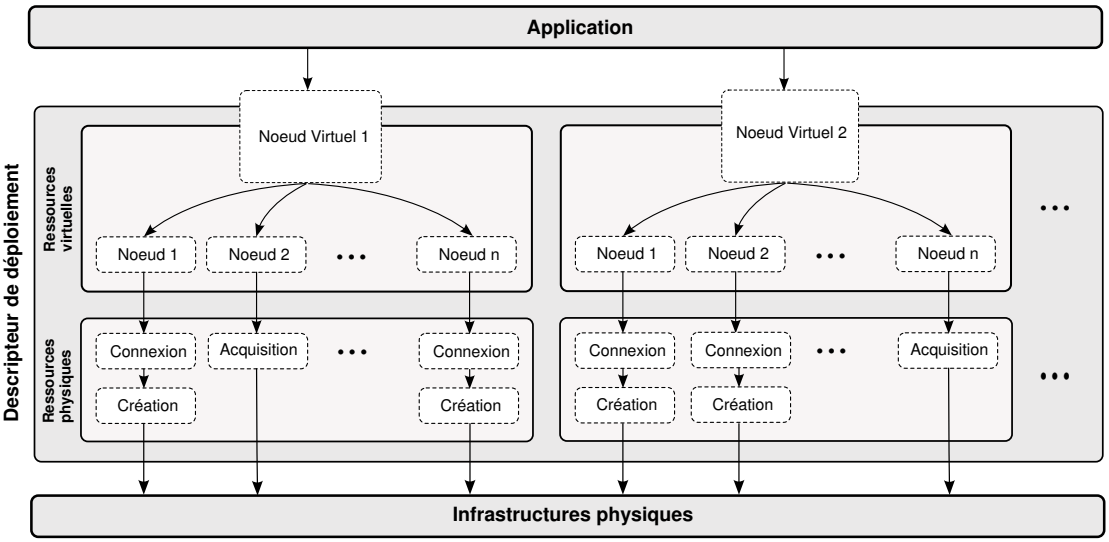

FIG. 3.5 – Modélisation du déploiement avec ProActive

Les ressources physiques peuvent en pratique être de deux natures différentes. Elles peuvent être récupérées au moment du déploiement :

- **par création** ; les noms de machines, les protocoles de connexion et de

création sont définis dans le descripteur de déploiement. Le mécanisme de déploiement de ProActive reconnaît un grand nombre de protocoles qui peuvent être utilisés dans un descripteur de déploiement : des protocoles de connexion directe tels que *ssh*, *rsh*, des protocoles de réservation de ressources sur grappes de machines dédiées tels que *lsf*, *pbs*, *sge*, *oar*, ou encore des protocoles qui permettent de communiquer avec un intergiciel de grille tel que *globus*, *unicore* ou encore *glite*.
Ces protocoles peuvent être combinés dans les descripteurs de déploiement de manière à pouvoir accéder à des ressources organisées de façon hiérarchique. Par exemple, on peut se connecter sur la machine frontale d'une grappe de machines par *ssh*, puis utiliser *oar* pour réserver un ensemble de machines, puis enfin créer une machine virtuelle sur chacune de ces machines.

– **par acquisition** ; Le mécanisme de déploiement permet aussi de se connecter sur une machine et de récupérer l'environnement d'exécution qui s'y trouve. Par exemple, il est possible d'utiliser le mécanisme de connexion offert par Java-RMI, le *lookup* pour se connecter sur une machine virtuelle et récupérer le ou les nœuds qui s'exécutent dans celle-ci.
L'intergiciel ProActive propose aussi une infrastructure pair-à-pair [CAR 06c] qui permet de mettre à disposition un ensemble de ressources préalablement créées. Cette infrastructure peut être vue comme un *fournisseur* de nœuds d'exécution : elle peut fournir à une application un ensemble de nœuds d'exécution déjà déployés. Lorsque l'application se termine, les nœuds sont rendus à l'infrastructure pair-à-pair, et peuvent alors être proposés à d'autres applications. Le mécanisme de déploiement permet de se connecter à cette infrastructure et d'acquérir un ou plusieurs nœuds.

3.2.2.3 Exemple de descripteur de déploiement

Les descripteurs de déploiement sont implémentés en pratique en XML. Nous donnons ici un exemple complet. Ce descripteur déploie deux nœuds virtuels, nommés respectivement `vn1` et `vn2`. La première partie d'un descripteur définit les nœuds virtuels, et spécifie la cardinalité de chaque nœud virtuel : la propriété `multiple` informe que le nœud virtuel peut être constitué de plusieurs nœuds d'exécution, alors que la propriété `unique` force le nœud virtuel a n'être constitué que d'un seul nœud d'exécution.

```
<virtualNodesDefinition>
  <virtualNode name="vn1" property="multiple"/>
  <virtualNode name="vn2" property="unique"/>
</virtualNodesDefinition>
```

La partie suivante définit l'association entre les nœuds virtuels et les environnements d'exécution, ici les machines virtuelles, qui vont être déployées ; le résultat de cette association est un ou un ensemble de nœuds d'exécution. Dans l'exemple, le nœud virtuel `vn1` est associé à trois machines virtuelles, et le nœud `vn2` à une seule machine virtuelle.

```
<mapping>
  <map virtualNode="vn1">
    <jvmSet>
      <jvmName value="jvm1"/>
      <jvmName value="jvm2"/>
      <jvmName value="jvm3"/>
    </jvmSet>
  </map>
  <map virtualNode="vn2">
    <jvmSet>
      <jvmName value="jvm1"/>
    </jvmSet>
  </map>
</mapping>
```

Enfin, les environnements d'exécution sont associés aux processus de déploiement réels. Les processus de déploiement peuvent être des chaînes de processus. Dans notre exemple :

- la machine virtuelle `jvm1` est créée sur la machine locale,
- la machine virtuelle `jvm2` est déployée sur une grappe de machines dédiées gérée par un ordonnanceur de tâches OAR,
- la machine virtuelle `jvm3` existe déjà sur une machine `remoteHost`, et est acquise par recherche RMI (*lookup*),
- enfin, la machine virtuelle `jvm4` est acquise à travers l'infrastructure pair-à-pair préalablement déployée.

```
<jvms>
  <jvm name="jvm1">
    <creation>
      <processReference refid="localJvm"/>
    </creation>
  </jvm>
  <jvm name="jvm2">
    <creation>
      <processReference refid="sshCluster"/>
    </creation>
  </jvm>
  <jvm name="jvm3">
    <acquisition>
      <serviceReference refid="rmiLookup"/>
    </acquisition>
  </jvm>
   <jvm name="jvm4">
    <acquisition>
      <serviceReference refid="p2pLookup"/>
    </acquisition>
  </jvm>
</jvms>
```

Les processus eux-mêmes sont définis dans la dernière partie du descripteur. Le processus `localJvm` crée une machine virtuelle sur la machine locale. La création locale est définie par la classe `process.JVMNodeProcess` de la libraire ProActive.

```
<processDefinition id="localJvm">
  <jvmProcess class="process.JVMNodeProcess"/>
</processDefinition>
```

Le processus `sshCluster` se connecte dans un premier temps à la machine frontale d'une grappe de machines dédiées, puis obtient grâce au processus `oarBooking` une machine sur laquelle il démarre une machine virtuelle à l'aide du processus `localJvm`.

```
<processDefinition id="sshCluster">
  <sshProcess class="process.ssh.SSHProcess" hostname="frontale.
     cluster.inria.fr">
    <processReference refid="oarBooking"/>
  </sshProcess>
</processDefinition>

 <processDefinition id="oarBooking">
   <oarProcess class="process.oar.OARSubProcess">
     <processReference refid="localJvm" />
     <commandPath value="/usr/bin/oarsub"/>
       <oarOption>
         <resources>nodes=1</resources>
       </oarOption>
     </oarProcess>
  </processDefinition>
```

Le processus `rmiLookup` se connecte sur le *rmiregistry* de la machine `remoteHost` et récupère le nœud d'exécution `PAjvm` qui s'exécute sur cette machine.

```
<services>
  <serviceDefinition id="rmiLookup">
    <RMIRegistryLookup url="rmi://remoteHost.inria.fr/PAjvm"/>
  </serviceDefinition>
</services>
```

Enfin, le processus `p2pLookup` dispose de deux points d'entrées dans l'infrastructure pair-à-pair (`registryHost1` et `registryHost2`); à l'aide de ces points d'entrée, le processus demande un nœud d'exécution.

```
<services>
  <serviceDefinition id="p2pLookup">
   <P2PService NodesAsked="1">
    <peerSet>
     <peer>rmi://registryHost1:3000</peer>
     <peer>rmi://registryHost2:3000</peer>
    </peerSet>
   </P2PService>
  </serviceDefinition>
 </service>
```

3.3 Impacts sur la conception d'un mécanisme de tolérance aux pannes

Dans cette section, nous allons analyser les propriétés et les contraintes du modèle ASP et de son implémentation ProActive afin de comprendre l'impact de ces différents points dans le cadre du développement d'un protocole de tolérance aux pannes par recouvrement arrière.

3.3.1 Création de points de reprise

Comme on l'a vu dans la section 2.2, la capture d'état pour la création d'un point de reprise est un élément majeur de la tolérance aux pannes par recouvrement arrière ; selon les approches choisies, on obtiendra des propriétés de performance, de portabilité et de transparence différentes. Nous cherchons ici l'approche la mieux adaptée aux propriétés et aux contraintes définies dans ce chapitre.

3.3.1.1 Approches possibles

La première propriété attendue est la portabilité des images ; la contrainte de portabilité 3.2.0.2 nécessite que l'image de l'état d'une activité puisse être utilisée pour redémarrer l'activité sur n'importe quel nœud d'exécution. Or un nœud d'exécution est un environnement qui abstrait la ressource physique réelle sous-jacente. La persistance au niveau du système d'exploitation ou utilisant des librairies spécifiques telles que [HOW 99] à un système d'exploitation donné est donc exclue.

Il est possible aussi d'utiliser une machine virtuelle Java modifiée, qui propose la persistance des activités, comme [BOU 99] ou [AGB 00]. Dans ce cas, les images d'états sont portables uniquement entre les machines virtuelles modifiées. La virtualisation des ressources offertes par Java est dans ce cas fortement diminuée ; la contrainte d'utilisation de Java standard 3.2.1.1 n'est plus respectée. La persistance au niveau de la machine virtuelle est donc exclue.

L'utilisation d'un compilateur Java modifié ou d'un préprocesseur spécifique peut apporter la persistance des activités tout en assurant la portabilité des images si le code généré peut être exécuté sur une machine virtuelle standard. Certaines approches telles que [TRU 00] sont exclues puisqu'elles nécessitent dans le code applicatif la signalisation des points de reprise potentiels ; la contrainte de transparence 3.2.0.1 n'est donc pas respectée. Mais de manière générale, l'approche par compilation, pour rester complètement portable, nécessite l'utilisation d'objets Java additionnels, appelés objets *contextes* qui reflètent l'état de l'activité réelle. Cette approche engendre donc un surcoût important en temps d'exécution et une augmentation de l'empreinte mémoire de l'application. Comme on l'a vu dans la section 2.2.1, certains préprocesseurs Java induisent un surcoût de 100% sur le temps d'exécution. Pour ces raisons de performance, nous excluons l'utilisation d'un compilateur spécifique.

Enfin, la persistance peut être implémentée en proposant une extension de la libraire telle que [SOU 96] ; dans ce cas, le code de l'application doit explicitement déclarer la persistance de certaines classes par héritage. Une application ProActive existante devrait donc être modifiée pour pouvoir être exécutée de façon tolérante aux pannes, ce qui va à l'encontre de la contrainte de transparence 3.2.0.1.

3.3.1.2 Approche choisie

Nous avons choisi de ne reposer sur aucun mécanisme extérieur, mais sur les possibilités du langage lui-même, de manière à assurer la portabilité des images et du mécanisme lui-même. Nous avons donc choisi comme mécanisme de persistance *par défaut* la sérialisation standard proposée par le langage Java. La forme sérialisée d'un objet Java est intrinsèquement portable sur toutes les machines virtuelles Java. De plus, la transparence est assurée par le fait que la sérialisation d'un objet ne fait pas obligatoirement intervenir l'utilisateur ; en effet, la définition des méthodes de sérialisation et de désérialisation n'est pas obligatoire.

En terme de performances, l'utilisation de la sérialisation n'implique aucun surcoût en temps ou en mémoire pendant une exécution sans capture. Mais c'est aussi un mécanisme standard, qui ne peut pas être modifié ; la capture des images ne peut donc pas être optimisée à l'aide des différentes techniques classiques telles que la capture incrémentale ou la capture non bloquante.

Cette approche est intéressante aussi parce qu'elle représente une alternative par défaut pour la persistance, mais laisse la possibilité à l'utilisateur d'intervenir ; en effet, l'utilisateur peut redéfinir les méthodes de sérialisation pour par exemple minimiser la taille des données à sauvegarder en sauvegardant uniquement des données significatives. Dans ce cas, le mécanisme de persistance n'est plus transparent, mais c'est un choix de l'utilisateur. Il dispose toujours d'une alternative totalement transparente qui permet en particulier d'exécuter de façon tolérante aux pannes des applications déjà écrites et compilées.

L'utilisation de la sérialisation comme mécanisme de persistance est rendue possible grâce à la propriété d'absence de partage de mémoire 3.1.1.1 qui nous assure qu'une activité est formée d'un ensemble d'objets Java qui ne peuvent pas être partagés par d'autres activités. De plus, la propriété des références réifiées sous forme de couple *stub-proxy* (propriété 3.2.1.2) permet d'assurer la validité des références après la sérialisation puis la désérialisation.

La difficulté d'utiliser la sérialisation comme mécanisme de persistance réside dans le fait que les fils d'exécution en Java, les objets de type `Thread`, ne sont pas sérialisables. De manière générale, le langage Java ne permet pas l'accès aux structures de contrôle de l'exécution, telles que la pile. Lorsqu'une activité est sérialisée, l'image résultante contient donc l'état de l'activité au niveau applicatif ainsi que son état au niveau méta (queue de requête en attente, ensemble des futurs en attente,...), mais pas l'état courant du fil d'exécution. Si une telle image est utilisée pour redémarrer une activité, le fil d'exécution repart dans son état par défaut, qui n'est pas forcément cohérent avec l'état de l'activité. En pratique,

cette incohérence se caractériserait par le fait que le service d'une requête qui avait commencé avant la capture de l'état serait repris *depuis le début* en cas de redémarrage de l'activité.

Pour éviter cette incohérence, nous utilisons la propriété sur les états capturables 3.1.3.1 qui nous dit que l'état du fil d'exécution d'une activité inactive (i.e. qui ne sert pas de requête) n'est pas significatif. La capture de l'état d'une activité inactive est donc cohérente, et le point de reprise résultant peut être utilisé pour redémarrer l'activité.

La persistance fournie ici est donc une persistance *faible* : si une demande de capture de l'état d'une activité est faite, cette capture ne peut avoir lieu que lorsque l'activité est inactive.

3.3.2 Cohérence des états globaux

Pour assurer de façon transparente à l'utilisateur la cohérence des états globaux formés durant l'exécution, les protocoles par points de reprise classiques (synchronisés ou induits par messages) font l'hypothèse que :

- soit un point de reprise peut être déclenché à tout moment durant l'exécution sur n'importe quelle activité,
- soit les réceptions de messages peuvent être repoussées jusqu'à ce qu'un point de reprise soit possible.

La première condition permet d'assurer que si un message marqueur dans le cas d'une approche synchronisée ou un message potentiellement orphelin dans le cas d'une approche induite par message est reçu, un point de reprise peut être déclenché *avant* la prise en compte respectivement du prochain message ou du message reçu. Le message potentiellement orphelin est donc délivré après le point de reprise et n'est plus orphelin dans l'état global contenant le point de reprise forcé.

La deuxième condition est plus subtile : si un point de reprise ne peut pas être déclenché avant la prise en compte d'un message potentiellement orphelin, alors c'est la réception du message qui doit être repoussée après le prochain point de reprise. Dans ce cas, le message est bien délivré après le point de reprise et n'est pas orphelin dans l'état global contenant ce point de reprise.

La première condition est vraie dans le cas où la persistance du système est forte. Or nous avons vu dans la section précédente que le mécanisme de persistance que nous avons retenu implique une persistance faible.

La deuxième condition requiert que les communications du système considéré soient *purement asynchrones*, c'est-à-dire que les réceptions de message soient totalement imprévisibles [CHA 96b]. Cette condition est en pratique rarement remplie par les systèmes réels qui soit assurent un ordonnancement de message particulier, soit donnent la possibilité à l'utilisateur d'influer sur l'ordonnancement des réceptions. Par exemple, l'intergiciel de communication MPI permet à l'aide de la primitive `MPI_RCV(...)` d'attendre un message en particulier, ce qui a pour effet de prédire la réception d'un message à un moment particulier de

l'exécution. Dans le contexte d'ASP, l'ordonnancement causal des réceptions de messages (contrainte 3.1.2.1) et le mode de communication par requête-réponse avec futur supposent donc que certaines réceptions de messages doivent avoir lieu avant des moments précis de l'exécution.

Dans ces différents cas, la solution consistant à repousser la réception des messages n'est donc pas applicable. En effet, l'exécution peut être bloquée en attente d'un message, par exemple dans le cas d'une attente par nécessité sur un futur. Si la réponse attendue est potentiellement orpheline, sa réception doit donc être repoussée jusqu'à ce que l'exécution atteigne un état stable, c'est à dire que le service de requête courant se termine. Or l'exécution est bloquée en attente de ce message : le système est donc bloqué en attente infinie.

Notre contexte ne permet donc pas la création transparente pour l'utilisateur d'états globaux cohérents. Les protocoles par journalisation sont une solution possible dans un environnement dans lequel la création d'états globaux cohérents est impossible. En effet, ce type de protocole crée chaque point de reprise de manière *indépendante*, et ne repose pas sur une synchronisation entre les activités durant l'exécution sans panne. La journalisation répond aussi aux contraintes d'ordonnancement de message, puisque tout événement non déterministe, en particulier les réceptions de message, doit être journalisé de façon stable.

Cependant, comme nous l'avons vu dans le chapitre 2, la journalisation n'est pas une solution idéale qui peut s'appliquer de façon efficace dans toutes les situations. Par exemple, dans le cas d'applications hautement communicantes, le surcoût induit par la journalisation peut devenir fortement prohibitif.

Nous voulons donc proposer pour ce type de situation une approche par points de reprise uniquement, moins coûteuse. La contrainte la plus forte dans ce cas est l'impossibilité de créer des états globaux cohérents durant l'exécution sans panne ; les protocoles classiques, qu'ils soient synchrones ou induits par message ne peuvent pas s'appliquer dans cette situation.

Il faut donc rendre recouvrable un état global *qui n'est pas cohérent*. Comme on l'a vu dans la section 2.3, il faut capturer un état global et suffisamment d'informations pour pouvoir contrôler la réexécution depuis cet état global non cohérent jusqu'à un état global cohérent de l'exécution de référence.

Nous pouvons cependant exploiter les caractéristiques particulières des états possibles dans une exécution ASP. En effet, si on ne considère pas le cas des services de requête imbriqués, un point de reprise est possible entre chaque service de requête (propriété 3.1.3.2).

3.3.3 Manipulation des messages

Pour contrôler la réexécution, il faut être capable de contrôler l'exécution des évènements non déterministes, et en particulier les réceptions de message. Les activités étant temporairement désynchronisées pendant la réexécution, il est par exemple possible que l'une d'entre elle reçoive un message trop tôt ou trop tard ; elle doit donc être capable de repousser ou d'attendre la réception d'un message.

Cette capacité est en général supposée par les protocoles de tolérance aux pannes par journalisation des messages. En effet, lorsqu'une activité est redémarrée depuis le dernier état enregistré, tous les messages qui avaient été reçus entre cet état et le moment de la panne sont renvoyés à l'activité. On suppose donc ici que l'activité est capable de recevoir *en avance* tous les messages dans son état de départ. Nous allons voir avec les cas des réponses en ASP que cette supposition n'est pas toujours vraie.

Nous avons vu que les réceptions de requête et de réponse ont des propriétés très différentes dans ASP. En effet, la contrainte d'ordonnancement causal 3.1.2.1 et la propriété de caractérisation de l'exécution 3.1.2.2 nous permettent d'affirmer que la réception d'une requête, tant qu'elle n'est pas servie, ne doit être ordonnée que par rapport aux autres réceptions de requêtes, de manière à assurer l'ordonnancement causal. Cet ordonnancement est représenté par l'ordre des requêtes dans la queue des requêtes en attente. Donc une activité est toujours dans un état dans lequel elle peut recevoir une requête. Par exemple, dans le cas d'un protocole de journalisation des messages, une activité est capable de recevoir en avance toutes les requêtes reçues entre le point de reprise et la panne ; il suffit que ces requêtes soient correctement ordonnées dans la queue des requêtes en attente.

Au contraire, l'ordre de réception des réponses n'est pas significatif pour l'exécution, mais une activité doit d'abord atteindre un certain état avant de pouvoir recevoir une réponse. En effet, la contrainte 3.1.2.2 nous dit qu'une réponse ne peut être reçue par une activité que si cette activité a déjà envoyé la requête correspondante, et a donc créé le futur associé à cette réponse. Si une réponse est reçue en avance, c'est-à-dire avant la création du futur, il n'y a pas de place pour cette réponse dans l'activité ; elle ne peut donc pas être prise en compte.

Le modèle ASP définit l'association entre une requête, la réponse correspondante et son futur par la connaissance commune d'un identifiant unique, généré lors de l'envoi de la requête. Donc une réponse qui est reçue en avance par une activité est estampillée avec un identifiant *qui n'a pas encore été généré* par l'activité réceptrice. Or le modèle ASP suppose actuellement que la génération des identifiants n'est pas forcément déterministe. Dans ce cas, le futur qui sera créé lors de l'envoi de la requête correspondante à la réponse en avance n'aura pas le même identifiant que la réponse. La mise à jour du futur par la réponse devient donc impossible.

Une solution possible serait de modifier le mécanisme de communication de ProActive, de manière à pouvoir prendre en compte les réponses qui n'ont pas encore de futur, et donc de modifier aussi le mécanisme des futurs lui-même. De plus, il faudrait préciser dans le modèle ASP une manière déterministe de créer les identifiants et ainsi pouvoir prévoir l'identifiant qui sera généré après la réception d'une réponse en avance.

Nous allons montrer dans le chapitre 4 qu'une modification intrusive du code de l'intergiciel et une modification du modèle existant, bien que relativement simples, ne sont pas nécessaires. Nous allons donc proposer un protocole qui as-

sure que durant la réexécution depuis un état global incohérent, *aucune réponse ne peut être reçue en avance* par une activité.

3.3.4 Causalité potentielle

La connaissance d'une causalité potentielle pour ASP va nous permettre de déterminer les relations causales qui doivent être conservées entre les évènements, sans avoir à utiliser de mécanisme de détection dynamique de la causalité. La propriété 3.1.5.1 nous dit que la relation de causalité pour ASP $\prec^{ASP}$ est vraie pour toutes les exécutions possibles, puisqu'elle découle de la sémantique du modèle, et non des spécificités d'une exécution en particulier.

Nous pouvons donc sans aucun surcoût à l'exécution et sans aucune modification de l'implémentation tirer parti de cette connaissance d'une causalité plus précise dans la conception d'un protocole adapté à ASP. Nous exploitons cette connaissance pour relâcher les conditions de cohérence, mais aussi la synchronisation nécessaire entre les activités pour assurer la cohérence de la réexécution, lors du recouvrement de l'application.

3.3.5 Implémentation dans ProActive

La contrainte de transparence du mécanisme de tolérance aux pannes 3.2.0.1 ne nous permet donc pas d'utiliser une approche explicite, en proposant à l'utilisateur une interface de programmation applicative spécifique à la tolérance aux pannes. Les aspects liés à la tolérance aux pannes doivent donc être gérés uniquement par l'intergiciel lui-même.

Nous gardons cependant comme objectif de minimiser l'intrusion du code gérant la tolérance aux pannes dans le code existant de l'intergiciel, de manière à faciliter la maintenance et l'évolutivité du code. La propriété 3.2.1.1 nous permet une intégration par interception, peu intrusive : nous ajoutons à la structure d'une activité présentée dans la figure 3.4 un méta-objet, unique responsable de la gestion de la tolérance aux pannes. Tout le code lié à la tolérance aux pannes doit être centralisé dans ce méta-objet, qui est informé par le protocole à méta-objet de ProActive des envois et des réceptions de messages, ainsi que du déclenchement des services de requêtes. Le protocole que nous proposons devra donc prendre des décisions uniquement en fonction de ces informations.

Nous devons bien sûr donner la possibilité à l'utilisateur de configurer le mécanisme de tolérance aux pannes, en fonction des conditions d'exécution. Si par exemple une application est déployée sur un environnement très instable, dans lequel la durée de vie des nœuds est très courte, il faut créer des lignes de recouvrement fréquemment. En effet, si la fréquence des pannes est supérieure à la fréquence de création de ligne de recouvrement, l'exécution ne finit pas. De plus, l'utilisateur doit aussi pouvoir spécifier certains paramètres comme la méthode d'accès à la mémoire stable.

Le problème posé ici est comment configurer le méta-objet responsable de la tolérance aux pannes. L'utilisateur ne connaît pas à l'avance les conditions dans lesquelles l'application sera déployée : les paramètres de tolérance aux pannes

pour une exécution donnée ne peuvent être connus qu'au moment du déploiement, en fonction des caractéristiques des ressources réelles. Les contraintes de transparence 3.2.0.1 et de portabilité 3.2.0.2 empêchent une configuration dans le code source de l'utilisateur, puisqu'il faudrait modifier et recompiler le code source avant un déploiement.

L'activité ne peut donc pas contenir les paramètres de tolérance aux pannes. Nous allons donc configurer non pas l'activité elle même, mais l'environnement d'exécution abstrait de ProActive, le nœud d'exécution (propriété 3.2.2.1). Pour permettre à l'utilisateur de décrire la configuration du mécanisme de tolérance aux pannes au moment du déploiement, nous pouvons utiliser les descripteurs existants dans ProActive.

3.4 Conclusion

Nous avons présenté dans ce chapitre les aspects du modèle ASP et de son implémentation ProActive qui ont un impact sur la conception d'un mécanisme de tolérance aux pannes, tant au niveau du protocole qu'au niveau de l'intégration dans l'intergiciel. Chacun de ces aspects sera pris en compte par le protocole que nous avons proposé durant cette thèse, et qui est présenté dans le chapitre suivant. Ce protocole de tolérance aux pannes par recouvrement arrière :

- se base sur la création d'état globaux et minimise le degré d'utilisation de journalisation de messages,
- assure la recouvrabilité des états globaux non cohérents formés pendant une exécution sans panne,
- évite les réceptions de réponses en avance, c'est-à-dire avant la création du futur associé,
- tient compte de la causalité particulière définie pour le modèle ASP pour relâcher la notion de cohérence et minimiser les synchronisations nécessaires lors de la réexécution.

Enfin, en termes d'implémentation, l'intégration de ce protocole se fera à l'aide du protocole à méta-objet existant, et un mécanisme de configuration de la tolérance aux pannes au moment du déploiement sera ajouter au mécanisme de déploiement existant.

Chapitre 4

Proposition

L'analyse de notre contexte nous a permis d'identifier les contraintes à respecter dans la définition d'un protocole de tolérance aux pannes pour ASP et ProActive. Nous présentons d'abord dans ce chapitre le protocole que nous avons développé durant cette thèse [BAU 05b, BAU 05a], ainsi que son implémentation dans ProActive. Enfin, nous présentons les évaluations de performances à travers différentes applications distribuées communicantes.

4.1 Protocole par point de reprise pour ASP

4.1.1 Création des états globaux

Nous ne pouvons donc pas supposer dans le cadre de ASP que les états globaux sont cohérents. Il peut exister dans ces états des requêtes ou des réponses orphelines ou en transit. Pour identifier de manière simple et efficace ces messages particuliers, nous utilisons l'approche induite par message avec estampillage des messages par l'index des points de reprises proposée dans [BRI 84].

Sur chaque activité, les points de reprise sont indexés de manière incrémentale. Un point de reprise est régulièrement déclenché sur chaque activité durant l'exécution ; la période de déclenchement de point de reprise est notée TTC. De cette manière, on assure que des états globaux de l'application sont régulièrement crées de manière à minimiser la quantité de travail à défaire en cas de panne. Le n^{ieme} point de reprise de l'activité i est noté C_i^n comme dans la figure 4.1.

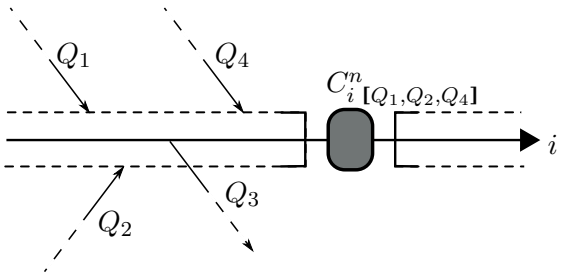

FIG. 4.1 – Point de reprise de l'activité i

On pourra préciser le contenu de la queue des requêtes en attente si nécessaire : on voit dans la figure 4.1 que le point de reprise $C_{i\ [Q_1,Q_2,Q_4]}^{n}$ contient les réceptions des requêtes Q_1, Q_2 et Q_4.

Comme dans [BRI 84], tous les messages partant d'une activité sont estampillés avec l'index du dernier point de reprise de l'activité émettrice. De cette manière, une activité qui reçoit un message est informée de l'index du dernier point de reprise de l'émetteur ; elle peut donc connaître la particularité du message qu'elle reçoit :

- si le message est estampillé avec une valeur supérieure à l'index du dernier point de reprise du receveur, alors le message reçu est orphelin,
- si le message est estampillé avec une valeur inférieure à l'index du dernier point de reprise du receveur, alors le message reçu est en transit.

Dans le cas de la réception d'un message orphelin, la stratégie adoptée par [BRI 84] est de déclencher un point de reprise, dit point de reprise "forcé", avant la prise en compte de ce message et de l'indexer avec la valeur estampillée sur le message. De cette manière, on assure qu'un état global n (formé des points de reprise portant le même index n) est un état global cohérent, donc une ligne de recouvrement possible en cas de panne.

Dans notre contexte, la prise de ce point forcé n'est pas possible ; il faut en effet attendre que l'activité atteigne un état stable. Comme nous l'avons vu dans la section 3.3.2, on ne peut pas repousser le message orphelin, sous peine de bloquer l'application. Ce message reste donc un message orphelin dans l'état global en cours de construction. C'est le cas de la requête Q_1 dans la figure 4.2 ; la prise du point de reprise $n+1$ doit être repoussée à la fin de du service $X(Q_0)$. La requête Q_1 reste orpheline dans l'état global formé des points de reprise $n+1$; sa réception fait partie de l'état de l'activité dans le point de reprise $C_{j\ [Q_1]}^{n+1}$.

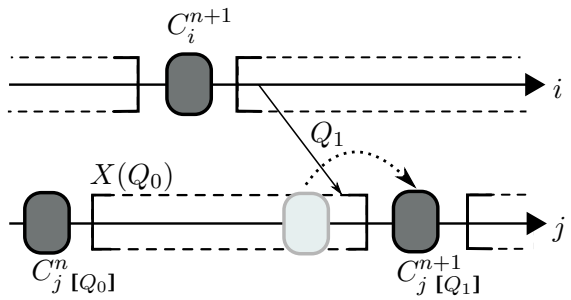

FIG. 4.2 – Le point de reprise grisé est impossible : Q_1 est un message orphelin

Notre contribution consiste en une solution qui assure que, bien qu'incohérent, un état global formé de point de reprise de même index soit recouvrable, et forme donc une ligne de recouvrement possible en cas de panne.

4.1.2 Prise en compte des messages orphelins

Un message orphelin dans un état global est un message qui est reçu dans cet état mais pas encore envoyé. Donc en cas de réexécution depuis cet état global, un duplicata de ce message est réémis : l'activité réceptrice reçoit dans ce cas ce message une nouvelle fois. Pour rendre l'état global recouvrable, on peut annuler le message dupliqué durant la réexécution, soit en évitant son envoi ou bien sinon sa réception. Lorsque cela est possible, une autre solution est de modifier l'état global pour qu'il ne contienne plus la réception du message orphelin ; en cas de réexécution, le message dupliqué peut donc être reçu normalement. Nous verrons par la suite dans quelle situation cette solution est possible.

La présence d'un message orphelin dans un état global pose un problème au niveau de la cohérence des données mais aussi au niveau de la synchronisation entre les activités. Pour traiter ces deux parties du problème il faut assurer les deux propriétés suivantes durant la réexécution :

- **équivalence des duplicatas** ; le contenu du message dupliqué qui est envoyé durant la réexécution doit être strictement le même que le message orphelin envoyé lors de la première exécution ;
- **cohérence de la réexécution** ; la synchronisation engendrée par la réception de ce message peut devoir être reproduite pendant la réexécution. En d'autres termes, les conséquences causales de la réception d'un message orphelin ne doivent pas avoir lieu *avant* cette réception.

La première condition est évidente : en cas de réexécution depuis $n+1$ dans la figure 4.2, l'activité j qui reçoit le message dupliqué Q_1 est dans un état dans lequel elle a *déjà reçu* ce message. Ce message peut donc avoir eu potentiellement des conséquences sur l'état de l'activité. Si le duplicata est ignoré par j, son contenu doit être strictement identique pour assurer la cohérence entre l'état réel de j et l'état de j supposé par i suite à l'envoi de Q_1.

La seconde condition est plus subtile. Elle assure que la réexécution est une exécution *possible* du système, c'est-à-dire qu'il n'y a pas de conséquences exécutées avant leurs causes, même si la réexécution se fait depuis un état global incohérent. Or si on annule l'envoi ou la réception d'un message orphelin pendant la réexécution, la synchronisation engendrée par ce message est perdue ; des conséquences peuvent alors être exécutée avant leurs causes. Par exemple dans la figure 4.2, lors de la réexécution, i démarre dans un état dans lequel elle n'a pas encore envoyé Q_1. Elle n'est donc pas dans un état dans lequel elle peut exécuter des conséquences causales de l'envoi du message Q_1. Or, j qui lui a déjà reçu Q_1 peut exécuter des conséquences de la réception de Q_1 : les conséquences causales d'un message *qui n'a pas encore été envoyé* peuvent donc être exécutées.

Selon la sémantique du système considéré, cette incohérence peut amener à un état global impossible. Par exemple, dans le cas d'un mécanisme d'accusé de réception pour les messages, il devient possible pendant une réexécution qu'une activité reçoive un accusé de réception pour un message qu'elle n'a pas encore envoyé. Dans le cadre d'ASP, cette incohérence peut mener à la réception d'une réponse par une activité qui n'a pas encore envoyé la requête correspondante. Par

exemple, dans la figure 4.3, la requête Q_1 est orpheline dans l'état global n, et R_1 est la réponse du service de Q_1 sur j. Supposons une réexécution depuis l'état global n : sans synchronisation adéquate, j peut servir Q_1 puis envoyer R_1 avant même que i ait envoyé Q_1. Dans ce cas, la contrainte 3.1.2.2 n'est pas respectée : i reçoit une réponse pour laquelle il n'existe pas encore de futur.

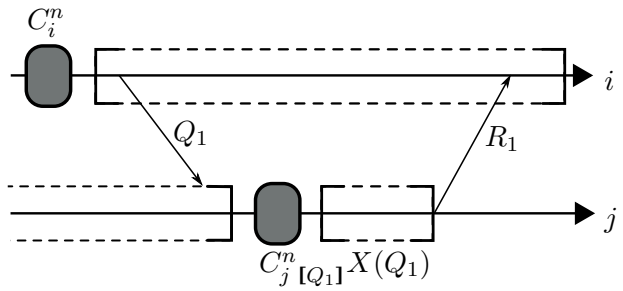

FIG. 4.3 – La réponse R_1 doit être reçue après l'envoi de Q_1

Pour résoudre de façon simple et efficace les problèmes de données et de synchronisation posés par les messages orphelins, nous introduisons le concept de promesse de requête.

4.1.2.1 Promesse de requête

Une promesse de requête est un substitut local pour une requête qui n'a pas encore été reçue durant une réexécution. A chaque promesse correspond une requête promise qui a été envoyé durant l'exécution de référence ; lorsque la requête promise est reçue durant la réexécution, cette promesse est automatiquement remplacée par la requête réelle.

Une promesse de requête peut être placée dans la queue de requêtes en attente d'une activité, et peut donc être ordonnée par rapport aux autres requêtes. En termes d'implémentation, une promesse de requête doit être de type polymorphiquement compatible avec le type requête.

Il doit exister une correspondance unique entre une promesse de requête et la requête promise. Dans le cadre des activités déterministes par morceaux et de communications FIFO du modèle ASP, la propriété 3.1.2.2 permet de caractériser une exécution uniquement par les listes des identifiants des émetteurs de requêtes reçues sur chaque activité. Donc, pour une exécution donnée, une réception de requête par une activité peut être identifiée de manière unique par l'identifiant de l'émetteur.

Une promesse de requête étant ordonnée par rapport aux autres réceptions de requête, elle ne doit donc contenir que l'identifiant de l'activité qui a émis la requête attendue ; cette information est suffisante pour identifier de manière unique la requête correspondante à la promesse.

Le service d'une promesse de requête est soumis à une attente par nécessité : quand une activité essaie de servir une promesse de requête, elle est bloquée tant que la promesse de requête n'est pas remplacée par la requête correspondante. La promesse d'une requête envoyée par i et placée dans la queue des requêtes en attente d'une activité j est notée $Q_{i,j}^{pmd}$. Quand une requête $Q_{i,j}$ est reçue par une activité j :

- si j est bloquée en attente par nécessité sur une promesse de requête $Q_{i,j}^{pmd}$ de l'émetteur i, alors j est débloquée et démarre le service de la requête $Q_{i,j}$,
- si j n'est pas bloquée en attente par nécessité, mais qu'il existe une ou plusieurs promesses de requête $Q_{i,j}^{pmd}$ de l'émetteur i, alors la plus ancienne promesse $Q_{i,j}^{pmd}$ est remplacée par la requête reçue,
- sinon $Q_{i,j}$ est ajoutée à la fin de la queue des requêtes en attente de j.

La promesse de requête va nous permettre de résoudre de manière simple les problèmes posés par les messages orphelins, en minimisant la synchronisation nécessaire ainsi que la quantité de données à journaliser. La promesse de requête va nous permettre :

- d'éviter les réceptions multiples des requêtes orphelines,
- d'assurer l'équivalence du contenu des duplicatas de messages orphelins,
- d'assurer la synchronisation engendrée par les requêtes orphelines lors de la réexécution, afin d'éviter les réceptions de réponse en avance.

4.1.2.2 Requêtes orphelines

Lorsqu'une requête $Q_{i,j}$ est reçue par j et identifiée orpheline, l'activité j programme un point de reprise au prochain état capturable. De plus, la requête orpheline est remplacée par une promesse $Q_{i,j}^{pmd}$ dans la queue de requête de j *au moment du prochain point de reprise* de j, et uniquement dans l'image utilisée pour créer le point de reprise. De cette manière, la réception de $Q_{i,j}$ est virtuellement annulée dans l'état global en cours de création, et repoussée après le point de reprise.

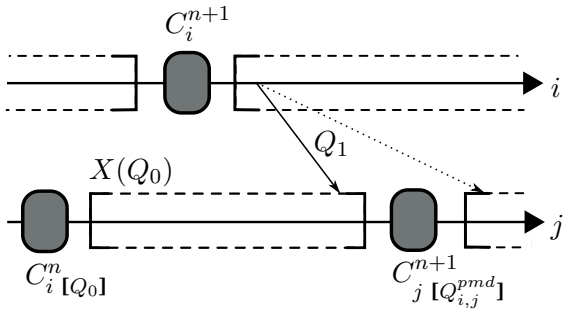

FIG. 4.4 – La réception de Q_1 est virtuellement repoussée après $C_{j\ [Q_{i,j}^{pmd}]}^{n+1}$

Dans l'exemple de la figure 4.4, on peut voir que la requête orpheline Q_1 est remplacée par une promesse dans le point de reprise $C_{j\ [Q_{i,j}^{pmd}]}^{n+1}$. En cas de reprise depuis le point $n+1$, j redémarre dans un état dans lequel il n'a pas *encore* reçu

la requête Q_1.

On suppose donc ici qu'une requête orpheline peut toujours être transformée en promesse de requête dans le point de reprise déclenchée par sa réception, donc qu'elle se trouve encore dans la queue des requêtes en attente. Sans considérer le cas de services imbriqués, la propriété d'occurrence des états capturables 3.1.3.2 assure qu'un point de reprise peut être pris entre chaque service. Donc la réception d'une requête orpheline déclenche un point de reprise *à la fin du service courant*. Par conséquent, une requête orpheline ne peut pas être déjà servie au moment du point de reprise, et se trouve donc forcément encore dans la queue de requête de l'activité. Une requête orpheline peut donc toujours être manipulée et remplacée dans la queue de requête par une promesse.

Nous allons donc supposer par la suite que les activités ne déclenchent pas de service imbriqué car l'état de l'activité avant un tel service n'est pas capturable. En effet, si le service imbriqué d'une requête *orpheline* est déclenché, l'activité ne peut pas prendre de point de reprise *avant* ce service. Par conséquent, cette requête orpheline sera déjà servie dans le prochain point de reprise possible, et ne sera donc plus dans la queue de requête. Nous reviendrons cependant sur ce problème dans la section 7.2.1 en proposant une solution pour rendre capturable l'état d'une activité avant un service imbriqué.

L'insertion d'une promesse de requête dans la queue des requêtes en attente permet de conserver l'ordre de réception des requêtes, puisque la promesse est ordonnée par rapport aux autres requêtes de la queue. La réception qui a été annulée est donc repoussée à la même position que pendant l'exécution de référence.

De plus, l'attente par nécessité sur le service d'une promesse de requête permet d'assurer la synchronisation engendrée par les requêtes orphelines pendant la réexécution, afin d'éviter la réception de réponse en avance. Dans le cas de la figure 4.3, la requête Q_1 est orpheline et est donc transformée en une promesse $Q_{i,j}^{pmd}$ dans le point de reprise C_i^n. Lorsque i redémarre depuis C_i^n, elle commence par le service de $Q_{i,j}^{pmd}$, et est donc bloquée en attente de la requête Q_1. Par conséquent, il est impossible qu'elle envoie la réponse R_1 avant que j ait envoyé Q_1.

De manière plus générale, le service d'une requête qui n'est pas encore reçue bloque l'activité jusqu'à la réception de cette requête. Il est donc impossible pour une activité de servir une requête qui n'a pas encore été envoyée. Par conséquent, il est impossible de recevoir une réponse avant d'avoir envoyé la requête correspondante. La contrainte 3.1.2.2 sur les réceptions de réponses est respectée.

La promesse de requête nous permet donc de réagir *a posteriori* dans le cas de messages orphelins, puisque la prise d'un point de reprise forcé est impossible. De plus, une promesse de requête est un objet simple qui ne contient que des informations disponibles localement, et qui est donc facile à construire durant l'exécution de référence. Enfin, la synchronisation fournie par l'attente par nécessite sur le service d'une requête est efficace et ciblée : seule l'exécution du passé des réponses potentiellement en avance est synchronisée, et donc bloquée si nécessaire. Le reste de l'exécution de l'activité, c'est-à-dire les réceptions de messages et le

service des requêtes reçues avant les requêtes orphelines, n'est soumis à aucune synchronisation.

4.1.2.3 Réponses orphelines

Lorsqu'une réponse $R_{i,j}$ est reçue par j et identifiée orpheline, l'activité j programme un point de reprise au prochain état capturable. Dans le cas des réponses, la création d'un substitut de réception qui permettrait de repousser virtuellement cette réception et d'assurer la synchronisation qu'elle engendre est impossible.

En effet, lorsqu'elle est reçue par une activité, une réponse met à jour le futur correspondant dans l'état de l'activité. Cette mise à jour peut potentiellement débloquer l'activité qui peut modifier *globalement* son état en fonction de la réponse reçue. Contrairement aux modifications induites par la réception d'une requête, cette modification est irréversible parce que l'impact sur l'état de l'activité n'est pas observable. Comme on l'a vu dans la section 3.3.5, l'implémentation du mécanisme de tolérance aux pannes ne doit reposer que sur l'interception des évènements clés de la vie d'une activité, définis par la propriété 3.2.1.1, et sur la délégation à un méta objet responsable de la tolérance aux pannes. On ne peut donc pas considérer que ces modifications d'état induites par l'utilisation d'un futur soient observables par ce méta objet ; un tel mécanisme d'observation serait très coûteux en pratique.

Par conséquent, on ne peut pas, comme dans le cas des requêtes, virtuellement annuler la réception d'une réponse orpheline. La synchronisation dûe à la réception d'une réponse orpheline n'est donc plus assurée pendant la réexécution. Nous allons voir dans la section 4.1.3 que cette perte de synchronisation peut entraîner une perte de l'ordonnancement des réceptions de messages, et comment compenser cette perte de synchronisation.

Le duplicata d'une réponse orpheline doit donc être ignoré par le récepteur lors de la réexécution. Dans le cadre de ASP et de ProActive, un duplicata de réponse est automatiquement ignoré par son récepteur. En effet, un duplicata de réponse ne peut pas avoir de futur en attente correspondant, puisque ce futur a déjà été mis à jour. Par conséquent, le duplicata d'une réponse orpheline n'est pas pris en compte par l'activité réceptrice.

Enfin, la propriété 3.1.2.2 nous assure que l'ordre des réceptions de réponse entre elles n'est pas significatif dans l'exécution. La seule condition à respecter sur la réception des réponses est qu'elles ne soient pas reçues en avance, et cette condition est forcément remplie dans le cas des réponses orphelines puisque, par définition, elles ont *déjà* été reçues dans l'état global.

4.1.2.4 Équivalence des duplicatas

Dans le cas des requêtes et des réponses, le protocole doit assurer la condition d'équivalence des duplicatas, c'est-à-dire l'égalité entre le contenu applicatif des messages orphelins et celui des duplicatas créés pendant la réexécution.

Le contenu applicatif d'un message $M_{i,j}$ est la conséquence de l'exécution de l'activité émettrice i depuis un état initial. Assurer que le message $M_{i,j}$ a le même contenu que durant l'exécution de référence revient à assurer que la réexécution est *équivalente* à l'exécution de référence, c'est-à-dire qu'elle produit les mêmes résultats. Il faut donc assurer l'équivalence de la réexécution avec l'exécution de référence tant que des duplicatas de messages orphelins peuvent être envoyés.

Par conséquent, notre protocole doit enregistrer durant l'exécution de référence suffisamment d'informations pour assurer la condition suivante :

Condition 1 *La réexécution doit être équivalente à l'exécution de référence tant que des duplicatas de messages orphelins peuvent être reçus.*

Nous verrons dans la section 4.1.4 comment l'utilisation de promesses de requête va permettre d'assurer cette équivalence.

4.1.3 Prise en compte des messages en transit

Les messages en transit dans un état global sont détectés par le récepteur à l'aide de l'index du dernier point de reprise de l'émetteur estampillé sur le message : si une activité reçoit un message dont l'estampille est inférieure à l'index courant, alors le message reçu est en transit.

Un message en transit dans un état global n est un message qui ne sera pas réémis par la réexécution depuis l'état global n. Lorsqu'un message en transit est identifié, il est journalisé et associé au dernier point de reprise par l'activité réceptrice. Ainsi, les réceptions de messages en transit sont réexécutées *artificiellement* au moment de la réexécution de l'activité depuis un point de reprise n. Nous verrons par la suite que cette journalisation sur la mémoire stable se fait en pratique en une seule fois ; l'activité journalise tous les messages en transit en mémoire volatile, puis sauvegarde le moment venu l'ensemble de ces messages sur la mémoire stable.

La réception artificielle des messages en transit pose le problème de la cohérence de la réexécution. En effet, ces réceptions étant artificielles, elles ne correspondent pas forcément à une exécution possible du système. De nouveau, nous distinguons le traitement des requêtes et des réponses.

4.1.3.1 Requêtes en transit

La réception artificielle d'une requête en transit peut causer une violation de l'ordonnancement causal des réceptions. En effet, la synchronisation qui assure cet ordonnancement lors d'une exécution normale ne peut plus avoir lieu. Par exemple dans la figure 4.5, en cas de reprise depuis la ligne de recouvrement n, la réception artificielle de la requête en transit Q_0 doit avoir lieu avant celle de Q_2 pour conserver l'ordonnancement causal. Or lorsque i redémarre, il n'attend pas la réception de R_1, puisque comme on l'a vu précédemment, la synchronisation engendrée par une réponse orpheline est perdue. L'ordonnancement entre les réceptions de Q_0 et de Q_2 est donc perdu.

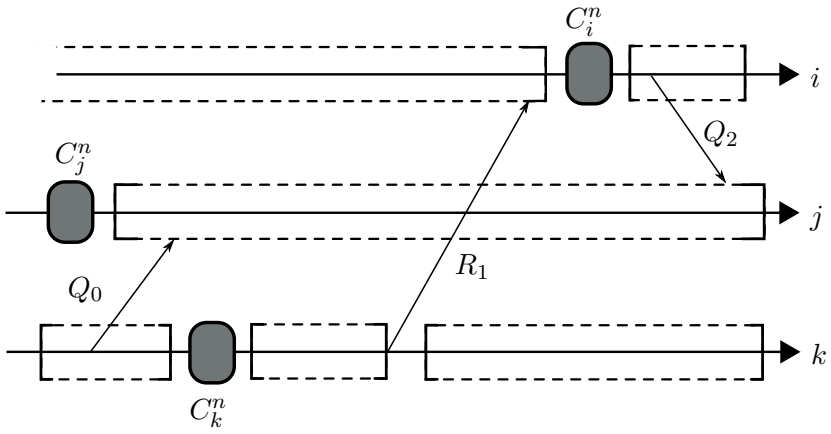

FIG. 4.5 – La réception du message en transit Q_0 doit précéder celle de Q_2

Par conséquent, notre protocole doit enregistrer durant l'exécution de référence suffisamment d'informations pour assurer la condition suivante :

Condition 2 *L'ordre de réception des requêtes doit être le même durant la réexécution que durant l'exécution de référence tant qu'il peut exister des messages en transit dans l'état global en cours de création dans l'exécution de référence.*

4.1.3.2 Réponses en transit

Dans le cas des réponses, la position de la réception elle-même n'est pas significative (propriété 3.1.2.2), mais doit avoir lieu après l'envoi de la requête correspondante (contrainte 3.1.2.2). Or, si l'activité redémarre dans un état dans lequel elle n'a pas encore envoyé la requête correspondante à une réponse en transit, cette réponse ne peut pas être reçue par l'activité ; le futur pour cette réponse n'existe pas encore.

Il faut donc assurer que l'activité contient déjà le futur de la réponse en transit, c'est-à-dire que l'envoi de la requête correspondante a été fait avant le dernier point de reprise, et fait donc partie de la capture d'état.

Supposons que l'envoi de la requête correspondante ne fasse pas partie de l'état. Alors cette requête est forcément orpheline. En effet, sa réception appartient à l'état global, puisqu'on suppose ici que l'envoi de la réponse correspondante appartient à ce même état, mais pas son envoi. C'est le cas par exemple dans la figure 4.6 : Q_0 est orpheline et sa réponse correspondante R_0 est en transit.

Or, comme nous l'avons vu dans la section 4.1.2, la propriété d'occurrence des états capturables 3.1.3.2 nous permet d'assurer qu'un point de reprise peut toujours être pris avant le service d'une requête orpheline. Le cas de la figure 4.6 est donc impossible : une requête orpheline n'est jamais servie avant le point de reprise qu'elle déclenche sur le récepteur. Le protocole assure donc que les réponses en transit peuvent toujours être reçues artificiellement au moment de la réexécution. En pratique, les réponses en transit sont journalisées par l'activité

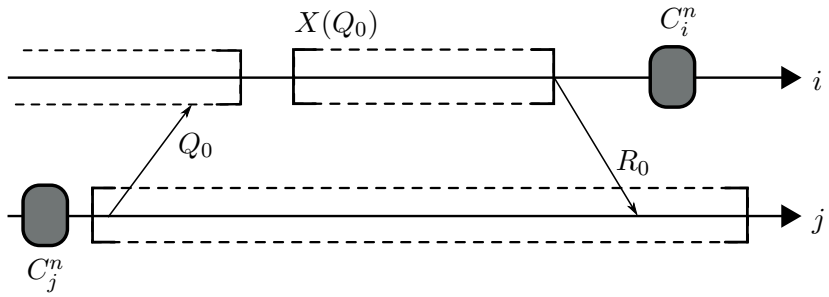

FIG. 4.6 – Un point de reprise est toujours pris avant le service $X(Q_0)$

réceptrice à la réception ; on note $\mathcal{R}_i^n$ le journal de réponses associé au point de reprise n de l'activité i. En cas de reprise depuis un point de reprise n, les futurs correspondants aux réponses contenues dans $\mathcal{R}_i^n$ sont directement mis à jour avant de redémarrer l'activité i.

4.1.4 Équivalence d'exécution

L'utilisation de promesse de requête permet donc d'éviter à l'activité réceptrice de recevoir deux occurrences d'une requête orpheline, et permet aussi d'assurer la cohérence minimum de la réexécution. Nous allons voir dans cette section comment les promesses de requête sont utilisées pour assurer les conditions d'équivalence de la réexécution **Condition 1** et **Condition 2**.

La propriété 3.1.2.2 du modèle ASP nous permet de caractériser l'exécution d'une activité uniquement par l'ordre de réception des requêtes sur cette activité. Par exemple, dans la figure 4.7, le contenu de la réponse R_1 est une conséquence de l'état représenté par le point de reprise C_j^{n+1} et du service de Q_0 *puis* du service de Q_1. Pour assurer l'équivalence de la réponse dupliquée en cas de réexécution, il faut donc disposer du point de reprise C_j^{n+1}, ainsi que de l'ordre de réception de Q_0 et Q_1.

Assurer l'équivalence de la réexécution donnée par la condition 1 revient donc à assurer le même ordre de réception des requêtes, ce qui assure aussi la condition 2. Nous introduisons pour cela l'historique de réception de requête, une liste de promesses de requête qui est associée à un point de reprise. On notera $\mathcal{H}_i^n$ l'historique de réception associé au point de reprise n de l'activité i : quand l'activité i prend le point de reprise C_i^n, l'historique $\mathcal{H}_i^n$ est *ouvert* sur l'activité i. Quand un historique est ouvert sur une activité, chaque réception de requête déclenche l'ajout d'une promesse de cette requête à la fin de l'historique ouvert.

Nous utilisons ici le fait que les activités en ASP sont déterministes par morceaux, et que les seuls évènements non déterministes sont les réceptions de requêtes. Bien sûr, cette restriction peut être levée puisque le déterminisme par morceaux assure que tous les évènements non déterministes sont observables. En pratique, cela implique que le méta-objet est systématiquement capable d'intercepter l'exécution d'un évènement non déterministe, et est donc capable de

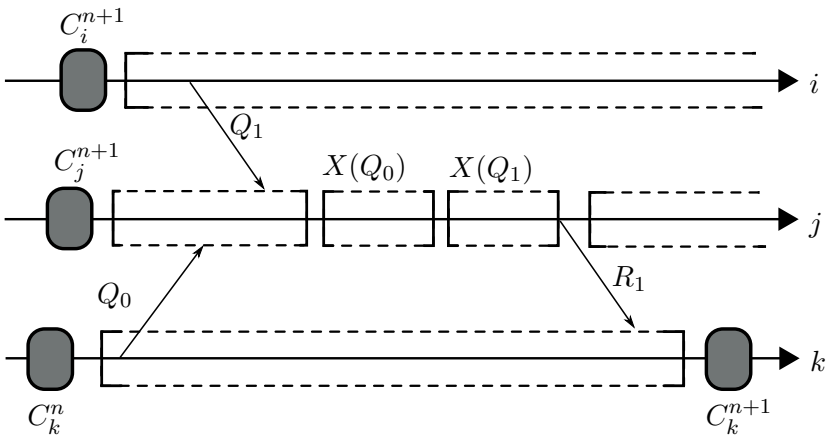

FIG. 4.7 – Le service de Q_0 doit précéder celui de Q_1 pour assurer l'équivalence de R_1

créer des promesses pour n'importe quel type d'évènement non déterministe. Par exemple, pour des évènements non déterministes de type tirage de valeur aléatoire, le méta-objet crée à l'exécution de l'évènement une promesse qui contient la valeur obtenue. Ainsi, en cas de réexécution, la promesse de l'évènement peut restituer la valeur de l'exécution de référence. Nous introduirons dans le chapitre 5 la généralisation de la promesse pour tout type d'évènement non déterministe, et montrerons que le raisonnement appliqué ici sur les réceptions de requête peut être appliqué sur d'autres types d'évènements non déterministes.

Un historique doit finalement être clos durant l'exécution de référence, c'est-à-dire ne plus ajouter de promesses dans cet historique et l'associer au point de reprise n sur la mémoire stable. Or il faut qu'il contienne suffisamment d'informations pour assurer les conditions 1 et 2. Ces deux conditions sont remplies dès qu'il ne peut plus y avoir ni de messages orphelins ni de messages en transit dans l'état global en cours de création ; il faut donc déterminer le moment à partir duquel tous les messages potentiellement orphelins ou en transit ont été reçus.

Dans le cas de notre modèle ASP, l'existence d'un rendez-vous sur les communications (propriété 3.1.2.1) permet de résoudre simplement ce problème. En effet, ce rendez-vous assure la propriété de futur commun (*strong common future*) entre deux activités communicantes [CHA 96b] : les évènements exécutés *après* l'envoi d'un message $M_{i,j}$ sur une activité i sont exécutés aussi forcément *après* la réception de ce message par j. En particulier, si un point de reprise est déclenché après l'envoi d'un message, le rendez-vous nous assure que ce point de reprise est pris après que le récepteur du message ait reçu ce message. Donc quand une activité prend un point de reprise, il ne peut pas y avoir dans le système de message envoyé par cette activité mais pas encore reçu, c'est-à-dire de message en transit envoyé par cette activité. Ainsi, lorsque l'état global n est terminé, il ne peut plus

y avoir dans le système de message ni orphelin ni en transit. En effet, s'il existe encore :

- un message orphelin, c'est que l'activité réceptrice n'a pas encore pris le point de reprise n,
- un message en transit, c'est que l'activité émettrice n'a pas encore pris le point de reprise n.

Dans ces deux cas, l'état global n ne serait pas terminé. Finalement, les historiques n sur les activités peuvent être clos à partir du moment où la *dernière* activité à prendre le point de reprise n stocke ce point de reprise en mémoire stable. Pour déclencher la clôture des historiques, nous utilisons un message non fonctionnel, noté M_n^{gs}, qui indique à son récepteur que l'état global n est terminé et donc que l'historique n peut être clos. La dernière activité qui a stocké son point de reprise n sur la mémoire stable est chargée d'envoyer le message M_n^{gs} à toutes les activités du système. On note que ce message est non fonctionnel, et n'est donc pas pris en compte par le protocole.

La clôture des historiques doit correspondre à une *coupe cohérente* de l'exécution. En effet, si une requête est orpheline dans la coupe formée par la clôture des historiques, une activité peut se bloquer en attente infinie sur une promesse. Prenons le cas de la figure 4.8 : la requête Q_2 est orpheline dans la coupe formée par les clôtures, notées $\Diamond$. En cas de réexécution depuis la ligne de recouvrement n, le contenu *mais aussi l'envoi lui-même* du message Q_2 est conditionné entre autre par l'ordre de service, donc de réception, des requêtes Q_0 et Q_1. Or, ces deux réceptions ne font pas partie de l'historique du point de reprise C_k^n, donc l'ordre de réception n'est pas assuré ; la réexécution peut être différente de l'exécution de référence. Dans ce cas, l'envoi de Q_2 peut ne pas avoir lieu, et l'activité j resterait infiniment en attente par nécessite sur la promesse de requête de Q_2.

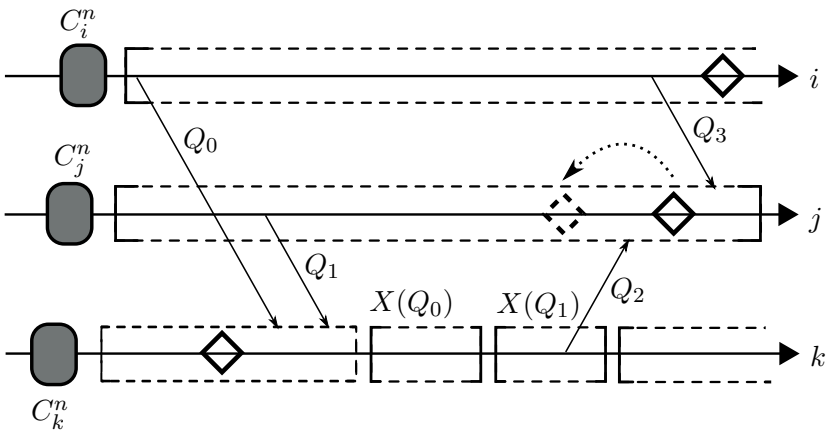

FIG. 4.8 – Les clôtures d'historiques doivent former une coupe cohérente

Nous appliquons le protocole proposé par Briatico dans [BRI 84] pour assurer la cohérence de la clôture des historiques : chaque message est estampillé avec l'index du dernier historique clos de l'émetteur du message. Lorsqu'une activité reçoit un message estampillé avec une valeur n supérieure à l'index du dernier historique clos localement, alors elle clôt l'historique n avant de prendre en compte le message. On peut ici directement appliquer cette solution, puisque contrairement aux points de reprise, les clôtures d'historiques peuvent être déclenchées à n'importe quel moment.

En pratique, la clôture d'historique nous permet aussi de journaliser sur la mémoire stable tous les messages en transit *en une seule fois*. En effet, un état global devient recouvrable, donc utilisable pour une reprise de l'application, à partir du moment où l'historique nécessaire est associé à chaque point de reprise le constituant. Donc, il n'est pas utile de journaliser en mémoire stable les messages identifiés en transit tant que l'historique n'est pas clos. En pratique, ces messages sont journalisés dans la mémoire *volatile* du récepteur, puis envoyé en une seule fois sur la mémoire stable avec l'historique. Les réponses sont placés dans le journal $\mathcal{R}_i^n$ prévu à cet effet, mais il n'est pas nécessaire d'avoir un journal pour les requêtes : les requêtes en transit sont placées *directement* dans l'historique. En effet, il existe par définition une place dans l'historique pour tous les messages en transit, puisque l'historique se ferme lorsqu'il n'y a plus de message en transit. Ainsi, lors de la réexécution depuis un point de reprise, les requêtes en transit n'ont pas besoin d'être réémises ; elles sont déjà dans la queue des requêtes en attente de l'activité.

4.1.5 Reprise après panne

La reprise après une panne dans le système est *globale* : toutes les activités doivent redémarrer depuis la ligne de recouvrement la plus récente. Une ligne de recouvrement est un état global stable ainsi que tous les historiques associés à chaque point de reprise.

Grâce à la synchronisation paresseuse introduite par les promesses de requêtes, le processus de reprise est asynchrone : les activités peuvent redémarrer depuis un point de reprise sans synchronisation globale préalable. Nous utilisons donc un numéro d'incarnation, introduit dans [STR 85], qui permet d'identifier les activités qui ont repris de celles qui n'ont pas encore repris. Le nombre de pannes détectées depuis le début de l'application est conservé en mémoire stable : lorsqu'une panne est détectée, ce nombre de pannes, noté $I^{effectif}$, est incrémenté et la reprise numéro $I^{effectif}$ est lancée. De plus, chaque activité conserve le numéro d'incarnation de sa dernière reprise, noté I_i, ainsi que l'index du point reprise utilisé pour la dernière reprise, noté LR_i.

Lorsque la panne d'une activité i est détectée, une activité j est choisie pour déclencher le processus de reprise. L'activité j détermine à partir des informations stockées sur la mémoire stable l'index de ligne de recouvrement n et le numéro d'incarnation de la reprise I, puis relance l'activité i sur une ressource disponible.

Avant de redémarrer, l'activité j envoie un message de reprise $M^{rec}_{n,I}$ à toutes les activités. Quand une activité reçoit un message de reprise $M^{rec}_{n,I}$, si l'incarnation de la reprise I est supérieure à son incarnation locale I_i, elle récupère le point de reprise C^i_n sur la mémoire stable, restaure son état à partir de C^i_n et, avant de redémarrer depuis ce point de reprise :

- met à jour son numéro d'incarnation local I_i, ainsi que LR_i,
- ajoute en queue de sa queue de requêtes en attente l'historique $\mathcal{H}^n_i$ associé à C^i_n,
- met à jour les futurs correspondants aux réponses contenues dans le journal $\mathcal{R}^n_i$ associé à C^i_n.

Pour éviter les communications entre incarnations différentes, les valeurs I_i et LR_i sont estampillées sur les messages. Lorsqu'une activité i reçoit un message de j :

- si $I_j < I_i$, alors le message est ignoré, et i envoie un message $M^{rec}_{LR_i,I_i}$ à j,
- si $I_j > I_i$, alors i démarre sa reprise depuis le point de reprise LR_j,
- sinon le message est délivré à i.

L'utilisation du numéro d'incarnation permet aussi de gérer les pannes pendant la reprise. En effet, même si la reprise globale n n'est pas terminée, une panne déclencherait une reprise avec un numéro d'incarnation différent. On repose ici sur l'hypothèse que l'incrémentation du numéro d'incarnation effectif $I^{effectif}$ est *atomique* ; deux reprises concurrentes ne peuvent pas avoir le même numéro d'incarnation. Manivannan propose dans [MAN 02] une solution intéressante qui ne suppose pas cette atomicité, mais qui repose sur des incréments du numéro d'incarnation qui sont fonctions de l'identifiant de l'activité qui déclenche la reprise. Dans ce cas, les reprises concurrentes peuvent être distinguées sans faire l'hypothèse que l'incrémentation du numéro d'incarnation est atomique.

4.1.6 Protocole

Nous résumons tout d'abord dans cette section le protocole proposé, puis proposons une description précise sous forme de pseudo-code dans la figure 4.9.

4.1.6.1 Synthèse

Comme la réexécution s'effectue depuis un état global incohérent, des problèmes de cohérence peuvent survenir. Nous résumons point par point comment ces différents problèmes sont gérés et résolus :

⇒ **Requête orpheline** : sa réception est remplacée par une promesse. Le problème de la double réception est donc écarté, et la position de cette réception est conservée durant la réexécution. Le contenu de la requête est assuré d'être le même que dans l'exécution de référence grâce aux historiques de réception.

⇒ **Réponse orpheline** : la réception du duplicata ne correspond à aucun futur et est donc ignorée. Le problème de la double réception est donc écarté. Le contenu de la réponse est assuré d'être le même que dans l'exécution de référence grâce à l'historique de réception.

⇒ **Requête en transit** : sa réception est journalisée dans l'historique de réception. Le message est donc reçu durant la réexécution, et sa position par rapport aux autres réceptions est conservée. L'ordonnancement causal est donc assuré, même pour les requêtes en transit.

⇒ **Réponse en transit** : sa réception est journalisée, et le futur correspondant est mis à jour avant la réexécution. Le message est donc reçu durant la réexécution. Le protocole assure que cette réponse ne peut pas être reçue en avance puisque l'envoi de la requête correspondante appartient au point de reprise.

⇒ **Réponse en avance** : l'attente par nécessité sur les promesses de requête permet d'assurer qu'aucune réponse ne peut être reçue en avance par une activité.

4.1.6.2 Algorithme

La figure 4.9 présente le protocole sous forme de pseudo-code. On notera que :
- la procédure `Initialisation()` est appelée à la création d'une activité,
- la procédure `TentativePoint()` est appelée entre chaque service de requête.

Chaque activité i maintient les valeurs suivantes :
- l'index N_i du dernier point de reprise pris,
- l'index N_i^{suiv} du prochain point de reprise à prendre,
- son numéro d'incarnation courant I_i,
- l'index du point de reprise utilisé lors de la dernière reprise,
- le compte à rebours TTC_i,
- l'historique de réception de requête $\mathcal{H}_i^k$ associé au point de reprise k,
- un journal de réponses $\mathcal{R}_i^k$ associé au point de reprise k.

Les valeurs transportées par un message $M_{i,j}$, et qui sont donc connues de l'activité réceptrice j sont :
- l'index N_i du dernier point de reprise de i,
- l'index H_i du dernier historique fermé par i,
- le numéro d'incarnation courant I_i de i,
- l'index LR_i du point de reprise utilisé pour la dernière reprise de i.

Enfin, nous notons $\mathbb{Q}_i^{rcv}$ la queue des requêtes en attente de l'activité i, et $\oplus$ l'opérateur de concaténation.

- `Initialisation()`
  $I_i = LR_i = N_i = 0$
  $N_i^{suiv} = 1$
  $TTC_i = TTC\_INIT$
  `TentativePoint()`

- `Reception(`$M_{i,j}$`)`
  si ($I_i == I_j$) *alors*
    si (`ouvert(`$\mathcal{H}_j^{N_j}$`)` *et* $N_j < H_i$) *alors* `fermer(`$\mathcal{H}_j^{N_j}$`)`
    si ($N_i < N_j$) *alors*
      si ($M_{i,j}$ *est une* $Q_{i,j}$) *alors* $\mathcal{H}_j^{N_j} \oplus Q_{i,j}$ *sinon* $\mathcal{R}_j^{N_j} \oplus R_{i,j}$
    sinon si (`ouvert(`$\mathcal{H}_j^{N_j}$`)`) *alors* $\mathcal{H}_i^k \oplus Q_{i,j}^{pmd}$
    si ($N_i > N_j$) *alors* $N_j^{suiv} = N_i$
  sinon si ($I_i > I_j$) *alors*
    i bloqué sur l'envoi de $M_{i,j}$
    `Recouvrement(`j, I_i, LR_i`)`
  sinon
    envoyer M_{LR_j,I_j}^{rec} à i, ignorer $M_{i,j}$

- `Reception(`M_n^{gs}`)`
  si (`ouvert(`$\mathcal{H}_i^{N_i}$`)` *et* $N_i < n$) *alors* `Fermer(`$\mathcal{H}_i^{N_i}$`)`

- `Fermer(`$\mathcal{H}_i^k$`)`
  stocker sur mémoire stable $\mathcal{H}_i^k$ et $\mathcal{R}_i^k$ avec C_i^k
  `ouvert(`$\mathcal{H}_i^k$`)` = faux

- `TentativePoint()`
  si ($N_i^{suiv} > N_i$) *alors*
    `Point(`N_i^{suiv}`)`
  sinon
    si ($TTC_i == 0$) *alors* `Point(`$N_i + 1$`)`

- `Point(`n`)`
  capturer C_i^n
  dans C_i^n, $\forall Q_{j,i} \in \mathbb{Q}_i^{rcv}$, *si* $N_j \geq n$ *alors* $Q_{i,j}$ remplacé par $Q_{i,j}^{pmd}$
  $N_i = n,\ TTC_i = TTC\_INIT$
  créer $\mathcal{H}_i^n$ et $\mathcal{R}_i^n$
  `ouvert(`$\mathcal{H}_i^n$`)` = vrai
  stocker C_i^n en mémoire stable
  si état global n complet *alors* envoyer à tous M_n^{gs}

- `Panne(`i`)`
  $I^{global} = I^{global} + 1$
  `Recouvrement(`i, I^{global}, n`)`
  envoyer à tous $M_{n,I^{global}}^{rec}$

- `Reception(`$M_{n,k}^{rec}$`)`
  si ($I_i < k$) *alors* `Recouvrement(`i, k, n`)` *sinon* ignorer $M_{n,k}^{rec}$

- `Recouvrement(`i, k, n`)`
  Stopper activité
  Récuperer C_i^n et $\mathcal{H}_i^n$ et $\mathcal{R}_i^n$
  Modifier l'état de l'activité à partir de C_i^n
  $\mathbb{Q}_i^{rcv} \oplus \mathcal{H}_i^n$
  $I_i = k\ , LR_i = n$
  Mettre à jour les futurs associés aux réponses dans $\mathcal{R}_i^n$
  Redémarrer activité

FIG. 4.9 – Protocole de point de reprise

4.1.7 Causalité potentielle

Le protocole que nous proposons ici dans le cadre d'ASP repose implicitement sur une relation de causalité entre les évènements spécifique au modèle ASP (différente de la relation de Lamport). En effet, la notion de cohérence utilisée dans la description du protocole suppose que certains évènements, bien que consécutifs sur la même activité, ne sont pas forcément causalement liés. Nous montrons informellement ici comment cette relation est exploitée implicitement, et donc comment la connaissance de cette relation est nécessaire pour pouvoir modéliser et prouver de façon formelle dans le chapitre 5 la recouvrabilité des états globaux formés durant l'exécution.

L'utilisation de promesses de requête pour virtuellement annuler la réception des requêtes orphelines repose sur une causalité locale partielle. En effet, la relation de causalité pour ASP définie dans la section 3.1.5 précise que, tant qu'elle n'est pas servie, la réception d'une requête n'est causalement liée qu'aux autres réceptions de requêtes. Le positionnement de la promesse par rapport aux autres réceptions est donc suffisant ; il n'y pas d'autre relation causale à conserver.

C'est la même idée qui permet l'utilisation d'une liste de promesses de requêtes pour réordonner les réceptions de requête en avance de façon transparente, c'est-à-dire sans que l'activité réceptrice intervienne, et sans utiliser un mécanisme de temporisation des messages extérieur à l'activité. Même si l'activité reçoit les requêtes dans un ordre différent et à des moments différents relativement à son exécution de référence, le fait que les promesses de requête puissent être ordonnées entre elles et par rapport aux autres réceptions de requêtes suffit à assurer la cohérence.

L'attente par nécessité sur la promesse d'une requête orpheline repose elle aussi sur une causalité locale partielle. On autorise en effet la réexécution de l'activité depuis le point de reprise, même si le duplicata de la requête orpheline *n'a pas encore été reçu*. De manière plus générale, les branches d'exécution causalement indépendantes de l'évènement qui n'a pas encore eu lieu peuvent être exécutées sans risque d'incohérence. L'activité peut donc servir les requêtes qui ont été reçues avant la ou les requêtes orphelines, et recevoir d'autres requêtes puisque la position de la requête est conservée par la promesse. L'attente par nécessité uniquement sur le service des requêtes orphelines est donc suffisante pour assurer une cohérence minimum de la réexécution, et éviter la réception de réponse en avance.

Dans le cadre de la causalité stricte de Lamport, cette réexécution n'est pas correcte. En effet, des évènements déterministes dont le passé local n'a pas encore eu lieu sont exécutés. Pour assurer la cohérence au sens de Lamport, il faudrait bloquer *complètement* l'exécution tant que tous les messages orphelins ne sont pas reçus. La connaissance d'une relation de causalité plus précise nous permet ici une synchronisation plus ciblée, et donc plus efficace.

Dans le cas des réponses, la correction du protocole repose sur le fait que le premier évènement qui est une conséquence causale de la réception d'une réponse est le premier accès au futur correspondant. Or, comme on l'a vu dans la section 3.1.5,

cet évènement est particulier dans le modèle ASP : il est (comme le service d'une promesse de requête) gardé par un mécanisme d'attente par nécessité. Cette attente par nécessité fait que la réception d'une réponse a toujours lieu *avant* le premier accès à un futur, et donc que la relation de causalité entre ces deux évènements est toujours conservée. Il n'y a donc pas de mécanisme particulier à ajouter au protocole pour assurer le maintien des relations de causalité vis-à-vis des réceptions de réponses, comme c'est le cas pour les requêtes.

4.1.8 Comparaison avec d'autres travaux

Nous comparons ici notre approche avec différentes solutions existantes, d'abord par rapport aux solutions induites par messages en général, puis par rapport aux solutions capables de recouvrir une application depuis un état global incohérent.

4.1.8.1 Approches induites par message

Le protocole que nous proposons est inspiré des protocoles de points de reprises induits par messages basés sur les index, introduits par Briatico et al. dans [BRI 84]. Cette approche ne met en jeu aucune synchronisation additionnelle que celle des messages de l'application pour la création des états globaux durant l'exécution. De plus, elle ne met pas en jeu un transfert d'information important entre les processus durant une exécution sans panne, puisque seul un nombre entier doit être ajouté sur les messages de l'application.

Ces propriétés permettent d'envisager un surcoût faible pendant une exécution sans panne. Pourtant, dans [ALV 99], Alvisi et al. identifient les points négatifs des protocoles de type induit par messages, parmi lesquels :

- l'augmentation du nombre de points de reprise forcés avec la taille du système et selon les motifs de communication,
- l'indéterminisme des prises de points forcés qui complique la récupération de l'espace sur la mémoire stable occupé par des points de reprise inutiles,
- et le besoin d'une persistance forte pour déclencher les points de reprise forcés.

Les points de reprise forcés sont donc le talon d'Achille des protocoles induits par message, et de nombreux travaux ont proposé de réduire ce nombre. Par exemple, dans [BAL 99], Baldoni et al. proposent un protocole qui autorise les activités à augmenter l'index de point de reprise sans en prendre réellement en fonction d'une relation d'équivalence entre certains points de reprises locaux consécutifs. Hélary et al. dans [HéL 97a] se basent sur la prévention durant l'exécution de cycle "zigzag" sur points de reprise pour éviter les points de reprise forcés inutiles ; les auteurs montrent expérimentalement que le nombre de points de reprise forcés n'excède pas 4% du nombre de point de reprise total. Les mêmes auteurs introduisent dans [HéL 97b] la propriété de précédence virtuelle et prouvent l'équivalence entre cette propriété et la propriété d'absence de cycle "zigzag" dans une exécution. Cette propriété est utilisée par Baldoni et al. dans [BAL 98] pour proposer une classification des protocoles induits par message en fonction du fait qu'ils assurent ou non la propriété de précédence virtuelle, donc dans quelle mesure ils évitent les points de reprise inutiles.

A la différence des protocoles classiques, notre protocole n'est pas sujet à l'accumulation de points de reprise forcés ; aucun point de reprise n'est inutile. En effet, dans les protocoles classiques, ces points de reprise sont pris après un point régulier pour éviter l'incohérence de l'état global, et donc pour créer un état global recouvrable. Or, notre protocole permet de rendre recouvrable un état global non cohérent ; il n'y a donc *jamais* besoin de déclencher un point de reprise en plus du point régulier.

Cependant, dans notre cas, si les points de reprise sont limités aux seuls points réguliers, une activité doit accéder *deux fois* à la mémoire stable pour envoyer un point de reprise utilisable pour une reprise : d'abord envoyer le point de reprise lui-même puis envoyer l'historique et les journaux de messages. La différence avec les protocoles classiques est que le nombre d'accès à la mémoire stable est par contre déterministe. La récupération de l'espace sur la mémoire stable est donc simplifiée ; dès qu'un état global n est stable et que tous les historiques ont été reçus, toute autre information peut être effacée de la mémoire stable. On retrouve ici les qualités d'une approche par point de reprise synchrone.

Finalement, bien que basée sur une approche induite par message, notre solution ne souffre pas des défauts identifiés par Alvisi et al. dans [ALV 99] cités précédemment.

En cas de panne d'une seule activité dans le système, toutes les activités doivent redémarrer depuis la dernière ligne de recouvrement, c'est-à-dire depuis le dernier état global complété avec un historique clos. Plusieurs solutions ont été proposées pour éviter à toutes les activités de devoir reprendre à cause de la panne d'une seule d'entre elles. Nous ne nous penchons pas ici sur les solutions basées uniquement sur la journalisation des messages, qui permettent la reprise uniquement de l'activité fautive, mais sur les approches basées sur la construction d'état globaux.

Manivannan et al. proposent dans [MAN 02] un protocole par points de reprise induits par message qui permet à une activité de reprendre systématiquement depuis son dernier point de reprise en cas de panne, indexé n. Toutes les activités ne sont alors pas forcées de reprendre après cette panne. Les activités qui n'ont pas encore pris le point de reprise n ne redémarrent pas ; elles déclenchent simplement le point de reprise n puis continuent leur exécution.

Cette solution intéressante n'est cependant pas applicable dans notre cas puisqu'elle ne considère pas l'existence potentielle de messages orphelins dans les états globaux. Dans ce cas, une activité qui n'a pas encore pris le point de reprise n et qui ne redémarre pas contient peut être un message orphelin, qui sera donc réémis par une activité qui doit reprendre et donc reçu une nouvelle fois.

D'autres solutions se basent sur l'utilisation de vecteurs d'horloges pour déterminer les dépendances causales entre les activités. Ces informations sur les dépendances sont utilisées pour ne faire redémarrer que les activités qui ont été causalement liées à l'activité tombée en panne depuis le dernier point de reprise. Bien qu'applicable dans le cadre de notre protocole, nous n'avons pas ajouté ce

mécanisme. En effet, nous ne nous plaçons pas dans un contexte particulier en ce qui concerne le type d'application, puisque le modèle ASP et l'implémentation ProActive sont applicables à n'importe quel type d'applications distribuées. Or, nous pensons qu'un tel mécanisme ne peut se révéler utile que si l'on considère des applications avec un motif de communication donné. Par exemple, pour des applications de type maître-esclaves ou de type *branch-and-bound*, un tel mécanisme peut éviter la reprise inutile d'un certain nombre d'éléments de l'application. Mais dans le cas général, et particulièrement dans le cadre d'applications de type SPMD (*Single Program Mutliple Data*) telles que celles que nous utilisons pour évaluer notre protocole, une telle optimisation serait inutile. En effet, les relations de causalité entre les activités forment très rapidement un graphe complet, par communication de proche en proche. Dans ce cas, les activités sont rapidement toutes dépendantes les unes des autres, et la panne de l'une d'entre elle entraîne la reprise de tout le système.

4.1.8.2 Recouvrabilité d'états globaux non cohérents

Schulz et al. proposent dans [SCH 04] une solution à une problématique similaire à la nôtre. Il propose un mécanisme de tolérance aux pannes par point de reprise *automatique* dans le cadre de l'intergiciel de communication MPI, en basant la persistance des processus sur un compilateur spécialisé qui fournit une persistance faible ; les captures d'états sont possibles pendant les points de captures potentiels définis par l'utilisateur dans le code source. Comme on l'a vu, la création automatique d'états globaux cohérents est impossible : la persistance utilisée est faible, et l'intergiciel de communication MPI ne suppose pas des communications purement asynchrones. Les auteurs proposent donc un protocole qui identifie et journalise les évènements qui causent l'incohérence durant la création des états globaux. En cas de reprise, la réexécution est contrainte jusqu'à atteindre un état global cohérent à l'aide des journaux d'évènements.

La solution proposée par Schulz et al. est basée sur une approche synchrone : un processus central est élu et devient responsable de la coordination nécessaire à la création d'un état global. Un état global est créé en quatre phases, qui sont synchronisées par le processus central par envoi de messages non fonctionnels.

Le protocole que nous avons proposé ne nécessite pas de processus central, et ne repose pas sur des phases qui doivent être synchronisées de manière forte. De plus, un seul envoi de message non fonctionnel est nécessaire, le message qui signale la complétion de l'état global en cours. Mais à la différence de Schulz et al., la réception de ce message par *toutes* les activités n'est pas primordiale. En effet, la clôture d'historique déclenchée par la réception de ce message se propage ensuite par les messages applicatifs. On pourrait donc imaginer d'envoyer cet unique message non fonctionnel à un sous-ensemble seulement des activités du système, choisie selon le motif de communication de l'application de manière à assurer une propagation de la clôture des historiques.

Le protocole de Schulz et al. traite le problème des messages orphelins en les identifiant du coté de l'émetteur, et en annulant l'émission de ces messages en

cas de réexécution. De cette façon, le récepteur ne reçoit pas deux fois le même message.

Pour que les messages orphelins puissent être identifiés du côté de l'émetteur et filtrés pendant la réexécution, il faut que les processus, avant de redémarrer l'exécution, s'échangent les identifiants des différents messages orphelins. En effet, pendant l'exécution de référence, un message orphelin ne peut être identifié *que par le récepteur* du message. La reprise du système doit donc passer par une phase de synchronisation qui n'est pas requise par notre protocole.

Même si les messages orphelins sont filtrés, il faut assurer l'équivalence des duplicatas. Cette équivalence est assurée grâce à un journal des évènements non déterministes qui est construit pendant l'exécution de référence. Mais aucune cohérence de la réexécution n'est assurée ; un message peut être reçu par un processus avant d'avoir exécuté localement le passé causal de ce message. Si dans le cas de [SCH 04] et du cadre de l'intergiciel MPI, cette incohérence n'a pas d'effet sur l'exécution, nous avons vu qu'elle pose problème dans notre contexte dans le cas des réceptions de réponse. De manière plus générale, l'approche de Schulz et al. ne permet pas de recréer la synchronisation générée par les messages orphelins, puisque les messages orphelins ne sont plus envoyés durant la réexécution.

On retrouve une approche similaire dans les travaux de Vaidya sur le problème des *points de reprise décalés* (*skewed checkpointing*) [VAI 99]. Pour diminuer la contention autour de l'accès à la mémoire stable, Vaidya propose d'étaler dans le temps les points de reprise de chaque processus. De cette manière, on peut éliminer la contention d'accès à la mémoire stable, mais on ne peut plus assurer la cohérence des états globaux. En effet, les points de reprises sont décalés durant l'exécution, et ne peuvent donc plus tenir compte des contraintes de cohérence. La solution de Vaidya se base sur le concept de point de reprise logique et point de reprise physique présenté dans [WAN 93]. Le point de reprise logique est similaire dans notre protocole à la clôture de l'historique.

Le protocole proposé par Vaidya présente les mêmes aspects de synchronisation forte que celle de Schulz et al., durant l'exécution mais aussi pendant la reprise pour identifier les messages orphelins. De plus, il ne prend pas en compte le cas des réceptions de message en avance.

4.2 Implémentation

Le protocole proposé pour ASP a été implémenté dans l'intergiciel ProActive, et des expérimentations ont été menées pour évaluer les performances de ce protocole. Ces expérimentations représentent aussi une validation expérimentale du protocole. Cependant, le chapitre 5 présente une formalisation des idées introduites par le protocole, ainsi qu'une preuve de la recouvrabilité des états globaux constitués.

L'objectif premier dans les orientations choisies pour cette implémentation est de minimiser l'intrusion dans le code actuel de ProActive, pour faciliter l'évolu-

tion et la maintenance du code. Nous avons choisi une approche par interception : nous avons ajouté un méta-objet à l'architecture de l'objet actif, le `FTManager`, qui est responsable des aspects de tolérance aux pannes durant l'exécution (analyse des messages entrant et sortant, prise de point de reprise,...). Il en résulte une intrusion très faible dans le code du coeur de ProActive. En plus d'une maintenabilité accrue du code, nous n'avons constaté aucun surcoût sur le temps d'exécution entre la version 2.1 de ProActive sans le code de tolérance aux pannes et les versions supérieures incluant le mécanisme de tolérance aux pannes (bien sûr lorsque ce mécanisme n'est pas activé).

Le second objectif est de proposer une tolérance aux pannes transparente pour l'utilisateur : elle doit être proposée comme un service offert par la librairie, et ne doit pas nécessiter de modification du code source de l'utilisateur. Nous avons proposé pour cela un système de configuration simple et extensible, les *technical services*.

4.2.1 Configuration

Nous voulons que la tolérance aux pannes soit vue comme un *service non fonctionnel* qui peut être activé et configuré par un utilisateur *au moment du déploiement*, sans avoir eu à en tenir compte dans le code source de son application. Nous avons donc proposé durant cette thèse un mécanisme de configuration générique pour les services non fonctionnels dans ProActive, telle que la tolérance aux pannes [CAR 06b], mais aussi la balance de charge automatique [BUS 05b]. Ce mécanisme permet à l'utilisateur de configurer de façon simple un service non fonctionnel donné dans le descripteur de déploiement utilisé pour déployer l'application ; cette configuration est appliquée dynamiquement aux activités s'exécutant sur les ressources déployées.

Du point de vue de l'utilisateur, une application ProActive est déployée sur des nœuds virtuels, qui sont un ensemble logique de nœuds d'exécution. Nous proposons ici de configurer l'environnement d'exécution, le nœud, de manière à ce que cette configuration soit transmise aux activités s'exécutant dans cet environnement.

Les ressources dans un déploiement ProActive pouvant être de nature différentes (créées ou acquises), un nœud doit pouvoir être reconfigurable dynamiquement. En effet, l'infrastructure pair-à-pair de ProActive maintient un ensemble de nœuds qui sont déployés et configurés par l'administrateur de l'infrastructure. Si ces nœuds sont acquis par un utilisateur au moment du déploiement, ils doivent pouvoir être reconfigurés si nécessaire, puisqu'une application doit pouvoir utiliser indifféremment des ressources créées ou acquises.

Nous avons ajouté au nœud ProActive un système de configuration par variable d'environnement, à la manière des paramètres de la machine virtuelle Java. Dans le cas de Java, on peut déclarer et définir, soit au démarrage de l'application une variable grâce à la déclaration `-Dproperty=value`, soit dans le code source grâce à l'interface `System.setProperty(property,value)`, une variable qui pourra être lue ou modifiée par tous les processus s'exécutant dans cette ma-

chine virtuelle. L'utilisateur de ProActive peut de la même manière déclarer et définir des variables qui seront accessibles par toutes les activités s'exécutant au sein d'un même nœud à l'aide des méthodes `Node.setProperty(property,value)` et `Node.getProperty(property)`.

A l'aide de ce système de configuration du nœud par variable d'environnement, nous proposons une solution de configuration générique, simple et extensible des services non fonctionnels qui peuvent être offerts par la librairie. Ce mécanisme est utilisé dans ProActive pour configurer la tolérance aux pannes au moment du déploiement de l'application.

4.2.1.1 Technical Services

Un *technical service* est un service non fonctionnel offert par la librairie qui peut être activé et configuré par une application au moment du déploiement. A chaque service non fonctionnel correspond une classe implémentant l'interface `TechnicalService`. Cette classe définit comment configurer un nœud d'exécution en fonction d'un ensemble de paramètres qui seront spécifiés au déploiement.

Du point de vue de l'utilisateur, un service non-fonctionnel est un ensemble de couples clé-valeur à définir dans le descripteur de déploiement, chacun de ces couples configurant un aspect du service non fonctionnel. En pratique, un service non fonctionnel est configuré dans le descripteur de déploiement dans le bloc XML `technicalServiceDefinitions`, comme dans l'exemple suivant. Ce bloc est lié à la définition d'un nœud virtuel ; la configuration définie est appliquée à tous les nœuds liés à ce nœud virtuel. Une configuration de service non fonctionnel peut être appliquée à plusieurs nœuds virtuels.

```
<technicalServiceDefinitions>
 <service id="service1" class="services.Service1">
  <arg name="name1" value="value1"/>
  <arg name="name2" value="value2"/>
  ...
 </service>
 <service id="service2" class="services.Service2">
 ...
</technicalServiceDefinitions>
```

Un service est identifié de manière unique dans le descripteur par son `id`. Cet identifiant est utilisé pour appliquer un service à un nœud virtuel. L'attribut `class` définit l'implémentation du service, une classe implémentant l'interface `TechnicalService` :

```
public interface TechnicalService {
      public void init(Map argValues);
      public void apply(Node node);
}
```

Les paramètres de configuration du service sont spécifiés par la balise `arg`. Ces paramètres sont passés à la méthode `init` sous forme d'une table associant le nom du paramètre et sa valeur. La méthode `apply` prend en paramètre un nœud

sur lequel un service doit être appliqué ; cette méthode est appelée automatiquement sur tous les nœuds qui sont acquis ou créés lors du processus de déploiement avant que ces nœuds ne soient passés à l'application.

Un nœud virtuel ne peut pour le moment être configuré que par *un seul* service non fonctionnel. Nous pensons en effet que deux services non fonctionnels correspondant à deux méta-objets différents et deux classes `TechnicalService` différentes ne peuvent pas être considérés par défaut comme compatibles. Ce problème de composition de service relève du domaine de la programmation par aspect [KIC 01], qui est en dehors du cadre de cette thèse.

```
<virtualNodesDefinition>
 <virtualNode name="virtualNode1" property="multiple" serviceRefid=
    "service1"/>
</virtualNodesDefinition>
```

4.2.1.2 Application à la tolérance aux pannes

Dans le cas de la tolérance aux pannes, nous avons donc fourni une classe `FaultToleranceService` ; elle définie comment la configuration de la tolérance aux pannes déclarée dans un nœud doit être appliquée sur les activités qui s'exécutent dans ce nœud. L'utilisateur doit spécifier dans le descripteur de déploiement la valeur du TTC, l'adresse de la mémoire stable, la disponibilité du nœud comme ressource utilisable en cas de panne, l'adresse d'un serveur de localisation potentiellement différente de l'adresse de mémoire stable. Il peut aussi ajouter, de façon optionnelle, l'utilisation des optimisations présentées dans la section 4.2.3, telles que l'envoi de point de reprise asynchrone ou la conservation en local d'une copie du dernier point de reprise.

```
<technicalServiceDefinitions>
 <service id="ft-service" class="services.FaultTolerance">
  <arg name="checkpointServer" value="rmi://host1/server"/>
  <arg name="locationServer" value="rmi://host2/LocationServer"/>
  <arg name="checkpointingMode" value="asynchronous"/>
  <arg name="TTC" value="60"/>
 </service>
</technicalServiceDefinitions>
```

L'exemple ci-dessus est une configuration de tolérance aux pannes avec deux serveurs distincts pour la localisation et le stockage des points de reprise, une période de point de reprise de 60 secondes. De plus, l'utilisateur ici spécifie que l'envoi des points de reprise doit se faire en mode asynchrone.

4.2.2 Mémoire stable, ressources et localisation

En plus du protocole de création d'états globaux et du protocole de reprise, nous avons intégré dans l'implémentation les différents services nécessaires au fonctionnement en pratique de la tolérance aux pannes :

- un service de stockage des points de reprise,
- un service de détection de pannes,

- un fournisseur de ressources qui peuvent être utilisées pour redémarrer la ou les activités défaillantes si le nœud dans lequel elles s'exécutaient n'est plus disponible,
- un service de localisation pour activités qui ont été redémarrées sur un autre nœud suite à une panne.

L'implémentation robuste de ces services n'étant pas un objectif de cette thèse, nous proposons une implémentation centralisée. Nous avons implémenté un serveur de tolérance aux pannes unique capable de rendre ces différents services, et basé sur le protocole de communication RMI. L'accès à ce serveur unique peut être spécifié en une seule fois dans la définition d'un *technical service* de tolérance aux pannes en utilisant l'argument `globalServer` :

```
<arg name="globalServer" value="rmi://host1/globalServer"/>
```

Cependant, chacun de ces services correspond à une interface spécifique et peut donc être implémenté de manière indépendante simplement en étendant le code existant.

4.2.2.1 Détection de pannes

Nous considérons ici qu'une activité est en panne si la connexion vers cette activité est impossible, c'est-à-dire si une exception est levée par la couche de communication. Même si l'activité soupçonnée n'est pas réellement en panne mais que le chemin réseau pour y accéder est coupé, le redémarrage de l'application est quand même possible. L'utilisation de numéros d'incarnations sur les messages permettra aux autres activités d'identifier une telle activité "zombie", d'ignorer ses messages et de terminer l'exécution de cette activité.

La détection de panne peut être réalisée par les activités entre elles. En effet, l'envoi de message étant soumis dans ProActive à un rendez-vous, une activité qui en contacte une autre doit attendre la fin de la connexion. Elle peut donc détecter d'éventuelles erreurs de connexion, et déclencher le processus de recouvrement si nécessaire.

Pourtant, cette solution pour la détection de pannes n'est pas suffisante. En effet, si l'activité tombée en panne n'est contactée par aucune autre, la panne n'est jamais détectée. Le serveur de tolérance aux pannes propose donc un service de détection de pannes, basé sur un mécanisme de *heartbeat messages*, ou messages de vie. Ce service est chargé de contacter régulièrement chacune des activités de l'application en appelant de *manière synchrone* une méthode dédiée sur l'activité. L'activité est considérée comme en panne si la connexion pour réaliser cet appel est impossible. Dans ce cas, une activité est choisie par le service pour déclencher le recouvrement de l'application.

4.2.2.2 Ressources

Ce serveur propose aussi un service de fourniture de ressources. L'activité chargée du recouvrement contacte ce service pour obtenir un nœud d'exécution

qui peut accueillir l'activité en panne. Les ressources peuvent être collectées par le service de deux manières :

- l'utilisateur peut au moment du déploiement spécifier à l'aide des *technical services* les nœuds qui peuvent recevoir des activités qui doivent être redémarrées. Si le *technical service* associé à un nœud définit la propriété `resourceNode` dans le descripteur de déploiement :

  ```
  <arg name="resourceNode" value="rmi://host1/ressourceServer"/>
  ```

  alors le nœud s'enregistre automatiquement comme ressource disponible en cas de panne auprès du service de ressource démarré sur la machine `host1`.
- le serveur peut aussi utiliser une infrastructure pair-à-pair sous-jacente pour récupérer des nœuds sur lesquels les activités pourront être redémarrées. Dans ce cas, le serveur doit être démarré avec l'option `-p2p rmi ://url1/entry1 rmi ://url2/entry2 ...`, où les URLs passées en paramètre sont des points d'entrée possibles du réseau pair-à-pair.

Ce service de ressource ne tient pas compte des problèmes de placement des activités ; le choix du nœud utilisé comme ressource pour redémarrer une activité ne suit pas de politique précise (ordre FIFO dans le cas de ressources enregistrées au déploiement, et indéterministe pour les ressources obtenues par le réseau pair-à-pair). Cependant, ProActive fournit un service de balance de charge qui pourrait être utilisé pour optimiser le placement des activités après un recouvrement.

Certaines solutions de tolérance aux pannes comme la plate-forme GatoStar [FOL 94] proposent une unification de la tolérance aux pannes et du placement des processus, en fonction de critères physiques tels que la charge des machines, mais aussi en fonction des caractéristiques de l'application, telles que le taux global de communication ou le temps processeur consommé.

4.2.2.3 Localisation

Enfin, le serveur de tolérance aux pannes implémente aussi un service de localisation. Lorsqu'une activité en panne est détectée par le service de détection ou par une autre activité, le service de localisation est prévenu et l'activité est signalée en panne. Dès qu'une activité redémarre, elle contacte le service de localisation pour préciser sa nouvelle localisation en lui envoyant une référence sur elle-même.

Si une activité i constate qu'une référence vers une activité j n'est plus valable, c'est-à-dire que la connexion est impossible, elle contacte le service de localisation et lui envoie cette référence invalide vers j. Plusieurs cas sont possibles :

- si l'activité j n'est pas signalée en panne, et que la référence invalide reçue est la même que la référence actuellement associée à j, alors j est potentiellement en panne,
- si l'activité j n'est pas signalée en panne, et que la référence invalide reçue n'est pas la même que la référence actuellement associée à j, alors la référence actuellement associée à j est renvoyée à l'appelant i,
- si l'activité j est signalée en panne, alors l'appel de i est temporisé jusqu'à ce que j ait contacté le serveur pour préciser sa nouvelle localisation.

Il faut noter que ce service de localisation ne repose pas sur l'existence d'un *processus* stable, mais seulement sur l'existence d'une mémoire stable. En effet, toutes les données sont mises à jour par les activités elles-mêmes : il existe sur la mémoire stable une table qui recense toutes les activités du système, ainsi que leur état supposé et leur localisation.

4.2.3 Optimisations

Nous présentons ici deux optimisations qui ont été implémentées avec le protocole proposé. Ces deux optimisations ont pour but de diminuer respectivement le surcoût du mécanisme de tolérance aux pannes pendant l'exécution et le temps de recouvrement de l'application après une panne.

4.2.3.1 Capture synchrone, envoi asynchrone

L'utilisation de la sérialisation Java rend impossible une capture d'état des activités non-bloquante. Bien que Java propose depuis la version 1.5 un mécanisme de *copy-on-write* pour les collections d'objets de type `List` et `Set`, ce mécanisme ne peut pas s'appliquer à un graphe d'objets quelconques, et donc en particulier à une activité. L'activité est donc forcément bloquée pendant la capture d'état.

Cependant, il est possible de séparer la capture proprement dite et l'envoi de l'image sur la mémoire stable, comme suggéré dans la section 2.2, par Sens dans [SEN 95] ou encore par Zheng et al. dans le cadre de l'intergiciel Charm++ [ZHE 04]. Nous avons donc implémenté un mécanisme d'envoi asynchrone des points de reprise. Lorsque la capture d'état est terminée, l'image sérialisée est stockée dans le méta-objet responsable de la tolérance aux pannes, et l'activité est débloquée. Un processus (une *thread* Java) lié à ce méta-objet est alors chargé d'envoyer le point de reprise sur la mémoire stable, en parallèle de l'exécution de l'activité.

Si cette optimisation permet de minimiser le temps d'inactivité, elle implique une utilisation de la mémoire quasiment doublée. En effet, deux images de l'activité, l'activité elle-même et la capture, doivent à un instant donné cohabiter dans le nœud d'exécution. L'envoi asynchrone du point de reprise est donc optionnel. Cette option peut être spécifiée par l'utilisateur dans la configuration de la tolérance aux pannes définie dans le descripteur de déploiement à l'aide de la propriété `checkpointMode` :

```
<arg name="checkpointingMode" value="asynchronous"/>
```

La valeur par défaut de la propriété `checkpointingMode` est `synchronous`.

4.2.3.2 Copie locale du dernier point de reprise

Nous avons aussi ajouté à l'implémentation une option qui permet à une activité de conserver localement une copie de son dernier point de reprise, comme proposé par Lemarinier et al. dans [LEM 04] dans le cadre d'un protocole par points de reprise coordonnés. Si le recouvrement de l'application est nécessaire depuis

ce point de reprise, l'activité n'a pas besoin d'accéder à la mémoire stable pour récupérer les données nécessaires. Cette optimisation minimise la contention sur la mémoire stable en cas de recouvrement et diminue de façon significative le temps de recouvrement de l'application, donc le surcoût engendré par une panne.

Cependant, une activité ne doit pas toujours repartir depuis son dernier point de reprise. Si une panne a lieu entre le déclenchement de la création d'un état global $n+1$ (c'est-à-dire quand au moins une activité prend le point de reprise $n+1$) et la clôture de l'historique de cet état global, le système doit repartir depuis l'état global précédent n. Donc les activités qui ont sauvegardé localement le point de reprise $n+1$ ne peuvent pas l'utiliser, et doivent récupérer le point de reprise n en mémoire stable. Ce cas est bien sûr détecté et géré automatiquement par le protocole de recouvrement.

Cette optimisation implique une augmentation de l'espace mémoire pour chaque activité : il faut qu'une copie complète de la capture d'état soit disponible localement. De plus, il serait possible de conserver sur chaque activité les *deux* derniers points de reprise. Dans ce cas, une activité disposerait toujours localement des données pour redémarrer, même si l'état global en cours de création n'est pas terminé. Mais cela impliquerait encore un doublement de l'espace requis pour une probabilité faible d'utilisation. En effet, en pratique, la durée de création des états globaux, donc la probabilité d'une panne pendant cette création, est toujours très faible par rapport à la durée de l'application.

Nous avons donc choisi de mettre cette optimisation en option, comme dans le cas de l'envoi asynchrone des points de reprise. Cette option peut être spécifiée par l'utilisateur dans la configuration de la tolérance aux pannes définie dans le descripteur de déploiement à l'aide de la propriété `recoveryMode` :

```
<arg name="recoveryMode" value="local"/>
```

La valeur par défaut de la propriété `recoveryMode` est `remote`.

4.3 Expérimentations

Nous présentons dans cette section les expérimentations réalisées avec l'implémentation. Ces expérimentations permettent d'évaluer les performances du protocole proposé dans divers contextes, mais représentent aussi une validation expérimentale de ce protocole.

Les performances d'un protocole de tolérance aux pannes sont caractérisées par trois impacts sur une exécution :

- le surcoût dû à l'implémentation du protocole dans l'environnement d'exécution,
- le surcoût dû la création et à la sauvegarde sur mémoire stable des points de reprise,
- le surcoût engendré par une panne et une reprise durant l'exécution.

Bien sûr, ces trois surcoûts dépendent des propriétés de l'application considérée, mais aussi de la fréquence des points de reprise. Une fréquence de point de reprise élevée engendre un surcoût élevé, mais une perte de temps réduite en cas

de panne. Nous allons dans cette section mesurer expérimentalement l'impact sur les performances de ces trois aspects.

4.3.1 Environnement d'évaluation

Les expérimentations présentées ici ont été menées sur la grappe de Sophia-Antipolis de la grille Grid5000 [CAP 05]. Cette grappe est constituée d'un premier ensemble de 105 IBM eServer 325 (2 AMD Opteron 246 2 GHz avec 1 Mo de cache et 2 Go de mémoire), reliés en Gigabit Ethernet. Le deuxième ensemble de machines est constitué de 56 Sun Fire X4100 (AMD Opteron 275 2,2 GHz avec 1 Mo de cache et 4 Go de mémoire), reliés entre eux et au premier ensemble en Gigabit Ethernet. Des noyaux Linux 2.6.9 sont déployés sur tous les nœuds, ainsi qu'une machine virtuelle Java Sun 1.5.0.07.

Dans les expérimentations présentées ici, les nœuds IBM eServer 325 sont utilisés pour déployer l'application, et le serveur global de tolérance aux pannes présenté dans la section 4.2.2 est deployé sur un nœud Sun Fire X4100.
Enfin, chacun des résultats présentés ici est une moyenne sur 10 exécutions consécutives. Pour des raisons de stabilité des résultats de ces exécutions consécutives, on notera que pour toutes les expérimentions, on déploie sur chaque nœud physique un seul nœud d'exécution ProActive, et que chacun de ces nœuds ne contient qu'une seule activité.

4.3.2 Applications d'évaluation

Nous avons mené des expérimentations sur trois applications différentes. Ces trois applications sont de type *Single Program Multiple Data* (Programme Unique Données Multiples).

Nous allons rapidement présenter ces applications, de manière à identifier leurs caractéristiques. Pour chacune de ces applications, nous présentons les temps d'exécution non tolérante aux pannes ainsi que la taille d'un point de reprise d'une activité pour chaque configuration, selon le nombre de nœuds utilisés et la taille des données initiales. La notation $\emptyset$ indique que le nombre de ressources est trop faible pour le calcul demandé. De plus, nous donnons le motif de communication pour une taille de donnée et un nombre de nœuds fixe, ainsi que la quantité de données transmise par les messages entre les différentes activités. Pour ces graphiques, l'abscisse donne l'identifiant de l'activité réceptrice, et l'ordonnée donne l'identifiant de l'activité émettrice. La densité de la couleur indique le nombre de messages envoyés, ou la quantité de données envoyée selon les cas.

Noyau IS (*Integer Sort*) Un tri d'entiers distribué type *bucket-sort*.

Cette application est une implémentation en ProActive du noyau IS de la bibliothèque de *benchmarks* NPB (*Nas Parallel Benchmarks*) [BAI 95]. Les données d'entrée pour les applications implémentées dans la bibliothèque NBP sont standardisées, réparties en six classes S, W, A, B, C et D selon leur taille. La classe S est la plus petite, la classe D est la plus grande. Nous testons ici IS avec les classes A et B, qui correspondent respectivement à des tableaux d'entiers de taille 2^{23} et 2^{25}.

Noyau IS						
Nombre de nœuds		2	4	8	16	32
Classe A	Temps d'exécution (sec)	4,04	3,72	3,9	4,42	5,27
	Taille d'un point de reprise (Mo)	86,08	43,06	22,89	11,32	5,61
Classe B	Temps d'exécution (sec)	∅	10,05	7,42	7,06	6,73
	Taille d'un point de reprise (Mo)	∅	186,4	88,12	44,6	11,56

FIG. 4.10 – Temps d'exécution et taille de point de reprise pour IS

Les figures 4.11 et 4.12 présentent respectivement les motifs de communication et la quantité de données transmise pendant l'exécution de IS en classe B sur 16 nœuds.

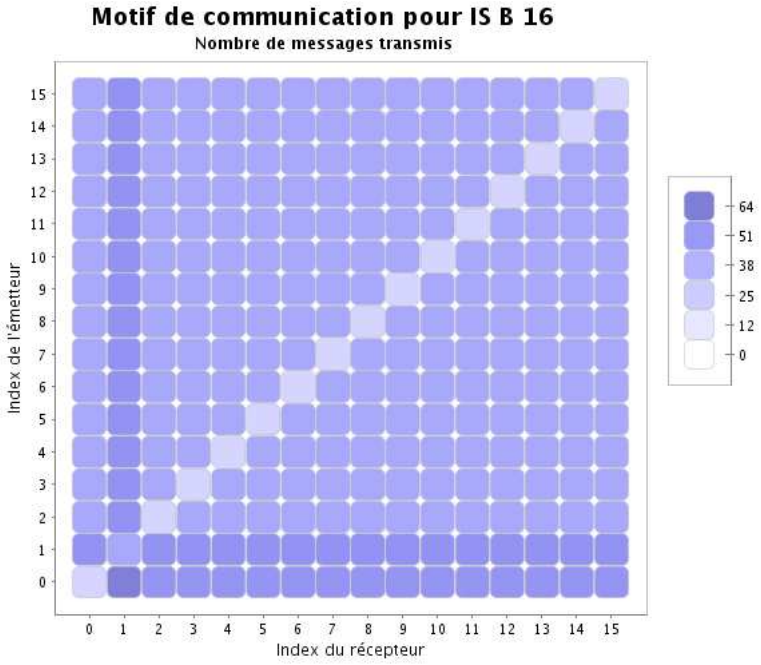

FIG. 4.11 – Motif de communication du noyau IS en classe B sur 16 nœuds

On peut voir que le nombre de messages envoyés durant l'exécution est faible (au maximum 65 messages). On constate aussi que le motif de communication est de type *all-to-all* régulier, c'est à dire que toutes les activités communiquent ensemble.

On remarque que l'activité indexée 1 émet et reçoit plus de messages que toutes les autres. Cette activité est utilisée dans notre implémentation comme réducteur : lorsqu'une valeur doit être calculée à partir de toutes les valeurs calculées par toutes les activités puis redistribuées à toutes les activités, c'est l'activité 1 qui collecte les valeurs puis redistribue le résultat.

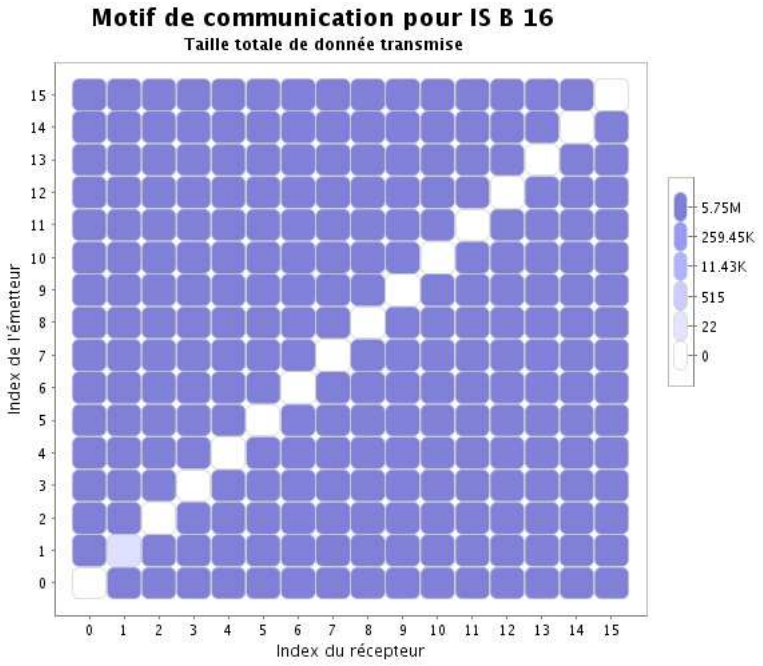

FIG. 4.12 – Quantité de données transmises du noyau IS en classe B sur 16 nœuds

Noyau CG (*Conjugate Gradient*) Une approximation de la plus petite valeur propre d'une matrice creuse.

Cette application est aussi une implémentation en ProActive d'un noyau de la bibliothèque de *benchmarks* NPB. Nous testons ici le noyau CG avec les classes A,B et C, qui correspondent respectivement à des matrices d'ordre 14000, 75000, 150000. On notera que les matrices, étant creuses, ne doivent pas être *entièrement* représentées en mémoire.

Noyau CG						
Nombre de nœuds		2	4	8	16	32
Classe A	Temps d'exécution (sec)	11,21	12,3	17,4	19,17	27,43
	Taille d'un point de reprise (Mo)	33,5	17,09	8,98	5,48	2,96
Classe B	Temps d'exécution (sec)	182,01	142,45	123,66	119,27	130,28
	Taille d'un point de reprise (Mo)	221,94	119,84	61,7	36,84	19,18
Classe C	Temps d'exécution (sec)	$\emptyset$	331,58	206,73	180,7	187,15
	Taille d'un point de reprise (Mo)	$\emptyset$	305,29	156,87	90,13	47,29

FIG. 4.13 – Temps d'exécution et taille de point de reprise pour CG

Les figures 4.14 et 4.15 présentent respectivement les motifs de communication et la quantité de données transmise pendant l'exécution de CG en classe C sur 16 nœuds.

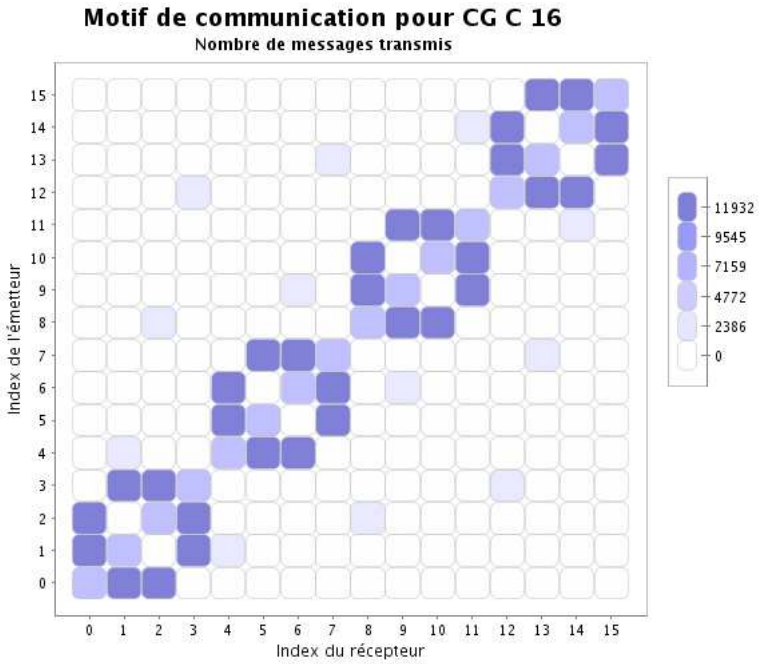

FIG. 4.14 – Motif de communication du noyau CG en classe C sur 16 nœuds

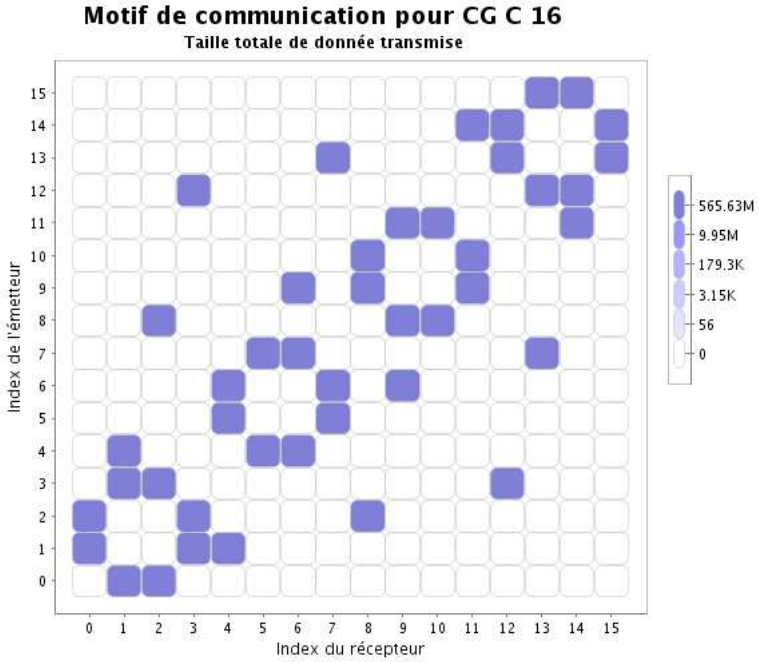

FIG. 4.15 – Quantité de données transmises du noyau CG en classe C sur 16 nœuds

C'est l'application testée qui engendre le plus de communication entre les activités (au maximum 11932 messages vers une même activité), la taille *moyenne*

d'un message étant petite (environ 48 Ko). On constate dans la figure que le motif de communication ne met pas en jeu de communication de type *all-to-all*, et qu'il n'y a pas d'activité utilisée comme point central de réduction.

Ce noyau est aussi très irrégulier, que ce soit au niveau du motif de communication (les activités cibles d'une activité donnée varient durant l'exécution) ou au niveau de la taille des données (les messages envoyées par une activité ont une taille très variable selon la cible et le moment de l'exécution).

Jacobi Résolution d'équations linéaires

Nous testerons aussi notre mécanisme de tolérance aux pannes avec l'application Jacobi. Cette application qui n'est pas une application d'évaluation standard nous permet cependant de contrôler plus finement les paramètres telle que la taille des données ou le nombre d'itération du calcul, puisque les données d'entrées ne sont pas standardisées.

La méthode de Jacobi est un algorithme qui détermine les solutions d'un système d'équations linéaires en réduisant itérativement une matrice de valeurs de type `double`. Une valeur dans la matrice est réduite en fonction de la valeur de ses quatre voisins cardinaux. Dans l'algorithme original, cette réduction itérative est stoppée lorsque la différence entre deux valeurs consécutives est inférieure à un seuil donné. Dans notre cas, nous fixons arbitrairement le nombre d'itérations à 500, de manière à pouvoir comparer les différents résultats.

Jacobi							
Nombre de nœuds		9	16	25	36	45	64
3000^2	Temps d'exécution (sec)	93,06	64,39	51,66	38,8	31,62	27,98
	Taille point de reprise (Mo)	15,68	8,83	5,66	3,93	2,89	2,22
5000^2	Temps d'exécution (sec)	226,61	134,1	96,94	80,51	67,81	56,15
	Taille point de reprise (Mo)	43,45	24,48	15,68	10,9	8,01	6,14
7000^2	Temps d'exécution (sec)	$\emptyset$	251,6	175,23	126,82	103,32	86,32
	Taille point de reprise (Mo)	$\emptyset$	47,94	30,7	21,3	15,67	12

FIG. 4.16 – Temps d'exécution et taille de point de reprise pour Jacobi

Les figures 4.17 et 4.18 présentent respectivement les motifs de communication et la quantité de données transmise pendant l'exécution de Jacobi sur une matrice carrée de 7000 de coté sur 16 nœuds.

La matrice qui doit être réduite est découpée et distribuée sur les nœuds disponibles, avec une activité par nœud. Le découpage est régulier, et est fait de manière à réduire le nombre de données à transmettre. Chaque activité réduit une sous-matrice de taille similaire, et doit communiquer avec au maximum 4 activités : les quatre activités avec lesquelles elle partage une bordure de sous-matrice.

On le constate sur les figures 4.17 et 4.18 : chaque activité communique avec au maximum quatre autres activités, et ce de manière parfaitement régulière. Le nombre et la taille des messages restent réguliers durant toute l'application. 1000 messages sont envoyés entre chaque activité, faisant transiter 6,69 Mo de données au total.

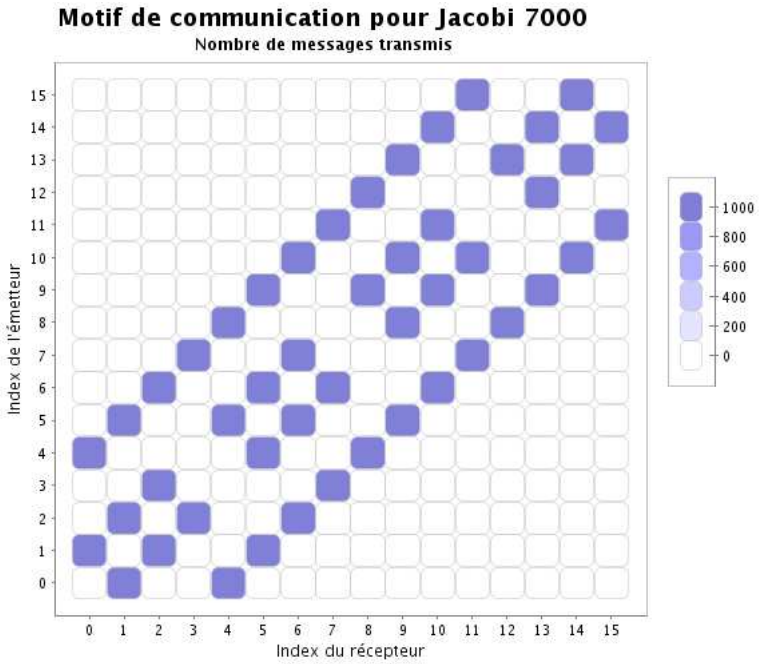

FIG. 4.17 – Motif de communication de Jacobi 7000 sur 16 nœuds

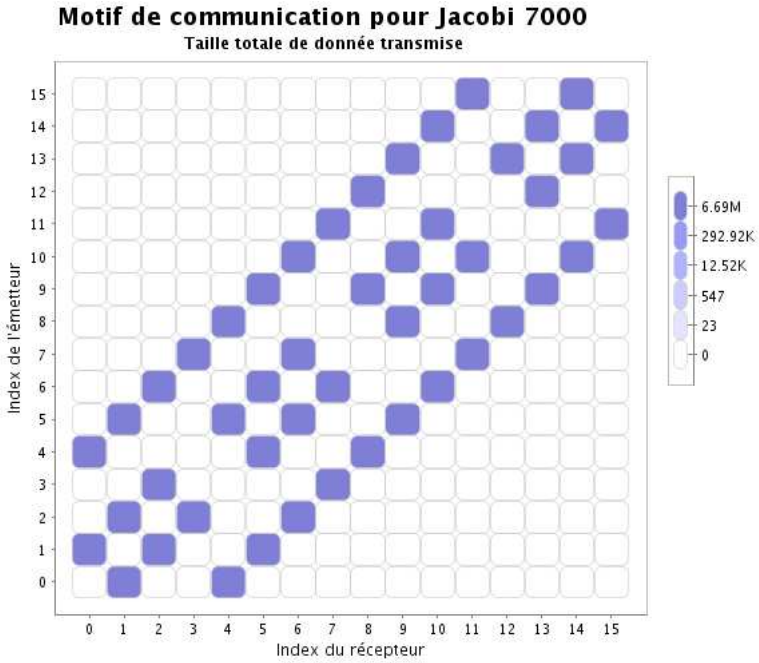

FIG. 4.18 – Motif de communication de Jacobi 7000 sur 16 nœuds

4.3.3 Coût de l'implémentation

Nous évaluons dans un premier temps le surcoût induit par l'implémentation du protocole dans ProActive, sans déclencher de point de reprise. Ce surcoût est dû en majeur partie au traitement des messages par le méta-objet responsable de la tolérance aux pannes, mais aussi à l'ajout de données sur les messages applicatifs (estampillage des messages).

Les graphiques de la figure 4.19 présentent le surcoût à l'exécution de l'implémentation sur le noyau IS. Le rapport est calculé de la manière suivante : $rapport = (T_{sans\ point\ de\ reprise} - T_{standard})/T_{standard}$, où $T_{standard}$ représente le temps d'exécution de référence, sans tolérance aux pannes, et $T_{sans\ point\ de\ reprise}$ représente le temps d'exécution avec la tolérance aux pannes activée, mais sans aucun point de reprise déclenché ($TTC = \infty$)

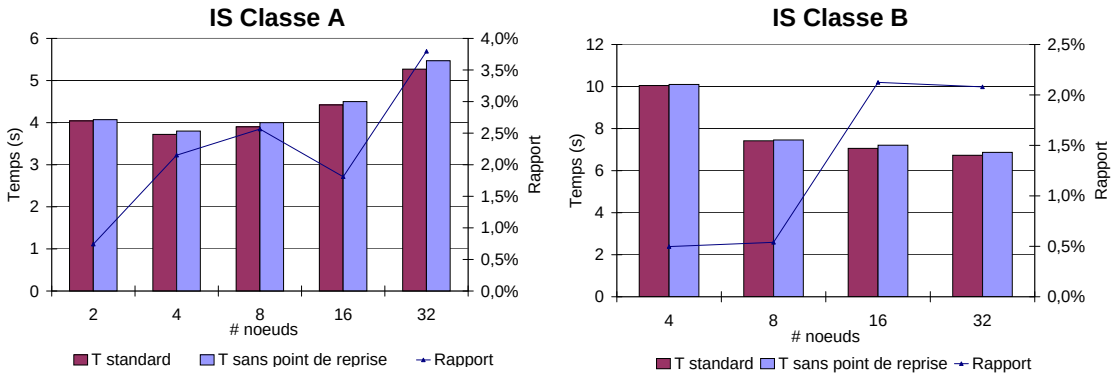

FIG. 4.19 – Surcoût de l'implémentation sur le noyau IS

On constate que le surcoût augmente globalement avec le nombre de nœuds, en restant inférieur à 4% en classe A, et à 2% en classe B. Cette augmentation avec le nombre de nœuds est dûe à l'augmentation du nombre de message transmis et à la diminution de la taille des messages. En effet, plus la taille du message diminue, plus le temps de traitement dû à la tolérance aux pannes est significatif par rapport au temps de traitement global du message (sérialisation, envoi sur le réseau,...). La différence de taille de message explique aussi la différence entre les surcoûts pour les classes A et B.

On peut donc s'attendre à une augmentation de ce surcoût pour le noyau CG, pour lequel la fréquence globale de communication est très importante avec des messages en moyenne de petite taille. C'est ce que l'on constate sur la figure 4.20, qui présente le surcoût à l'exécution sur le noyau CG en classe A, B et C. Ce surcoût reste globalement inférieur à 6%, avec une pointe à 7% pour la classe B sur 32 nœuds. On constate aussi que le surcoût augmente avec le nombre de nœuds, sauf dans le cas de la classe A. Ceci s'explique par le fait que même sur un petit nombre de nœuds, la taille des messages est très petite, et donc le surcoût du traitement des messages atteint son maximum dès 2 nœuds.

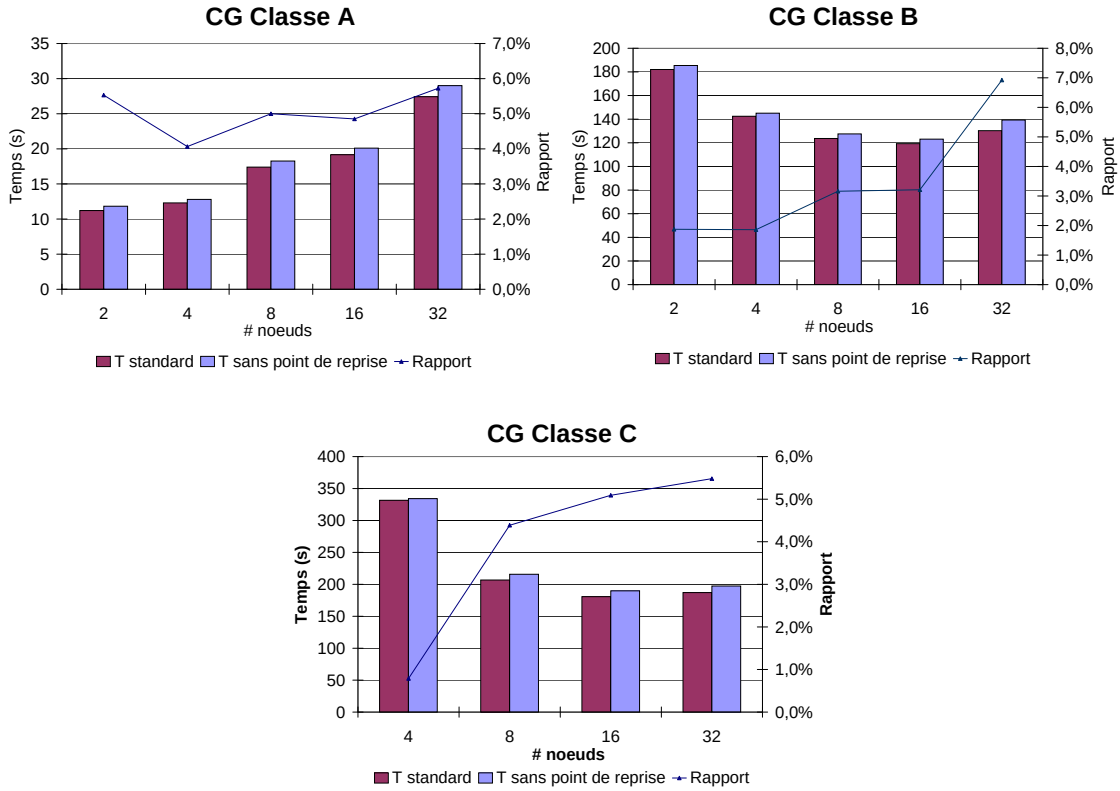

FIG. 4.20 – Surcoût de l'implémentation sur le noyau CG

Enfin, la figure 4.21 présente le surcoût à l'exécution sur l'application Jacobi sur des matrices de dimension 3000 et 7000. On constate que l'évolution du surcoût suit le même comportement que précédemment, mais aussi qu'il est globalement plus faible que dans le cas de noyau IS et CG, inférieur à 3%. Ceci s'explique par la meilleure capacité de passage à l'échelle de l'application Jacobi par rapport aux implémentations des noyaux IS et CG en ProActive. En effet, le temps de communication est mieux recouvert par le temps de calcul dans le cas de Jacobi. Par conséquent, le surcoût sur les communications est lui aussi partiellement masqué.

Le graphique 4.22 présente le surcoût sur les communications dû à l'implémentation de manière plus générale. Nous avons déployé deux activités. L'une d'elle envoie 500 Mo de données aléatoires à l'autre sous forme d'envoi de requêtes, avec différentes tailles de paramètre de la requête. Le paramètre est un tableau de type `byte`. Le nombre d'appels consécutifs est suffisant pour atteindre 500 Mo de données envoyées. On notera dans ce graphique que l'abscisse, la taille des messages, est logarithmique.

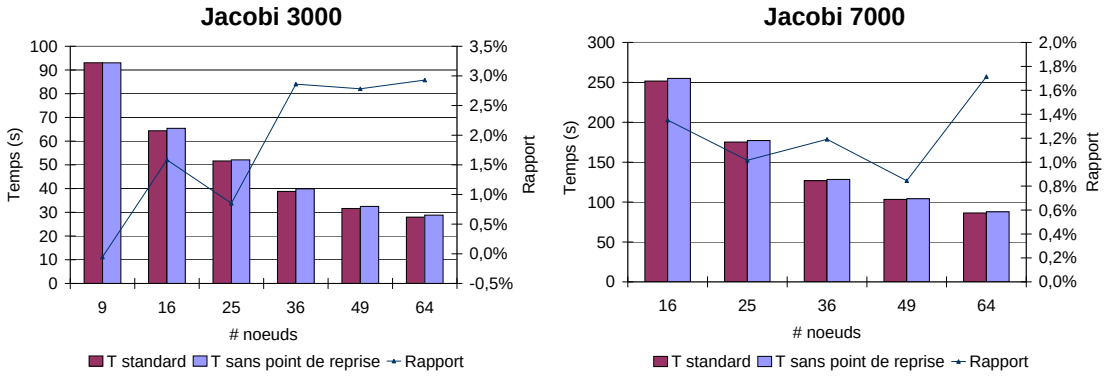

FIG. 4.21 – Surcoût de l'implémentation sur l'application Jacobi

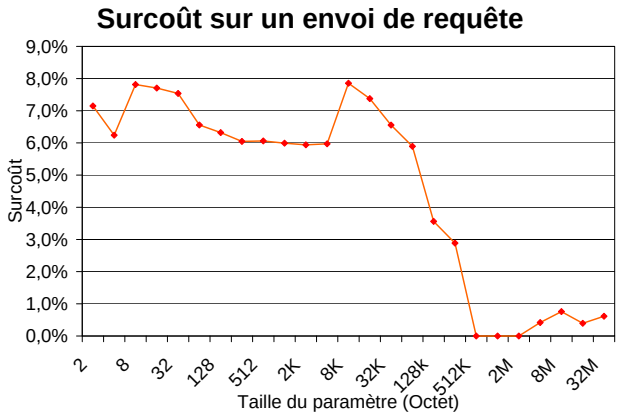

FIG. 4.22 – Surcoût sur les communications

On constate que le surcoût unitaire sur les communications est au maximum de 8% pour des messages de très petite taille, et devient négligeable à partir de messages de taille 512 Ko. On note ici que les données transmises étant des tableaux de `byte`, le temps de sérialisation des paramètres est minimum. Cette mesure est donc pessimiste puisque le temps de référence sans tolérance aux pannes est minimal.

4.3.4 Coût des points de reprise

Nous évaluons maintenant le surcoût induit par la prise d'un point de reprise durant l'exécution. Dans un premier temps, nous évaluons ce surcoût sur des exécutions pendant lesquelles un seul point de reprise est déclenché. Nous présentons le temps d'exécution supplémentaire par rapport à l'exécution standard, avec les deux modes synchrone et asynchrone d'envoi du point de reprise.

La figure 4.23 présente les temps d'exécution supplémentaires pour chacun des noyaux IS et CG dans différentes classes. On note que dans le cas du noyau IS, le mode d'envoi asynchrone n'a pas été testé ; le temps total de l'application n'étant pas suffisant, l'application termine *avant* que les points de reprise soient stockés en mémoire stable. En effet, nous considérons que l'envoi du point de reprise, bien qu'exécuté en parallèle de l'activité, peut influer sur le temps d'exécution puisqu'il consomme du temps CPU et de la bande passante.

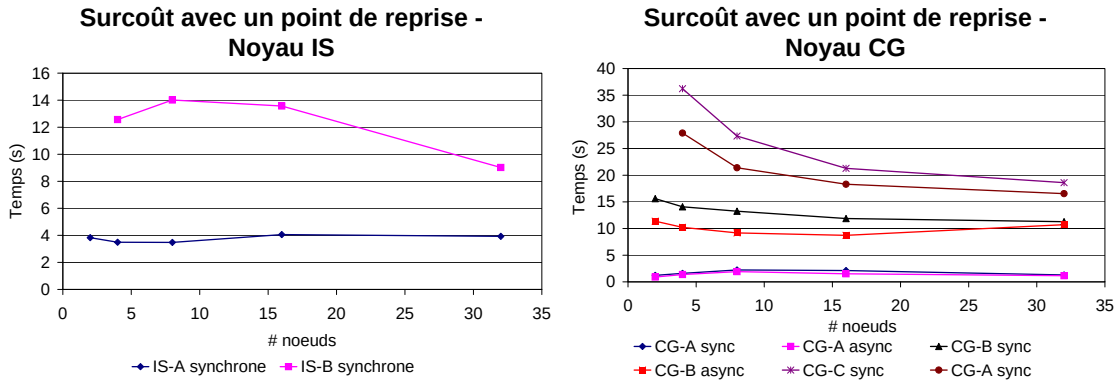

FIG. 4.23 – Surcoût avec un point de reprise pour les noyaux IS et CG

On constate d'abord bien sûr que le surcoût en temps augmente avec la taille des points de reprise ; au maximum 36,2 secondes pour CG en classe C sur 4 nœuds, avec un point de reprise de 305,3 Mo.

De plus, ce surcoût diminue avec le nombre de nœuds ; la taille des points de reprise diminuant, le temps de capture et d'envoi diminue en moyenne pour chaque activité. On constate qu'à partir d'une taille minimale, le surcoût ne diminue plus avec le nombre de nœuds ; par exemple pour IS classe A ou CG classe A, il reste constant.

Enfin, on peut constater que l'utilisation du mode d'envoi asynchrone réduit jusqu'à 30% le temps de point de reprise, ici dans le cas de CG en classe C sur 4 nœuds. Le gain obtenu par le mode asynchrone diminue avec la taille des points de reprise ; on peut constater par exemple que le gain dans le cas de CG classe A ne dépasse pas 2%. Par conséquent, ce gain diminue aussi avec le nombre de nœuds, la taille des points de reprise diminuant aussi.

Dans le cas de l'application Jacobi (figure 4.24), la différence entre les expérimentations est moins nette, les résultats des différentes expérimentations étant plus instables dans le cas de Jacobi que dans le cas des noyaux NAS. Pour les matrices de dimensions 3000 et 5000, le surcoût engendré reste de l'ordre de 5 secondes en mode synchrone et de 3 secondes en mode asynchrone. Pour ces dimensions, la taille des points de reprise étant relativement faible, le surcoût reste stable, comme on l'a constaté pour le noyaux CG classe A. La taille des points de reprise dans le cas de la matrice 7000 devient significative ; on constate que sur 16 nœuds, l'optimisation d'envoi asynchrone des points de reprise permet de diviser par deux le temps supplémentaire dû au point de reprise.

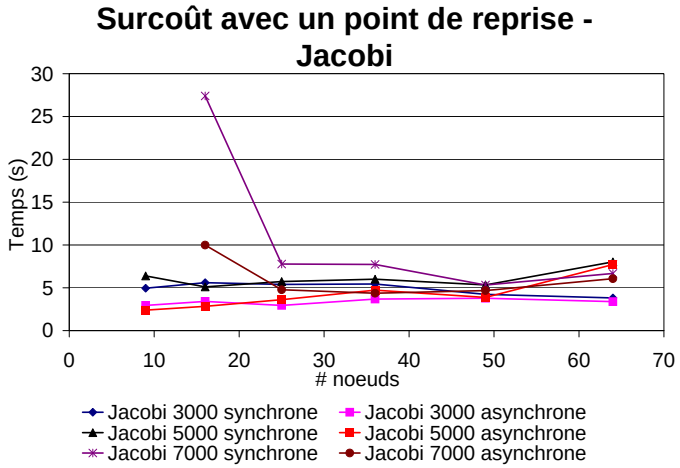

FIG. 4.24 – Surcoût avec un point de reprise pour l'application Jacobi

Bien sûr, les temps présentés ici dépendent fortement des caractéristiques de la mémoire stable employée, ici un serveur dédié sur la même grappe de machines. Nous avons donc d'abord dans un premier temps isolé le temps dû à un point de reprise pour une seule activité, en fonction de la taille de ce point. La figure 4.25 présente le temps pris par cette activité pour capturer son état puis l'envoyer sur la mémoire stable. On notera que les activités utilisées pour ce test ne contiennent qu'un tableau de type `byte` ; comme précédemment, le temps de sérialisation, donc de capture de l'état, est dans ce cas minimum.

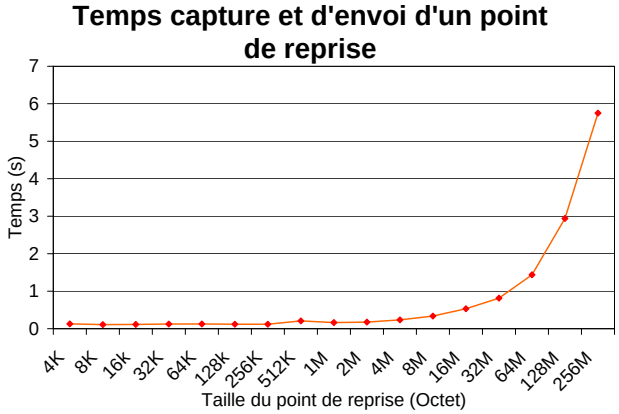

FIG. 4.25 – Temps de capture et d'envoi sur la mémoire stable d'une seule activité

La figure 4.25 montre que l'accès au serveur est rapide ; le temps d'envoi pour une seule activité devient significatif lorsque le point de reprise atteint plusieurs dizaines de mégaoctets. Ce test ne tient pourtant pas compte de la contention d'accès au serveur. Nous avons donc réalisé des tests avec le même type d'activités (ne contenant qu'un tableau de `byte`) qui ne communiquent pas entre elles, mais qui déclenchent un point de reprise au même moment. La figure 4.26 présente res-

pectivement en fonction du nombre de nœuds, les temps de points de reprise dans le cas où la taille totale de donnée à envoyer est de 1 Go, et dans le cas où chaque point de reprise fait 64 Mo. Pour le premier cas, la taille de chaque activité est modifiée pendant le test pour que la taille totale de toutes les activités atteigne 1 Go. Chaque courbe présente le temps moyen pris par les activités, mais aussi le maximum parmi toutes les activités. Ce temps maximum est en effet significatif, puisque on considère ici des applications fortement couplées ; l'arrêt d'une activité entraîne très rapidement l'arrêt global du calcul.

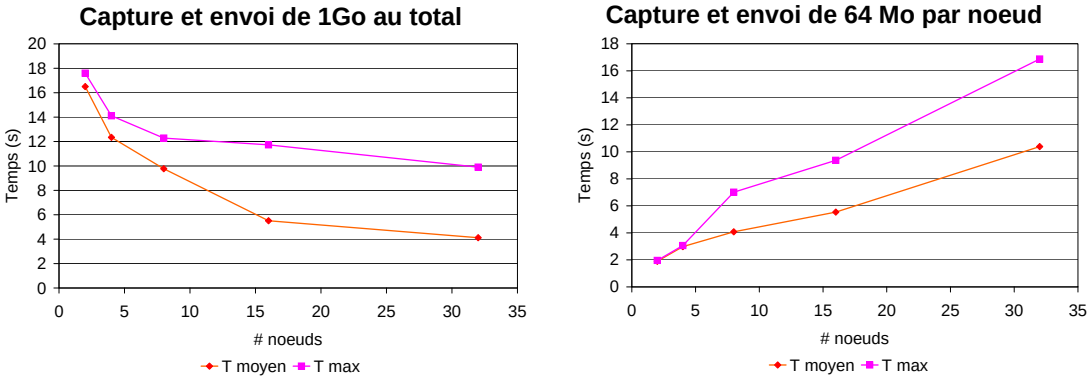

FIG. 4.26 – Temps de capture et d'envoi sur la mémoire stable d'activités concurrentes

On constate que l'augmentation du nombre de nœuds influe fortement sur le temps d'accès à la mémoire stable. Le temps d'envoi d'un point de reprise de 64 Mo est multiplié par 10 lorsque le système passe de 1 à 32 nœuds. Cette augmentation est bien sûr prévisible puisque la taille totale des données à stocker augmente avec le nombre de nœud.

Lorsque la taille totale de donnée à envoyer est fixe quelque soit le nombre de nœuds, ce qui est généralement le cas pour une application, on voit que le temps moyen et le temps maximum de point de reprise diminuent et se stabilisent vers une valeur minimum, ici autour de 4 secondes en moyenne. Pourtant, la quantité totale de donnée à envoyer reste la même ; cette diminution s'explique par le fait qu'en Java, toute communication entraine une sérialisation et une déserialisation des données à transmettre, un processus relativement coûteux par rapport au temps de transfert sur le réseau proprement dit. Cependant, contrairement au transfert sur le réseau, ce processus de sérialisation et de déserialisation tire parti de la parallélisation : il est plus rapide de déserialiser sur la mémoire stable 32 points de reprise de 32 Mo en parallèle que 2 points de reprise de 512 Mo. Le temps de point de reprise diminue donc sur chaque activité avec l'augmentation du nombre total d'activités.

Nous pouvons quand même constater le phénomène de contention sur l'accès à la mémoire stable : cette augmentation de la contention est caractérisée par le fait que le temps maximum diminue moins vite que le temps moyen avec le nombre de nœuds. En d'autres termes, les activités ont de moins en moins de données à

envoyer, donc le temps moyen diminue, mais l'attente maximum pour accéder à la mémoire stable augmente, et compense cette diminution pour les activités les plus malchanceuses.

Enfin, nous avons évalué le surcoût global induit par le protocole. Ce surcoût pour une application donnée est dépendant de deux facteurs : la fréquence de point de reprise représentée par la valeur du TTC et le nombre de nœuds. L'impact de la fréquence de point de reprise est caractérisé dans la figure 4.27 : pour CG en classe C et Jacobi sur une matrice de taille 7000, le surcoût global est donné en fonction du TTC. Pour cette expérimentation, nous avons porté le nombre d'itérations de calcul de 500 à 1000 pour l'application Jacobi, de manière à pouvoir tester des valeurs de TTC suffisamment grandes.

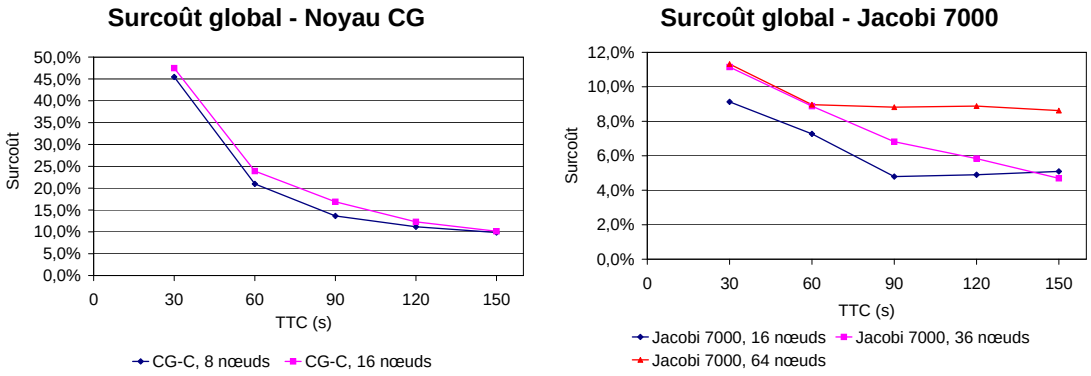

FIG. 4.27 – Surcoût global en fonction du TTC

On constate que ce surcoût diminue avec l'augmentation du TTC jusqu'à se stabiliser autour d'une valeur minimum de 10% pour CG sur 8 et 16 nœuds, et entre 5% et 9% selon le nombre de nœuds pour Jacobi. En particulier, nous avons constaté dans la figure 4.28 que le nombre de points de reprise pris dans toutes les expérimentations diminue proportionnellement à l'augmentation du TTC.

On peut donc conclure que le nombre de points de reprise et le surcoût global diminuent de façon linéaire avec la *fréquence* de point de reprise. En effet, notons que les graphiques 4.27 et 4.28 présentent le surcoût en fonction de la *période* de point de reprise. Or, les courbes obtenues sont de la forme $f(x) = a\frac{1}{x} + b$ avec x la période. Comme la fréquence est donnée par l'inverse de la période, cette fonction avec x la fréquence serait de la forme $f(x) = a'x + b'$, donc linéaire. Finalement, on ne retrouve pas comme dans les approches induites par message classiques d'augmentation imprévisible du nombre de points de reprise total à cause des points de reprise forcés.

4.3.5 Coût des pannes

Pour des raisons techniques, l'évaluation *précise* du coût d'une panne lors d'une exécution est difficile. Il est difficile d'isoler dans ce surcoût les parties dues

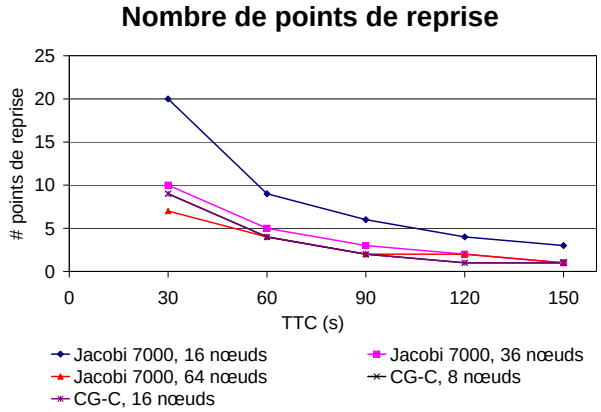

FIG. 4.28 – Nombre de points de reprise en fonction du TTC

au temps de détection de la panne, à la redistribution des données nécessaires pour le recouvrement, à la période de resynchronisation des activités et enfin au temps de calcul perdu. En particulier, le temps de calcul perdu causé par une panne dépend totalement du moment auquel cette panne a lieu. Par exemple, une panne qui a lieu quelques secondes après la création d'une ligne de recouvrement ne cause la perte que de quelques secondes de temps de calcul. Si au contraire cette panne a lieu quelques secondes avant la création d'une ligne de recouvrement, le temps de calcul perdu peut être important.

Pour tenter d'extraire le surcoût induit par une panne de manière générale, nous déclenchons une panne durant toutes les exécutions 10 secondes après la création d'une ligne de recouvrement. Plus précisément, la machine virtuelle de la dernière activité à avoir envoyé un point de reprise sur la mémoire stable est stoppée au bout de 10 secondes par un signal `kill`. On peut donc approximer ici le temps de calcul perdu à 10 secondes, bien que cette mesure ne puisse être stricte.

Au moment de stopper l'activité, le service de détection de panne est réveillé et découvre immédiatement l'activité stoppée ; le temps de détection de la panne peut donc être négligé dans cette expérience.

Nous évaluons donc le surcoût causé par une panne durant une exécution du noyau CG en classe B et C présenté dans la figure 4.29, et de l'application Jacobi avec des matrices de dimension 5000 et 7000 dans la figure 4.30. Dans chacun des graphiques, une ligne en pointillés noirs indique le temps de calcul perdu, puisque la panne est déclenchée 10 secondes après le dernier point de reprise.

Pour toutes ces expérimentations, nous présentons les temps de reprise avec et sans la conservation d'une copie du point de reprise en local, comme présenté dans la section 4.2.3, de manière à évaluer le gain apporté par cette optimisation.

Le temps de recouvrement augmente avec le nombre de nœuds, mais la vitesse de cette augmentation diminue avec le nombre de nœuds dans le cas de CG : les courbes sont croissantes concaves. Dans le cas de Jacobi, le temps de recouvre-

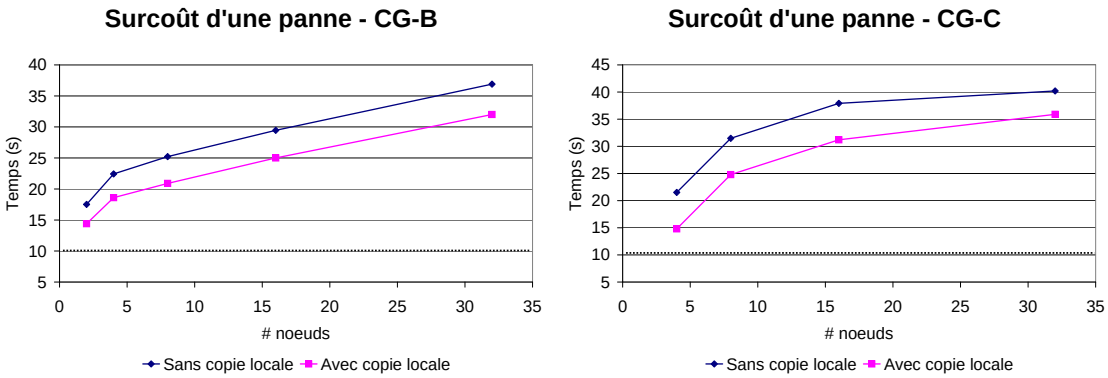

FIG. 4.29 – Surcoût causé par une panne pour le noyau CG

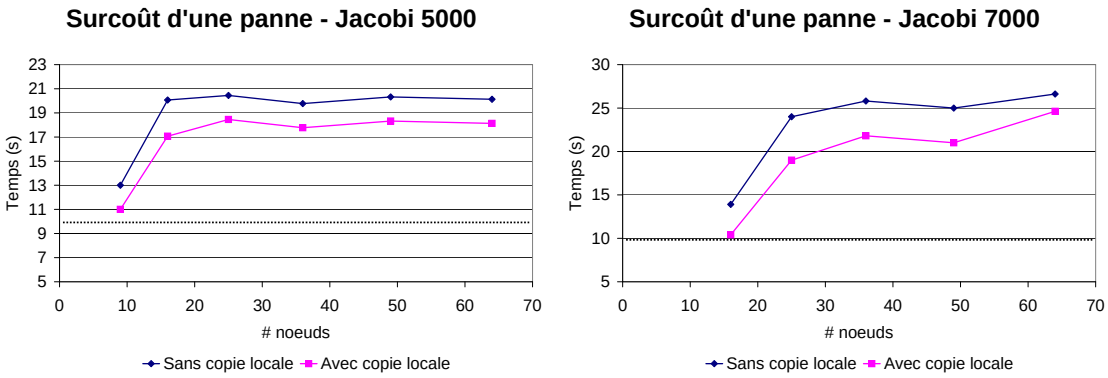

FIG. 4.30 – Surcoût causé par une panne pour l'application Jacobi

ment semble même se stabiliser à une valeur maximum d'environ 20 secondes dans le cas d'une matrice de dimension 5000 et de 25 secondes dans le cas d'une matrice 7000.

Ce ralentissement de l'augmentation du surcoût induit par une panne, ou la quasi-stabilisation dans le cas de Jacobi, s'explique par le fait que la récupération des points de reprise sur la mémoire stable s'effectue en parallèle. De manière symétrique à ce qu'on a vu dans le cas de la figure 4.26 (capture et envoi de 1 Go de donnée), même si la contention augmente, le temps moyen et maximum diminue car le processus de sérialisation sur la mémoire stable tire parti de la parallélisation. Il est en effet plus rapide de sérialiser 64 objets de 12 Mo en parallèle sur une même machine que 16 objets de 48 Mo : il est donc plus rapide pour 64 activités de récupérer en parallèle 12 Mo de données chacune sur la mémoire stable, que pour 16 activités de récupérer 48 Mo chacune.

Cette quasi-stabilisation du temps de recouvrement est aussi une conséquence du caractère asynchrone du la mécanisme de reprise dans le protocole proposé : il n'y a pas de phase de synchronisation stricte qui augmenterait de façon régu-

lière le temps de recouvrement avec le nombre de nœuds. La resynchronisation des activités se fait sans blocage global grâce aux promesses de requêtes, qui ne synchronisent l'activité que si nécessaire, et uniquement localement.

Enfin, on observe que dans toutes les expérimentations, le gain obtenu grâce à la copie locale du dernier point de reprise reste quasiment constant jusqu'à 49 nœuds. Dans le cas de CG, le gain obtenu sur le temps de recouvrement est de l'ordre de 4 secondes en classe B et de l'ordre de 7 secondes en classe C. Pour Jacobi, on observe un gain de 2,5 secondes pour la matrice 5000 et 4 secondes pour la matrice 7000.

4.4 Conclusion

Nous avons présenté dans ce chapitre le protocole que nous avons développé durant cette thèse pour le modèle ASP. Ce protocole tient compte mais aussi tire parti des spécificités de ce modèle et de son implémentation ProActive, avec en particulier la possibilité de reprendre depuis des états globaux incohérents, et la capacité à éviter les réceptions de réponse en avance.

Cette solution se base essentiellement sur l'utilisation de promesses de réception de requête. Elles permettent tout d'abord d'assurer de façon uniquement locale la synchronisation nécessaire pour éviter les réceptions de réponses en avance. De plus, elles sont utilisées pour réordonner les réceptions de requêtes durant la réexécution, et ainsi assurer l'équivalence de la réexécution avec l'exécution de référence.

Nous avons aussi décrit dans ce chapitre l'implémentation de ce protocole dans l'intergiciel ProActive, ainsi qu'un mécanisme de configuration simple d'utilisation à travers les descripteurs de déploiement. Cette implémentation fait actuellement partie de la version téléchargeable de ProActive, et est disponible avec une documentation et des exemples d'utilisation. Elle constitue aussi une base pour l'intégration d'autres protocoles de tolérance aux pannes : nous avons par exemple implémenté sur cette base un protocole simple de journalisation pessimiste par le récepteur. Ce protocole simple est un premier élément de l'extension pour le contexte des grilles de calcul présentée dans le chapitre 6.

Enfin, cette implémentation nous a permis d'évaluer les performances du protocole dans un contexte d'utilisation réel. Nous avons constaté dans le cadre d'applications communicantes de type SPMD un surcoût global sur le temps d'exécution variant de 10% à 45% pour le noyau CG et de 4% à 12% pour l'application Jacobi, selon le nombre de nœuds et la valeur du TTC.

A titre de comparaison, le protocole proposé par Schulz et al. dans [SCH 04], qui possède des propriétés similaires au nôtre dans le contexte de l'intergiciel MPI, induit un surcoût du même ordre de grandeur. Pour une application de traitement de matrice type SPMD sur 16 nœuds, le surcoût global reste inférieur à 50% ; dans des conditions de taille de point de reprise et de valeur du TTC similaire, notre protocole induit un surcoût de 9% à 45,4%. On notera que cette

comparaison n'a de sens qu'en termes d'*ordre de grandeur*, puisque les contextes de développement et d'expérimentation sont différents.

Chapitre 5

Formalisation et preuves : cohérence promise

Nous proposons dans ce chapitre de formaliser les concepts introduits par le protocole développé pour ASP. Ce travail a été réalisé en collaboration avec Ludovic Henrio de l'équipe OASIS [CAR 04a, CAR 06d]. Cette formalisation a pour but premier de prouver la correction du protocole proposé ; nous montrons que les états globaux formés, bien que non cohérents, sont toujours recouvrables.

Le second objectif de cette modélisation est d'identifier précisément les hypothèses requises par notre approche : nous allons montrer que la capacité de recouvrement depuis un état global incohérent n'est pas dépendante des particularités du modèle ASP. De manière plus générale, le formalisme proposé ici peut s'appliquer dans d'autres systèmes, toujours dans le cadre de protocole de tolérance aux pannes par recouvrement arrière depuis un état global non cohérent.

Après avoir introduit le formalisme élémentaire nécessaire, nous généralisons le concept de promesse de requête introduit dans le chapitre 4 en promesse d'évènement. Nous supposons dans un premier temps qu'une promesse d'évènement est capable de synchroniser de manière globale l'exécution, sans tenir compte de la possibilité de créer en pratique un tel outil. Sur la base des promesses d'évènement, nous présentons la *cohérence promise*, ou $\mathcal{P}$-cohérence. La $\mathcal{P}$-cohérence est un outil qui va permettre de prouver la recouvrabilité d'états globaux non cohérents contenant des promesses d'évènement ; à travers la définition d'une réduction d'un état $\mathcal{P}$-cohérent nous prouvons que la réexécution depuis un état $\mathcal{P}$-cohérent est toujours possible, ne peut pas bloquer et dirige toujours l'exécution vers un état global cohérent de l'exécution de référence. Cette réduction repose sur la définition théorique des promesses d'évènement, c'est-à-dire capables de synchronisation globale sur l'exécution.

L'objectif est ensuite de montrer que cette réduction d'un état $\mathcal{P}$-cohérent peut être respectée en pratique en ne contrôlant l'exécution que de manière locale sur chaque activité, et en particulier sans supposer que les promesses d'évènement synchronisent l'exécution de manière globale. Pour cela, nous définissons la réduction *locale*, c'est-à-dire une réduction qui ne tient compte que de l'état local d'une activité et qui ne suppose pas de synchronisation particulière entre les ac-

tivités ; on ne voit plus l'exécution comme une suite de réduction de l'état global mais comme des suites de réduction d'états locaux en parallèle.

Nous montrons ensuite que cette réduction est réaliste, c'est-à-dire que le contrôle sur l'exécution d'une activité nécessaire pour respecter cette réduction est réalisable dans la plupart des systèmes.

Enfin, nous identifions les trois hypothèses qui doivent être respectées par un état global $\mathcal{P}$-cohérent et par le système considéré pour que la réduction locale soit équivalente à la réduction globale, et donc que, dans le cadre d'un contrôle uniquement local de l'exécution, cet état global soit recouvrable.

Les preuves des propriétés et des théorèmes présentés dans cette section sont disponibles dans l'annexe A.

5.1 Formalisme élémentaire

Cette section présente d'abord formellement la notion d'équivalence d'exécution, puis revient sur le concept de causalité potentielle introduit dans la section 2.7. Nous considérons par la suite que toute variable libre est universellement quantifiée à gauche de l'équation. Par exemple, $f(x) \Rightarrow g(y)$ est équivalent à $\forall x, y,\ f(x) \Rightarrow g(y)$.

5.1.1 Exécution distribuée et équivalence

Une exécution donnée $S \xrightarrow{e^1 \dots e^n} S'$ peut être caractérisée par un ensemble partiellement ordonné $(E, \prec^{hb})$. L'ordre $e^1 \prec^{hb} \dots \prec^{hb} e^n$ des évènements réduits lors de l'exécution est une des différentes extensions linéaires possibles de $(E, \prec^{hb})$. Il existe donc plusieurs exécutions caractérisées par le même ensemble ; on parle d'exécutions équivalentes. En effet, considérons deux évènements qui s'exécutent sur deux activités indépendantes. Si au cours de deux exécutions consécutives, l'ordre de leur exécution est inversé, le résultat de l'exécution ne s'en trouvera pas modifié. Les deux exécutions sont différentes mais équivalentes. Nous formalisons ici cette notion en définissant une relation d'équivalence entre toutes les exécutions possibles à partir d'un état initial donné S^0 :

Définition 5.1.1 (Exécutions équivalentes) *Deux exécutions* $S \xrightarrow{e^1} S_1^1 \dots S_1^{n-1} \xrightarrow{e^n} S_1^n$ *et* $S \xrightarrow{e^{\sigma(1)}} S_2^1 \dots S_2^{n-1} \xrightarrow{e^{\sigma(n)}} S_2^n$ *sont dites* équivalentes *si, pour tout* i*, deux états globaux* S_1^i *et* S_2^i *obtenus après l'exécution du même ensemble d'évènements* $\{e^1, \dots, e^i\}$ *dans un ordre* différent *sont équivalents :*

$$\forall i \leq n, \left(S \xrightarrow{e^1 \dots e^i} S_1^i \wedge S \xrightarrow{e^{\sigma(1)} \dots e^{\sigma(i)}} S_2^i \wedge (\forall j \leq i \Rightarrow \sigma(j) \leq i) \right) \Rightarrow S_1^i \equiv S_2^i$$

où σ *est une permutation.*

La notion d'équivalence d'exécution repose sur l'équivalence d'état, notée $\equiv$. Cette équivalence peut être la stricte égalité des états résultants, mais cette égalité est dans la pratique difficile à vérifier. Il peut être par exemple plus simple, mais cohérent avec la notion d'équivalence, de considérer que deux états finaux de

deux exécutions d'une même application sont équivalents si les résultats fournis sont égaux.

La notion d'équivalence entre exécutions permet d'identifier les évènements minimaux e dans un état donné S dans une exécution $(E, \prec^{hb})$, c'est-à-dire $\forall e' \in E, \nexists e' \prec^{hb} e$. Nous notons $\setminus$ l'opérateur de soustraction d'ensembles tel que $E \setminus E' = \{e | e \in E \wedge e \notin E'\}$. Dans l'état global S, l'ensemble des évènements qui peuvent être exécutés en conservant l'équivalence de l'exécution sont les évènements minimaux :

Propriété 5.1.1 (Évènement minimal) *Soit $\mathcal{C}$ une coupe cohérente d'une exécution $(E, \prec^{hb})$ et $S_{\mathcal{C}}$ l'état caractérisé par la coupe $\mathcal{C}$.*

$$S \xrightarrow{*} S_{\mathcal{C}} \xrightarrow{e} S' \xrightarrow{*} S_1 \textit{ caractérisée par } (E, \prec^{hb}) \Rightarrow e \textit{ est minimal pour } \prec \textit{ dans } E \setminus \mathcal{C}$$

5.1.2 Définition d'une causalité potentielle

La relation de Lamport est une relation de causalité potentielle définie de manière unique et applicable à toutes les exécutions réparties. Déterminer une relation plus précise dépend de l'application considérée et de son environnement d'exécution. Nous définissons dans un premier temps les conditions qu'un ordre partiel entre évènements doit remplir pour définir une relation de causalité potentielle correcte, puis nous expliquons comment définir cette relation pour un système donné.

Définition 5.1.2 (Causalité potentielle) *Un ordre partiel $\prec$ définit une relation de causalité potentielle si il est suffisant pour caractériser les exécutions équivalentes : deux exécutions dans lesquelles on permute seulement des évènements qui ne sont pas causalement liés doivent être équivalentes. Donc, si $\prec$ est une relation de causalité potentielle, alors*

$$\forall S, S_1, S_2, \sigma, \begin{cases} S \xrightarrow{e^1 \ldots e^n} S_1 \wedge S \xrightarrow{e^{\sigma(1)} \ldots e^{\sigma(n)}} S_2 \\ \sigma \textit{ une permutation préservant } \prec \end{cases} \Rightarrow S_1 \equiv S_2$$

L'ensemble des exécutions équivalentes selon une relation de causalité potentielle $\prec$ est donc défini par $(E, \prec)$: toutes les extensions linéaires de $(E, \prec)$ sont des exécutions équivalentes.

Pour déterminer une relation de causalité potentielle correcte, Tarafdar et al. proposent dans [TAR 98b] de déterminer dynamiquement les relations de causalité qui lient les évènements locaux durant l'exécution. Cette solution met donc en jeu un mécanisme spécifique qui est capable d'identifier et de restituer les relations de causalité entre évènements. C'est une solution proche d'un environnement de *debugging* : toutes ou parties des actions effectuées par une activité doivent être contrôlées et journalisées. Elle nécessite donc une modification de l'environnement d'exécution, et peut engendrer un surcoût prohibitif à l'exécution.

Enokido et al. proposent dans [ENO 98] une solution pour déterminer *statiquement* une relation de causalité potentielle correcte pour un système à objets communiquant. Ils utilisent la notion de *compatibilité* entre opérations sur un objet pour déterminer un ordre partiel sur la réception des messages. Cette approche a déjà été abordée dans [LEO 94].

Nous proposons de généraliser cette notion de compatibilité au niveau de l'évènement, de manière à définir statiquement une causalité potentielle entre tous les évènements valables pour toute exécution possible du système. La sémantique du système considéré nous permet de définir des règles de compatibilité entre les classes d'évènements. Par exemple, la connaissance de la sémantique de ASP nous a permis de définir dans la section 3.1.5 une relation de causalité potentielle vraie pour toute exécution ASP, en fonction des compatibilités entre les différents types d'évènement.

Donc, si la sémantique du système autorise l'échange de l'exécution de deux évènements, ces évènements sont *compatibles* :

Définition 5.1.3 (Évènements compatibles) *Deux évènements e et e' sont compatibles, noté $e \bowtie e'$, s'ils peuvent être exécutés dans n'importe quel ordre, et que l'effet de leur exécution est indépendant de cet ordre :*

$$e \bowtie e' \Leftrightarrow \begin{cases} \left(\exists S, S_1, S_2,\ S \xrightarrow{ee'} S_1 \wedge S \xrightarrow{e'e} S_2\right) \ \wedge \\ \left(\forall S', S_1, S_2,\ S' \xrightarrow{ee'} S_1 \wedge S' \xrightarrow{e'e} S_2 \Rightarrow S_1 \equiv S_2\right) \end{cases}$$

Les relations de compatibilité entre les évènements nous permettent de définir statiquement pour un système donné une relation de causalité potentielle correcte :

Définition 5.1.4 (Compatibilité et causalité potentielle) *L'ordre $\prec$ est un ordre de causalité potentielle déduit de la relation de compatibilité $\bowtie$:*

$$e \prec e' \textit{ ssi } \left(S \xrightarrow{*} S^1 \xrightarrow{e} S^2 \xrightarrow{*} S^3 \xrightarrow{e'} S^n \wedge \neg(e \bowtie e')\right) \vee \left(e \prec e'' \wedge e'' \prec e'\right)$$

Dans le cas d'une exécution caractérisée par la relation de Lamport $(E, \prec^{hb})$, un ordre de causalité potentielle correcte $\prec$ peut être définit par toutes les permutations sur $\prec^{hb}$ autorisées par la relation de compatibilité, avec la clôture transitive :

$$e \prec e' \textit{ ssi } \left(e \prec^{hb} e' \wedge \neg(e \bowtie e')\right) \vee \left(\exists e'', e \prec e'' \wedge e'' \prec e'\right)$$

Ces deux définitions sont équivalentes si l'on considère que pour tout système, l'envoi d'un message n'est pas compatible avec sa réception. Cette hypothèse est réaliste puisque rendre compatible l'envoi et la réception d'un message revient à autoriser à recevoir un message avant de l'avoir envoyé.

5.1.3 Déterminisme

Considérer un ordre de causalité potentielle et une relation de compatibilité entre les évènements nous permet de définir un prédicat qui identifie les évènements déterministes. Un évènement est déterministe si, une fois que son exécution est possible, il est nécessairement exécuté et a le même effet dans toutes les exécutions possibles.

Définition 5.1.5 (Évènement déterministe) *Soit Det un prédicat tel que pour tous les états globaux S d'une exécution $(E, \prec)$, tels que $S_0 \xrightarrow{e_1..e_n} S$, pour toute réduction $\xrightarrow{e}$ depuis l'état S, si $Det(e)$ est vrai, alors e est exécuté nécessairement après S dans une des exécutions représentées par $(E, \prec)$.*

$$\forall S', S \xrightarrow{e} S' \Rightarrow \neg Det(e) \vee (e \in E \backslash \{e_1..e_n\} \wedge e \textit{ est minimal pour } \prec \textit{ dans } E \backslash \{e_1..e_n\})$$

De façon équivalente,

$$Det(e) \Rightarrow \forall S, e', S', S'' \left(S \xrightarrow{e} S' \wedge S \xrightarrow{e'} S'' \Rightarrow \exists S_1, S_2, S' \xrightarrow{e'} S_1 \wedge S'' \xrightarrow{e} S_2 \wedge S_1 \equiv S_2 \right)$$

Cette formalisation du déterminisme est particulièrement adaptée à la causalité potentielle. En effet, un évènement est déterministe si il est *compatible avec tous les évènements qui pourraient être exécutés à sa place*. Plus formellement, nous donnons donc la propriété suivante :

Propriété 5.1.2 (Déterminisme et compatibilité)

$$S \xrightarrow{e} S' \wedge Det(e) \Rightarrow \forall e' (S \xrightarrow{e'} S'' \Rightarrow e \bowtie e')$$

Notons qu'il est toujours correct de considérer un évènement déterministe comme un évènement non déterministe, puisque un évènement déterministe est un évènement non déterministe peut toujours remplir les conditions définies par le prédicat Det.

5.2 $\mathcal{P}$-cohérence

Nous présentons dans cette section la $\mathcal{P}$-cohérence, une condition de recouvrabilité sur un état global non cohérent, qui repose sur l'utilisation de promesses d'évènement. Nous définissons donc d'abord la promesse d'évènement, ainsi que sa sémantique.

5.2.1 Promesse d'événement

Une promesse d'événement, notée $\mathcal{P}(e)$, est un substitut pour un évènement non déterministe e dans un état S qui permet de contrôler une exécution distribuée. Une promesse d'évènement contient suffisamment d'informations pour rendre un évènement non déterministe d'une exécution de référence déterministe pendant la réexécution.

5.2.1.1 Sémantique

Une promesse d'évènement n'existe que par rapport à une exécution de référence ; les informations nécessaires à la création de la promesse sont collectées durant cette exécution. Durant une réexécution depuis S, la présence de la promesse $\mathcal{P}(e)$ dans S assure que l'évènement e aura lieu, et que :

1. le contenu applicatif de e dans la réexécution soit identique à celui de e dans l'exécution de référence,
2. la position de e dans la réexécution soit la même que celle de e dans l'exécution de référence.

De manière générale, une promesse d'évènement $\mathcal{P}(e_1)$ assure que l'exécution $S \xrightarrow{\mathcal{P}(e_1)e_2...e_ne_1} S'$ est *équivalente* à $S \xrightarrow{e_1e_2...e_n} S''$. La promesse d'évènement $\mathcal{P}(e_1)$ assure que l'évènement e_1 aura lieu au cours de toutes les exécutions possibles. Lorsque l'évènement e_1 a lieu, il prend la place de la promesse $\mathcal{P}(e_1)$.

Une promesse d'évènement est d'abord un élément de **journalisation** : une promesse doit assurer que, durant une réexécution, le contenu applicatif de l'évènement promis soit le même que durant l'exécution de référence.

Les informations nécessaires pour assurer la propriété de journalisation sont bien sûr dépendantes du contexte. Si l'évènement promis est forcément régénéré pendant la réexécution, la promesse n'a pas besoin de contenir le contenu applicatif de l'évènement promis. Ce contenu est assuré par le contexte d'être le même. Dans le cas contraire, la promesse doit contenir les informations suffisantes pour recréer l'évènement ; c'est le cas de la journalisation d'un message *avec son contenu* dans un protocole de type journalisation pessimiste.

Une promesse d'évènement est aussi un élément de **synchronisation** : elle positionne dans l'exécution l'évènement promis, relativement à l'exécution des autres évènements causalement liés. Le positionnement d'un évènement e dans une exécution correspond à deux conditions :

- les évènements qui sont une conséquence causale de e ne doivent pas être exécutés avant e,
- et les évènements qui font partie du passé causal de e doivent être exécutés avant e.

5.2.1.2 Correspondance entre promesses et évènements

La création et la manipulation de promesses d'évènement repose d'abord sur la capacité à identifier un évènement particulier et à observer son exécution, de manière à pouvoir créer une correspondance *unique* entre une promesse et l'évènement promis. Par exemple, les protocoles de journalisation causale utilisent le déterminant (section 2.5.3) des messages pour identifier de manière unique la réception d'un message.

Il faut donc que l'exécution des évènements non déterministes soit *observable*, c'est-à-dire que l'on puisse identifier ces évènements et le moment où ils sont exécutés. La capacité d'observer les évènements est à la base du concept de déterminisme par morceaux. En effet, le déterminisme par morceaux modélise chaque exécution locale comme une suite d'exécutions déterministes, séparées par des évènements non déterministes. Par conséquent, tout événement non déterministe qui a lieu durant l'exécution doit être observable. Nous nous plaçons donc dans ce contexte ; les exécutions considérées doivent être déterministes par morceaux.

Cette hypothèse de déterminisme par morceaux est souvent étendue à l'hypothèse que les seuls évènements non déterministes sont les réceptions de message. Cette approximation vient du fait que les réceptions de messages sont des évènements non déterministes qui sont facilement observables ; la réception d'un message déclenche le plus souvent sur l'activité réceptrice l'exécution d'un code qui peut être observé.

De manière plus générale, la notion d'évènement observable est fortement liée à l'implémentation du système considéré, ainsi qu'à la granularité de l'évènement, c'est-à-dire ce qui est considéré comme un évènement élémentaire. Par exemple, considérons un environnement de *debugging* pour le langage Java. Une granularité possible dans ce cas est l'instruction élémentaire (*bytecode*) ; l'exécution d'une instruction par la machine virtuelle est considérée comme un évènement élémentaire. L'objectif d'un tel environnement d'exécution est de rendre *tout* évènement observable, de manière à pouvoir contrôler l'exécution le plus précisément possible, par exemple stopper l'exécution avant ou après un évènement donné.

Pour tout évènement non déterministe e d'une exécution de référence, il existe une relation unique entre e et son évènement correspondant e' dans la réexécution. Nous définissons l'opérateur de correspondance $\lhd$:

Définition 5.2.1 (Opérateur de correspondance) $\mathcal{P}(e) \lhd e'$ *si et seulement si* e' *est l'évènement* correspondant *à* e.

On peut donc créer une promesse d'évènement pour tous les évènements non déterministes de l'exécution. Ce lien entre les promesses d'évènement et le déterminisme par morceaux nous permet de définir le déterminisme par morceaux de la façon suivante :

Propriété 5.2.1 (Déterminisme par morceaux et promesses) *Dans une exécution déterministe par morceaux, tous les évènements non déterministes sont observables :*

$$S \xrightarrow{e} S' \wedge \neg Det(e) \wedge S \xrightarrow{e'} S'' \Rightarrow S'' \xrightarrow{e} S''' \vee \mathcal{P}(e) \lhd e'$$

Par la suite, e représente un évènement non promis, $\mathcal{P}(e)$ une promesse d'évènement et $e_{\mathcal{P}}$ un évènement ou une promesse d'évènement. De plus, notons que la compatibilité peut être étendue de façon triviale pour les promesses d'évènement :

$$e \bowtie e' \Rightarrow \mathcal{P}(e) \bowtie e' \ \wedge \ e \bowtie \mathcal{P}(e') \ \wedge \ \mathcal{P}(e) \bowtie \mathcal{P}(e')$$

5.2.2 Définition de la $\mathcal{P}$-cohérence

Nous définissons dans cette section la $\mathcal{P}$-cohérence. Nous montrons la recouvrabilité d'une coupe $\mathcal{P}$-cohérente dans le cadre de l'utilisation de promesses d'évènement et d'une synchronisation globale, c'est-à-dire si la sémantique des promesses définie dans la section précédente est assurée. En particulier, nous supposons dans cette première partie que les promesses d'évènement sont capables de régénérer un évènement et de le positionner dans l'exécution sans contrainte. De

plus, nous supposons qu'il existe une synchronisation globale qui assure la cohérence de l'exécution, c'est à dire qu'une synchronisation globale empêche l'exécution sur toutes les activités des évènements déterministes qui sont la conséquence d'évènements qui n'ont pas encore eu lieu. A partir de ces deux hypothèses, nous définissons les règles de réduction d'un état global $\mathcal{P}$-cohérent.

La $\mathcal{P}$-cohérence est un outil de preuve ; cette réduction globale *théorique* va nous permettre de prouver que toutes les exécutions possibles depuis un état global $\mathcal{P}$-cohérent sont équivalentes à une exécution qui amène à un état global cohérent de l'exécution de référence. Nous montrons aussi qu'il existe toujours une réduction possible depuis un état global $\mathcal{P}$-cohérent, et donc que l'exécution ne peut pas bloquer.

La $\mathcal{P}$-cohérence est basée sur l'utilisation de promesses d'évènement. Nous définissons donc d'abord la notion de $\mathcal{P}$-coupe, une coupe qui peut contenir des promesses d'évènement. Une $\mathcal{P}$-coupe assure aussi une cohérence locale :

- tous les évènements qui précédent causalement un évènement déterministe e de la $\mathcal{P}$-coupe appartiennent à la $\mathcal{P}$-coupe, mais peuvent être promis,
- tous les évènements non déterministes qui précédent causalement un évènement non déterministe (promis ou non) de la $\mathcal{P}$-coupe appartiennent à la $\mathcal{P}$-coupe, mais peuvent être promis.

La notion de cohérence locale assurée par la définition d'une $\mathcal{P}$-coupe est le pendant de la cohérence locale assurée par la définition d'une coupe par Mattern : une coupe doit contenir le passé local de tous les évènements qu'elle contient.

Pour définir la $\mathcal{P}$-coupe, nous étendons la notion d'appartenance d'un évènement à un état dans le cadre des promesses d'évènement. Soit $\mathcal{C}_p$ un ensemble d'évènements, on note $e \in_{\mathcal{P}} \mathcal{C}_p$ si et seulement si e *ou* une promesse $\mathcal{P}(e)$ pour l'évènement appartient à $\mathcal{C}_p$:

$$e \in_{\mathcal{P}} \mathcal{C}_p \Leftrightarrow e \in \mathcal{C}_p \vee \mathcal{P}(e) \in \mathcal{C}_p$$

Définition 5.2.2 ($\mathcal{P}$-coupe) *La coupe $\mathcal{C}_p$ est une $\mathcal{P}$-coupe d'une exécution $(E, \prec)$ si et seulement si $e \in_{\mathcal{P}} \mathcal{C}_p \Rightarrow e \in E$ et :*

$$\forall e, e',\ (e \in_{\mathcal{P}} \mathcal{C}_p), \begin{cases} e' \prec_i e \wedge Det(e) \Rightarrow e' \in_{\mathcal{P}} \mathcal{C}_p \\ e' \prec_i e \wedge \neg Det(e) \Rightarrow e' \in_{\mathcal{P}} \mathcal{C}_p \vee Det(e') \end{cases}$$

Une $\mathcal{P}$-coupe $\mathcal{C}_p$ d'une exécution $(E, \prec)$ est donc un ensemble partiellement ordonné d'évènements $(\mathcal{C}_p, \prec')$; l'ordre $\prec'$ est appelé l'ordre induit par l'exécution sur la $\mathcal{P}$-coupe $\mathcal{C}_p$:

$$(e, e' \in_{\mathcal{P}} \mathcal{C}_p \wedge e \prec e') \Leftrightarrow \big(e \prec' e' \vee \mathcal{P}(e) \prec' e' \vee e \prec' \mathcal{P}(e') \vee \mathcal{P}(e) \prec' \mathcal{P}(e')\big)$$

Nous définissons une coupe[1] $\mathcal{P}$-cohérente comme une $\mathcal{P}$-coupe particulière qui assure une cohérence minimale entre les évènements s'exécutant sur des activités

[1]Une coupe $\mathcal{P}$-cohérente est forcément une $\mathcal{P}$-coupe ; par la suite, nous utiliserons indifféremment coupe ou $\mathcal{P}$-coupe pour désigner une coupe $\mathcal{P}$-cohérente

différentes : tous les évènements non déterministes qui précédent causalement un évènement appartenant à une coupe $\mathcal{P}$-cohérent appartiennent à cette coupe, mais peuvent être promis.

Définition 5.2.3 ($\mathcal{P}$-cohérence) *Une $\mathcal{P}$-coupe $\mathcal{C}_p$ est une coupe $\mathcal{P}$-cohérente d'une exécution $(E, \prec)$ si et seulement si*

$$\forall e, e',\ (e \in_{\mathcal{P}} \mathcal{C}_p), e' \prec e \Rightarrow \left(e' \in_{\mathcal{P}} \mathcal{C}_p \vee Det(e')\right)$$

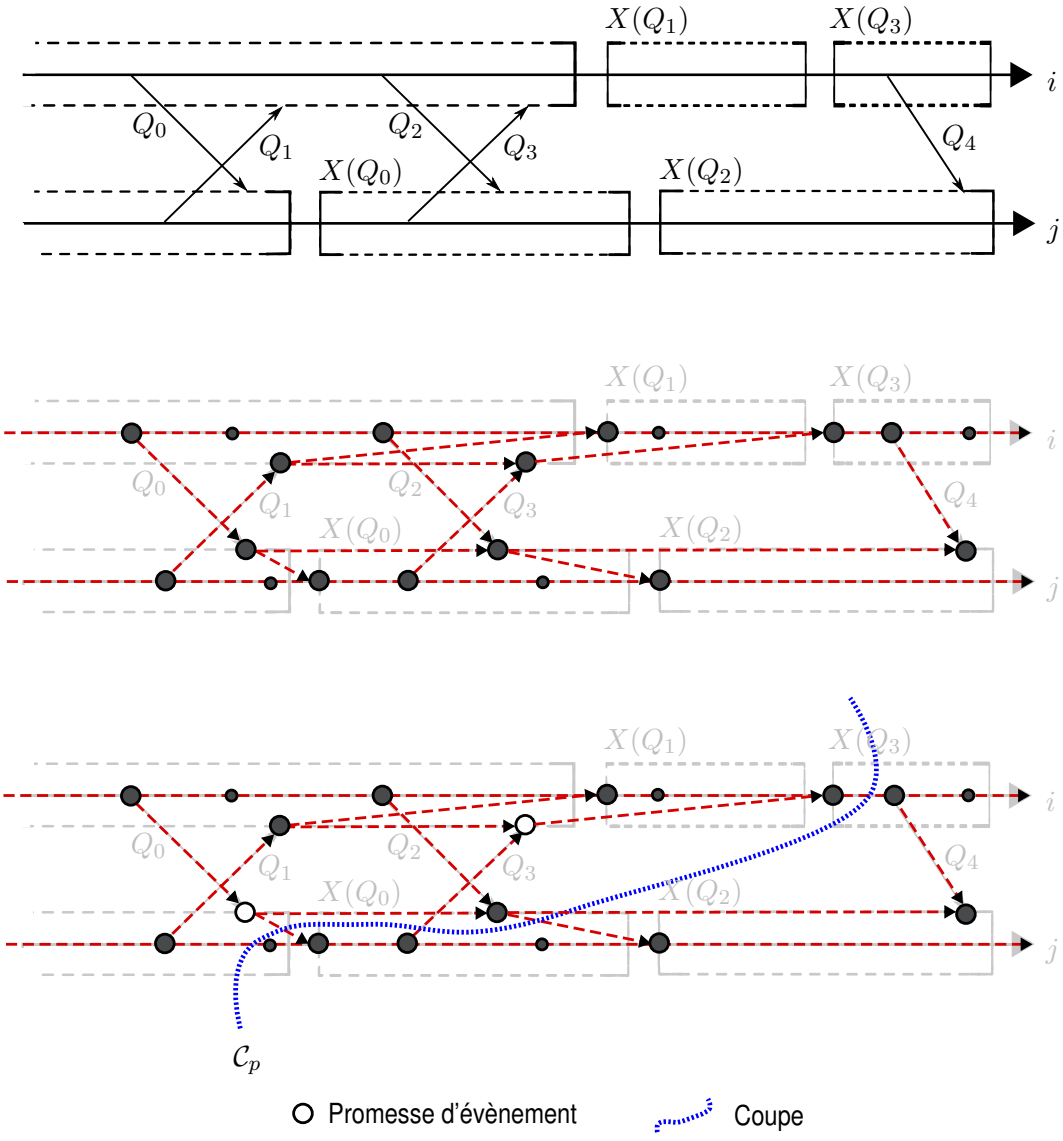

FIG. 5.1 – Coupe $\mathcal{P}$-cohérente

La figure 5.1 présente une coupe $\mathcal{P}$-cohérente d'une exécution ASP. Par souci de clarté, on considère dans cet exemple une exécution sans envoi de réponse ; le même raisonnement est bien sûr applicable dans une exécution avec des réponses. La coupe $\mathcal{C}_p$ est $\mathcal{P}$-cohérente. En particulier, le fait que $\mathcal{C}_p$ soit $\mathcal{P}$-cohérente

implique que si le service de Q_3 est dans $\mathcal{C}_p$, alors la réception de Q_3 *ou une promesse* de la réception de Q_3 est dans $\mathcal{C}_p$; ici on a une promesse de cet évènement.

De plus, si la réception de Q_2 est dans $\mathcal{C}_p$, alors toutes les réceptions précédentes doivent être dans $\mathcal{C}_p$, promises ou non. Ici, on a une promesse sur la réception de Q_0.

La réception de Q_3 étant promise dans $\mathcal{C}_p$, $\mathcal{C}_p$ reste $\mathcal{P}$-cohérente même sans contenir l'envoi de Q_3 car cet envoi est un évènement déterministe. Cependant, tous les évènements non déterministes précédant causalement la promesse de réception de Q_3 doivent appartenir à ou être promis dans $\mathcal{C}_p$. Ici, la réception de Q_1 et une promesse de la réception de Q_0 sont dans $\mathcal{C}_p$.

5.2.3 Réduction d'une coupe $\mathcal{P}$-cohérente

La notion de réduction d'évènement étant définie dans la section 2.3.1 sur une coupe *cohérente* d'une exécution, nous identifions deux coupes cohérentes qui entourent une coupe $\mathcal{P}$-cohérente $\mathcal{C}_p$: la plus petite coupe cohérente qui contient tous les évènements (promis ou non) contenus dans $\mathcal{C}_p$, notée $\mathcal{C}_p^{\sqcup}$, et la plus grande coupe cohérente contenue dans $\mathcal{C}_p$, notée $\mathcal{C}_p^{\sqcap}$.

Définition 5.2.4 ($\mathcal{C}_p^{\sqcup}$ et $\mathcal{C}_p^{\sqcap}$) *Soit $\mathcal{C}_p$ une coupe $\mathcal{P}$-cohérente. Nous définissons $\mathcal{C}_p^{\sqcup}$ comme la plus petite coupe cohérente de l'exécution $(E, \prec)$ qui contient tous les évènements e tels que $e \in_{\mathcal{P}} \mathcal{C}_p$:*

$$\mathcal{C}_p^{\sqcup} = \{e | \exists e' \in_{\mathcal{P}} \mathcal{C}_p,\ e \preceq e'\}$$

Et nous définissons $\mathcal{C}_p^{\sqcap}$ comme la plus grande coupe cohérente de l'exécution $(E, \prec)$ contenue dans $\mathcal{C}_p$:

$$\mathcal{C}_p^{\sqcap} = \{e | \forall e', e' \preceq e \Rightarrow e' \in \mathcal{C}_p\}$$

Nous prouvons en annexe que l'ensemble d'évènements $\{e | \forall e', e' \preceq e \Rightarrow e' \in \mathcal{C}_p\}$ est bien la plus petite coupe cohérente contenant $\mathcal{C}_p$, et que $\{e | \forall e', e' \preceq e \Rightarrow e' \in \mathcal{C}_p\}$ est bien la plus grande coupe cohérente de l'exécution $(E, \prec)$ contenue dans $\mathcal{C}_p$.

Notons que $\mathcal{C}_p^{\sqcap}$ et $\mathcal{C}_p^{\sqcup}$ sont des coupes de l'exécution de référence. De plus, $\mathcal{C}_p^{\sqcap}$ est incluse dans $\mathcal{C}_p$, donc que $\mathcal{C}_p^{\sqcap}$ est forcément une coupe de la réexécution. Nous allons montrer que $\mathcal{C}_p^{\sqcup}$ est une coupe de la réexécution une fois que toutes les promesses sont remplacées par les évènements promis.

Nous prouvons grâce à la définition de la $\mathcal{P}$-cohérence les propriétés suivantes sur les coupes $\mathcal{C}_p^{\sqcup}$ et $\mathcal{C}_p^{\sqcap}$: tous les évènements non déterministes dans $\mathcal{C}_p^{\sqcup}$ doivent appartenir ou être promis dans $\mathcal{C}_p$. De plus, trivialement, il existe forcément une exécution depuis $\mathcal{C}_p^{\sqcap}$ vers $\mathcal{C}_p^{\sqcup}$, car ce sont deux coupes cohérentes de la même exécution.

Propriété 5.2.2 ($\mathcal{C}_p^{\sqcup}$ et $\mathcal{C}_p^{\sqcap}$) *Pour toute coupe $\mathcal{P}$-cohérente $\mathcal{C}_p$, on a :*

- $e \in \mathcal{C}_p^{\sqcup} \wedge \neg Det(e) \Rightarrow e \in_{\mathcal{P}} \mathcal{C}_p$
- $\mathcal{C}_p^{\sqcap} \xrightarrow{*} \mathcal{C}_p^{\sqcup}$

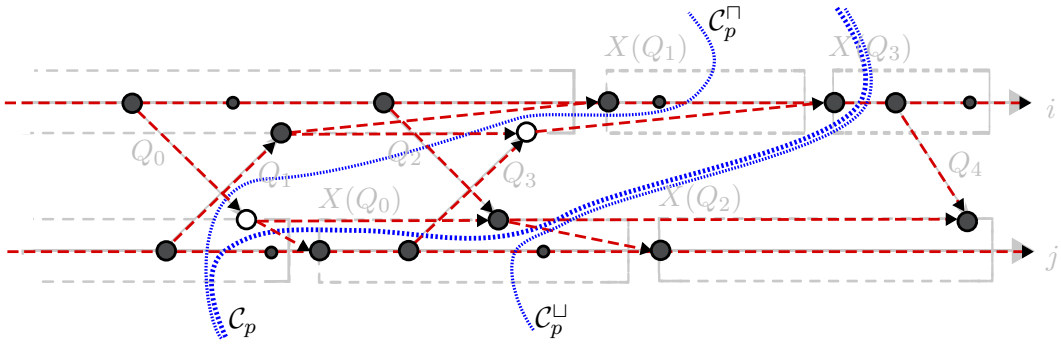

FIG. 5.2 – Coupes $\mathcal{C}_p^{\sqcup}$ et $\mathcal{C}_p^{\sqcap}$

La figure 5.2 présente les coupes $\mathcal{C}_p^{\sqcap}$ et $\mathcal{C}_p^{\sqcup}$ correspondant à la coupe $\mathcal{P}$-cohérente présentée dans la figure 5.1. Par exemple, $\mathcal{C}_p^{\sqcap}$ ne peut pas contenir le service de la requête Q_3 car la réception de Q_3 est une promesse dans $\mathcal{C}_p$. $\mathcal{C}_p^{\sqcup}$ doit contenir l'envoi de Q_3 pour rester cohérente, bien que cet envoi ne soit pas contenu dans la coupe $\mathcal{C}_p$.

A partir de ces deux coupes, nous pouvons définir les règles de réduction d'une coupe $\mathcal{P}$-cohérente, puis prouver que toutes les réductions possibles à partir de ces règles représentent des exécutions équivalentes à l'exécution qui mènent à la coupe cohérente $\mathcal{C}_p^{\sqcup}$. Nous prouvons donc la recouvrabilité d'une coupe $\mathcal{P}$-cohérente. Une fois encore, on voit que ces règles de réduction supposent une vue globale du système ; on suppose dans cette section qu'il est toujours possible de connaître la coupe cohérente $\mathcal{C}_p^{\sqcap}$, ainsi que toutes les exécutions possibles depuis cette coupe cohérente.

Définition 5.2.5 (Réduction sur une coupe $\mathcal{P}$-cohérente) *Soit $(\mathcal{C}_p, \prec)$ une coupe $\mathcal{P}$-cohérente d'une exécution $(E, \prec^0)$. Nous définissons $\mathcal{C}_p \xrightarrow{e} \mathcal{C}'_p$ la réduction d'un évènement $e \notin \mathcal{C}_p$ sur une coupe $\mathcal{P}$-cohérent $\mathcal{C}_p$. $(\mathcal{C}'_p, \prec')$ est l'ensemble partiellement ordonné d'évènements résultant de cette réduction tel que :*

- ***Évènement déterministe :*** *Si $\mathcal{C}_p^{\sqcap} \xrightarrow{e} \mathcal{C}' \wedge Det(e)$ alors e est ajouté dans $\mathcal{C}_p$ après les évènements de $\mathcal{C}_p^{\sqcap}$.*

$$\mathcal{C}'_p = \mathcal{C}_p \oplus \{e\}$$

- ***Évènement correspondant à une promesse :*** *Si $\neg Det(e) \wedge \exists \mathcal{P}(e') \in \mathcal{C}_p$ tel que $\mathcal{P}(e') \lhd e$, alors $\mathcal{C}'_p$ est la $\mathcal{P}$-coupe $\mathcal{C}_p$ avec $\mathcal{P}(e')$ remplacé par e'. En particulier, e' prend la position de $\mathcal{P}(e')$.*

$$\mathcal{C}'_p = \mathcal{C}_p\{\mathcal{P}(e') \leftarrow e'\}$$

- ***Évènement non déterministe sans promesse :*** *Si $\mathcal{C}_p^{\sqcap} \xrightarrow{e} \mathcal{C}' \wedge \neg Det(e) \wedge \forall \mathcal{P}(e') \in \mathcal{C}_p, \mathcal{P}(e') \not\lhd e$ alors e est ajouté à la fin de la coupe $\mathcal{C}_p$.*

$$\mathcal{C}'_p = \mathcal{C}_p \oplus \{e\}$$

Avec les notations suivantes :

$$\mathcal{C}_p\{\mathcal{P}(e') \leftarrow e'\} = \mathcal{C}_p \backslash \{\mathcal{P}(e)'\} \cup \{e'\} \textit{ avec pour tout } e_\mathcal{P}, e'_\mathcal{P} \in \mathcal{C}_p\{\mathcal{P}(e') \leftarrow e'\}$$

$$e_\mathcal{P} \prec' e'_\mathcal{P} \textit{ ssi } \begin{cases} e_\mathcal{P} = e' \wedge \mathcal{P}(e') \prec e'_\mathcal{P} \ \vee & e' \textit{ prend la position de } \mathcal{P}(e') \\ e'_\mathcal{P} = e' \wedge e_\mathcal{P} \prec \mathcal{P}(e') \ \vee & e' \textit{ prend la position de } \mathcal{P}(e') \\ e_\mathcal{P} \neq e' \wedge e'_\mathcal{P} \neq e' \wedge e_\mathcal{P} \prec e'_\mathcal{P} & \prec' \textit{ est égal à } \prec \textit{ sauf pour } e' \end{cases}$$

$$\mathcal{C}_p \oplus \{e\} = \mathcal{C}_p \cup \{e\} \textit{ avec pour tout } e_\mathcal{P}, e'_\mathcal{P} \in \mathcal{C}_p \oplus \{e\}$$

$$e_\mathcal{P} \prec' e'_\mathcal{P} \textit{ ssi } \begin{cases} \neg Det(e) \wedge e'_\mathcal{P} = e \wedge \neg e \bowtie e_\mathcal{P} \ \vee & \textit{si } \neg Det(e), e \textit{ à la fin de } \mathcal{C}_p \\ Det(e) \wedge e'_\mathcal{P} = e \wedge e_\mathcal{P} \in \mathcal{C}_p^\sqcap \wedge \neg e \bowtie e_\mathcal{P} \ \vee & \textit{sinon } e \textit{ après } \mathcal{C}_p^\sqcap \\ Det(e) \wedge e_\mathcal{P} = e \wedge e'_\mathcal{P} \notin \mathcal{C}_p^\sqcap \wedge \neg e \bowtie e'_\mathcal{P} \ \vee & \textit{mais avant les autres évènements} \\ e_\mathcal{P}, e'_\mathcal{P} \in \mathcal{C}_p \wedge e_\mathcal{P} \prec e'_\mathcal{P} \ \vee & \prec' \textit{ contient } \prec \\ \exists e''_\mathcal{P} \in \mathcal{C}'_p,\ e_\mathcal{P} \prec' e''_\mathcal{P} \wedge e''_\mathcal{P} \prec' e'_\mathcal{P} & \textit{clôture transitive} \end{cases}$$

La notation $\mathcal{C}_p\{\mathcal{P}(e') \leftarrow e'\}$ définit l'ensemble d'évènements et l'ordre entre ces évènements obtenu lorsqu'un évènement est exécuté et qu'une promesse contenue dans l'état global représenté par $\mathcal{C}_p$ est remplacée par cet évènement. La notation $\mathcal{C}_p \oplus \{e\}$ définit l'ensemble d'évènement et l'ordre entre ces évènements obtenu lorsqu'un évènement est exécuté et est ajouté dans l'état global représenté par la coupe $\mathcal{C}_p$. Assurer qu'une exécution réelle respecte les règles de réductions définies ici implique que les manipulations $\mathcal{C}_p\{\mathcal{P}(e') \leftarrow e'\}$ et $\mathcal{C}_p \oplus \{e\}$ soient possibles durant l'exécution, c'est à dire que l'exécution de certains évènements puisse être d'une certaine manière "déplacée". Nous verrons par la suite que le contrôle sur l'exécution nécessaire pour pouvoir manipuler les évènements de cette manière peut être réalisé dans la plupart des systèmes.

Notons que la réduction d'une coupe $\mathcal{P}$-cohérente est notée $\rightarrow$ de la même manière que la réduction sur une coupe cohérente définie dans la section 2.3.1. Il n'y a pas d'ambiguïté dans cette notation, puisqu'une coupe cohérente peut-être vue comme une coupe $\mathcal{P}$-cohérente particulière : si $\mathcal{C}_p$ est une coupe cohérente dans la définition 5.2.5 , alors la réduction est identique à celle d'un état global cohérent. En particulier, $\mathcal{C}_p \oplus \{e\}$ correspond à l'exécution de l'évènement e sur la coupe $\mathcal{C}_p^\sqcap = \mathcal{C}_p$, et $\mathcal{C}_p\{\mathcal{P}(e') \leftarrow e'\}$ n'est jamais utilisé.

La figure 5.3 présente une suite de réductions possibles à partir de la coupe $\mathcal{P}$-cohérente $\mathcal{C}_p$ présentée dans la figure 5.1. Les évènements réductibles sont surlignés en gris, et les évènements réduits sont entourés en pointillés gras.

Dans cet exemple, le premier évènement réduit est la réception de la requête Q_3. Cette réduction est possible à ce moment puisque la réception est promise ; elle peut par définition avoir lieu n'importe quand. De plus, la synchronisation globale assure que des conséquences de cette réception ne pourront pas avoir lieu sur les autres activités avant son passé. Par exemple, l'envoi par i de la requête Q_4 est bloqué par la synchronisation globale tant que j n'a pas envoyé la requête Q_3.

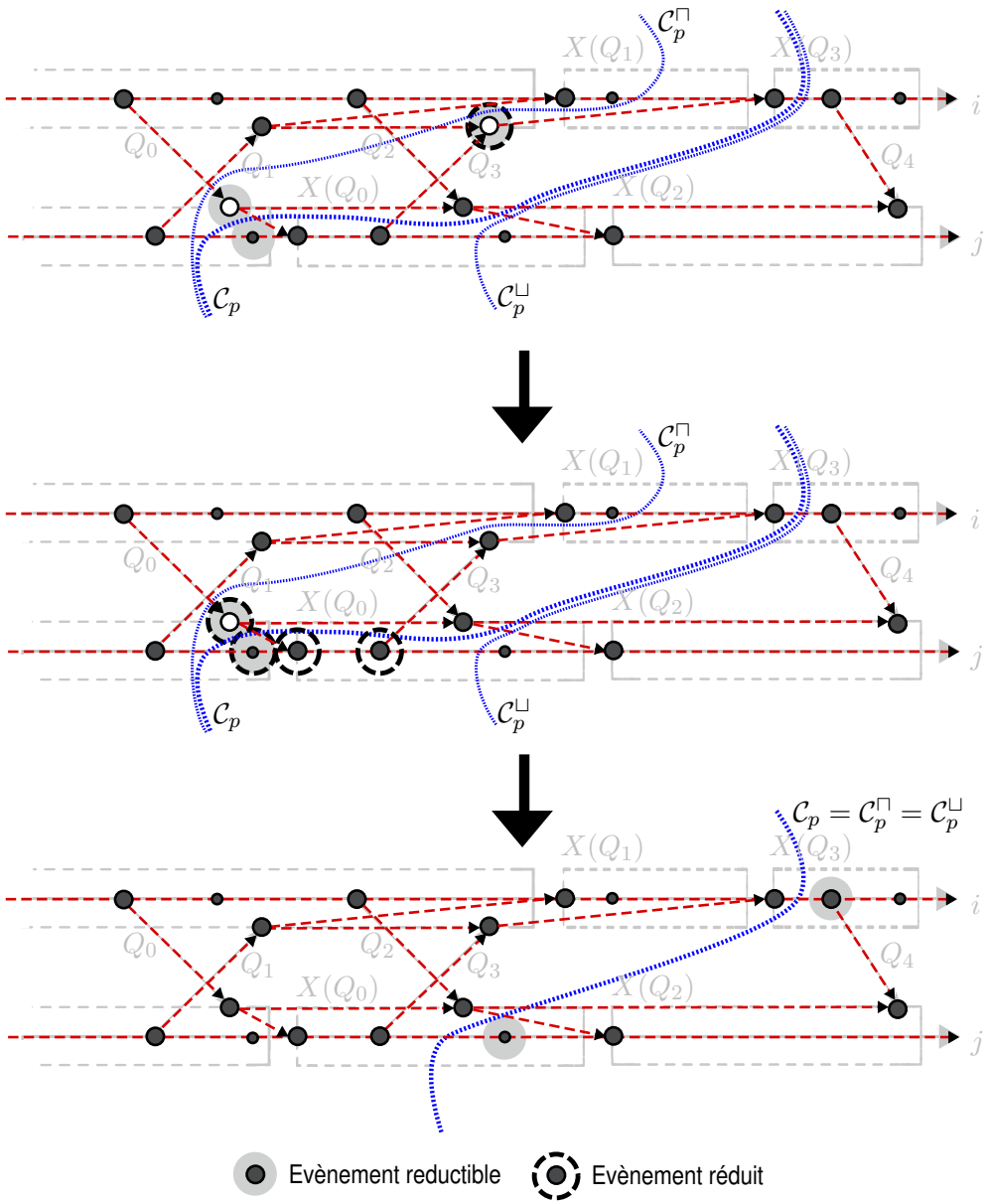

FIG. 5.3 – Réduction d'une coupe $\mathcal{P}$-cohérente

La seconde étape de l'exemple montre les quatre réductions qui doivent avoir lieu pour débloquer la synchronisation globale sur l'envoi de Q_4 : ces quatre évènements sont le passé de l'envoi de Q_4. Une fois que ces quatre évènements ont été exécutés, on peut voir que la coupe $\mathcal{C}_p$ a été réduite à une coupe cohérente de l'exécution de référence. L'exécution normale peut donc continuer.

La cohérence de la réexécution est exprimée dans la définition 5.2.5 par le fait que seules des réductions sur la coupe cohérente $\mathcal{C}_p^{\sqcap}$ peuvent avoir lieu ; l'exécution d'un évènement *avant* son passé causal est donc impossible. Cependant, la

conséquence d'un évènement qui n'a pas encore été exécuté peut déjà appartenir à $\mathcal{C}_p$; cette conséquence à été exécutée pendant l'exécution de référence et est dans la coupe. Mais l'existence de la synchronisation globale assure qu'aucune conséquence suivante ne peut être exécutée ; la cohérence de la réexécution est donc toujours conservée.

5.2.3.1 Preuve de la recouvrabilité d'une coupe $\mathcal{P}$-cohérente

Nous montrons ici la recouvrabilité d'une coupe $\mathcal{P}$-cohérente en prouvant que toutes les exécutions possibles depuis cette coupe passent finalement par un état global cohérent de l'exécution de référence, et ne peuvent jamais bloquer à cause de la synchronisation.

La première propriété montre que la réduction d'une coupe $\mathcal{P}$-cohérente par la réduction 5.2.5 donne toujours une coupe $\mathcal{P}$-cohérente. Les propriétés de la coupe de départ sont donc conservées durant la réexécution.

Propriété 5.2.3 (La réduction conserve la $\mathcal{P}$-cohérence) *La réduction sur une coupe $\mathcal{P}$-cohérente produit une coupe $\mathcal{P}$-cohérente d'une exécution E' (qui peut être différente de E après $\mathcal{C}_p^{\sqcup}$). Si $\mathcal{C}_p$ est une coupe $\mathcal{P}$-cohérente de l'exécution $(E, \prec)$, alors*

$$\mathcal{C}_p \xrightarrow{e} \mathcal{C}'_p \Rightarrow \exists (E', \prec') \text{ telle que } \mathcal{C}'_p \text{ est une coupe } \mathcal{P}\text{-cohérente de } (E', \prec')$$

La théorème suivant assure qu'il existe toujours une suite de réductions possibles depuis une coupe $\mathcal{P}$-cohérente $\mathcal{C}_p$ jusqu'à atteindre la coupe supérieure $\mathcal{C}_p^{\sqcup}$. L'exécution représentée par cette suite de réductions génère donc tous les évènements correspondants à toutes les promesses contenues dans la coupe $\mathcal{C}_p$, et ne peut pas être bloquée à cause de la synchronisation globale.

Théorème 5.2.1 (Exécution possible) *Il existe une exécution depuis la coupe $\mathcal{C}_p$ vers la coupe cohérente $\mathcal{C}_p^{\sqcup}$:*

$$\mathcal{C}_p \xrightarrow{*} \mathcal{C}_p^{\sqcup}$$

Ce théorème signifie qu'il existe une réduction qui imite l'exécution de référence jusqu'à ce que toutes les promesses d'évènement soient remplacées par les évènements promis. Nous montrons maintenant par le théorème 5.2.2 que toutes les exécutions possibles sont équivalentes jusqu'à $\mathcal{C}_p^{\sqcup}$, en particulier équivalentes à l'exécution possible du théorème 5.2.1. Ainsi nous montrons que toutes les exécutions possibles génèrent les évènements promis pour toutes les promesses d'évènement contenues dans une coupe $\mathcal{P}$-cohérente, et ne peuvent pas être bloquée par la synchronisation globale.

Le théorème 5.2.2 affirme que $\mathcal{C}_p^{\sqcup}$ est aussi une coupe cohérente de la réexécution depuis la coupe $\mathcal{P}$-cohérente $\mathcal{C}_p$, donc que la réexécution depuis $\mathcal{C}_p$ est équivalente à l'exécution de référence jusqu'à la coupe $\mathcal{C}_p^{\sqcup}$.

Théorème 5.2.2 (Exécution contrôlée) *On a*

$$\mathcal{C}_p \xrightarrow{*} \mathcal{C}'_p \Rightarrow \exists \mathcal{C}' \text{ cohérente}, \mathcal{C}_p^{\sqcup} \xrightarrow{*} \mathcal{C}' \wedge \mathcal{C}'_p \xrightarrow{*} \mathcal{C}'$$

Les théorèmes 5.2.1 et 5.2.2 démontrent finalement la recouvrabilité d'une coupe $\mathcal{P}$-cohérente : toutes les exécutions possibles depuis une coupe $\mathcal{P}$-cohérente sont équivalentes à une exécution passant exactement par une coupe cohérente de l'exécution de référence.

5.2.3.2 $\mathcal{P}$-cohérence et inclusion

Nous étendons enfin l'inclusion entre les coupes pour les coupes $\mathcal{P}$-cohérentes. Cette définition de l'inclusion sera utilisée dans les preuves de la section suivante.

Définition 5.2.6 ($\subseteq$ pour les coupes $\mathcal{P}$-cohérentes) *Une coupe $\mathcal{P}$-cohérente $\mathcal{C}_p$ est inclue dans une autre $\mathcal{C}'_p$ si tous les évènements et promesses d'évènement dans $\mathcal{C}_p$ sont dans $\mathcal{C}'_p$, même si des promesses dans $\mathcal{C}_p$ sont remplacées par leur évènements correspondant dans $\mathcal{C}'_p$.*

$$\mathcal{C}_p \subseteq \mathcal{C}'_p \Leftrightarrow \begin{cases} e \in \mathcal{C}_p \Rightarrow e \in \mathcal{C}'_p \\ \mathcal{P}(e) \in \mathcal{C}_p \Rightarrow e \in_{\mathcal{P}} \mathcal{C}_p \end{cases}$$

Nous prouvons la propriété suivante : deux coupes $\mathcal{P}$-cohérentes incluses l'une dans l'autre peuvent toujours se réduire vers une même troisième coupe $\mathcal{P}$-cohérente. Cette propriété sera utilisée par la suite pour la preuve de la recouvrabilité d'une coupe $\mathcal{P}$-cohérentes dans le cadre d'une synchronisation uniquement locale de l'exécution.

Propriété 5.2.4 (Réduction de coupes $\mathcal{P}$-cohérentes incluses) *Soient $\mathcal{C}_p$ et $\mathcal{C}'_p$ deux coupes $\mathcal{P}$-cohérentes* d'une même exécution $(E, \prec)$*, alors on a :*

$$\mathcal{C}_p \subseteq \mathcal{C}'_p \Rightarrow \exists \mathcal{C}''_p \text{ coupe } \mathcal{P}\text{-cohérente de } (E, \prec), \mathcal{C}_p \xrightarrow{*} \mathcal{C}''_p \wedge \mathcal{C}'_p \xrightarrow{*} \mathcal{C}''_p$$

Enfin, nous définissons la notation suivante pour tout ensemble d'évènement E :

$$E \subseteq_{\mathcal{P}} \mathcal{C}_p \quad \Leftrightarrow \quad \forall e \in E, e \in \mathcal{C}_p \vee \mathcal{P}(e) \in \mathcal{C}_p$$

5.3 Cadre pratique : synchronisation locale

La définition des promesses d'évènement de la section précédente suppose qu'une promesse engendre une synchronisation *globale* qui permet d'assurer que l'ordre d'exécution des évènements respecte l'ordre causal et assure l'équivalence de la réexécution. Un tel mécanisme de synchronisation globale serait coûteux et complexe, voir impossible selon les implémentations, à mettre en oeuvre.

Nous allons maintenant définir un cadre pratique d'application de la $\mathcal{P}$-cohérence. L'idée est de réduire le contrôle nécessaire sur l'exécution à un contrôle uniquement local ; on ne suppose plus qu'une promesse d'évènement appartenant à une

activité puisse modifier l'exécution d'une autre activité, comme par exemple forcer une autre activité à attendre un évènement donné. La synchronisation générée par une promesse de requête n'est plus qu'une synchronisation locale.

Nous commençons donc par définir dans la section 5.3.1 la réduction sur un état *local*, ou réduction locale[2], dans le cadre d'une exécution de référence, c'est-à-dire dans le cas où l'état local considéré fait parti d'un état global cohérent. Cette définition va nous permettre d'établir des relations entre la réduction globale et la réduction locale, que nous exploiterons ensuite dans le cadre de la $\mathcal{P}$-cohérence.

Nous exposons dans la section 5.3.2 ensuite les capacités de contrôles de l'exécution locale que nous supposons ici, et montrons qu'elles sont réalisables en pratique.

Enfin, nous définissons dans la section 5.3.3 la réduction locale d'une coupe $\mathcal{P}$-cohérente, qui peut être respectée grâce à ces capacités de contrôle de l'exécution. Cette définition nous permet enfin d'identifier dans quel cadre cette réduction locale reste équivalente à la réduction globale d'une coupe $\mathcal{P}$-cohérente, et donc que la propriété de recouvrabilité est conservée. Nous formalisons ce cadre sous la forme de trois hypothèses qui portent sur la coupe $\mathcal{P}$-cohérente elle-même et sur le système.

Nous nous focaliserons par la suite sur les évènements qui créent la synchronisation entre les activités, c'est-à-dire les envois et les réceptions de messages. En effet, les évènements non déterministes d'un autre type comme la génération d'un nombre aléatoire rentrent dans le cadre de la $\mathcal{P}$-cohérence présentée précédemment : puisqu'ils n'engendrent pas de synchronisation entre les activités, il n'y a pas de différence entre le point de vue global et le point de vue local. Le fait que le mécanisme de tolérance aux pannes soit capable d'intercepter l'exécution des évènements non déterministes est suffisant : le contenu applicatif de l'évènement, par exemple la valeur aléatoire générée, peut être journalisée et restituée en cas de réexécution par le mécanisme. On peut donc toujours créer une promesse pour ce type d'évènement.

5.3.1 Réduction locale sur une coupe cohérente

Nous considérons dans cette section une exécution distribuée non plus comme un ensemble de réductions d'évènement sur un état *global*, mais sur différents états locaux. Nous définissons donc la réduction d'un état local par un évènement, ou réduction locale, et montrons son équivalence avec la réduction globale. En d'autres termes, toute réduction locale sur un état local appartenant à une coupe cohérente correspond à une réduction globale possible.

Nous notons s_i un état local quelconque de l'activité i, et $S\big|_i$ la restriction de l'état global S aux évènements exécutés sur l'activité i. Naturellement, on a

[2]Une réduction locale ne tient compte que de l'état local de l'activité

$e \in S \Leftrightarrow \exists i, e \in S\big|_i$. Tout état global S est de la forme $\{s_0 ... s_n\}$. La réduction locale d'un état local s_i par un évènement e vers un état s'_i est notée $s_i \xrightarrow{e}_L s'_i$. Enfin, si l'évènement e est exécuté sur l'activité i, nous notons $e \in A_i$.

Dans cette section, nous décrivons les restrictions et les hypothèses qui sont généralement faites sur le système dans la littérature. Nous montrons que ces hypothèses sont nécessaires pour qu'une réduction locale corresponde toujours à une réduction globale possible, c'est-à-dire pour que pour tout évènement $e \in A_i$:

$$s_i \xrightarrow{e}_L s'_i \;\Leftrightarrow\; \exists S, S'\; S\big|_i = s_i \;\wedge\; S \xrightarrow{e} S' \;\wedge\; S'\big|_i = s'_i$$

Cette équivalence entre la réduction globale et la réduction locale dans le cas d'une exécution depuis un état global *cohérent* est un outil essentiel pour la suite. Elle va nous permettre d'établir une relation entre les réductions globale et locale depuis une coupe $\mathcal{P}$-cohérente.

5.3.1.1 Synchronisation entre les activités

La synchronisation entre les activités ne se fait que par envoi et réception de messages. En particulier, les relations de causalité entre les activités ne peuvent se former qu'à travers un message. Nous posons donc l'hypothèse suivante :

Hypothèse 5.3.1 (La causalité se propage par message)

$$e_1 \prec e_2 \wedge e_1 \in A_i \wedge e_2 \in A_j \wedge i \neq j \Rightarrow \exists e'_1, e'_2 \textit{ tels que } (e'_1, e'_2) \in \Gamma \wedge e_1 \preceq e'_1 \prec e'_2 \preceq e_2$$

Donc, si aucune communication n'est mise en jeu, la réduction locale ne dépend que de l'état local s_i : tous les évènements internes peuvent être réduit de la même façon pour toutes les réductions possibles depuis un même état local.

Hypothèse 5.3.2 (Réduction des évènements internes)

$$\forall e,\, \nexists e', (e', e) \in \Gamma \wedge S \xrightarrow{e} S_1 \wedge S\big|_i = S'\big|_i \;\Rightarrow\; \left(S' \xrightarrow{e} S'_1 \wedge S_1\big|_i = S'_1\big|_i \wedge \left(i \neq j \Rightarrow S'_1\big|_j = S'\big|_j\right)\right)$$

5.3.1.2 Communications entre les activités

Nous formalisons ici l'hypothèse faite sur les communications dans la section 2.3.1 : les communications sont fiables, il n'y a ni perte ni duplication ni altération des messages. Donc tout message envoyé est finalement reçu :

Hypothèse 5.3.3 (Tout message envoyé est reçu) *Soit un message* $(e, e') \in \Gamma$*, alors on a*

$$e \in S \Rightarrow \exists S_1, S_2,\; S' \xrightarrow{*} S_1 \xrightarrow{e'} S_2$$

et localement

$$s_i \xrightarrow{e}_L s'_i \textit{ avec } s'_i = S\big|_i \Rightarrow \exists S',\; S \xrightarrow{*} S' \wedge s_j \xrightarrow{e'}_L s'_j,\; \textit{avec } s_j = S'\big|_j$$

Nous formalisons aussi l'hypothèse symétrique triviale : tout message reçu a été envoyé précédemment.

Hypothèse 5.3.4 (Tout message reçu est envoyé) *Soit un message* $(e, e') \in \Gamma$*, alors on a*

$$S \xrightarrow{e'} S' \Rightarrow e \in S$$

et localement

$$s_j \xrightarrow{e'}_L s'_j, \text{ avec } s_j = S'\big|_j \Rightarrow \exists S,\ S \xrightarrow{*} S' \wedge s_i \xrightarrow{e}_L s'_i \text{ avec } s'_i = S\big|_i$$

Enfin, nous pouvons supposer sans restriction qu'un envoi de message est forcément déterministe. L'exécution de l'envoi d'un message peut être conditionné par un évènement non déterministe qui le précède, mais l'envoi lui-même ne peut pas être non déterministe.

Hypothèse 5.3.5 (Tout envoi de message est déterministe)

$$(e, e') \in \Gamma \Rightarrow Det(e)$$

5.3.1.3 Ordonnancement des messages

Si on considère la réduction d'évènement d'un point de vue local, il faut alors tenir compte d'un potentiel ordonnancement des réceptions de messages, dépendant du système considéré. Nous proposons ici de spécifier l'ordonnancement des réceptions de message comme un ensemble d'évènements, pour pouvoir considérer dans ce formalisme n'importe quel ordonnancement de message.

L'ensemble $Odt_\prec(e)$ contient, en fonction de l'envoi e et du passé causal de cet envoi, l'ensemble des réceptions qui doivent être exécutés avant la réception correspondante e' sur l'activité de destination. Cet ensemble ne contient que les évènements qui ne font pas partie du passé causal de l'envoi du message par transitivité de $\prec$: il exprime l'ordonnancement des réceptions qui n'est pas exprimé dans la définition de $\prec$ pour le système considéré.

Définition 5.3.1 (Ordonnancement des messages)

$$\forall (e, e') \in \Gamma, e' \in A_i, Odt_\prec(e) = \left\{ e_0 \middle| \begin{array}{l} e_0 \in A_i \wedge \neg e_0 \prec e \ \wedge\ \exists S, S_1,\ e_1 \preceq e \Rightarrow e_1 \in S \wedge \\ S \xrightarrow{e_0} S_1 \wedge \nexists S_2,\ S \xrightarrow{e'} S_2 \ \wedge \exists S', S'_1\, S\big|_i = S'\big|_i \wedge S' \xrightarrow{e'} S'_1 \end{array} \right\}$$

L'ensemble d'évènements $Odt_\prec(e)$ ne peut dépendre que du passé causal de e et de e lui-même. Donc l'ensemble $Odt_\prec(e)$ est une fonction de $\{e' | e' \prec e\} = \bigcup_{e' \prec e} e$. Nous supposons ici que cette fonction d'union est l'union des résultats d'une autre fonction dépendante de e sur chaque évènement du passé de e. De façon plus formelle, on suppose ;

Hypothèse 5.3.6 (Partition de $Odt_\prec(e)$**)** *Pour tout envoi de message* e*, il existe une fonction* odt_e *telle que*

$$Odt_\prec(e) = \bigcup_{e' \prec e} odt_e(e')$$

Notons d'abord que cette hypothèse n'est pas utilisée dans la cadre des preuves sur la réduction locale depuis une coupe cohérente puisque l'ordonnancement des messages est par définition assuré par l'exécution, mais sera utilisée par la suite dans le cadre de la $\mathcal{P}$-cohérence.

Cette hypothèse exprime le fait que l'ordonnancement d'une réception peut être vu comme une conséquence des évènements de son passé *en les considérant un par un* en tenant compte de l'ordre : chaque évènement du passé d'une réception a ou n'a pas d'impact sur son ordonnancement. En d'autres termes, la présence d'une réception dans $Odt_{\prec}(e)$ ne peut pas être la conséquence de la *conjonction* de deux évènements du passé de e. Cette hypothèse est réaliste : elle est vérifiée par les ordonnancements de message classiques tels que l'ordonnancement causal ou encore le FIFO point-à-point. Prenons le cas de l'ordonnancement causal. On peut alors définir $Odt_{\prec}(e)$ de la manière suivante :

$$\forall (e, e') \in \Gamma, e' \in A_i, Odt_{\prec}(e) = \{e'_0 | (e_0, e'_0) \in \Gamma \wedge e_0 \prec e \wedge e'_0 \in A_i\}$$

où A_i est l'activité qui reçoit le message correspondant à l'envoi e. Dans ce cas, $Odt_{\prec}(e)$ est bien partitionnable de la manière suivante :

$$\forall (e, e') \in \Gamma, e' \in A_i, Odt_{\prec}(e) = \bigcup_{e_0 \prec e} \{e'_0 | (e_0, e'_0) \in \Gamma \wedge e'_0 \in A_i\}$$

5.3.1.4 Relations entre réduction locale et réduction globale

Nous supposons finalement qu'une réduction locale $s_i \xrightarrow{e}_L s'_i$ n'est possible que si il existe une réduction globale depuis un état global contenant s_i, et que, si e est une réception, alors tous les évènements contenus dans l'ensemble $Odt_{\prec}(e')$ (où e' est l'envoi correspondant à la réception e) ont été exécutés.

Hypothèse 5.3.7 (Correspondance entre réduction locale et globale)

$$s_i \xrightarrow{e}_L s'_i \Leftrightarrow \begin{cases} \exists S, S',\ S\big|_i = s_i \wedge S'\big|_i = s'_i \wedge S \xrightarrow{e} S' & \wedge \\ \exists e', (e', e) \in \Gamma \Rightarrow Odt_{\prec}(e') \subseteq s_i \end{cases}$$

Les définitions et hypothèses précédentes permettent enfin de prouver la propriété suivante : toutes les réductions locales possibles sur un état local s_i appartenant à un état global S correspondent à une réduction globale possible sur l'état global S, et réciproquement.

Propriété 5.3.1 (Equivalence entre réduction locale et globale) *Sous les hypothèses sur les communications 5.3.3 et 5.3.4, les hypothèses 5.3.2 and 5.3.7 sont équivalentes à la propriété suivante :*

Soient les états globaux $S = \{s_0 \dots s_i \dots s_n\}$ *et* $S' = \{s_0 \dots s'_i \dots s_n\}$*, alors*

$$s_i \xrightarrow{e}_L s'_i \Leftrightarrow S \xrightarrow{e} S'$$

Notons que cette propriété permet de montrer que les hypothèses classiques sur le système présentées ici sont suffisantes puisque $s_i \xrightarrow{e}_L s'_i \Rightarrow S \xrightarrow{e} S'$, mais aussi qu'elles sont nécessaires puisque $s_i \xrightarrow{e}_L s'_i \Leftarrow S \xrightarrow{e} S'$.

5.3.2 Contrôle de l'exécution

Dans le cadre idéal (section 5.2.1), la synchronisation engendrée par une promesse d'évènement $\mathcal{P}(e)$ doit assurer *globalement* que les évènements qui sont une conséquence causale de e ne sont pas exécutés avant e, et les évènements qui font partie du passé causal de e sont exécutés avant e. Nous montrons dans cette section qu'il est réaliste en pratique d'assurer ces deux propriétés *localement*, et nous décrivons le contrôle sur l'exécution locale nécessaire.

Notons que d'un point de vue local, les seuls évènements qui peuvent être exécutés avant leurs causes sont les réceptions de message. En effet, ce sont les seuls évènements qui ne sont pas déclenchés par l'activité elle-même. Pour assurer l'ordre local, il nous faut donc considérer uniquement les deux cas suivants :

- une réception est exécutée *en retard* : si l'état local courant s de l'activité contient une promesse pour cet évènement, l'activité ne doit pas exécuter d'évènement qui soit une conséquence de cet réception en retard tant qu'elle n'a pas eu lieu.
- une réception est exécutée *en avance* : l'état local courant s de l'activité ne contient pas encore tout le passé de cette réception. Dans ce cas, il doit être possible de la réordonner par rapport aux évènements qui n'ont pas encore eu lieu et qui sont ou non promis.

5.3.2.1 Mécanisme de contrôle de l'exécution

Nous supposons l'existence sur chaque activité d'un mécanisme dédié à la tolérance aux pannes capable d'agir sur l'exécution de façon uniquement locale. En particulier, nous ne supposons pas que deux mécanismes de deux activités distinctes puissent échanger des informations. Nous pouvons de façon réaliste supposer que ce mécanisme :

- connaît les relations causales *locales* qui lient les évènements entre eux. En effet, nous nous plaçons dans un cadre où la relation de causalité potentielle entre les évènements locaux est définie statiquement en fonction de la sémantique du système considéré, à partir des relations de compatibilité entre évènements. Cette relation de causalité est donc toujours vraie et peut être utilisée dans le mécanisme dédié. On notera ici que l'on *ne suppose pas* que ce mécanisme est capable de restituer les relations entre évènements dûes aux communications, mais seulement celles définies par l'ordre local $\prec_i$.

  Le mécanisme doit être capable d'exploiter ces relations ; il faut donc qu'il soit capable d'intercepter certains types d'évènement qui sont utilisés pour définir $\prec_i$. Le mécanisme peut cependant, selon les possibilités de l'implémentation, exploiter une approximation de $\prec_i$, selon les types d'évènement qui sont interceptés ou non.
- est capable d'intercepter l'exécution des tous les évènements non déterministes. Cette supposition est une conséquence directe de l'hypothèse du déterminisme par morceaux, puisque tout évènement non déterministe doit être identifiable. Nous supposons donc que le mécanisme peut, si nécessaire, journaliser puis restituer le contenu d'un évènement non déterministe.

– et enfin est capable d'agir sur l'exécution de l'activité en la bloquant et la redémarrant de manière transparente.

5.3.2.2 Réception en retard : attente par nécessité

Dans le cas d'une réception en retard par rapport à un état s, une attente par nécessité permet d'éviter l'exécution de conséquences locales de cet évènement. Comme nous disposons d'un mécanisme capable d'identifier les relations causales locales et de stopper l'exécution, cette attente par nécessité sur l'exécution des conséquences des promesses d'évènement est réalisable. La sémantique de cette synchronisation est la suivante : si une activité i tente d'exécuter un évènement qui *n'est pas une réception de message* et qui est une *conséquence causale* d'une promesse d'évènement $\mathcal{P}(e)$ dans l'état courant s, alors l'exécution est immédiatement bloquée. L'activité i est mise en attente sur $\mathcal{P}(e)$. Lorsque l'évènement e' tel que $\mathcal{P}(e) \lhd e'$ est exécuté, e' se substitue alors à $\mathcal{P}(e)$ dans l'état s_i et i est débloquée si elle était est en attente sur $\mathcal{P}(e)$.

Cette restriction d'une synchronisation globale vers une synchronisation locale a une conséquence sur la création de promesses d'évènement. En effet, on suppose ici qu'il est suffisant de bloquer l'exécution *locale* pour assurer qu'il n'y aura pas d'évènement conséquence d'une promesse qui soit exécuté dans *tout le système* ; pour que cela soit vrai, il faut que cette promesse n'ait pas eu de conséquences sur les autres activités. Dans le cas contraire, une attente locale n'est plus suffisante. Dans le cadre de l'utilisation de l'attente par nécessité locale comme moyen de synchronisation sur les promesses d'évènement, nous imposons donc une condition sur la transformation d'une réception e par une promesse $\mathcal{P}(e)$ dans un état local s_i : il ne doit pas y avoir dans s_i d'envoi de message qui soit une conséquence de e. On note cependant que l'état s_i peut contenir des évènements qui sont des conséquences causales de e mais qui ne sont pas des envois. Nous avons donc la propriété suivante :

Propriété 5.3.2 (Promesse de réception)

$$\mathcal{P}(e) \in s \Rightarrow \nexists (e', e'') \in \Gamma, e' \in s \wedge e \prec_i e'$$

La conséquence de cette propriété est qu'il n'est pas toujours possible de transformer un évènement dans un état local s_i en promesse d'évènement. Plus précisément, il n'est pas toujours *utile* de transformer un évènement en promesse. En effet, si la propriété 5.3.2 n'est pas respectée, l'attente par nécessité n'est plus suffisante pour assurer la position de l'évènement promis. Dans la section suivante, nous verrons comment l'hypothèse 5.3.10 **Ordonnancement promis** permet de prendre en compte le cas de ces évènements qui ne peuvent pas être promis.

5.3.2.3 Réception en avance : positionnabilité

Le cas des réceptions en avance se rencontre dans les protocoles de tolérance aux pannes par journalisation des messages : en cas de reprise, on suppose que

tous les messages peuvent être reçus par l'activité puis réordonnés si nécessaire. En particulier, les messages journalisés et renvoyés artificiellement à l'activité doivent pouvoir être réordonnés entre eux mais aussi par rapport aux messages qui sont envoyés normalement par d'autres activités. Le réordonnancement se fait généralement en utilisant les déterminants de messages, qui indiquent la position de la réception par rapport aux autres. Il est donc réaliste d'utiliser des promesses d'évènement pour réordonner les messages en avance, qui peuvent contenir un déterminant de message.

Pourtant, nous n'allons pas considérer que l'on peut créer une promesse pour *toutes* les réceptions en avance. Plus généralement, nous ne considérons pas que tous les messages peuvent être reçus n'importe quand puis réordonnées artificiellement. En effet, l'hypothèse sous-jacente est ici que les réceptions de messages sont toujours des évènements compatibles avec tous les autres types d'évènement. Dans le cas contraire, exécuter une réception pourrait rendre l'état local incohérent, puisque des conséquences seraient exécutées avant les causes. Par exemple, comme nous l'avons vu dans les sections 3.1.2 et 3.3.3, les réceptions de réponse en ASP ne peuvent pas avoir lieu avant l'envoi de la requête correspondante.

Une solution possible pour que l'état local reste cohérent serait de repousser artificiellement la réception. Il faudrait donc un mécanisme capable de retarder l'exécution de certains évènements, puis de déclencher leur exécution à un moment précis, en fonction de l'état courant de l'activité. En pratique, cela peut être implémenté sous la forme d'un tampon de messages reçus. Ce tampon doit être capable de délivrer les messages au bon moment, et placé comme intermédiaire entre l'activité et la couche de communication.

Plutôt que de supposer l'existence systématique d'un tel mécanisme, nous considérerons par la suite le cas des réceptions non positionnables à travers l'hypothèse 5.3.9 **Positionnement des réceptions** qui sera définie dans la section suivante. Cette hypothèse pose que si une réception ne peut pas être manipulée et repoussée, alors son passé local doit avoir été exécuté par le récepteur avant son exécution.

5.3.3 Réduction locale d'une coupe $\mathcal{P}$-cohérente

De façon symétrique à la définition d'une réduction locale sur les coupes cohérentes dans la section 5.3.1, nous allons définir ici une réduction locale sur une coupe $\mathcal{P}$-cohérente. L'objectif est de montrer que cette réduction locale est équivalente à la réduction globale sur une coupe $\mathcal{P}$-cohérente, donc que les propriétés de cette réduction globale sont conservées pour la réduction locale. Cette équivalence repose sur trois hypothèses concernant :

- les états locaux formant la coupe,
- la capacité de positionner des messages reçus en avance, c'est-à-dire avant l'exécution de leur passé par l'activité réceptrice,
- la conservation de l'ordonnancement de message durant la réexécution.

Nous formalisons ces trois hypothèses, puis prouvons que dans ce cas, les propriétés de la réduction globale sont conservées par la réduction locale.

5.3.3.1 Coupe de recouvrement

Une différence majeure avec la réduction locale sur les coupes cohérentes est que la réduction définie ici est forcément *une réexécution*, c'est-à-dire depuis un état global formé d'un ensemble de captures d'état faites pendant l'exécution de référence, mais ne formant pas un état global cohérent. Cette coupe $\mathcal{P}$-cohérente depuis laquelle la réexécution est lancée, ou *coupe de recouvrement*, sera notée par la suite $\mathcal{C}_p^0 = \{c_1^0 \dots c_n^0\}$. Dans cette coupe $\mathcal{C}_p^0$:

- les réceptions orphelines sont remplacées par des promesses de réceptions *si cela est possible*, c'est-à-dire si la propriété 5.3.2 est assurée,
- les états locaux qui constituent la coupe de recouvrement sont cohérents : si e appartient à l'état local s, tous les évènements locaux du passé de e appartiennent *ou sont promis* dans s.

Ces deux points se formalisent par l'hypothèse suivante :

Hypothèse 5.3.8 (Cohérence des états locaux)

$$e, e' \in A_i \wedge e \prec e' \wedge e' \in \mathcal{C}_p^0 \Rightarrow e \in \mathcal{C}_p^0 \vee \begin{cases} \mathcal{P}(e) \in \mathcal{C}_p^0 \ \wedge \\ \exists e'',\ (e'', e) \in \Gamma \wedge e'' \notin \mathcal{C}_p^0 \end{cases}$$

Nous supposons que les messages en transit dans la coupe seront réémis artificiellement. En particulier, nous supposons que les messages en transit peuvent être réémis et donc reçu *n'importe quand*, sans aucune synchronisation avec la réexécution. De plus, nous supposons que la réception des messages orphelins qui ne correspondent à aucune promesse n'est jamais exécutée.

Nous avons vu qu'il était facile en pratique d'identifier les messages orphelins ou en transit sur le récepteur grâce à un estampillage des messages par l'index du dernier point de reprise de l'émetteur. Il est donc raisonnable de poser que les messages en transit peuvent être journalisés puis réémis au moment de la réexécution, et que les duplicatas de messages orphelins qui ne sont pas promis peuvent être identifiés et ignorés pendant la réexécution.

5.3.3.2 Définition de la réduction locale

Pour pouvoir définir la réduction d'une coupe $\mathcal{P}$-cohérente de manière locale, il faut identifier une coupe locale qui correspond à un état possible de l'activité, c'est-à-dire qui ne contient ni de promesse d'évènement ni de conséquence sans ses causes. En pratique, nous identifions pour toutes c_i la coupe locale $c_i^{\sqcap}$ comme la plus grande coupe locale ne contenant pas de promesse d'évènement. La réduction locale définie ici se base sur les réductions *possibles* sur $c_i^{\sqcap}$, puisque $c_i^{\sqcap}$ correspond à un état réel s_i de l'activité i.

De façon informelle, la coupe c'_i résultante de la réduction d'une coupe locale c_i appartenant à une coupe $\mathcal{P}$-cohérent est obtenue :

- soit en remplaçant une promesse par un évènement correspondant si elle existe dans c_i,
- soit en repoussant un évènement non déterministe à la fin de la coupe c_i,
- soit en exécutant un évènement déterministe sur l'état local réel $c_i^{\sqcap}$.

La coupe locale $c_i^{\sqcap}$ est en fait une approximation locale de l'état global cohérent $\mathcal{C}_p^{\sqcap}$ (section 5.2.3). La réduction locale doit assurer la cohérence de cet état local durant la réexécution. Nous devons donc éviter qu'une promesse d'évènement ou un évènement qui ne peut pas encore appartenir à $c_i^{\sqcap}$ ait des conséquences sur la réduction. En pratique, il suffit d'assurer l'attente par nécessité sur les conséquences de promesses d'évènement, c'est-à-dire empêcher l'exécution des conséquences locales des promesses. Nous montrerons par la suite qu'assurer cette cohérence locale assure la conservation de la $\mathcal{P}$-cohérence de manière globale.

Définition 5.3.2 (Réduction locale d'une coupe $\mathcal{P}$-cohérente) *Nous notons $c_i \xrightarrow{e}_L c'_i$ la réduction locale d'un évènement $e \notin c_i$ sur la coupe locale c_i, avec $c_i = \mathcal{C}_p\big|_i$ et $\mathcal{C}_p$ une coupe $\mathcal{P}$-cohérente. Si la réduction de e est possible sur $c_i^{\sqcap}$ vers un état s'_i, alors e se réduit sur c_i de la façon suivante :*

$$c_i^{\sqcap} \xrightarrow{e}_L s'_i \Rightarrow \begin{cases} c_i \xrightarrow{e'}_L c_i\{\mathcal{P}(e') \leftarrow e'\} & \textit{si } \exists \mathcal{P}(e') \in \mathcal{C}_p, \mathcal{P}(e') \lhd e \\ c_i \xrightarrow{e}_L c_i \oplus \{e\} & \textit{sinon} \end{cases}$$

Avec les notations de la définition 5.2.5 redéfinies avec l'ordre local $\prec_i$:

$$c_i\{\mathcal{P}(e') \leftarrow e'\} = c_i \backslash \{\mathcal{P}(e)'\} \cup \{e'\} \textit{ avec pour tout } e_{\mathcal{P}}, e'_{\mathcal{P}} \in c_i\{\mathcal{P}(e') \leftarrow e'\}$$

$$e_{\mathcal{P}} \prec'_i e'_{\mathcal{P}} \textit{ ssi } \begin{cases} e_{\mathcal{P}} = e' \wedge \mathcal{P}(e') \prec_i e'_{\mathcal{P}} \ \vee & e' \textit{ prend la position de } \mathcal{P}(e') \\ e'_{\mathcal{P}} = e' \wedge e_{\mathcal{P}} \prec_i \mathcal{P}(e') \ \vee & e' \textit{ prend la position de } \mathcal{P}(e') \\ e_{\mathcal{P}} \neq e' \wedge e'_{\mathcal{P}} \neq e' \wedge e_{\mathcal{P}} \prec_i e'_{\mathcal{P}} & \prec'_i \textit{ est } \prec_i \textit{ sauf pour } e' \end{cases}$$

$$c_i \oplus \{e\} = c_i \cup \{e\} \textit{ avec pour tout } e_{\mathcal{P}}, e'_{\mathcal{P}} \in c_i \oplus \{e\}$$

$$e_{\mathcal{P}} \prec'_i e'_{\mathcal{P}} \textit{ ssi } \begin{cases} \neg Det(e) \wedge e'_{\mathcal{P}} = e \wedge \neg e \bowtie e_{\mathcal{P}} \vee & \textit{si } \neg Det(e), e \textit{ à la fin de } c_i \\ Det(e) \wedge e'_{\mathcal{P}} = e \wedge e_{\mathcal{P}} \in c_i^{\sqcap} \wedge \neg e \bowtie e_{\mathcal{P}} \ \vee & \textit{sinon } e \textit{ après } c_i^{\sqcap} \\ Det(e) \wedge e_{\mathcal{P}} = e \wedge e'_{\mathcal{P}} \notin c_i^{\sqcap} \wedge \neg e \bowtie e'_{\mathcal{P}} \vee & \textit{mais avant les autres évènements} \\ e_{\mathcal{P}}, e'_{\mathcal{P}} \in \mathcal{C}_p \wedge e_{\mathcal{P}} \prec_i e'_{\mathcal{P}} \ \vee & \prec'_i \textit{ contient } \prec_i \\ \exists e''_{\mathcal{P}} \in \mathcal{C}'_p,\ e_{\mathcal{P}} \prec'_i e''_{\mathcal{P}} \wedge e''_{\mathcal{P}} \prec'_i e'_{\mathcal{P}} & \textit{clôture transitive} \end{cases}$$

Nous étendons de façon triviale la notation de la réduction locale $\xrightarrow{e}_L$ sur une coupe $\mathcal{C}_p = \{c_1 \dots c_n\}$ vers une coupe $\mathcal{C}'_p = \{c'_1 \dots c'_n\}$ de la manière suivante :

$$\mathcal{C}_p \xrightarrow{e}_L \mathcal{C}'_p \Leftrightarrow \mathcal{C}_p = \{c_1 \dots c_n\} \wedge c_i \xrightarrow{e}_L c'_i \wedge \mathcal{C}'_p = \{c_1 \dots c'_i \dots c_n\}$$

Comme on peut le voir, les manipulations d'évènements $c_i\{\mathcal{P}(e') \leftarrow e'\}$ et $c_i \oplus \{e\}$ reposent maintenant sur une connaissance de l'ordre local $\prec_i$ uniquement.

Dans le cas de $c_i\{\mathcal{P}(e') \leftarrow e'\}$, soit la réception est en retard, et l'attente par nécessité assure que l'activité l'attend, et donc qu'elle retrouve sa place, soit elle est en avance et la promesse de réception permet de la replacer par rapport aux autres réceptions.

Dans le cas de $c_i \oplus \{e\}$, e peut être un événement déterministe ; dans ce cas, il est réduit sur l'état local $c_i^{\sqcap}$, ce qui correspond à l'exécution normale d'un évènement, comme défini dans la section 5.3.1. Si e est une réception et qu'elle est en avance, alors on doit être capable de la repositionner (le cas "si $\neg Det(e)$, e à la fin de c_i"). Il faut donc que cette réception soit positionnable. Dans le cas contraire, c_i *doit contenir* le passé causal de e'. Nous formalisons par la suite la notion de positionnement d'une réception dans l'hypothèse 5.3.9.

Enfin, notons qu'il n'y a pas d'ambiguïté introduite par la notation commune entre la réduction locale d'une coupe cohérente et la réduction locale d'une coupe $\mathcal{P}$-cohérente. En effet, si la coupe $\mathcal{C}_p$ considérée dans la définition 5.3.2 est cohérente, alors on a $c_i = c_i^{\sqcap}$, donc la seule réduction possible est $c_i^{\sqcap} \xrightarrow{e}_L s_i'$, avec $s_i' = c_i^{\sqcap} \oplus \{e\}$, ce qui correspond à la réduction d'une coupe cohérente.

5.3.3.3 Positionnement des réceptions

Nous formalisons ici sous forme d'une hypothèse les capacités de réception d'un message en avance discutées dans la section 5.3.2.3. Nous distinguons donc deux cas, selon que la réception est de nature positionnable ou non. Les réceptions positionnables doivent pouvoir être prises en compte en avance, c'est-à-dire avant leur passé local, sans modifier le cours de l'exécution jusqu'au moment voulu. Dans le cas d'une réception non positionnable, le passé de la réception doit être contenu dans l'état local au moment de son exécution.

Hypothèse 5.3.9 (Positonnement des réceptions)

$$\forall \mathcal{C}_p = \{c_1 \dots c_n\},\ \forall e, e_0 \in A_i,\ c_i \xrightarrow{e}_L c_i' \wedge (Det(e_0) \vee e_0 \in_{\mathcal{P}} \mathcal{C}_p) \wedge e_0 \prec e \Rightarrow$$
$$e_0 \in \mathcal{C}_p \quad \vee \quad \left(\neg Det(e) \wedge \exists e',\ (e', e) \in \Gamma \wedge \left(\exists c_i'' c_i \xrightarrow{e_0}_L c_i'' \ \Leftrightarrow \exists c_i''',\ c_i' \xrightarrow{e_0}_L c_i'''\right)\right) \textit{ avec } c_i'' \xrightarrow{e}_L c_i'''$$

On peut montrer que, trivialement, dans une exécution depuis une coupe cohérente, on a toujours le cas de gauche $e_0 \in \mathcal{C}_p$; on n'a donc jamais besoin de positionner artificiellement la réception d'un message. De plus, nous pouvons voir que si une réception est déterministe, alors il faut que $e_0 \in \mathcal{C}_p$ soit assuré, puisque l'autre possibilité implique $\neg Det(e)$. Par conséquent, une réception déterministe est toujours non positionnable. En effet, la réduction locale définie en 5.3.2 place *toujours* un évènement déterministe juste après la coupe locale $c_i^{\sqcap}$ puisqu'il ne peut pas y avoir de promesse pour cet évènement qui permettrait de le positionner. Notons que l'utilisation d'un mécanisme de temporisation comme discuté dans la section 5.3.2.3 permettrait de voir toute réception comme un évènement non déterministe, et rendrait donc positionnable n'importe quelle réception.

5.3.3.4 Ordonnancement des messages

Pendant la réexécution, une partie de la synchronisation entre les activités est perdue à cause des messages orphelins et des messages en transit. En d'autres termes, certains des éléments de $Odt_{\prec}(e)$ ne peuvent plus être déterminés dynamiquement en fonction du passé de e. Notons $Odt_{\prec'}(e)$ l'ensemble des évènements qui peuvent encore être déterminés dynamiquement en fonction du passé de e avec la perte potentielle de synchronisation ; $Odt_{\prec'}(e)$ représente la partie de l'ordonnancement de message qui est restitué par la réexécution pour l'envoi de message (e, e'). Comme on a vu avec l'hypothèse de partitionnement 5.3.6, on peut voir $Odt_{\prec'}(e)$ comme :

$$Odt_{\prec'}(e) = \bigcup_{e' \prec' e} odt_e(e')$$

où $e' \prec' e$ si et seulement si il existe au moins une suite d'évènements $e_1 \dots e_n$ telle que $e' \preceq e_1 \prec \dots \prec e_n \prec e$ et $\{e_1 \dots e_n\} \cap \mathcal{C}_p^0 = \emptyset$. En effet, si un évènement de la suite $e_1 \dots e_n$ appartient déjà à la coupe de recouvrement $\mathcal{C}_p^0$, cet évènement ne sera pas exécuté durant la réexécution, donc la chaîne de causalité est brisée. Dans ce cas, $odt_e(e')$ ne peut pas être restitué par la réexécution. En remplaçant l'existence d'au moins une suite d'évènements par l'ensemble de toutes les suites possibles, nous pouvons donc approximer un sur-ensemble de $Odt_{\prec'}(e)$:

$$Odt_{\prec'}(e) \supseteq \bigcup_{e' \in A} odt_e(e'), \text{ avec } A = \{e' | e' \prec e \wedge \nexists e_1 \in \mathcal{C}_p^0,\ e' \preceq e_1 \prec e\}$$

Nous voulons assurer que toutes les réceptions qui ne font plus partie de $Odt_{\prec'}(e)$ correspondent à une promesse dans la coupe $\mathcal{P}$-cohérente de recouvrement. De cette manière, l'ordonnancement perdu est recréé artificiellement. De manière plus formelle, nous voulons assurer que tout au long de la réexécution (i.e. pour toute réception de la réexécution), on a

$$Odt_{\prec}(e) \backslash Odt_{\prec'}(e) \subseteq_{\mathcal{P}} \mathcal{C}_p$$

qui est vérifié si on suppose que $\mathcal{C}_p^0$ est telle que

$$\bigcup_{\substack{e' \prec e \\ \exists e_1 \in \mathcal{C}_p^0,\ e' \preceq e_1 \prec e}} odt_e(e') \subseteq_{\mathcal{P}} \mathcal{C}_p^0$$

En effet, d'après l'approximation précédente d'un sur-ensemble de $Odt_{\prec'}(e)$, on a

$$Odt_{\prec}(e) \backslash Odt_{\prec'}(e) \subseteq \bigcup_{e' \prec e \wedge e' \notin A} odt_e(e') = \bigcup_{\substack{e' \prec e \\ \exists e_1 \in \mathcal{C}_p^0,\ e' \preceq e_1 \prec e}} odt_e(e') \subseteq_{\mathcal{P}} \mathcal{C}_p^0 \subseteq \mathcal{C}_p$$

Nous pouvons donc identifier tous les évènements qui font parti de $Odt_{\prec}(e)$ mais qui ne font pas parti de $Odt_{\prec'}(e)$. Considérons un message (e, e'), et la partie du passé de l'envoi e qui précède causalement un évènement e_0 de la coupe

$\mathcal{C}_p^0$. Tout l'ordonnancement généré par cette partie du passé, c'est-à-dire les évènements définis par $odt_e(e_i)$ tel que $e_i \prec e_0$, doit être restitué. Nous posons donc l'hypothèse que ces évènements doivent appartenir, promis ou non, à la coupe de recouvrement $\mathcal{C}_p^0$.

Hypothèse 5.3.10 (Ordonnancement promis)

$$\forall e\, \exists e_2,\, (e, e_2) \in \Gamma,\, \forall e_1 \in \mathcal{C}_p^0,\, e_1 \prec e,\, \forall e',\, e' \preceq e_1,\, odt_e(e') \subseteq_{\mathcal{P}} \mathcal{C}_p^0$$

Notons que cette hypothèse implique que pour tout message $(e, e') \in \Gamma$ en transit, l'ensemble complet $Odt_{\prec}(e)$ est inclus, promis ou non, dans $\mathcal{C}_p^0$.

Cette hypothèse nous permet de montrer que durant toute la réexécution depuis $\mathcal{C}_p^0$, l'ordonnancement des messages perdu à cause des messages orphelins ou en transit est restitué sous la forme de promesses de réception. L'hypothèse 5.3.10 implique de façon triviale la propriété 5.3.3.

Propriété 5.3.3 (Ordonnancement restitué)

$$\forall \mathcal{C}_p \supseteq \mathcal{C}_p^0,\, Odt_{\prec}(e) \backslash Odt_{\prec'}(e) \subseteq_{\mathcal{P}} \mathcal{C}_p$$

Enfin, nous étudions l'impact de cette propriété sur l'hypothèse 5.3.7. L'hypothèse 5.3.7 de correspondance entre la réduction locale et une réduction globale reste vraie dans le cas d'une réduction sur $c_i^{\sqcap}$ puisque $c_i^{\sqcap}$ correspond à un état réel possible s_i. Cette hypothèse ainsi que l'hypothèse 5.3.10 nous permettent de prouver une relation de correspondance entre la réduction locale sur $c_i^{\sqcap}$ et une réduction globale sur un état global cohérent. La différence avec le cas strictement cohérent est que les évènements qui doivent être ordonnées à cause de l'ordonnancement de messages peuvent être promis.

Propriété 5.3.4 (Correspondance entre réduction locale et globale)

$$c_i^{\sqcap} \xrightarrow{e}_L c_i' \Leftrightarrow \begin{cases} \exists S, S',\, S\big|_i = c_i^{\sqcap} \wedge S \xrightarrow{e} S' & \wedge \\ (e', e) \in \Gamma \Rightarrow Odt_{\prec}(e') \subseteq_{\mathcal{P}} c_i \end{cases}$$

Nous pouvons maintenant prouver que les propriétés de la réduction globale sont conservées, et donc qu'une coupe $\mathcal{P}$-cohérente est recouvrable un cadre de synchronisation et contrôle d'exécution uniquement local.

5.3.3.5 Preuve de la recouvrabilité dans le cadre d'une synchronisation locale

Nous prouvons tout d'abord que la réduction locale maintient la $\mathcal{P}$-cohérence.

Propriété 5.3.5 (La réduction locale maintient la $\mathcal{P}$-cohérence) *La réduction locale sur une coupe $\mathcal{P}$-cohérente produit une coupe $\mathcal{P}$-cohérente d'une exécution E' (qui peut être différente de E après $\mathcal{C}_p^{\sqcup}$). Si $\mathcal{C}_p$ est une coupe $\mathcal{P}$-cohérente de l'exécution $(E, \prec)$, alors*

$$\mathcal{C}_p \xrightarrow{e}_L \mathcal{C}_p' \Rightarrow \exists (E', \prec')\ \textit{tel que}\ \mathcal{C}_p'\ \textit{est une coupe}\ \mathcal{P}\textit{-cohérente de}\ (E', \prec')$$

Nous déterminons ensuite la relation entre la coupe locale $c_i^{\sqcap}$ et la coupe cohérente $\mathcal{C}_p^{\sqcap}$ pour une coupe $\mathcal{P}$-cohérente $\mathcal{C}_p$ donnée. En effet, nous avons affirmé précédemment que la coupe locale $c_i^{\sqcap}$ est une approximation locale de la coupe cohérente $\mathcal{C}_p^{\sqcap}$. En fait, nous pouvons montrer que tout au long de la réexécution, la coupe $\mathcal{C}_p^{\sqcap}$ est inclue dans l'ensemble des $c_i^{\sqcap}$.

Propriété 5.3.6 ($\mathcal{C}_p^{\sqcap}\big|_i$ inclus dans $c_i^{\sqcap}$) *Après chaque réduction locale, on a :*

$$\mathcal{C}_p^{\sqcap} \subseteq \{c_1^{\sqcap}, \ldots, c_n^{\sqcap}\}$$

Nous pouvons finalement montrer que la réduction locale définie ici conserve les propriétés de la réduction globale définie en 5.2.5. Dans un premier temps, nous prouvons que la réduction globale sans blocage définie dans le théorème 5.2.1 est toujours faisable avec la réduction locale :

Propriété 5.3.7 (Réduction sans blocage possible) *L'exécution possible définie par le théorème 5.2.1 peut être faîte par la réduction locale $\rightarrow_L$:*

$$\mathcal{C}_p \xrightarrow{e} \mathcal{C}'_p \wedge (Det(e) \vee \exists \mathcal{P}(e) \textit{ minimale dans } \mathcal{C}'_p \backslash \mathcal{C}_p\ ) \Rightarrow \mathcal{C}_p \xrightarrow{e}_L \mathcal{C}'_p$$

Par conséquent, il existe donc toujours une suite de réductions locales qui réduisent la coupe de recouvrement $\mathcal{C}_p^0$ vers une coupe cohérente de l'exécution de référence. Nous prouvons à travers la propriété 5.3.7 que le théorème 5.2.1 d'exécution possible dans le cas de la réduction globale reste vrai dans le cas de la réduction locale :

Théorème 5.3.1 (Exécution possible) *Il existe une exécution depuis la coupe $\mathcal{C}_p$ vers la coupe cohérente $\mathcal{C}_p^{\sqcup}$:*

$$\mathcal{C}_p \xrightarrow{*}_L \mathcal{C}_p^{\sqcup}$$

La réduction locale ne peut pas être strictement équivalente à la réduction globale sur une coupe $\mathcal{P}$-cohérente puisque $c_i^{\sqcap}$ ne donne qu'une *approximation* locale de la coupe $\mathcal{C}_p^{\sqcap}$ qui est utilisée dans la définition de la réduction globale. Cependant, nous montrons que la réduction locale est *finalement* équivalente à la réduction globale, c'est-à-dire toute suite de réductions locales fini par atteindre un état atteignable par réduction globale.

Propriété 5.3.8 (Etats globaux communs) *Toute suite de réductions locales finit par atteindre un état atteignable par réduction globale :*

$$\mathcal{C}_p \xrightarrow{*}_L \mathcal{C}'_p \;\Rightarrow\; \mathcal{C}'_p \xrightarrow{*}_L \mathcal{C}''_p \wedge \mathcal{C}_p \xrightarrow{*} \mathcal{C}''_p$$

Par conséquent, nous pouvons montrer que le théorème 5.2.2 d'exécution contrôlée dans le cas de la réduction globale reste vrai dans le cas de la réduction locale : toutes les exécutions possibles sont équivalentes jusqu'à une coupe cohérente de l'exécution de référence. En effet, la propriété 5.3.8 assure que :

$$\mathcal{C}_p \xrightarrow{*}_L \mathcal{C}'_p \;\Rightarrow\; \mathcal{C}'_p \xrightarrow{*}_L \mathcal{C}''_p \wedge \mathcal{C}_p \xrightarrow{*} \mathcal{C}''_p$$

De plus, le théorème 5.3.1 nous assure qu'il existe au moins une suite de réductions locales possible depuis la coupe $\mathcal{C}''_p$ jusqu'à la coupe *cohérente* $\mathcal{C}''^{\sqcup}_p$: $\mathcal{C}''_p \xrightarrow{*}_L \mathcal{C}''^{\sqcup}_p$.

Comme on a $\mathcal{C}_p^{\sqcup} \subseteq \mathcal{C}_p''^{\sqcup}$, la propriété 5.2.4 nous dit qu'il existe un suite de réductions globales telle que $\mathcal{C}_p^{\sqcup} \xrightarrow{*} \mathcal{C}_p''^{\sqcup}$. Nous pouvons donc conclure :

$$\mathcal{C}_p \xrightarrow{*}_L \mathcal{C}'_p \;\Rightarrow\; \mathcal{C}'_p \xrightarrow{*}_L \mathcal{C}''^{\sqcup}_p \wedge \mathcal{C}_p^{\sqcup} \xrightarrow{*} \mathcal{C}''^{\sqcup}_p$$

Théorème 5.3.2 (Exécution contrôlée)

$$\mathcal{C}_p \xrightarrow{*}_L \mathcal{C}'_p \;\Rightarrow\; \exists \mathcal{C} \textit{ cohérente}, \mathcal{C}'_p \xrightarrow{*}_L \mathcal{C} \wedge \mathcal{C}_p^{\sqcup} \xrightarrow{*} \mathcal{C}$$

Finalement, nous pouvons conclure grâce aux théorèmes 5.3.1 et 5.3.2 qu'une coupe $\mathcal{P}$-cohérente qui respecte les hypothèses 5.3.8, 5.3.9 et 5.3.10 est un état global recouvrable : toutes les réexécutions possibles depuis une telle coupe $\mathcal{P}$-cohérente sont équivalentes à une exécution passant exactement par une coupe cohérente de l'exécution de référence.

5.3.3.6 Cas idéal : pas de réception orpheline non promise

Nous étudions ici les propriétés d'un cas particulier : celui où *toutes* les réceptions orphelines dans une coupe $\mathcal{P}$-cohérente $\mathcal{C}_p$ peuvent être remplacées par une promesse, c'est-à-dire lorsque pour toutes les réceptions orphelines e, on a $\nexists e' \in s, (e', e'') \in \Gamma \wedge e \prec_i e'$ (propriété 5.3.2). Nous voulons montrer que dans ce cas, toute réduction locale de c_i correspond à une réduction globale possible de $\mathcal{C}_p$. Nous posons donc l'hypothèse suivante :

Hypothèse 5.3.11 (Toutes les réceptions orphelines sont promises) *Il n'existe pas de réception orpheline non promise dans $\mathcal{C}_p^0$:*

$$\forall (e, e') \in \Gamma, e \notin \mathcal{C}_p^0 \Rightarrow e' \notin \mathcal{C}_p^0$$

donc la propriété 5.3.2 permet d'affirmer qu'il n'y a pas d'envoi de messages qui soit conséquence d'une réception orpheline :

$$\forall (e, e') \in \Gamma, e \notin \mathcal{C}_p^0 \wedge \mathcal{P}(e') \in \mathcal{C}_p^0 \Rightarrow \nexists e'' \in \mathcal{C}_p^0, e'' \succ_i e' \wedge (e'', e_2) \in \Gamma$$

Nous pouvons alors prouver que dans ce cas, la coupe locale $c_i^{\sqcap}$ n'est plus une approximation locale de la coupe $\mathcal{C}_p^{\sqcap}$, mais est exactement $\mathcal{C}_p^{\sqcap}\big|_i$:

Propriété 5.3.9 ($c_i^{\sqcap}$ dans le cas idéal) *A tous moment de la réexécution, on a :*

$$\mathcal{C}_p^{\sqcap} = \{c_1^{\sqcap}, \ldots, c_n^{\sqcap}\}$$

Cette propriété nous permet finalement de prouver que, si toutes les réceptions orphelines sont promises, alors toute réduction locale de $\mathcal{C}_p$ correspond à une réduction globale possible de $\mathcal{C}_p$. En effet, dans ce cas, lorsque l'état local c_i est réduit, on est sûr que tout son passé *global* a déjà été exécuté puisqu'il est contenu dans $\mathcal{C}_p^{\sqcap}$. La réduction locale devient donc identique à la réduction globale :

Théorème 5.3.3 (Equivalence des réductions dans le cas idéal)

$$\mathcal{C}_p \xrightarrow{e}_L \mathcal{C}'_p \;\Rightarrow\; \mathcal{C}_p \xrightarrow{e} \mathcal{C}'_p$$

Cette étude du cas sans réception orpheline non promise est intéressante car elle montre que l'ordonnancement à restituer avec des promesses d'évènement est minimisé : il ne dépend plus que des évènements qui sont *déjà* dans la coupe de recouvrement $\mathcal{C}_p^0$, et plus des évènements exécutés durant la réexécution.

En effet, lorsque l'état local c_i est réduit, tout son passé global a déjà été exécuté donc, lorsqu'un message est reçu, tout le passé de l'envoi de ce message a déjà été exécuté. Par conséquent, tous les évènements e' possibles dans l'hypothèse d'ordonnancement promis 5.3.10 *appartiennent déjà* à la coupe de recouvrement $\mathcal{C}_p^0$. L'ordonnancement des messages à restituer avec des promesses d'évènement se réduit donc à l'ordonnancement généré par des évènements qui sont déjà dans la coupe de recouvrement $\mathcal{C}_p^0$, et en particulier celui généré par les messages en transit.

On peut aussi montrer que, dans ce cas idéal, le nombre de réceptions en avance possibles durant la réexécution est de la même manière minimisé : dès que tout l'ordonnancement promis a eu lieu, il ne peut plus y avoir de réception en avance.

En effet, nous pouvons montrer que si tout l'ordonnancement promis a eu lieu, alors la propriété de correspondance entre la réduction globale et la réduction locale depuis une coupe $\mathcal{P}$-cohérent 5.3.4 assure directement l'hypothèse de correspondance entre la réduction globale et la réduction locale depuis une coupe cohérente 5.3.7. D'abord, on a toujours $\mathcal{C}_p^{\sqcap} = \{c_1^{\sqcap}, \ldots, c_n^{\sqcap}\}$, donc $c_i^{\sqcap}$ et c_i' dans la propriété 5.3.4 sont des coupes locales appartenant à une coupe globale *cohérente* puisque $\mathcal{C}_p^{\sqcap}$ est cohérente : la propriété 5.3.4 assure donc la première partie de l'hypothèse 5.3.7 $\exists S, S',\ S\big|_i = s_i \wedge S'\big|_i = s_i' \wedge S \xrightarrow{e} S'$. De plus, comme tout l'ordonnancement promis à eu lieu, la partie gauche $Odt_{\prec}(e') \subseteq_{\mathcal{P}} c_i$ de la propriété 5.3.4 peut s'écrire $Odt_{\prec}(e') \subseteq s_i$, ce qui vérifie la deuxième partie de l'hypothèse 5.3.7 $\exists e', (e', e) \in \Gamma \Rightarrow Odt_{\prec}(e') \subseteq s_i$.

Finalement, la propriété d'équivalence entre la réduction globale et la réduction locale depuis une coupe cohérente 5.3.1 nous permet d'affirmer que dés que tout l'ordonnancement promis à eu lieu, la réduction locale devient équivalente à la réduction locale depuis une coupe cohérente. La réexécution correspond alors à une exécution possible *sans contrôle*. Par conséquent, dans le cas sans réception orpheline non promise, la réception de message en avance est impossible dès que tout l'ordonnancement promis (qui est minimal) a eu lieu.

5.3.4 Synthèse

Nous résumons dans cette section sur les trois hypothèses qui permettent d'assurer que la réduction locale d'une coupe $\mathcal{P}$-cohérente conserve les propriétés de la réduction globale. Il est difficile de synthétiser ces hypothèses sous la forme de conditions uniquement sur la coupe de recouvrement pour n'importe quel système ; en effet, deux des trois hypothèses portent sur les états locaux qui forment la coupe de recouvrement, mais *aussi* sur le système considéré. Nous montrons cependant qu'il est possible de spécifier des conditions sur la coupe de recouvrement en précisant le cadre d'application au niveau de l'ordonnancement des messages et au niveau des propriétés des réceptions non positionnables.

5.3.4.1 Hypothèse 5.3.8 : Cohérence des états locaux

Cette hypothèse porte uniquement sur les états locaux qui sont capturés pendant l'exécution de référence, c'est-à-dire sur les points de reprise qui forment la coupe de recouvrement. Ces états locaux doivent être localement cohérents, et peuvent être séparés en deux parties :

- une partie gauche : une image de l'état de l'activité au moment de la prise du point de reprise. Dans cette image, seules des réceptions orphelines peuvent être promises ;
- une partie droite : un ensemble de promesses d'évènement ordonnées entre elles. Cet ensemble de promesses d'évènement correspond à l'historique de réception des requêtes dans le cadre du protocole pour ASP proposé dans cette thèse.

On peut donc se demander si les réceptions orpheline *doivent ou ne doivent pas* être transformées en promesses d'évènement. En particulier, le formalisme s'applique encore et prouve la recouvrabilité de la coupe de recouvrement même si *aucune* réception orpheline n'est transformée en promesse. Par exemple, ce formalisme peut s'appliquer et prouver la correction des approches proposées par Schulz et al. dans [SCH 04] ou par Vaidya dans [VAI 99] dans lesquelles les réceptions orphelines restent dans la coupe de recouvrement.

Cependant, il vaut toujours mieux, si cela est possible, transformer une réception orpheline en promesse. En effet, l'attente par nécessité générée par cette promesse permet d'éviter des réceptions de message en avance, et permet aussi de rétablir une partie de l'ordonnancement des messages sans utiliser de promesse à droite de l'état local. L'utilisation de promesse pour remplacer les réceptions orphelines permet par conséquent de réduire le nombre de promesses à droite de l'état local, et donc de réduire la période pendant laquelle la réexécution est contrôlée. On peut le voir avec l'étude du cas idéal développée dans la section 5.3.3.6 : si toutes les réceptions orphelines sont remplacées par des promesses, l'ordonnancement des messages à restitué et le nombre de réceptions en avance possibles sont minimisés.

5.3.4.2 Hypothèse 5.3.10 : Ordonnancement promis

Cette hypothèse porte uniquement sur les promesses d'évènement de la partie droite de l'état local. On dit qu'un évènement e est *avant la coupe de recouvrement* si il existe un évènement e' dans la coupe qui précède causalement e. Cette hypothèse se résume alors de la façon suivante : "pour toute réception e_0 d'un message (e, e_0) qui a lieu durant la réexécution, l'ensemble $odt_e(e')$, pour tous les e' du passé causal de e qui sont avant la coupe, doit être promis dans la partie droite de l'état local". Dans cas, l'ordonnancement perdu des messages est restitué artificiellement pendant la réexécution.

De manière générale, l'évènement e dans $odt_e(e')$ correspond à des envois de message qui peuvent être placés arbitrairement loin dans le futur ; plus précisé-

ment tant que la réexécution n'a pas atteint globalement une coupe cohérente de l'exécution de référence. De plus, l'évènement e' dans $odt_e(e')$ correspond à tout évènement qui a dans son futur un évènement appartenant à la coupe de recouvrement $\mathcal{C}_p^0$. Par conséquent, l'ensemble des réceptions à promettre dans la partie droite de l'état local pour assurer l'hypothèse *Ordonnancement promis* est très difficilement définissable dans le cas général, c'est-à-dire pour n'importe quel ordonnancement de message. Une spécification *exacte* de cet ensemble impose la connaissance du modèle considéré et de sa sémantique. Cependant, il est aussi possible de restreindre les e et e' à considérer en faisant l'hypothèse suivante sur l'ordonnancement des messages :

$$\forall (e, e_0) \in \Gamma, odt_e(e') \subseteq \{e'' | (e', e'') \in \Gamma\}$$

Cette hypothèse signifie pour $odt_e(e')$:

- que les seuls évènements e' à considérer sont des envois de message et que $odt_e(e')$ est soit la réception du message envoyé par l'évènement e', soit vide. Comme seul l'ordonnancement dû aux évènements e' précédant causalement un évènement de la coupe de recouvrement est perdu (et doit être restitué par des promesses), ceci permet de ne considérer pour e' que :
  - les envois de messages en transit dans la coupe de recouvrement,
  - et les envois de messages qui précédent causalement la réception d'un message orphelin et qui ne sont pas reçus dans la coupe de recouvrement.
- et qu'il existe une approximation de $odt_e(e')$ qui ne dépend pas de e. Ceci permet de s'abstraire de e dans la définition de $odt_e(e')$.

Cette hypothèse est vérifiée par les ordonnancements de message classiques, tels que l'ordonnancement causal, le FIFO point-à-point ou encore le FIFO unvers-tous. En effet, comme on l'a vu dans la section 5.3.1, une fonction $odt_e(e')$ correcte pour l'ordonnancement causal est :

$$\forall (e, e_0) \in \Gamma, e_0 \in A_i, odt_e(e') = \{e'_0 | (e', e'_0) \in \Gamma \wedge e'_0 \in A_i\}$$

De plus, une fonction $odt_e(e')$ correcte pour l'ordonnancement FIFO point-à-point est :

$$\forall (e, e_0) \in \Gamma, e_0 \in A_i, odt_e(e') = \{e'_0 | (e', e'_0) \in \Gamma \wedge e'_0 \in A_i \wedge e \in A_j \wedge e' \in A_j\}$$

On voit tout d'abord dans ces définitions que tous les évènements e' possibles sont des réceptions. De plus, l'ordonnancement causal est une approximation correcte de l'ordonnancement FIFO point-à-point ; Charron-Bost et al. ont montré dans [CHA 96b] que l'ensemble des exécutions possibles avec l'ordonnancement causal est inclus dans l'ensemble des exécutions possibles avec les autres types d'ordonnancement classiques. En particulier, toutes les exécutions possibles avec un ordonnancement causal sont possibles avec un ordonnancement FIFO point-à-point. L'hypothèse précédente est donc bien vérifiée. Plus généralement, elle est vérifiée pour tous les ordonnancements de messages qui peuvent être approximés par l'ordonnancement causal, c'est-à-dire les ordonnancements tels que toutes les exécutions possibles avec un ordonnancement causal sont possibles avec l'ordonnancement considéré.

On peut conclure que pour les ordonnancements approximables par l'ordonnancement causal, les réceptions à promettre pour assurer l'hypothèse *Ordonnancement promis* sont forcément des réceptions de messages en transit ou des réceptions de messages envoyés avant des messages orphelins. Il est donc suffisant de promettre les réceptions de message en transit, et les réceptions de messages dont l'envoi précède causalement un message orphelin. Soit plus formellement, l'hypothèse générale d'ordonnancement promis 5.3.10 peut se restreindre à l'hypothèse sur la coupe de recouvrement suivante :

Hypothèse restreinte 5.3.1 (Ordonnancement causal promis)

$$\forall (e,e') \in \Gamma,\ \begin{pmatrix} e \in \mathcal{C}_p^0 \wedge e' \notin \mathcal{C}_p^0 \quad \vee \\ \exists (e_1,e_2) \in \Gamma, e_1 \notin \mathcal{C}_p^0 \wedge e_2 \in \mathcal{C}_p^0 \wedge e \prec e_2 \wedge e' \notin \mathcal{C}_p^0 \end{pmatrix} \Rightarrow \mathcal{P}(e') \in \mathcal{C}_p^0$$

5.3.4.3 Hypothèse 5.3.9 : Positionnement des réceptions

Cette hypothèse est constituée de deux parties distinctes. On considère deux types de réception de message. D'abord, les positionnables, qui doivent être non déterministes et pour lesquelles on peut créer des promesses. On assure donc que si ces messages arrivent en avance, on est capable de repousser leur exécution, et donc assurer l'équivalence de la réexécution. Comme on l'a vu précédemment, c'est l'hypothèse faite par les protocoles de journalisation des messages : elle se base le plus souvent sur le fait qu'il existe un tampon de réception qui temporise les réceptions en avance. Par exemple, dans le cas d'ASP, c'est la queue des requêtes en attente qui joue le rôle de ce tampon ; les réceptions de requête sont donc positionnables.

Le deuxième type de réception est les réceptions non positionnables dont tout le passé causal doit avoir été exécuté par le récepteur avant de pouvoir être elles-mêmes exécutées. Dans ce cas, même si notre formalisme permet de prendre en compte de telles réceptions, il est difficile de spécifier de manière générale les conditions qui assurent que le passé de la réception soit exécuté.

Cependant, en faisant une hypothèse sur le système, plus particulièrement sur les propriétés des réceptions non positionnables, nous pouvons restreindre l'hypothèse de positionnement des réceptions 5.3.9 à une hypothèse sur la coupe de recouvrement uniquement.

Supposons que l'ensemble des évènements qui doivent être exécutés pour qu'une réception non positionnable e' avec $(e,e') \in \Gamma$ puisse être exécutée est donné par l'ensemble $Pos(e')$ (si le message est positionnable, $Pos(e') = \emptyset$). L'hypothèse que l'on fait sur le système est que $Pos(e')$ ne peut contenir que des évènements qui précédent causalement l'envoi e, donc que $Pos(e)$ vérifie $(e,e') \in \Gamma \ \wedge \ e_0 \in Pos(e') \Rightarrow e_0 \prec e$. En particulier, les évènements qui précède la réception à cause de l'ordonnancement des messages ne peuvent pas appartenir a $Pos(e)$. En effet, nous avons vu que l'ordonnancement des messages est restitué grâce aux promesses de réception, donc des évènements qui sont virtuellement dans l'état sans avoir été exécutés. Or, par définition, une réception non positionnable ne peut avoir lieu que si son passé a été *réellement* exécuté.

Sous cette hypothèse, nous pouvons donc spécifier des conditions sur la coupe de recouvrement $\mathcal{C}_p^0$. Soit un message (e, e') dont la réception e' est non positionnable. Si (e, e') est en transit dans $\mathcal{C}_p^0$, alors $Pos(e') \subseteq \mathcal{C}_p^0$. Sinon, il ne doit pas exister de message orphelin entre les éléments de $Pos(e')$ et e. Ainsi, lorsque e est exécuté, les éléments de $Pos(e)$ le sont aussi.

Hypothèse restreinte 5.3.2 (Positionnement hors ordonnancement)

$$\forall (e,e') \in \Gamma,\ \begin{cases} e \in \mathcal{C}_p^0 \wedge e' \notin \mathcal{C}_p^0 \Rightarrow Pos(e') \subseteq \mathcal{C}_p^0 \\ e \notin \mathcal{C}_p^0 \wedge e_0 \in Pos(e') \Rightarrow \nexists (e_1,e_2) \in \Gamma \wedge e_1 \notin \mathcal{C}_p^0 \wedge e_2 \in \mathcal{C}_p^0 \wedge e_0 \preceq e_1 \prec e_2 \prec e \end{cases}$$

5.3.4.4 Bilan

Pour conclure cette section, nous récapitulons dans le tableau 5.4 les hypothèses à vérifier pour assurer la recouvrabilité d'une coupe $\mathcal{P}$-cohérente dans le cadre d'un contrôle d'exécution local. Ces hypothèses sont séparées en deux catégories, selon qu'elles portent sur le système ou sur la coupe de recouvrement. Pour pouvoir être spécifiées, ces hypothèses tiennent compte des restrictions sur l'ordonnancement de message et sur les propriétés des réceptions non positionnables proposées précédemment.

OBJECTIF	CONDITIONS SUR LE SYSTÈME	CONDITIONS SUR LA COUPE DE RECOUVREMENT
Créer une coupe de recouvrement	aucune	Hypothèse 5.3.8 : les états locaux sont cohérents et seules les réceptions orphelines sont promises dans les états locaux
Promettre des réceptions orphelines	Déterminisme par morceaux	Propriété 5.3.2 : il n'y a pas d'envoi de message conséquence d'une promesse dans la coupe
Assurer l'équivalence des duplicatas	Capable de gérer les réceptions en avance	Définition 5.2.2 : la coupe de recouvrement est $\mathcal{P}$-cohérente.
Respecter l'ordonnancement des messages	Capable de gérer les réceptions en avance *et* ordonnancement approximable par l'ordonnancement causal	Hypothèse restreinte 5.3.1 : les réceptions en transit et de message dont l'envoi précède une réception orpheline non promise sont promises
Gérer les réceptions *positionnables* en avance	L'exécution de la réception peut-être repoussée sans contrainte	aucune
Gérer les réceptions *non positionnables* en avance	Pour tout message (e, e'), on peut définir $Pos(e')$ et $Pos(e')$ ne contient que des évènements qui précédent causalement e	Hypothèse restreinte 5.3.2 : pas de message orphelin non promis entre les évènements dans $Pos(e')$ et l'envoi e. Si (e, e') en transit, $Pos(e')$ est inclus dans la coupe

FIG. 5.4 – Bilan des hypothèses sur le système et sur la coupe de recouvrement

Nous résumons enfin le rôle des promesses d'évènement dans la coupe de recouvrement : nous distinguons trois types de promesse :

- les promesses qui permettent d'assurer l'équivalence des duplicatas, et qui sont spécifiées par la définition de la $\mathcal{P}$-cohérence,
- les promesses qui permettent d'assurer le respect de l'ordonnancement de messages, et qui sont spécifiées par l'hypothèse d'ordonnancement promis 5.3.10,
- et enfin les promesses qui remplacent les réceptions orphelines, et qui permettent de recréer la synchronisation perdue par les messages orphelins.

Notons que l'intersection de ces trois ensembles n'est pas vide : certaines promesses sont spécifiées à la fois par la définition de la $\mathcal{P}$-cohérence et par l'hypothèse d'ordonnancement promis 5.3.10.

5.4 Preuve de correction du protocole pour ASP

Nous prouvons finalement la correction du protocole proposé dans le chapitre 4 : nous montrons tout d'abord que le contrôle sur l'exécution nécessaire est réalisable dans le modèle ASP, en particulier que les promesses peuvent être créées et que l'attente par nécessité peut être assurée. Enfin, nous montrons que les coupes de recouvrement créées durant l'exécution respectent les conditions imposées par la $\mathcal{P}$-cohérence et par les trois hypothèses de la réduction locale.

5.4.1 Conditions sur le modèle ASP

5.4.1.1 Création de promesses

Dans le protocole proposé dans le chapitre 4, on ne crée des promesses de réception que pour les requêtes (section 4.1.2). Une réception de requête est positionnable grâce à la queue de réception des requêtes, qui joue le rôle de tampon stockant les messages en avance.

De plus, l'exécution des activités est déterministe par morceaux : on peut identifier de manière unique les réceptions de requête et donc avoir une relation unique $\lhd$ entre une promesse de réception et la réception promise.

Enfin, nous avons vu que les promesses sont toujours créées pour des requêtes qui ne sont pas encore servies, que ce soit pour des requêtes orphelines (section 4.1.2) ou des requêtes dans l'historique de réception (section 4.1.4). Ces requêtes se trouvent donc dans la queue des requêtes en attente. Par conséquent :

- on peut toujours remplacer une requête par la promesse correspondante dans l'état de l'activité,
- il n'y a jamais d'envoi de message qui soit la conséquence d'une promesse de réception d'une requête orpheline ; la propriété 5.3.2 est toujours assurée.

Donc, dans le cadre de notre protocole pour ASP proposé dans le chapitre 4, il ne peut pas y avoir de réception de requête orpheline non promise.

5.4.1.2 Synchronisation locale

L'objectif ici est de montrer que l'exécution d'une activité suit la réduction locale d'une coupe $\mathcal{P}$-cohérente (définition 5.3.2). Nous rappelons tout d'abord que $c_i^{\sqcap}$ est une approximation locale du plus grand état global cohérent inclus dans la coupe de recouvrement. Nous avons vu dans la section 5.3.3 que la réduction locale est correcte si elle ne réduit que des évènements qui pourraient être réduits sur $c_i^{\sqcap}$. En d'autres termes, la réduction doit correspondre à une exécution locale *possible* depuis l'état caractérisé par $c_i^{\sqcap}$.

Les coupes locales c_i et $c_i^{\sqcap}$ ne différent que par des évènements qui ont dans leur passé soit des promesses d'évènement soit des évènements qui n'appartiennent pas à c_i. Dans le protocole, on peut montrer que c_i et $c_i^{\sqcap}$ ne différent que par des réceptions de requête. Montrons cette propriété par récurrence :

- elle est vraie dans la coupe de recouvrement car les requêtes promises ne sont jamais servies dans l'état de l'activité,
- elle est trivialement conservée par une réception de requête,
- elle est conservée par une réception de réponse car les réponses ne sont jamais reçues en avance (montré dans la section suivante) et appartiennent donc immédiatement à $c_i^{\sqcap}$,
- elle est conservée par le service de requête. En effet, les requêtes déjà servies ne sont pas promises. De plus, le service des requêtes étant FIFO, c'est la première (minimale au sens causal) requête qui est servie. Si cette requête est promise, l'exécution est bloquée en attente de la réception de la requête correspondante, sinon elle appartient à $c_i^{\sqcap}$ et est servie ; le service appartient donc à $c_i^{\sqcap}$.
- elle est conservée par l'exécution des évènements internes, qui sont forcément conséquence d'un service par la définition $\prec^{ASP}$. En effet, le service d'une requête appartient forcément à $c_i^{\sqcap}$, donc ces évènements internes appartiennent aussi à $c_i^{\sqcap}$.

On peut donc conclure que l'exécution locale respecte toujours la réduction locale d'une coupe $\mathcal{P}$-cohérente.

5.4.1.3 Réceptions non positionnables

Dans le modèle ASP, la réception d'une réponse n'est pas positionnable ; elle doit donc toujours être exécutée après son passé causal. En effet, on a vu qu'une réponse doit être reçue après la création du futur, donc après l'envoi de la requête correspondante. La définition de $\prec^{ASP}$ nous indique que le seul passé causal de la réception d'une réponse sur le récepteur est le passé causal de son envoi. En particulier, la réception d'une réponse n'est pas ordonnée par rapport aux autres réceptions de message. Donc, dans le cadre d'ASP, l'hypothèse générale de positionnement des réceptions 5.3.9 se réduit à l'hypothèse contrainte 5.3.2 sur la coupe de recouvrement. L'ensemble $Pos(e)$ où e est une réception de réponse ne contient que l'envoi de la requête correspondante à cette réponse.

5.4.1.4 Ordonnancement des messages

Enfin, comme on l'a vu dans la section 3.1.2, les réceptions de requêtes respectent l'ordonnancement causal. On peut donc réduire l'hypothèse générale d'ordonnancement promis 5.3.10 à l'hypothèse contrainte 5.3.1 sur la coupe de recouvrement.

5.4.2 Conditions sur la coupe de recouvrement

5.4.2.1 $\mathcal{P}$-cohérence

Nous montrons tout d'abord que les coupes de recouvrement créées durant l'exécution sont des $\mathcal{P}$-coupes (définition 5.2.2). Pour cela il suffit de montrer que :

$$\forall e, e',\ (e \in_{\mathcal{P}} c_i), \begin{cases} e' \prec_i e \wedge Det(e) \Rightarrow e' \in_{\mathcal{P}} c_i & \text{(a)} \\ e' \prec_i e \wedge \neg Det(e) \Rightarrow e' \in_{\mathcal{P}} c_i \vee Det(e') & \text{(b)} \end{cases}$$

Nous avons montré que les seuls évènements qui diffèrent dans c_i et $c_i^{\sqcap}$ sont des réceptions de requêtes, donc des évènements non déterministes. Donc, si $e' \prec_i e \wedge Det(e)$, alors $e \in c_i^{\sqcap}$ et $e' \in c_i$, ce qui vérifie le cas (a). Si on a $\neg Det(e)$, alors e est une réception de requête. Si e' est déterministe, alors (b) est vérifié automatiquement. Si e' est non déterministe, alors e' est une réception de requête et, comme la clôture de l'historique est cohérente (section 4.1.4), on a $e' \prec_i e \Rightarrow e' \in_{\mathcal{P}} c_i$, ce qui vérifie (b).

Ensuite, nous montrons la $\mathcal{P}$-cohérence de la coupe de recouvrement :

$$\forall e, e',\ (e \in_{\mathcal{P}} \mathcal{C}_p), e' \prec e \Rightarrow \big(e' \in_{\mathcal{P}} \mathcal{C}_p \vee Det(e')\big)$$

On applique le même raisonnement que dans le cas (b) précédent, en considérant la cohérence de l'historique de façon *globale*. Si e' est déterministe, alors la relation est vérifiée automatiquement. Si e' est non déterministe, alors e' est une réception de requête et, comme la clôture de l'historique est cohérente, on a $e' \prec_i e \Rightarrow e' \in_{\mathcal{P}} c_i$, ce qui vérifie la relation.

On peut donc conclure que les coupes de recouvrement formées par le protocole sont $\mathcal{P}$-cohérentes.

5.4.2.2 Hypothèse de cohérence des états locaux 5.3.8

La vérification de cette hypothèse est triviale. Elle découle directement de la construction des coupes de recouvrement. D'abord, elles sont constituées d'un ensemble de point de reprise représentant forcément un état local *cohérent*. De plus, seules les réceptions de requêtes orphelines sont remplacées par des promesses dans cet état local.

5.4.2.3 Hypothèse restreinte d'ordonnancement causal des réceptions 5.3.1

Étant dans le cadre d'un ordonnancement causal, l'hypothèse générale 5.3.10 est restreinte à l'hypothèse 5.3.1. Les coupes de recouvrement doivent vérifier

que les réceptions de message en transit et de message précédant causalement une réception de requête orpheline non promise sont promises. Soit :

$$\forall (e,e') \in \Gamma,\ \begin{pmatrix} e \in \mathcal{C}_p^0 \wedge e' \notin \mathcal{C}_p^0 \quad \vee & \text{(a)} \\ \exists (e_1,e_2) \in \Gamma, e_1 \notin \mathcal{C}_p^0 \wedge e_2 \in \mathcal{C}_p^0 \wedge e \prec e_2 \wedge e' \notin \mathcal{C}_p^0 & \text{(b)} \end{pmatrix} \Rightarrow \mathcal{P}(e') \in \mathcal{C}_p^0$$

Le protocole assure que les messages en transit sont promis. Donc dans le cas (a), on a bien $\mathcal{P}(e') \in \mathcal{C}_p^0$.

Si l'envoi d'un message (e,e') précède causalement la réception d'un message orphelin (e_1,e_2), alors la réception e' a lieu *avant* la réception e_2 (propriété *strong common future* assurée par le rendez-vous sur les communications, section 4.1.4). Or $e_2 \in_{\mathcal{P}} \mathcal{C}_p^0$ (message orphelin), l'historique est clos après e_2 donc après e'. Par conséquent, on a $e' \in_{\mathcal{P}} \mathcal{C}_p^0$, ce qui vérifie (b).

5.4.2.4 Hypothèse restreinte de positionnement hors ordonnancement 5.3.2

Les réceptions de réponses sont non positionnable, mais ne sont pas ordonnées entre elles : nous sommes dans le cadre restreint de positionnement hors ordonnancement. Il faut donc assurer que

$$\forall (e,e') \in \Gamma,\ \begin{cases} e \in \mathcal{C}_p^0 \wedge e' \notin \mathcal{C}_p^0 \Rightarrow Pos(e') \subseteq \mathcal{C}_p^0 \\ e \notin \mathcal{C}_p^0 \wedge e_0 \in Pos(e') \Rightarrow \nexists (e_1,e_2) \in \Gamma \wedge e_1 \notin \mathcal{C}_p^0 \wedge e_2 \in \mathcal{C}_p^0 \wedge e_0 \preceq e_1 \prec e_2 \prec e \end{cases}$$

Soit une réponse (e,e'), et sa requête correspondante (e_0,e'_0). La réception non positionnable e' est forcément une réception de réponse. De plus, $Pos(e')$ est l'envoi de la requête correspondante à la réponse : $Pos(e') = \{e_0\}$.

La relation de causalité potentielle $\prec^{ASP}$ nous permet d'affirmer que l'envoi de la réponse est une conséquence causale *locale* de la réception de la requête : $e'_0 \prec_i^{ASP} e$.

Si la réponse est en transit, alors la réception e'_0 est dans la coupe de recouvrement puisqu'elle a eu lieu sur l'activité qui envoie la réponse (Hypothèse 5.3.8 : les états sont localement cohérents). Or il n'y a pas de requête orpheline, donc l'envoi de la requête appartient aussi à la coupe de recouvrement : $Pos(e') = \{e_0\} \subseteq \mathcal{C}_p^0$, ce qui vérifie le cas (a).

Pour les réponses qui ne sont pas en transit, on peut appliquer un raisonnement similaire. Le seul message (e_1,e_2) qui vérifie $e_0 \preceq e_1 \prec e_2 \prec e$ est (e_0,e'_0) car $e'_0 \prec_i^{ASP} e$. Or, comme on l'a vu dans la section 5.4.1, (e_0,e'_0) ne peut pas être orphelin non promis puisque c'est une requête. Donc $e_1 \notin \mathcal{C}_p^0 \wedge e_2 \in \mathcal{C}_p^0$ ne peut pas être vérifié, ce qui vérifie le cas (b).

5.4.3 Résumé

Nous avons montré que l'exécution d'une activité en ASP avec une attente par nécessité sur le service d'une requête qui est promise respecte bien la réduction locale d'une coupe $\mathcal{P}$-cohérente. De plus, nous avons montré que les coupes de recouvrement respectent les conditions imposées par la définition de la $\mathcal{P}$-cohérence et par les trois hypothèses qui permettent de restreindre la synchronisation à une

synchronisation uniquement locale. Nous pouvons donc conclure que les états globaux formés par notre protocole dans le cadre d'ASP sont des états globaux recouvrables.

5.5 Conclusion

Nous avons présenté dans ce chapitre la formalisation des concepts introduits par le protocole de tolérance aux pannes par points de reprise proposé pour le modèle ASP dans le chapitre 4. Cette formalisation a permis de définir la cohérence promise, ou $\mathcal{P}$-cohérence, une condition de recouvrabilité sur des états globaux basée sur l'utilisation de promesses d'évènement.

Cette formalisation a d'abord pour objectif de prouver de façon formelle la correction du protocole pour ASP, c'est-à-dire de prouver que les états globaux utilisés pour recouvrir l'application après une panne sont recouvrables. Elle est aussi une généralisation du protocole défini pour ASP : nous avons en effet généralisé les propriétés du système considéré de manière à pouvoir spécifier la $\mathcal{P}$-cohérence d'un état global dans n'importe quel système, quelle que soit la relation de causalité découlant de la sémantique de ce système, ou encore quel que soit l'ordonnancement des messages.

La formalisation et les preuves nécessaires ont été faites en deux étapes. Nous avons tout d'abord défini la $\mathcal{P}$-cohérence, ainsi que la réduction globale d'une coupe $\mathcal{P}$-cohérente. Ces deux définitions se basent sur l'existence de promesses d'évènement capable de synchroniser *globalement* l'exécution distribuée : on suppose qu'une promesse d'évènement est capable de bloquer l'exécution de ses conséquences causales sur n'importe quelle activité du système. Dans ce cadre idéal, nous avons prouver formellement la recouvrabilité d'une coupe $\mathcal{P}$-cohérente.

La seconde étape a consisté à se ramener à un cadre d'utilisation réaliste, et en particulier à considérer que les promesses d'évènement ne peuvent synchroniser que l'exécution *locale*, c'est-à-dire l'exécution de l'activité qui doit exécuter l'évènement promis. Nous nous reposons alors sur un mécanisme d'attente par nécessité qui évite l'exécution des évènements locaux qui sont des conséquences causales d'une promesse. Nous avons donc défini la réduction locale d'une coupe $\mathcal{P}$-cohérente, qui ne repose plus que sur un contrôle de l'exécution locale. Enfin, nous avons isolé les trois hypothèses à faire sur la coupe $\mathcal{P}$-cohérente et sur le système pour que cette réduction locale soit équivalente à la réduction globale définie précédemment : nous prouvons ainsi la recouvrabilité d'une coupe $\mathcal{P}$-cohérente dans ce cadre d'un contrôle local réaliste de l'exécution.

Comme ces trois hypothèses portent sur la coupe $\mathcal{P}$-cohérente considérée *et* sur le système, il n'a pas été possible de synthétiser ces hypothèses sous forme de conditions sur la coupe de recouvrement pour *n'importe quel* système. Cependant, en restreignant l'ordonnancement à un ordonnancement approximable par l'ordonnancement causal, et en restreignant les liens de causalités des réceptions non positionnables avec les autres évènements, nous avons pu définir des condi-

tions précises sur la coupe de recouvrement pour les systèmes respectant ces restrictions. On notera que ces restrictions ne sont posées que pour permettre une spécification des conditions sur la coupe de recouvrement, mais ne sont en aucun cas nécessaires à la correction des propriétés des coupes $\mathcal{P}$-cohérentes, et donc des théorèmes démontrant la recouvrabilité de ces coupes.

Enfin, nous avons appliqué les résultats obtenus dans ce chapitre au modèle ASP et à notre protocole. Nous avons montré en particulier que les hypothèses nécessaires à la recouvrabilité d'une coupe $\mathcal{P}$-cohérente dans le cadre d'un contrôle de l'exécution locale sont vérifiées. Nous avons ainsi prouvé la correction de notre protocole.

Chapitre 6

Extension pour la grille

Le modèle ASP, par son modèle de service de méthode distante et asynchrone, et son implémentation ProActive par sa gestion complète de l'hétérogénéité, font de cet intergiciel une solution efficace dans le cadre des systèmes à large échelle tels que les grilles de calcul. Nous avons voulu proposer une solution pour adapter notre mécanisme de tolérance aux pannes à ce contexte particulier.

Nous proposons donc dans cette section une extension du protocole décrit précédemment adaptée au contexte des grilles de calcul. Cette extension se base sur la notion de groupes de recouvrement : le protocole du chapitre 4 est appliqué dans chacun des groupes d'activités de manière indépendante, et les communications entre les groupes sont soumises à un protocole additionnel de journalisation pessimiste par le récepteur. Ainsi, lorsqu'un groupe doit reprendre suite à une panne d'une de ses activités, tous les messages envoyés avant la panne par d'autres groupes peuvent être renvoyés durant la réexécution sans faire reprendre ces autres groupes.

6.1 Grille de calcul et tolérance aux pannes

Nous présentons dans un premier temps les spécificités des grilles de calcul, ainsi qu'une approche possible pour la tolérance aux pannes par recouvrement arrière dans ce contexte : les groupes de recouvrement.

6.1.1 Grilles de calcul

Le concept de grille de calcul a été introduit récemment par Foster et al. dans [FOS 99]. Une grille de calcul est une agrégation de ressources indépendantes, le plus souvent des grappes de machines dédiées (*clusters*), qui sont reliées entre elles par des liens non dédiés. Ce type de système introduit de nouvelles problématiques, avec en particulier :

- **L'hétérogénéité** - Elle se retrouve aussi bien au niveau des architectures des ressources disponibles qu'au niveau de la qualité des liens réseaux qui relient les ressources entre elles. Les architectures physiques sont variées, et difficilement prévisibles au moment du déploiement d'une application. De plus, l'hétérogénéité des ressources implique une différence de qualité

des ressources, donc la probabilité d'occurrence d'une panne peut fortement varier d'une grappe à l'autre.
Les liens réseaux qui relient les grappes entre elles sont souvent des liens non dédiés, et donc des liens à forte latence et à faible débit. Au contraire, les liens réseaux au coeur d'une grappe sont des liens rapides ; l'hétérogénéité du réseau est donc très forte.

- **le nombre de nœuds** - L'objectif des grilles de calcul est de réunir des groupes de machines pour atteindre un nombre de ressources disponibles plus grand que la taille de chaque groupe. Le nombre de ressources à considérer augmente donc d'un ordre de grandeur.
- **la volatilité** - L'augmentation de la volatilité des ressources est une conséquence directe de l'augmentation du nombre de nœuds. La probabilité globale qu'un des éléments du système tombe en panne est d'autant plus grande que le nombre d'éléments est grand.
- **la taille des données** - Les grilles de calcul permettent de traiter des données de taille jusqu'alors très difficilement manipulable par une application déployée sur une seule grappe de machine.
- **la structure des applications** - Pour exploiter efficacement une grille de calcul, les applications doivent tenir compte de la structure de la grille. En particulier, la faiblesse des liens entre les différentes grappes doit être prise en compte ; on va donc déployer les éléments de l'application de façon à minimiser le nombre de communications inter-grappes.

Le support de la tolérance aux pannes a été identifié dans [FOS 99] comme un défi majeur pour le déploiement des grilles de calcul. Nous décrivons ici une approche possible pour relever ce défi.

6.1.2 Une solution possible : les groupes de recouvrement

Elnozahy et al. présentent dans [ELN 04] une étude intéressante sur l'adaptation des protocoles de tolérance aux pannes par recouvrement arrière aux systèmes à très large échelle. Ils mettent en particulier en avant le fait que l'accès à la mémoire stable devient un point crucial dans les performances des protocoles de tolérance aux pannes par recouvrement arrière pour les systèmes à large échelle. De plus, en terme de temps de recouvrement, l'augmentation du nombre de ressources rend les approches basées sur une reprise globale de toutes les activités difficilement utilisables dans un tel contexte.

Les auteurs concluent leur étude en proposant en perspective différentes solutions qui semblent s'adapter à ce nouveau contexte. En particulier, ils proposent l'utilisation de *groupes de recouvrement* pour confiner les effets d'une panne à un sous-ensemble uniquement du système. La division en groupes de recouvrement peut aussi, selon les interactions nécessaires entre les groupes, être une solution au problème de l'accès à la mémoire stable en rendant indépendantes les différentes mémoires stables utilisées par chacun des groupes. La contention d'accès sur chaque mémoire stable devient alors une fonction du nombre d'éléments dans le groupe et non plus de la taille totale du système.

6.1.2.1 Groupes de recouvrement

La notion de groupe de recouvrement a été introduite par Sistla et al. dans [SIS 89], et développée par Vaidya dans [VAI 93]. L'objectif est de confiner l'effet d'une panne au sein d'un groupe de processus, et donc d'éviter que cette panne n'influe sur tout le système. Pour cela, on tire parti des propriétés des solutions de tolérance aux pannes par journalisation des messages. En effet, la journalisation permet d'isoler les effets d'une panne à la seule activité concernée ; seule l'activité tombée en panne doit recouvrir depuis son dernier point de reprise. On dit que l'activité est l'unité de recouvrement (*Recovery Units, RUs*) [STR 85].

Vaidya introduit par opposition aux RUs les unités de recouvrement distribuées (*Distributed Recovery Units, DRUs*) ; on ne considère plus l'activité comme unité de recouvrement, mais un *groupe* d'activités. Dans chaque groupe, un protocole de tolérance aux pannes par point de reprise est appliqué, et les communications entre les groupes sont soumises à un protocole de journalisation. Des passerelles de communication sont intercalées entre les groupes, et journalisent les messages inter-groupes. Cette journalisation peut être pessimiste comme dans le cas de [SIS 89] ou de [BOU 03b], ou optimiste [A.L 91]. Une approche optimiste impose un protocole de synchronisation entre les passerelles, alors qu'une approche pessimiste rend complètement indépendant chacun des groupes. Cette journalisation entre les groupes permet de confiner les effets d'une panne au groupe : comme dans le cas d'une RU, seul le groupe concerné doit reprendre depuis la dernière ligne de recouvrement. De plus, elle permet d'isoler les groupes du point de vue du protocole utilisé : Conan dans [CON 96] utilise des groupes de recouvrement pour appliquer différents protocoles de tolérance aux pannes sur différentes parties de l'application.

6.1.2.2 Validation d'état global

Lorsqu'un groupe reprend suite à une panne, l'état global du système est temporairement incohérent ; le groupe qui a repris doit rattraper son état d'avant la panne pour que l'état global redevienne cohérent. En particulier, le contenu des duplicatas de message inter-groupes qui sont réémis durant la réexécution du groupe doit être équivalent à celui des messages originaux pour assurer la cohérence avec les états des activités des autres groupes.

L'hypothèse de déterminisme par morceaux suffit à assurer que la réexécution d'une RU est équivalente à l'exécution de référence si les messages journalisés sont traités dans le même ordre que durant l'exécution de référence. Ce n'est plus vrai dans le cas d'un *groupe* d'activités : un groupe de recouvrement ne peut pas être considéré comme une unité d'exécution déterministe par morceaux. Considérons l'état global d'un groupe de recouvrement ; les réceptions de messages au sein de ce groupe sont des évènements internes non déterministes. Par conséquent, il n'est plus suffisant en cas de réexécution d'assurer l'ordre des réceptions de messages venant de l'extérieur du groupe pour assurer l'équivalence de la réexécution.

Ce problème est similaire au problème connu sous le nom de *communication avec le monde extérieur* ; les éléments du monde extérieur, comme par exemple un

périphérique d'affichage, ne peuvent pas être sous le contrôle du protocole de tolérance aux pannes. Par conséquent, lors d'une communication avec cet élément, le protocole de tolérance aux pannes doit pouvoir garantir qu'en cas de réexécution, le système retrouvera exactement le même état qu'il avait atteint au moment de la communication avec l'élément extérieur. Dans ce cas, la cohérence entre le système et le monde extérieur est conservée.

La validation d'état global (*output-commit*) est une solution au problème de communication avec le monde extérieur : avant une telle communication, le protocole sauvegarde l'état global du système, ou suffisamment d'informations pour assurer que le système retrouvera cet état en cas de réexécution. La validation d'état global est donc aussi une solution dans le cadre des groupes de recouvrement : un envoi de message vers un groupe distant est considéré comme un envoi vers le monde extérieur, et doit donc déclencher une validation d'état global.

Il existe dans la littérature différentes solutions qui permettent la validation d'état global avant l'envoi d'un message au monde extérieur, ou dans notre cas un message inter-groupe. Dans [SIS 89] et [JOH 88], un envoi de message inter-groupe déclenche la création d'une ligne de recouvrement. Avec cette approche, la latence d'envoi d'un message inter-groupe est très grande ; il faut que l'émetteur attende que la ligne de recouvrement soit créée et stockée en mémoire stable. Pour améliorer cette latence d'envoi, Johnson utilise dans [JOH 93] des vecteurs de dépendances entre les activités pour éviter une validation d'état *global* ; seules les activités causalement liées à l'envoi du message inter-groupe doivent valider leur état. Par ailleurs, la validation des états de ces activités peut se faire par point de reprise ou par journalisation optimiste des messages.

Les protocoles par journalisation des messages proposent de manière générale une latence d'envoi plus faible que les approches qui reposent sur des points de reprise, car elles ne nécessitent généralement pas de coordination préalable avec d'autres activités dans le groupe. En particulier dans le cas de la journalisation pessimiste, la validation d'état n'est même pas nécessaire puisque tous les évènements non déterministes, donc en particulier ceux qui précèdent l'envoi d'un message inter-groupe, sont systématiquement journalisés en mémoire stable.

Un exemple intéressant est le protocole par journalisation causale Manetho [ELN 92b]. La solution proposée dans Manetho réduit fortement la latence d'envoi de message inter-groupe. En effet, ce protocole repose sur le fait que chaque message applicatif porte avec lui son histoire causale sous la forme d'un graphe d'antécédence d'évènements. Ce graphe d'antécédence est utilisé en cas de réexécution du système pour forcer l'équivalence avec l'exécution de référence. Donc lorsqu'un message inter-groupe est émis, il suffit de stocker en mémoire stable le graphe d'antécédence qui lui est associé.

6.1.2.3 Adéquation avec la grille

La notion de groupe est intrinsèque aux grilles de calcul ; l'idée est de considérer chaque grappe d'une grille comme un groupe de recouvrement. Nous montrons

ici que cette solution, par son approche de type “diviser pour régner”, est particulièrement adaptée. Reprenons les problématiques introduites par les grilles de calcul (section 6.1.1) :

- **L'hétérogénéité** - La différence de qualité des ressources d'une grappe à l'autre peut être compensée par un protocole adapté (ou paramétré de façon adaptée) dans chaque groupe.
- **le nombre de nœuds** - Les protocoles qui sont appliqués dans les groupes ne s'appliquent que sur les membres du groupe. L'ordre de grandeur du nombre de nœuds à considérer n'est plus le nombre total de nœuds mais le nombre de nœuds par groupe.
- **la volatilité** - Les effets des pannes sont confinés au seul groupe concerné. La reprise suite à une panne est donc plus rapide puisqu'elle ne concerne que les membres du groupe.
- **la taille des données** - la taille totale de données à stocker en mémoire stable est répartie sur les différentes mémoires stables *indépendantes*. Il n'est plus nécessaire de disposer d'une mémoire stable unique capable de stocker la totalité des données manipulées par l'application.
- **la structure des applications** - le surcoût principal lié à l'utilisation de groupes de recouvrement provient de la nécessité de valider l'état global avant l'envoi d'un message inter-groupe et de la journalisation de ce message. Le surcoût principal est donc dû aux communications entre les grappes qui sont supposées être minimisées par la structure de l'application.

Deux aspects plus spécifiques à la tolérance aux pannes doivent cependant tenir compte des particularités d'une grille de calcul : la gestion des ressources et la localisation. Nous montrons en quoi les solutions proposées dans la section 4.2.2 doivent être adaptées.

D'abord, le fournisseur de ressources doit tenir compte de l'infrastructure physique ; une activité doit autant que possible redémarrer dans le même groupe de ressources physiques, par exemple la même grappe de machines. Dans le cas contraire, la structure d'origine de l'application n'est plus conservée ; deux activités communiquant de manière intensive peuvent alors être séparées, et devoir communiquer à travers un lien réseau lent. Pour conserver la structure de l'application, il faut que les ressources soient gérées par groupe ; en pratique, nous allons déployer un fournisseur de ressources tel que celui proposé dans la section 4.2.2 par groupe de ressources physiques. Chaque fournisseur de ressources ne gère que des ressources physiques proches, c'est-à-dire reliées par un lien réseau rapide.

Le problème de la localisation d'une activité redémarrée se complexifie dans le cas de groupes de recouvrement indépendants. En effet, si l'on suppose que les activités de groupes différents ne partagent pas de mémoire stable commune, la solution proposée dans la section 4.2.2 n'est plus applicable. En effet, cette solution suppose que toutes les activités puissent accéder à la table de localisation qui se trouve sur la mémoire stable. Il faut donc trouver une solution distribuée

qui ne suppose plus de point central ; nous proposons une telle solution dans la section 6.2.2.

La solution des groupes de recouvrement dans le contexte des grilles de calcul est utilisée par le protocole de tolérance aux pannes pour les grilles de calcul MPICH-V3 [BOU 03b]. Ce protocole se base sur l'utilisation de mémoires de canal, introduite dans le protocole MPICH-V1 [BOS 02], et d'un journal d'évènements stable (*event logger*) introduit dans [BOU 03a]. Les groupes de recouvrement sont séparés par les mémoires de canal qui journalisent de façon pessimiste les messages inter-groupe. Dans les groupes de recouvrement, les protocoles MPICH-V1 ou MPICH-V2 peuvent être indifféremment utilisés.

Cette solution ne nécessite pas de mécanisme de validation d'état global, grâce à l'utilisation d'un journal d'évènements stable. Lorsqu'un message (intra-groupe ou inter-groupe) est reçu par une activité, cette activité journalise de manière asynchrone le déterminant de cette réception sur un *event logger*. Par conséquent, toute l'histoire causale d'un groupe pouvant être récupérée sur un *event logger*, il n'est pas nécessaire de déclencher un mécanisme particulier au moment de l'envoi d'un message inter-groupe, contrairement à une approche comme Manetho.

Le protocole MPICH-V3 utilise un *Master Dispatcher*, un serveur stable supposé accessible depuis tous les groupes de recouvrement. L'avantage d'un tel serveur central est de pouvoir gérer la panne d'un groupe entier : ce *Master Dispatcher* peut alors redémarrer l'ensemble du groupe sur d'autres ressources. De plus, le problème de détection de panne et de localisation d'une activité redémarrée après une panne est simplifié : le *Master Dispatcher* connaît l'état et la localisation de toutes les activités à tout instant.

Nous proposons ici une solution similaire à MPICH-V3 appliquée à notre contexte. Les activités sont regroupées suivant la répartition géographique des grappes de machines : un groupe d'activités correspond à un groupe physique, c'est-à-dire une grappe de machines. Le protocole proposé dans le chapitre 4, étendu pour proposer un mécanisme de validation d'état global, peut être appliqué dans chaque groupe de manière indépendante. En particulier, nous retenons la solution de journalisation pessimiste des messages inter-groupes : cette approche pessimiste permet de rendre les mémoires stables utilisées par les différents groupes *totalement indépendantes*. La mémoire stable n'est donc pas supposée accessible depuis l'extérieur du groupe. Cette indépendance permet par exemple d'utiliser comme mémoire stable le système de fichier partagé comme NFS dans chaque grappe de machines. En effet, si ce système de fichier peut être utilisé par les activités s'exécutant dans la grappe de machines, les activités s'exécutant sur une autre grappe de machines ne peuvent le plus souvent pas y accéder.

A la différence de MPICH-V3, nous n'allons pas supposer l'existence d'un serveur stable central, le *Master Dispatcher*. Nous pensons en effet que l'existence d'un tel *processus* stable qui peut contacter toutes les activités du système est une hypothèse forte dans un contexte de grille de calcul. Nous allons donc proposer une solution distribuée au problème de localisation des activités entre les groupes. Par contre, la solution proposée ici ne gère pas la panne d'une grappe de machines complète puisqu'il n'y a pas de processus central capable de détecter

cette panne globale.

6.2 Groupes de recouvrement dans ProActive

Nous décrivons dans cette section comment nous avons ajouté à l'implémentation du protocole proposé dans le chapitre 4 la possibilité de déployer des groupes de recouvrement indépendants, en particulier avec des mémoires stables différentes qui ne sont pas supposées accessibles entre les groupes.

Nous avons donc ajouté à l'implémentation du mécanisme de tolérance aux pannes existant :

- un système de gestion des groupes : une activité doit savoir si le message qu'elle reçoit ou qu'elle émet est inter-groupe ou intra-groupe ;
- un système de localisation entre les groupes : si une activité tombe en panne et est redémarrée sur une autre ressource, les activités des autres groupes doivent pouvoir connaître sa nouvelle localisation ;
- un mécanisme de validation d'état global et de journalisation de message ; les messages inter-groupes déclenchent une validation d'état global du côté de l'émetteur, et sont journalisés de façon pessimiste sur la mémoire stable du récepteur.

La figure 6.1 résume ce mécanisme. Les messages inter-groupes en rouge sont journalisés de façon pessimiste par l'activité réceptrice sur la mémoire stable de son groupe. Le protocole proposé dans le chapitre 4 est appliqué entre les activités d'un même groupe, à travers les communications intra-groupes en noir (on le nommera par la suite le *protocole interne*). Ces activités ne stockent leurs points de reprise que sur la mémoire stable de leur groupe.

6.2.1 Gestion des groupes

Le notion de groupe dans notre cas est très fortement reliée à la notion de mémoire stable. En effet, pour pouvoir appliquer le protocole interne au sein de chaque groupe, il faut que les activités partagent une mémoire stable commune. C'est donc l'utilisation de la même mémoire stable, et dans les expérimentations du même serveur de tolérance aux pannes, qui crée la notion de groupe. Chaque mémoire stable doit pouvoir être identifiée de manière unique, et cet identifiant devient l'identifiant du groupe lié à cette mémoire stable. L'utilisation de protocole de gestion de groupe *dynamique* tels que les approches par "bavardage" (*gossip-based*) comme [GAN 03] n'est pas requise dans notre cas, puisque dans le cadre de la grille, les groupes sont utilisés pour représenter l'infrastructure physique sous-jacente qui est fixe.

Chaque activité doit donc au démarrage accéder à la mémoire stable, et récupérer cet identifiant de groupe, noté Grp_i. Cet identifiant est estampillé sur tous les messages envoyés par une activité. Ainsi, une activité peut déterminer si un message reçu a été envoyé par une activité du même groupe ou non.

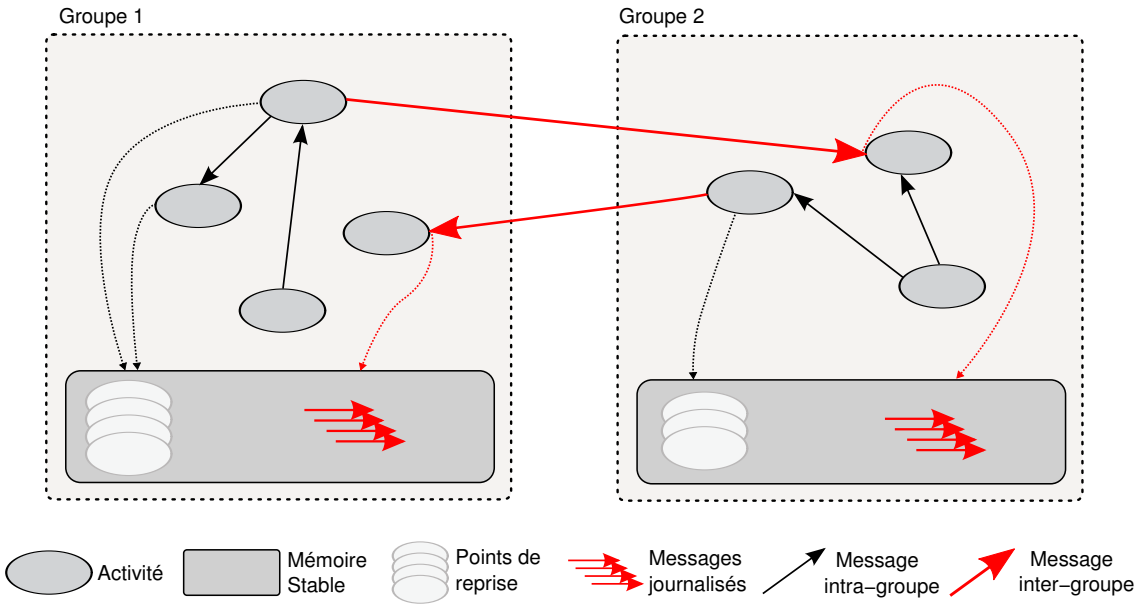

FIG. 6.1 – Groupes de recouvrement

Une activité émettrice doit aussi savoir *avant d'envoyer un message* si ce message est inter-groupe ou intra-groupe, de manière à pouvoir déclencher une validation d'état si nécessaire. Pour cela, nous maintenons sur la mémoire stable une table d'appartenance au groupe, qui est donc partagée par toutes les activités d'un même groupe. En pratique, cette table correspond à la table de localisation utilisée pour localiser les objets redémarrés au sein du groupe (section 4.2.2.3).

Pour des raisons de performance, chaque activité n'accède qu'une seule fois à cette table pour une activité cible donnée ; le résultat, c'est-à-dire si l'activité cible est dans le même groupe ou non, est conservé sur l'activité par la suite.

6.2.2 Localisation inter-groupe

Le service de localisation au sein d'un groupe proposé dans la section 4.2.2.3 repose sur le fait que chaque activité peut accéder à la table de localisation, donc à la mémoire stable. Or dans le cadre de groupes de recouvrement, nous ne pouvons plus faire la supposition que les activités d'un groupe Grp_i peuvent accéder à la mémoire stable d'un groupe Grp_j. Le problème se pose donc lorsqu'une activité i du groupe Grp_i tente de contacter une activité j du groupe Grp_j qui a redémarré et a donc changé de localisation.

Nous proposons d'utiliser les activités du groupe Grp_j lui-même comme intermédiaires entre l'activité i qui recherche la nouvelle localisation et la table de localisation du groupe Grp_j. Nous avons donc ajouté un mécanisme de messages non fonctionnels permettant à l'activité i de demander à une autre activité du groupe Grp_j la dernière localisation connue de l'activité j. Ces messages non fonctionnels permettent aux méta objets `FTManager` de communiquer sans passer par le mécanisme de communication propre de l'activité, et ne perturbent donc pas

l'activité.

Pour que ce système de délégation fonctionne, il faut que l'activité i du groupe Grp_i connaisse d'autres activités du groupe Grp_j que j. En effet, dans le cas inverse, i ne peut plus contacter aucune activité du groupe Grp_j, et ne peut donc plus retrouver la localisation de j.

Il faut donc un système pour gérer les références inter-groupes : si une activité acquiert une référence vers une activité d'un autre groupe, il faut obtenir et maintenir des points d'entrée vers ce groupe, c'est-à-dire des références vers des activités de ce groupe. Dans notre exemple, le fait que i ait une référence sur j implique que le groupe Grp_i possède et maintient un ensemble de points d'entrée vers le groupe Grp_j. En pratique, une table de localisation pour le groupe Grp_j, ou *table d'entrées vers* Grp_j notée $\mathbb{T}_{Grp_i \rightarrow Grp_j}$, est créée sur la mémoire stable du groupe Grp_i au moment où une activité de Grp_i acquiert une référence vers une activité du groupe Grp_j.

Grâce au mécanisme de communication non fonctionnelle entre les activités (plus précisément entre leurs méta-objets `FTManager` respectifs), chaque activité peut :

- fournir une table d'entrées vers son groupe de recouvrement à une autre activité,
- fournir la dernière localisation connue d'une activité appartenant à son groupe de recouvrement.

6.2.2.1 Création d'une table d'entrées

Les références étant réifiées sous forme de couple *stub-proxy* dans ProActive, nous avons ajouté un mécanisme de détection des références entrantes dans un groupe, en interceptant la désérialisation des couples *stub-proxy*. Lorsqu'une référence vers une activité j est reçue, et donc désérialisée par une activité i du groupe Grp_i, le mécanisme de création de la table d'entrées est déclenché : si l'activité j n'est dans aucune table sur la mémoire stable, alors un appel est fait sur cette référence pour connaître le Grp_j. Soit

- il existe une table d'entrées $\mathbb{T}_{Grp_i \rightarrow Grp_j}$ vers le groupe Grp_j sur la mémoire stable du groupe Grp_i, alors la référence vers l'activité j est ajoutée à cette table,
- il n'existe pas de table d'entrées $\mathbb{T}_{Grp_i \rightarrow Grp_j}$ vers le groupe Grp_j, alors une demande de table d'entrées est faite sur cette référence. Le groupe Grp_i obtient une table d'entrées vers le groupe Grp_j. Cette table est sauvegardée sur la mémoire stable du groupe Grp_i.

6.2.2.2 Maintenance d'une table d'entrées

L'activité qui a reçu la première référence vers le groupe distant est chargée de maintenir la table d'entrées au moyen d'un système de messages de vie (*heartbeat messages*). Encore une fois, nous avons ajouté un mécanisme non fonctionnel qui ne perturbe pas l'activité concernée ; on utilise une *thread* dédiée qui est créée et associée à l'activité. En particulier, cette *thread* ne fait pas partie de l'état de

l'activité. Une variable du méta objet `FTManager` indique si l'activité est chargée ou non de la maintenance d'une table de localisation de groupe distant. Ainsi, en cas de reprise, l'activité peut recréer cette *thread* dédiée.

L'activité (sa *thread* dédiée) doit régulièrement vérifier la validité des références dans la table de localisation de groupe qui lui est affectée. Plutôt que de toutes les vérifier une par une, nous avons optimisé le système de message de vie de la façon suivante. Nous avons ajouté à la table de localisation d'un groupe un numéro de version qui est incrémenté à chaque fois qu'une localisation est modifiée dans la table[1]. Ainsi, lorsque l'activité envoie un message de vie pour tester une référence, elle ajoute à ce message le dernier numéro de version connu de la table d'entrée. Si ce numéro de version est identique, alors le résultat de cet appel est vide, et la table d'entrées est considérée comme à jour ; les autres références ne sont pas testées. Si le numéro de version est inférieur, alors le résultat de cet appel est une table qui contient les changements de localisation qui ont eu lieu entre le numéro de version inférieur et le numéro de version actuel.

Ce mécanisme permet de minimiser le nombre de communications inter-groupes pour maintenir les tables d'entrées vers les groupes distants à jour.

6.2.3 Validation d'état global

Le mécanisme de validation d'état global assure, lorsqu'un message inter-groupe est envoyé, qu'en cas de panne et de reprise du groupe émetteur, ce message sera réémis avec le même contenu que dans l'exécution de référence. Ce problème d'équivalence du contenu est similaire au problème posé par les messages orphelins présenté dans la section 4.1.2 ; nous utilisons donc une solution similaire, l'historique de réception des requêtes. En assurant l'ordre de réception des requêtes intra-groupes, nous allons assurer l'équivalence du contenu des messages inter-groupes dupliqués.

La solution que nous proposons est de stocker sur la mémoire stable l'ensemble des historiques de réception du groupe, ou *historique global*, à chaque fois qu'un message à destination d'un autre groupe de recouvrement est émis. Chaque historique constituant l'historique global est associé au point de reprise correspondant déjà stocké sur la mémoire stable. C'est ce qui est fait au moment de la clôture de l'historique dans le protocole interne. Ainsi, comme dans le cas des messages orphelins précédemment, le passé de ces messages est rejoué en cas de réexécution et le duplicata du message est assuré d'être équivalent à l'original.

6.2.3.1 Capture de l'historique global

La technique utilisée dans la section 4.1.4 pour identifier et stocker l'historique global est asynchrone ; des messages spécifiques sont envoyés à toutes les

[1]Ce numéro de version ne correspond pas forcément au numéro d'incarnation car une seule panne peut entraîner plusieurs changements de localisation si des activités étaient coallouées.

activités qui déclenchent la clôture et le stockage de l'historique à la réception de ce message. Cette approche implique donc une très grande latence sur les envois des messages inter-groupes ; il faut que l'émetteur attende que toutes les activités aient stocké leur historique courant sur la mémoire stable. Le temps de latence sur l'envoi d'un message inter-groupe n'est donc pas borné et dépend fortement du nombre d'activités dans le groupe.

Or, dans le contexte d'une application communicante déployée sur une grille de calcul, nous pouvons supposer que le nombre de messages inter-groupes est minimisé *par rapport au nombre total de messages*, de par la structure de l'application. Pour autant, les messages inter-groupes ne peuvent pas être considérés comme rares durant l'exécution. Si on prend le cas par exemple d'applications SPMD comme le noyau CG ou Jacobi présentées dans la section 4.3, le nombre de communications entre groupes est au moins de l'ordre du nombre d'itérations de l'application.

Nous proposons donc pour minimiser le temps de latence sur l'envoi d'un message inter-groupe de maintenir sur chaque activité une estimation suffisante de l'historique global. De cette manière, la validation d'état global peut être faite de manière indépendante par une activité : elle doit simplement stocker sur la mémoire stable sa vue locale de l'historique global avant d'envoyer un message inter-groupe.

La difficulté dans ce cas est de maintenir sur chaque activité une vue *locale* suffisante de l'historique global courant. Nous allons pour cela utiliser une solution inspirée de certains protocoles de tolérance aux pannes par journalisation causale tels que Manetho [ELN 92b]. Dans Manetho, chaque message porte avec lui son graphe d'antécédence, c'est à dire le graphe de dépendance causale de tous les évènements non déterministes qui ont été exécutés avant l'envoi de ce message. Ce graphe d'antécédence permet à l'activité réceptrice de maintenir une vue du passé causal de l'application sous forme d'un graphe d'antécédence global. Ainsi, lorsqu'une validation d'état global doit être faite, l'activité doit simplement stocker ce graphe sur la mémoire stable.

Dans notre cas, l'historique de réception des messages est utilisé pour représenter le passé causal d'un message ; chaque message doit donc porter l'ensemble des historiques de réception qui précèdent causalement son envoi. Ainsi, lorsqu'un message est reçu par une activité, celle-ci peut mettre à jour sa vue locale de l'historique global.

6.2.3.2 Maintenance de la vue locale de l'historique global

Pour maintenir la vue locale de l'historique global, nous utilisons une technique proche des vecteurs d'horloges (*vector clock*) [BAL 02]. Chaque activité i maintient une table d'historiques, noté $\mathbb{H}_i$, qui représente la vue de i de l'historique global : $\mathbb{H}_i[j]$ est l'historique de j supposé par i. En particulier, $\mathbb{H}_i[i]$ représente l'historique de i et est donc toujours exact.

De manière à pouvoir comparer et mettre à jour les différents historiques de

réception, chaque historique $\mathbb{H}_i[j]$ doit être indexé *de façon absolue* : quand un élément est ajouté dans un historique, son index, c'est-à-dire son rang dans l'historique, doit rester constant même si d'autres éléments de l'historique sont effacés. En effet, pour minimiser la taille des vues locales, lorsqu'une activité i stocke en mémoire stable $\mathbb{H}_i$, elle efface le contenu de $\mathbb{H}_i$ (sauf le dernier élément de chaque historique, de manière à conserver l'index du dernier élément stocké en mémoire stable). Ainsi, lorsque l'activité i propage par la suite $\mathbb{H}_i$ sur les messages de l'application, elle indique aux récepteurs les éléments de l'historique global qui sont sauvegardés sur la mémoire stable, et qui n'ont donc plus besoin d'être maintenus : ces éléments peuvent être effacés des vues locales des récepteurs.

Finalement, lorsqu'un message $M_{i,j}$ est envoyé par une activité i, le vecteur $\mathbb{H}_i$ est ajouté au message. Lorsque ce message est reçu, l'activité j met à jour sa vue locale de l'historique global $\mathbb{H}_j$ en fonction de la vue locale de i, à la manière des vecteurs d'horloges. Cette mise à jour a deux fonctions :

- elle ajoute les éléments manquants à la fin de chaque historique, de manière à propager la progression de l'historique global ;
- elle efface aussi les éléments au début de chaque historique devenus inutiles, de manière à minimiser la taille des vues locales, et ainsi réduire la taille des informations à ajouter sur les messages.

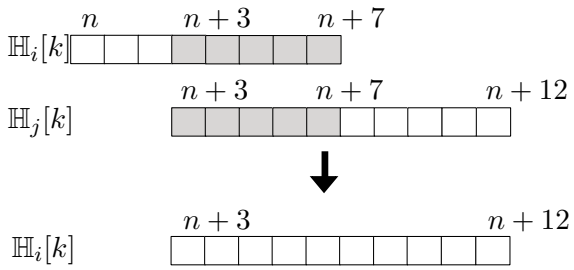

FIG. 6.2 – Mise à jour des vues locales de i

La figure 6.2 résume le mécanisme de mise à jour : l'activité i met à jour sa vue locale de l'historique de réception de l'activité k en fonction de la vue locale d'une activité j. La partie antérieure à la partie commune en gris peut être effacée, et la partie postérieure à la partie commune doit être ajoutée. On note que l'index des éléments est absolu : le premier élément de $\mathbb{H}_i[k]$ après la mise à jour porte l'index $n+3$.

6.2.3.3 Conservation de l'ordonnancement causal des réceptions entre les groupes

Nous montrons ici que la sauvegarde de l'historique global est aussi nécessaire sur *réception* d'un message inter-groupe. Prenons le cas de l'exemple de la figure 6.3 : les activités i et j,k forment deux groupes de recouvrement. Comme on peut le voir, les réceptions de Q_0 et de Q_2 sont causalement liées : Q_0 doit être

reçue avant Q_2. Or, si le groupe formé de j et k doit recouvrir depuis l'état global n, le message Q_2 est renvoyé artificiellement par le mécanisme de tolérance aux pannes, et n'est donc plus synchronisé avec la réception de Q_0. Il faut donc que les réceptions de ces deux messages fassent partie de l'historique de réception utilisé pendant la réexécution. Par conséquent, l'activité j doit sauvegarder son historique sur la mémoire stable au moment de la réception du message inter-groupe Q_2. Enfin, comme l'historique global doit constituer une coupe cohérente de l'exécution de référence, j ne peut pas sauvegarder *seulement* son historique locale ; il doit sauvegarder sa vue locale $\mathbb{H}_j$ de l'historique global.

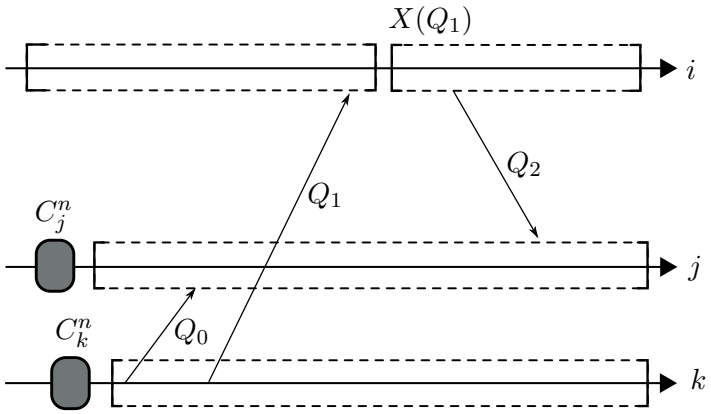

FIG. 6.3 – La réception de Q_0 doit précéder celle de Q_2 en cas de reprise depuis n

6.2.3.4 Cohabitation des deux protocoles

Le protocole de maintenance de l'historique global ne rentre pas en conflit avec le protocole interne. Il suppose que les historiques de réception de requête sont maintenus par les activités *tout le long de l'exécution*. La clôture d'historique du protocole interne sur réception du message M_n^{gs} déclenche encore sur chaque activité une sauvegarde sur la mémoire stable de l'historique local avec le dernier point de reprise (section 4.1.4) ; la différence est que même après cette clôture, l'activité *continue* à maintenir son historique local. Ainsi, l'historique global peut être maintenu durant toute l'exécution.

Lors d'une validation d'état global, comme lors de la clôture du protocole interne, les historiques de réceptions constituant l'historique global doivent être associés aux points de reprises formant la dernière ligne de recouvrement ; ils doivent pouvoir être ajoutés à la fin de la queue des requêtes en attente en cas de reprise depuis cette ligne (section 4.1.5).

Dans le cas de la clôture du protocole interne, chaque historique local est systématiquement associé au dernier point de reprise pris par l'activité. En effet, c'est cette clôture qui rend le dernier état global formé recouvrable, c'est-à-dire

qui transforme le dernier état global en ligne de recouvrement.

Dans le cas d'une validation d'état global, chaque historique local constituant l'historique global doit être associé *à tous les derniers points de reprise* pris par l'activité jusqu'à celui qui fait partie de la dernière ligne de recouvrement. En effet, le dernier point de reprise de chaque activité ne fait pas forcément partie de la dernière ligne de recouvrement. Notons N_{rec} l'index de la dernière ligne de recouvrement stockée en mémoire stable. Lorsqu'une activité a pris un point de reprise n mais que la dernière ligne de recouvrement est $N_{rec} < n$, le point de reprise n ne peut pas encore être utilisé pour redémarrer l'activité en cas de panne. Or il faut que l'état global soit validé même en cas de reprise depuis l'état global $N_{rec} < n$. Par conséquent, lorsqu'une activité valide l'état global, il faut qu'elle stocke chaque historique local constituant l'historique global avec tous points de reprise n déjà stockés sur la mémoire stable tels que $n > N_{rec}$, ainsi qu'avec les points de reprise N_{rec}.

6.2.4 Journalisation et filtrage des duplicatas

6.2.4.1 Journalisation pessimiste par le récepteur

Les messages inter-groupes sont journalisés de façon pessimiste par l'activité réceptrice : le message reçu est stocké sur la mémoire stable du groupe de l'activité réceptrice avant d'être délivré, c'est-à-dire avant d'être mis dans la queue des requêtes en attente, ou placé dans le futur correspondant. Si ce message est une requête, comme l'historique de réception est systématiquement stocké sur la mémoire stable sur réception d'un message inter-groupe, cette requête est placée directement dans l'historique au lieu d'une promesse. De cette manière, l'ordre de réception est conservé, et les requêtes journalisées n'ont pas besoin d'être renvoyées artificiellement au moment de la reprise d'une activité : l'historique étant ajouté à la queue des requêtes en attente au moment de la reprise, elle contient déjà les requêtes journalisées.

Dans le cas des réponses, elles sont sauvegardées sur un journal sur la mémoire stable qui n'a pas besoin d'être ordonné, et sont renvoyées artificiellement avant la reprise de l'activité concernée.

Ces deux mécanismes existent déjà dans la version du protocole sans groupe, puisque les messages en transit doivent être journalisés, dans l'historique pour les requêtes et dans un journal spécifique pour les réponses (section 4.1.3).

6.2.4.2 Filtrage des messages dupliqués

Des messages dupliqués vont être créés par la réexécution d'un groupe : comme dans les protocoles de journalisation classiques, ces messages qui ont déjà été reçus et été journalisés par des activités des autres groupes doivent être filtrés. Pour être filtrés, il faut soit qu'ils ne soient pas émis, soit qu'ils soient ignorés par le récepteur. Pour cela, nous supposons d'abord que les messages sont indexés de la façon suivante : l'index $M_{i,j}.id = n$ indique que le message $M_{i,j}$ est le n^{me} envoyé par l'activité i. Ainsi, le couple (i, $M_{i,j}.id$) permet d'identifier de manière unique

le message $M_{i,j}$.

En pratique, à cause du rendez-vous sur les communications du modèle ASP, il est plus efficace de ne pas émettre les messages dupliqués plutôt que de les ignorer du coté du récepteur. Dans ce cas, on évite pendant la réexécution le temps d'attente dû au rendez-vous sur les communications inter-groupes, qui sont supposées être des communications longue distance, donc coûteuses en temps. Nous proposons d'utiliser une table $LastSent_i$ maintenue par chaque activité i : $LastSent_i[j]$ indique l'index du dernier message envoyé à j par i, j étant forcément une activité d'un *autre* groupe de recouvrement. Ainsi, lorsqu'une activité i valide l'état global avant d'envoyer un message inter-groupe, c'est-à-dire stocke $\mathbb{H}_i$ sur la mémoire stable, elle stocke aussi la table $LastSent_i$. De cette façon, en cas de réexécution de l'activité i, la table $LastSent_i$ peut être récupérée, et l'activité i peut annuler l'émission des messages inter-groupes à destination de j dont l'index est inférieur à $LastSent_i[j]$. L'activité i n'a donc pas à subir l'attente dûe au rendez-vous sur les communications inutiles.

Pourtant, ce mécanisme n'est pas suffisant : supposons que i envoie un message inter-groupe $M_{i,j}$ à j et stocke $LastSent_i$ sur la mémoire stable. Supposons alors qu'*avant d'envoyer effectivement le message*, l'activité i tombe en panne. Dans ce cas, $LastSent_i$ contient $M_{i,j}.id$, et $M_{i,j}$ sera donc filtré par i pendant la réexécution alors qu'il n'a jamais été reçu par j.

Une solution possible serait de stocker l'index $LastSent_i[j]$ sur la mémoire stable *après* avoir envoyé $M_{i,j}$, mais cette solution implique un accès supplémentaire à la mémoire stable. Plutôt que d'ajouter un accès à la mémoire stable, nous ne filtrons les messages de i vers j que jusqu'à l'index $LastSent_i[j] - 1$ durant la réexécution, et nous utilisons une table complémentaire $LastRcvd_i$ du coté du récepteur j. Chaque activité maintient cette table $LastRcvd_i$ qui indique pour chaque activité j l'index du dernier message reçu. A chaque modification, cette table est stockée sur la mémoire stable en même temps que le message inter-groupe reçu ; elle ne nécessite donc pas d'accès supplémentaire à la mémoire stable. Ainsi, lorsque l'activité j reçoit un message $M_{i,j}$ de i, la valeur de contenue dans $LastRcvd_j[i]$ supérieure à $M_{i,j}.id$ lui indique que ce message est un duplicata et qu'il doit être ignoré.

6.2.5 Exemple récapitulatif

La figure 6.4 résume les différents mécanismes de maintenance d'historique global, de validation d'état global et de journalisation présentés dans cette section, combinés avec le protocole interne. Elle présente une exécution possible d'un groupe de recouvrement formé de trois activités i, j et k. Pour simplifier la figure, nous n'avons pas montré les services de requêtes. De plus, nous supposons ici que la ligne de recouvrement $n-1$ est terminée, donc que $N_{rec} = n - 1$.

Pour chaque réception de requête, la vue locale de l'historique global du récepteur après la mise à jour est donnée. On voit par exemple qu'à la réception de Q_3, l'activité k met à jour $\mathbb{H}_k[j]$ en y ajoutant une promesse de requête $Q_{i,j}^{pmd}$: une

requête envoyée par i a été reçue par j (Q_1).

Les requêtes Q_2 reçue par i et Q_4 envoyée par k sont des requêtes inter-groupes. Lorsque i reçoit Q_2, elle met à jour $\mathbb{H}_i[i]$ en y ajoutant non pas une promesse de requête mais la requête Q_2 elle-même, puis valide l'état global : elle stocke $\mathbb{H}_i$ avec les points de reprise n mais aussi $n-1$ puisque la clôture n n'a pas encore eu lieu. Comme $\mathbb{H}_i$ contient la requête Q_2, i journalise Q_2 en même temps que la validation d'état global. Elle peut ensuite effacer les éléments contenus dans $\mathbb{H}_i$: on peut voir que Q_2 n'est plus dans $\mathbb{H}_i[i]$ à la réception de Q_5 par i.

Lorsque k envoie Q_4, elle valide l'état global en stockant $\mathbb{H}_k$ sur la mémoire stable. Encore une fois, la clôture n n'ayant pas encore eu lieu sur k, $\mathbb{H}_k$ est stocké avec les points de reprise n et $n-1$.

Lorsque l'état global n est achevé, la clôture d'historique n est déclenchée par le protocole interne ($\Diamond$) : chaque activité stocke sur la mémoire stable son historique local $\mathbb{H}_i[i]$ avec le point de reprise n. L'état global n devient la dernière ligne de recouvrement. Par conséquent, lorsque i valide l'état global avant l'envoi de la requête inter-groupe Q_8, il stocke sur la mémoire stable sa vue locale $\mathbb{H}_i$ uniquement avec les points de reprise n.

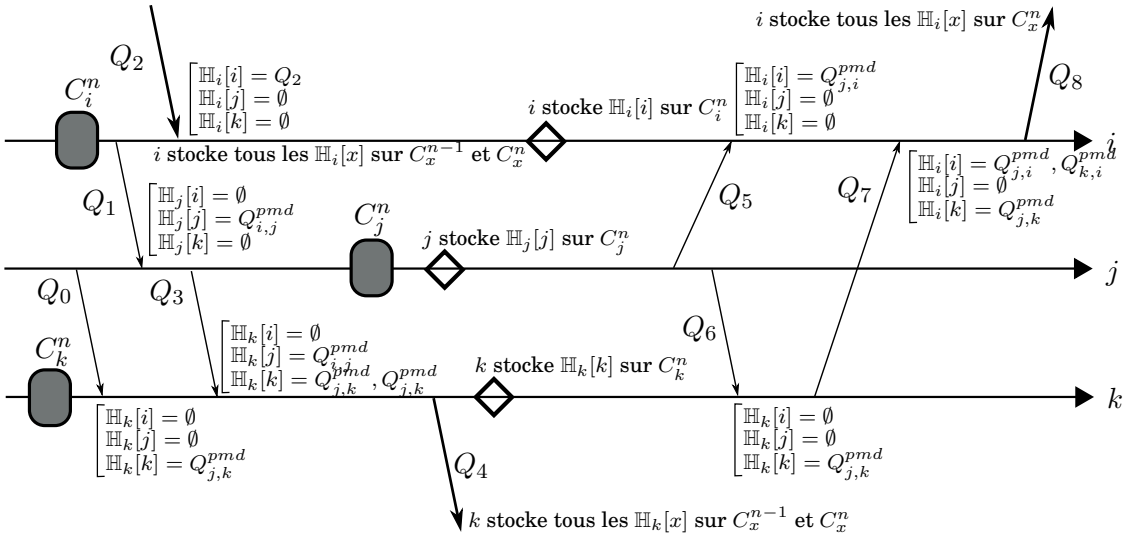

FIG. 6.4 – Exécution d'un groupe de recouvrement : Q_2, Q_4 et Q_8 sont des requêtes inter-groupes

6.3 Modification du protocole

Le protocole proposé dans la section 4.1.6 doit être étendu avec le protocole de maintenance des vues locales de l'historique global, et avec le mécanisme de validation d'état global. Nous renommons les procédures définies dans la section 4.1.6 pour le protocole sans groupe de recouvrement en les préfixant par `interne_`.

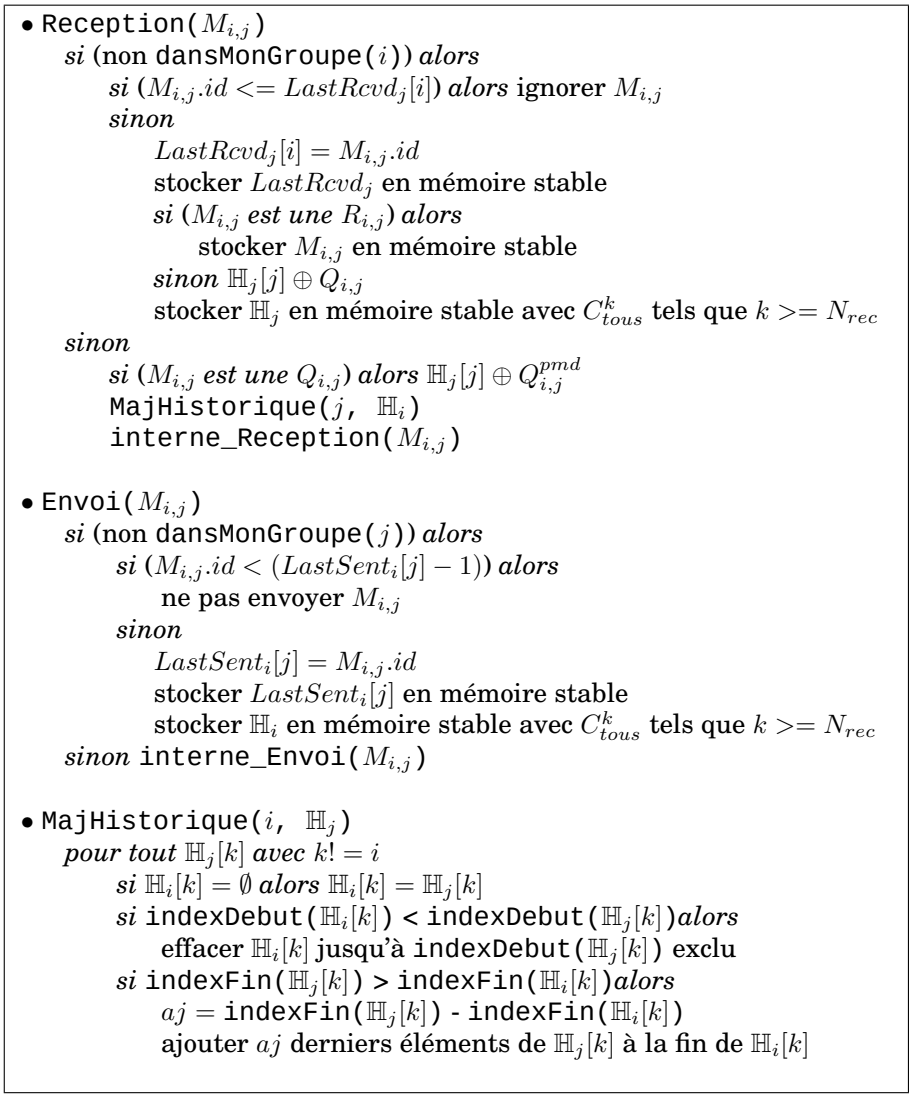

FIG. 6.5 – Extension du protocole pour les groupes de recouvrement

La figure 6.5 présente l'extension du protocole sous forme de pseudo-code.

Chaque activité i maintient les valeurs suivantes :
- la table $\mathbb{H}_i$, qui contient les historiques connus par i des autres activités du groupe,
- la table $LastRcvd_i$, avec $LastRcvd_i[j]$ indiquant l'index du dernier message reçu de l'activité j,
- la table $LastSent_i$, avec $LastSent_i[j]$ indiquant l'index du dernier message envoyé à l'activité j.

De plus, un message intra-groupe $M_{i,j}$ envoyé par i transporte la table $\mathbb{H}_i$, qui est donc connue de j à la réception de $M_{i,j}$.

On notera C^k_{tous} pour désigner tous les points de reprise stockés sur la mémoire stable dont l'index est k.

Comme les historiques contenus dans $\mathbb{H}_i$ sont indexés de manière absolue, nous disposons de deux fonctions `indexDebut(`$\mathbb{H}_i[k]$`)` et `indexFin(`$\mathbb{H}_i[k]$`)` qui renvoient respectivement l'index absolu du premier élément de $\mathbb{H}_i[k]$, et celui du dernier élément de $\mathbb{H}_i[k]$.

Enfin, le mécanisme de gestion des groupes présenté dans la section 6.2.1 est utilisé à travers un prédicat `dansMonGroupe(`i`)`. Ce prédicat renvoie vrai si l'activité i est dans le même groupe de recouvrement que l'activité appelante, faux sinon.

6.4 Implémentations et expérimentations

Nous avons étendu l'implémentation existante du protocole de manière à pouvoir déployer pour une application ProActive plusieurs groupes de recouvrement. Cette implémentation s'est entièrement intégrée dans l'implémentation existante, à l'exception du mécanisme de réception des réponses. En effet, comme nous l'avons vu dans la section 3.3.3, l'utilisation d'un mécanisme de journalisation des messages implique que tous les messages doivent pouvoir être reçus à n'importe quel moment. Donc pour être compatible avec la journalisation pessimiste des messages inter-groupe, nous avons ajouté un mécanisme de temporisation des réceptions de réponses. Si une activité reçoit une réponse en avance, cette réponse est placée dans un tampon. Lors de la création d'un futur suite à l'envoi d'une requête inter-groupe, l'activité vérifie qu'il n'existe pas dans ce tampon de réponse correspondante au futur créé. Si c'est le cas, cette réponse est retirée du tampon et est utilisée pour mettre à jour le futur.

Cependant, il faut noter que ce mécanisme de temporisation n'intervient que pour les messages inter-groupe, puisqu'une réponse intra-groupe ne peut jamais être reçue avant la création de son futur (section 4.1.2.2). De plus, le mécanisme de temporisation des réceptions de réponses n'est utilisé que si plusieurs groupes de recouvrement sont déployés.

Enfin, seul le serveur de localisation proposé dans la section 4.2.2 a été modifié pour prendre en compte les groupes de recouvrement : le mécanisme des tables de localisation inter-groupe a été intégré à l'implémentation d'origine.

6.4.1 Spécification des groupes de recouvrement

Comme on l'a vu dans la section 6.2.1, la notion de groupe de recouvrement est liée à la mémoire stable utilisée. Les groupes se forment donc par mémoire stable commune. En pratique, les groupes de recouvrement sont spécifiés sous forme de nœuds virtuels : on associe chacun des nœuds virtuels à un *technical service* différent (section 4.2.1), chacun spécifiant un serveur de tolérance aux pannes différent. Les groupes sont *automatiquement* découverts au déploiement de l'application. Par exemple, le descripteur de déploiement suivant permet de déployer une application sur 2 groupes de recouvrement indépendants :

```
...
<virtualNodesDefinition>
  <virtualNode name="groupeRec1" serviceRefid="service1"/>
  <virtualNode name="groupeRec2" serviceRefid="service2"/>
</virtualNodesDefinition>
...
<technicalServiceDefinitions>
 <service id="service1" class="services.FaultTolerance">
   <arg name="globalServer" value="rmi://host1/globalServer"/>
   <arg name="checkpointingMode" value="asynchronous"/>
   <arg name="TTC" value="60"/>
 </service>
 <service id="service2" class="services.FaultTolerance">
   <arg name="globalServer" value="rmi://host2/globalServer"/>
   <arg name="checkpointingMode" value="synchronous"/>
   <arg name="TTC" value="100"/>
 </service>
</technicalServiceDefinitions>
...
```

Comme on peut le voir, chacun des groupes peut définir une configuration adaptée aux ressources physiques sous-jacentes. Par exemple, supposons que les machines formant le premier groupe disposent de plus de mémoire que les machines du deuxième groupe. On peut alors utiliser dans le premier groupe l'option d'envoi des points de reprise en asynchrone (section 4.2.3), et rester en synchrone dans le deuxième groupe pour minimiser la taille mémoire nécessaire.

6.4.2 Expérimentations

Nous présentons dans cette section les expérimentations réalisées avec l'implémentation des groupes de recouvrement dans ProActive. L'objectif est d'évaluer l'impact sur les performances induit par les groupes de recouvrement par rapport à la version du chapitre 4, puis d'évaluer le gain apporté en terme de vitesse de recouvrement.

6.4.2.1 Environnement d'évaluation

Les expérimentations présentées ici ont été menées sur la grille de calcul Grid5000 [CAP 05]. Cette grille est constituée de machines réparties en 13 grappes se situant dans 9 villes de France. On distingue quatre architectures différentes (Itanium, Xeon, G5 et Opteron).

Les tests de comparaison entre le protocole avec et sans groupe de recouvrement ont été réalisés sur les grappes de Sophia-Antipolis (utilisée pour les expérimentations du chapitre 4) et de Rennes. La seconde grappe est constituée de 99 HP ProLiant DL145G2 (2 AMD Opteron 246 2 GHz avec 1 Mo de cache et 2 Go de mémoire), reliés en Gigabit Ethernet. Des noyaux Linux 2.6.13 sont deployés sur tous les nœuds, ainsi qu'une machine virtuelle Java Sun 1.5.0.07. Le lien réseau entre les deux grappes a une latence de 9,1 ms, et une bande passante de 40,2

Mo/s [COT 06]. Pour les tests sans groupe de recouvrement, un serveur de tolérance aux pannes unique est déployé sur un nœud Sun Fire X4100 de la grappe de Sophia-Antipolis. Dans le cas des groupes de recouvrement, un deuxième serveur est déployé sur un nœud HP ProLiant DL145G2 de la grappe de Rennes.

Les tests de passage à l'échelle du protocole avec groupe ont été réalisés sur 4, 5 et 6 grappes de la grille Grid5000 : Sophia-Antipolis, Rennes, Lille, Bordeaux, Nancy et Lille. Les temps de latence des liens inter-grappes varient de 1,5ms entre Toulouse et Bordeaux à 13,4ms entre Toulouse et Lille. Les bandes passantes varient de 26,9 Mo/s entre Toulouse et Lille à 190 Mo/s entre Toulouse et Bordeaux [COT 06].

Pour les mêmes raisons de stabilité des résultats que dans les expérimentations de la section 4, un seul nœud ProActive est déployé par nœud physique, et chacun de ces nœuds ne contient qu'une seule activité.

Enfin, nous menons les expérimentations avec l'application Jacobi présentée dans la section 4.3.2.

6.4.2.2 Évaluation du surcoût introduit par les groupes de recouvrement

Nous évaluons dans un premier temps le surcoût introduit par le protocole de validation globale, c'est-à-dire par la maintenance des vues locales de l'historique global. Ce surcoût est d'abord dû au fait que les messages intra-groupes doivent porter leur passé causal sous forme d'une table d'historiques de réception. De plus, lorsqu'un message intra-groupe est reçu, l'activité réceptrice doit comparer et mettre à jour sa vue locale de l'historique global.

Nous avons évalué ce surcoût sur l'application Jacobi. L'application a été déployée sur la seule grappe de Sophia-Antipolis, de manière à pouvoir comparer les résultats obtenus avec les expérimentations du chapitre 4. Il n'y a *qu'un seul* groupe de recouvrement dans ce cas, et aucune validation globale n'est déclenchée durant l'expérience.

Le graphique 6.6 présente donc le surcoût induit par le protocole de maintenance de la vue locale de l'historique global. Nous constatons que ce surcoût est relativement élevé, et augmente avec le nombre de nœuds. Il atteint 100% dans le cas d'une matrice 5000 sur 64 nœuds. Notons cependant qu'à titre de comparaison, la surcoût dû à la maintenance des graphes d'antécédence dans le protocole Manetho sur 16 nœuds varie de 6% à 30% selon les applications [BOU 05], contre 20 à 30% dans notre cas sur l'application Jacobi. Il faut noter aussi que cette évaluation est pessimiste puisque aucune validation d'état global n'est déclenchée : aucune partie des historiques ne peut donc être effacée durant l'exécution, et la taille de l'historique global croît continuellement.

Cet aspect de l'implémentation pourrait être optimisé pour diminuer le surcoût de maintenance de l'historique global. En particulier, il serait possible d'utiliser une approche par journalisation asynchrone systématique des historiques sur une mémoire stable dédiée ; les historiques locaux seraient sauvegardés en parallèle de l'activité sur une mémoire stable *à chaque modification*. Cette approche

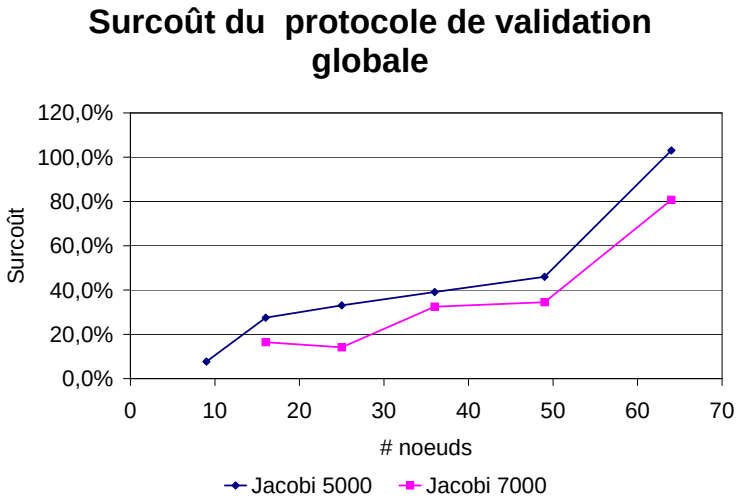

FIG. 6.6 – Surcoût induit par le maintien des vues locales de l'historique global

est utilisée dans le protocole de journalisation causale MPICH-V2, et est comparée aux approches classiques de type Manetho par Bouteiller et al. dans [BOU 05]. Les auteurs ont montré qu'une telle journalisation systématique pouvait réduire le surcoût d'un facteur 2 à 10 par rapport à une approche par mise à jour sur réception, selon l'application testée et sur des expérimentations sur 16 nœuds.

L'autre partie du surcoût introduit par les groupes de recouvrement provient de la journalisation pessimiste des messages par l'activité réceptrice. Pour évaluer ce surcoût de manière unitaire, nous avons déployé deux activités chacune dans un groupe de recouvrement différent, et évalué le temps d'envoi de 500 Mo de données aléatoires sous forme de requêtes. Le paramètre est un tableau de type `byte`, et la taille de ce tableau est adaptée en fonction du nombre d'envois pour obtenir 500 Mo de données. Enfin, cette expérience a été réalisée en déployant les deux activités sur la même grappe de machines, puis sur deux grappes distantes (Sophia-Antipolis et Rennes).

La figure 6.7 nous présente les résultats de cette expérience. Nous constatons que le surcoût dû à la journalisation pessimiste du message par le récepteur reste faible (de l'ordre de 10% maximum pour des messages de 32 Mo) dans le cas de 2 grappes de machines distantes, alors que ce surcoût est particulièrement élevé dans le cas d'une seule grappe de machines (jusqu'à 160 %). Cette différence s'explique par la latence élevée et la bande passante faible du lien qui relie les deux grappes : le surcoût relatif de la journalisation est compensé par le coût unitaire d'un envoi de message entre deux grappes. Cette expérience montre en particulier que l'utilisation d'un protocole de journalisation pessimiste classique, c'est-à-dire journaliser *tous* les messages, induirait un surcoût très élevé à l'exécution.

Comme une utilisation correcte des groupes de recouvrement implique que chaque groupe représente des grappes de machines distinctes, nous pouvons conclure que le surcoût dû à la journalisation reste faible par rapport au surcoût dû au protocole de maintenance de l'historique globale.

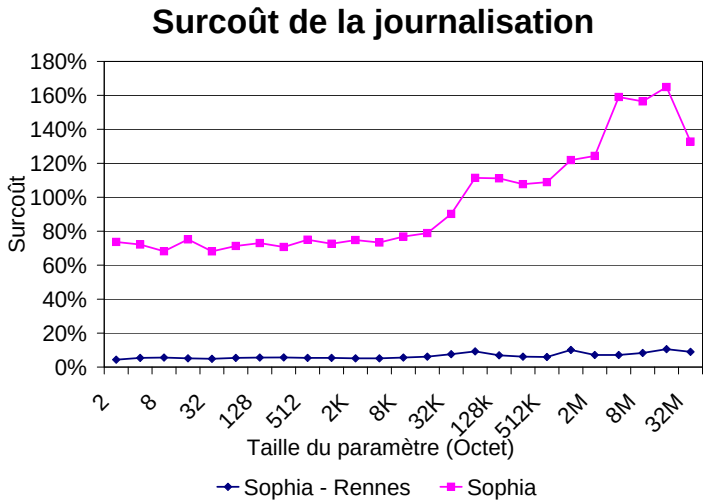

FIG. 6.7 – Surcoût induit par la journalisation pessimiste des messages

Enfin, nous avons comparé le surcoût global induit par le protocole sans et avec groupes de recouvrement, mais aussi le temps d'exécution supplémentaire induit par une panne. Nous avons utilisé l'application Jacobi avec des matrices 7000 et 9000, et un TTC de 100 secondes. Pour les tests avec une panne, une panne est déclenchée 50 secondes après la formation de la deuxième ligne de recouvrement, c'est-à-dire approximativement entre la formation de deux lignes consécutives.

Les points de reprise sont dans les deux cas pris de manière asynchrone. De plus, l'optimisation de conservation locale du dernier point de reprise n'est pas utilisée ici.

Les graphiques de la figure 6.8 présentent les résultats de cette expérimentation. On constate que le surcoût du protocole avec groupe de recouvrement est plus élevé que sans groupe, mais aussi qu'il augmente plus vite avec le nombre de nœuds. Il atteint 57% dans le cas d'une matrice 9000 sur 100 nœuds.

On constate à l'inverse que l'impact d'une panne durant l'exécution est proportionnellement moins important dans le cas du protocole avec groupe : le temps de recouvrement est plus important avec le protocole sans groupe, jusqu'à 70 secondes de plus dans le cas d'une matrice 9000 sur 100 nœuds sur un temps d'exécution total de référence de 260 secondes.

Cette rapidité de la reprise dans le cas des groupes de recouvrement est d'abord dûe au fait que seule une partie de l'application doit reprendre après la panne. De plus, les messages inter-groupes ne sont pas réémis durant la réexécution du groupe concerné par la panne. Ainsi, le temps mis pour refaire le calcul perdu entre la dernière ligne de recouvrement et la panne est plus petit que le temps mis durant l'exécution de référence, puisque les envois de nombreux messages sur les liens réseaux faibles sont évités.

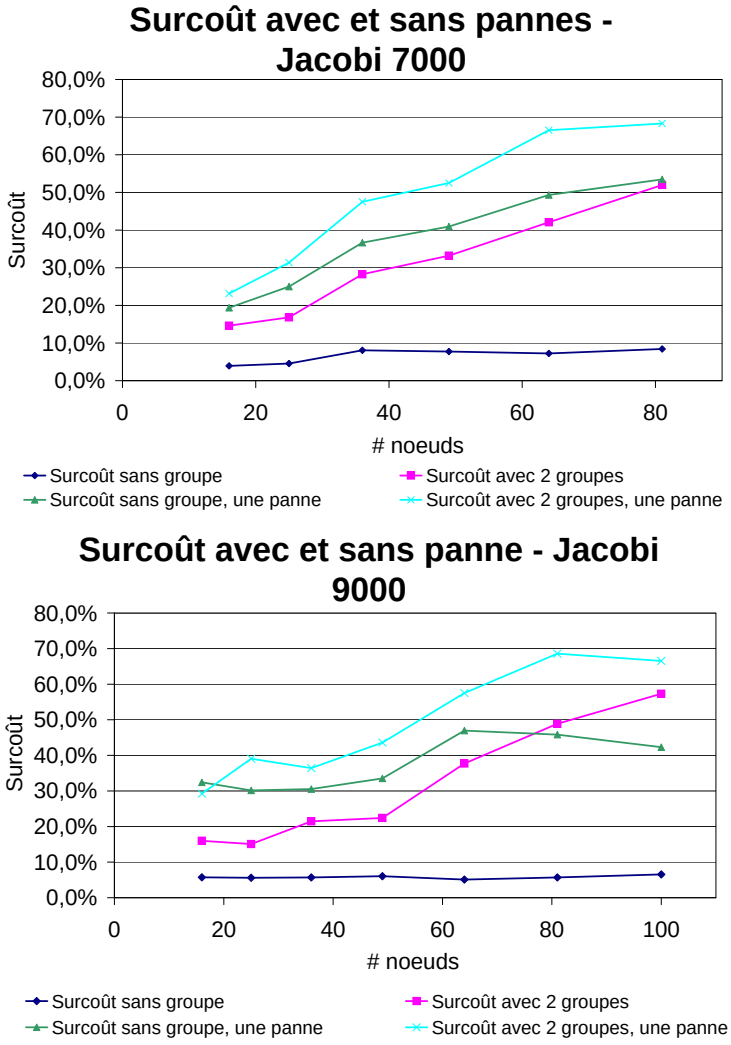

FIG. 6.8 – Surcoût sans et avec groupe de recouvrement, sans panne et avec une panne

Même si le surcoût sans panne est plus élevé, il peut donc être plus efficace d'utiliser le protocole avec groupe si la probabilité de panne est forte. En particulier, l'augmentation du nombre de nœuds augmentant la probabilité de panne, il peut être plus efficace d'utiliser le protocole avec groupe même si son surcoût augmente avec le nombre de nœuds.

Nous avons donc voulu évaluer pour les expérimentations précédentes à partir de combien de pannes l'utilisation du protocole avec groupe permet un temps total d'exécution plus faible que sans groupe. En considérant que le temps de recouvrement dû à une panne ne dépend pas du nombre total de pannes à partir du moment où les pannes n'ont pas lieu pendant une période de reprise, nous avons extrapolé les résultats de l'expérience précédente. Nous avons donc isolé le temps d'exécution supplémentaire dû à la panne, puis avons cumulé ce temps au temps total d'exécution autant de fois que de pannes supposées.

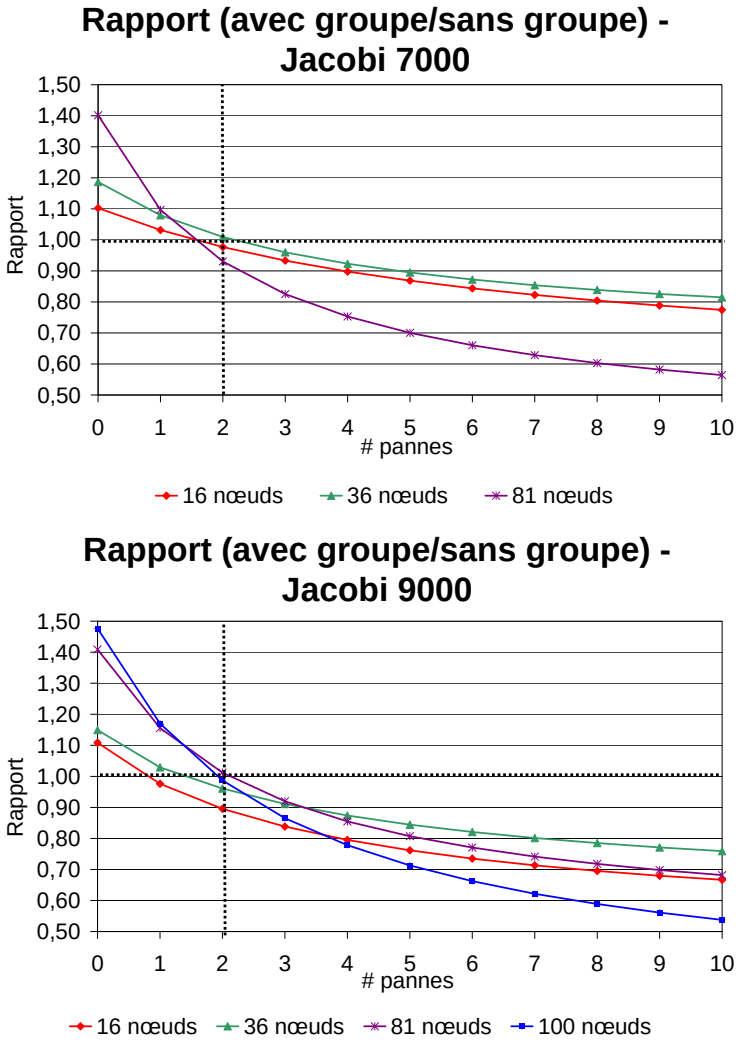

FIG. 6.9 – Rapport entre les temps avec et sans groupe de recouvrement

Les graphiques de la figure 6.9 présentent les résultats de cette extrapolation, sous la forme du rapport $T\ avec\ groupes / T\ sans\ groupe$ en fonction du nombre de pannes. On peut constater que dans tous les cas, pour les expérimentations précédentes, le temps total d'exécution est plus faible avec le protocole avec groupes à partir de deux pannes, le rapport devenant inférieur à 1. De plus, avec une matrice 9000 déployée sur 100 nœuds, le temps total d'exécution avec 10 pannes est réduit de moitié dans le cas de l'utilisation de groupes de recouvrement.

Enfin, nous avons évalué le surcoût induit par les groupes de recouvrement à plus large échelle en fonction du nombre de groupes, donc en pratique en fonction du nombre de grappes différentes utilisées. Nous avons déployé pour cela une matrice 16000 sur 4, 5 et 6 grappes distantes, avec de 100 à 324 nœuds. La valeur du TTC est de 100 secondes.

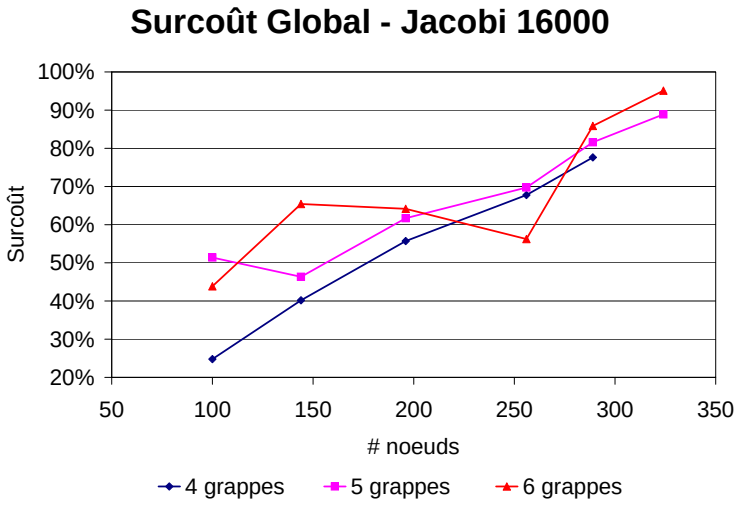

FIG. 6.10 – Surcoût global des groupes de recouvrement

Le graphique 6.10 présente les résultats de l'expérimentation sur plusieurs grappes. On constate tout d'abord deux points singuliers, sur 100 nœuds répartis sur 5 grappes, et sur 256 nœuds répartis sur 6 grappes. Ces points singuliers sont la conséquence d'une très forte instabilité des résultats, bien que les valeurs présentées ici soient la moyenne de 10 exécutions consécutives. Cette instabilité est dûe au fait que les liens réseaux entre les grappes ne sont pas dédiés exclusivement à Grid5000, et donc que les temps de communication peuvent varier fortement. Nous avons constaté en pratique que les résultats d'une expérience peuvent varier de 50% selon l'heure à laquelle elle est réalisée.

Hormis ces deux points singuliers, nous constatons que pour un nombre de nœuds relativement petit (de 100 à 200 nœuds), le surcoût augmente avec le nombre de groupes : il est de 25% pour 100 nœuds sur 4 groupes et de 45% sur 100 nœuds pour 6 groupes. En effet, l'augmentation du nombre de groupes entraîne une augmentation du nombre de messages journalisés et du nombre de validations d'état global déclenché durant l'exécution, donc une augmentation du surcoût.

Nous constatons aussi que cette différence s'efface avec l'augmentation du nombre total de nœuds : sur 296 nœuds, les surcoûts sont quasiment égaux, autour de 80% pour 4, 5 et 6 groupes, et sur 324 nœuds autour de 90% pour 5 et 6 groupes. Ce rapprochement des surcoûts s'explique par l'augmentation du surcoût dû au protocole de maintenance des vues locales de l'historique global. On a vu en effet que ce surcoût augmente rapidement avec les nombres de nœuds du groupe : dans le cas d'un grand nombre de nœuds avec un petit nombre de groupes, ce surcoût est plus important. Il compense donc le fait que le nombre de journalisations et de validations d'état global diminue avec le nombre de groupe.

6.5 Conclusion

Nous avons présenté dans ce chapitre une extension pour les grilles de calcul du protocole proposé dans le chapitre 4. Cette extension est basée sur la notion de groupe de recouvrement et sur une journalisation pessimiste des messages par le récepteur. Nous avons de plus ajouté dans le protocole existant un mécanisme permettant une validation d'état global rapide à tout moment de l'exécution.

Cette solution par regroupement des activités est adaptée au regroupement des ressources en grappes intrinsèque aux grilles de calcul. Elle permet en particulier de répartir les mémoires stables, et ce de façon totalement indépendante : la mémoire stable utilisée par un groupe est indépendante des autres. Nous avons aussi proposé dans ce cadre de mémoire stable répartie un mécanisme de localisation adapté qui ne nécessite pas de point central stable.

Cette extension a été implémentée dans ProActive. Elle a été intégrée à l'implémentation du protocole simple sans pour autant modifier les performances de ce dernier dans le cas d'un déploiement sans groupe de recouvrement. La spécification des groupes de recouvrement se base sur le même mécanisme de configuration que celui du protocole simple ; les activités se regroupent autour d'une mémoire stable commune spécifiée au moment du déploiement.

Cette implémentation nous a permis d'évaluer les performances de l'extension dans un contexte d'utilisation réel. Nous avons d'abord constaté que le surcoût induit est majoritairement dû au mécanisme de maintenance de l'historique global permettant la validation d'état global. Ce surcoût variant reste cependant comparable à celui induit dans des conditions similaires par le protocole Manetho, qui utilise une méthode proche de la nôtre. Nous prévoyons d'améliorer ces performances en utilisant une approche par journalisation pessimiste des historiques, comme proposé dans [BOU 05].

Nous avons de plus comparé le surcoût global avec celui induit par le protocole simple. Bien que le premier soit de 4 à 10 fois plus elevé pour des exécutions sans panne, nous avons estimé qu'il devient inférieur à partir de 2 à 3 pannes durant l'exécution, jusqu'à être 2 fois plus petit pour 10 pannes.

Enfin, dans le cadre d'un déploiement sur 4, 5 et 6 grappes de machines réparties sur l'ensemble de la France avec un groupe par grappe, nous avons observé un surcoût variant de 25% à 94% selon le nombre de nœuds et le nombre de groupes. Nous avons constaté en particulier que pour un grand nombre total de nœuds, le nombre de groupes n'influe quasiment pas sur le surcoût global ; ceci est la conséquence de l'augmentation rapide du surcoût dû au protocole de maintenance de l'historique global avec le nombre de nœuds dans un groupe. Nous prévoyons cependant de réduire ce surcoût en utilisant une technique de journalisation pessimiste des historiques, de manière à ce que le surcoût global diminue avec le nombre de groupes.

Chapitre 7

Conclusion et perspectives

Dans cette thèse, nous nous sommes intéressés à l'étude de la tolérance aux pannes par points de reprise dans le cas où la création d'états globaux cohérents est impossible. Notre objectif premier était de proposer une solution de tolérance aux pannes efficace, transparente et portable pour le modèle ASP et pour son implémentation en Java, ProActive. C'est le développement de cette solution qui nous a amené à étudier de manière plus globale le problème du recouvrement d'une application depuis un état global non cohérent.

7.1 Bilan

Les protocoles de tolérance aux pannes par points de reprise synchronisés ou induits par message classiques se basent sur la création des états globaux cohérents qui sont directement recouvrables. Pourtant, nous avons vu que l'hypothèse de persistance forte des processus faite par les protocoles classiques pour assurer la cohérence des états globaux peut avoir des conséquences sur l'efficacité ou la portabilité du mécanisme de tolérance aux pannes. Malgré cela, le problème du recouvrement depuis un état global non cohérent dans le cadre de ce type de protocole a été peu étudié dans la littérature.

Nous avons contribué à combler ce manque tout d'abord en proposant un protocole par points de reprise et son implémentation ne supposant pas que les états globaux soient cohérents. Cette implémentation fait aujourd'hui partie de la version téléchargeable de l'intergiciel ProActive. Elle est fournie avec un mécanisme de configuration simple d'utilisation et avec l'ensemble des services nécessaires à son fonctionnement, tels que la gestion des ressources ou la localisation d'activité après une reprise.

Nous avons montré à travers des expérimentations réalistes utilisant des applications réparties communicantes que notre solution et son implémentation présentent de bonnes performances, et induisent en particulier un surcoût faible sur le temps d'exécution.

Nous avons aussi contribué de manière plus générale à l'étude du recouvrement depuis un état global non cohérent en définissant formellement une nouvelle condition de recouvrabilité, la $\mathcal{P}$-cohérence, basée sur la notion de promesse

d'évènement. Cette définition s'intègre dans un formalisme événementiel capable de prendre en compte la sémantique de n'importe quel système ; elle est donc applicable dans un cadre général.

Nous nous sommes basés dans un premier temps sur des promesses d'évènement capables de contrôler et de synchroniser l'exécution de manière globale, sans prendre en compte la faisabilité d'un tel outil. Dans le cadre d'une telle synchronisation globale, nous avons prouvé formellement la recouvrabilité d'un état global $\mathcal{P}$-cohérent.

Nous avons ensuite montré que l'hypothèse forte d'une synchronisation globale n'est pas nécessaire pour assurer la recouvrabilité d'un état global $\mathcal{P}$-cohérent. Nous avons donc proposé une approche possible se basant uniquement sur un mécanisme de synchronisation locale simple, l'attente par nécessité. Nous avons identifié dans ce cadre réaliste les trois hypothèses qui doivent être vérifiées par l'état global et le système considérés pour conserver les propriétés prouvées dans le cadre d'une synchronisation globale. Finalement, nous avons montré que la recouvrabilité d'un état global $\mathcal{P}$-cohérent est conservée avec une synchronisation locale, et donc que l'hypothèse forte d'une synchronisation globale n'est pas nécessaire.

Enfin, en appliquant ce formalisme au modèle ASP, nous avons prouvé la correction de notre protocole en montrant que les états globaux formés durant l'exécution sont toujours recouvrables.

Enfin, nous avons contribué plus spécifiquement aux travaux de notre équipe de recherche dans le domaine des grilles de calcul en proposant une extension de notre protocole par points de reprise adaptée à ce contexte. Cette extension se base sur la constitution automatique de groupes de recouvrement au déploiement de l'application. Elle permet une répartition indépendante des mémoires stables et un confinement des effets d'une panne au seul groupe concerné.

Cette extension a été intégrée à l'implémentation existante. Nous avons évalué ses performances sur la grille de calcul Grid5000. Nous avons constaté que le surcoût sur le temps d'exécution sans panne est largement dû au protocole additionnel de maintien de l'historique global utilisé au sein de chaque groupe. Nous prévoyons d'étudier une solution alternative, basée sur la journalisation pessimiste de l'histoire causale de l'exécution, qui permettrait de minimiser ce surcoût.

Finalement, nous avons montré que le confinement des pannes assuré par les groupes de recouvrement permet de minimiser considérablement le temps d'exécution supplémentaire induit par la présence de pannes. Par conséquent, malgré un surcoût sur le temps d'exécution sans panne plus important que l'approche sans groupe, nous démontrons que notre extension présente de meilleures performances dans le contexte des grilles de calcul, où les ressources sont particulièrement volatiles.

7.2 Perspectives

Deux perspectives d'extension de notre travail nous semblent prometteuses. La première concerne le problème des services imbriqués : nous avons vu dans la section 4.1.2.2 que nous ne considérions pas le cas d'une politique de service

définie à l'aide de services imbriqués. Dans la section suivante, nous revenons sur ce point et proposons une solution.

La seconde perspective traite de la configuration du mécanisme de tolérance aux pannes dans un cadre dans lequel une infrastructure pair-à-pair telle que celle de ProActive est l'unique fournisseur de ressources de calcul.

7.2.1 Service de requêtes imbriqués

Nous avons vu que, dans le modèle ASP, l'utilisateur peut modifier la politique de service en déclenchant explicitement des services *imbriqués* grâce à la primitive $Serve(M)$, où M est suffisant pour identifier la requête à servir - par exemple le nom d'une méthode. Le problème lié aux services imbriqués est que l'état de l'activité juste avant un tel service n'est pas un état capturable (au sens défini dans la section 3.1.3) par sérialisation de l'activité : l'image obtenue ne contient pas le fait que l'exécution a atteint le déclenchement du service imbriqué.

Prenons un exemple de la méthode `foo()` suivante. On suppose ici que l'exécution de cette méthode correspond au service d'une requête envoyée par une activité distante. La méthode de l'intergiciel ProActive `serveOldest(String methodName)` déclenche le service de la plus ancienne requête appelant la méthode `methodName` de la queue de requête de l'activité. Les méthodes `internalComputation()` et `endComputation()` effectuent des calculs et modifient l'état de l'activité en conséquence.

```
public void foo(){
  internalComputation();
  ProActive.serveOldest("bar");
  endComputation();
}
```

Supposons qu'un point de reprise soit pris avant le service de `bar()`. Dans ce point de reprise, la seule indication concernant la position du flux d'exécution au moment de la capture est la requête qui était en cours de service au moment de la capture, ici la requête `foo()`. En particulier, le fait que la méthode `internalComputation()` a déjà été exécutée n'est pas indiqué dans le point de reprise ; il faudrait pour cela être capable de sérialiser aussi l'état de la *thread* de l'activité.

Si l'activité reprend depuis ce point de reprise, elle va redémarrer le service de `foo()`, et donc exécuter *une nouvelle fois* la méthode `internalComputation()`. Les modifications produites par l'exécution de cette méthode vont être à nouveau effectuées ; l'état de l'activité ne correspond donc plus à l'état attendu après l'exécution de `internalComputation()`. De manière générale, une activité qui reprend depuis un point de reprise déclenche le service de la requête en cours au moment de la capture *depuis le début* de la méthode correspondant au service.

Par conséquent, une activité ne peut pas prendre de point de reprise avant le service d'une requête imbriquée. Si cette requête est orpheline est qu'elle est servie par une activité, alors on ne peut plus supposer comme dans la section 4.1.2.2 qu'une requête orpheline n'est jamais servie, et se trouve donc toujours dans la queue de requête au moment du point de reprise.

Une solution possible serait d'autoriser dans le protocole le fait qu'une requête orpheline puisse déjà être servie, et donc d'autoriser les requêtes orphelines non promises. On a vu en effet dans la section 5.3.4 qu'une coupe $\mathcal{P}$-cohérente peut être recouvrable même si aucun message orphelin n'est promis. Mais dans ce cas, comme on l'a vu dans la section 4.1.3, il serait possible de recevoir des réponses en avance, c'est-à-dire avant la création du futur associé à cette réponse. Il faudrait donc un mécanisme additionnel dans l'implémentation de ProActive qui permette de pouvoir temporiser la réception de n'importe quelle réception de réponse. De plus, comme on l'a vu dans la section 5.3.4, il faudrait aussi augmenter la taille de l'historique puisque le nombre de messages à réordonner artificiellement pour respecter l'ordonnancement de message augmente avec le nombre de messages orphelins non promis.

Nous proposons ici une solution alternative qui permettrait de rendre recouvrable l'état d'une activité avant le service d'une requête imbriquée. Ainsi, la prise d'un point de reprise serait possible avant *n'importe quel* service de requête, et les propriétés de notre protocole seraient conservées. De plus, cette approche permet d'augmenter le nombre d'occurrences d'état capturable durant l'exécution. De cette façon, on réduit le décalage entre les points de reprise constituant les lignes de recouvrement, donc on réduit la taille des historiques et le nombre de messages en transit à journaliser. Le surcoût pendant l'exécution mais aussi le temps de recouvrement après une panne sont donc réduits.

Pour être capturable avant un service imbriqué, il faut que l'état *sérialisé* de l'activité indique si les instructions précédant un service imbriqué ont été exécutées ou non avant la capture. L'état peut donc être rendu capturable en modifiant le code de l'application. En effet, reprenons l'exemple de la méthode `foo()`. Si une variable *d'état* de l'activité indique si le code précédant le service imbriqué a été exécuté ou non, alors l'état avant ce service est capturable.

```
private boolean internalComputationDone = false;
...
public void foo(){
  if (!internalComputationDone){
    internalComputation();
  }
  this.internalComputationDone = true;
  ProActive.serveOldest("bar");
  endComputation();
}
```

Cette solution est applicable par l'utilisateur lui-même, mais le mécanisme de tolérance aux pannes n'est alors plus transparent. De plus, si la modification à apporter dans notre exemple est simple, elle peut devenir complexe dans le cas de plusieurs services imbriqués les uns dans les autres. Nous voulons donc proposer un préprocesseur de code source (ou de *bytecode*) qui modifie le code original de manière à assurer que l'état des activités avant un service imbriqué est capturable. Ce mécanisme pourrait se baser sur une technique utilisant un

objet *contexte*, indiquant la position du flux d'exécution sous la forme d'une pile de services de requête. Cet objet étant un objet Java standard, il peut être capturé avec l'état de l'activité et donc indiquer au moment de la réexécution la position du flux d'exécution avant la capture.

Cette technique est par exemple utilisée par le préprocesseur JavaGo [SEK 99] qui fournit une persistance forte pour les applications Java. Nous avons vu cependant dans la section 3.3.1 que nous excluions une telle approche à cause de l'impact sur les performances. En effet, dans la cas de JavaGo, l'objet contexte doit indiquer *à l'instruction près* la position du flux d'exécution, et doit par conséquent mettre à jour l'objet contexte entre chaque instruction. Nous avons vu que cette mise à jour pouvait induire un surcoût de 100% sur l'exécution (section 2.2.1).

On ne retrouverait pas cet impact prohibitif sur les performances dans notre cas. En effet, on doit pouvoir connaître la position du flux d'exécution uniquement *par rapport aux services de requêtes*, et pas par rapport à toutes les instructions exécutées. Par conséquent, l'objet contexte ne devrait plus être mis à jour qu'au début et à la fin de chaque service de requête, imbriquée ou non ; l'impact sur les performances serait donc minime par rapport à celui induit par une solution telle que JavaGo.

7.2.2 Tolérance aux pannes et infrastructure pair-à-pair

Nous travaillons actuellement avec Alexandre di Costanzo de l'équipe OASIS sur le concept de *familles* de nœud d'exécution dans le cadre de l'infrastructure pair-à-pair de ProActive. Nous nous plaçons ici dans un contexte où *toutes* les ressources utilisées par les applications sont acquises à travers l'infrastructure pair-à-pair. Nous avons en effet vu dans la section 3.2.2.2 qu'une application peut être déployée sur des nœuds existants gérés par l'infrastructure pair-à-pair. De plus, nous avons vu dans la section 4.2.2.2 que l'implémentation du serveur de ressources qui fournit des nœuds pour redémarrer les activités tombées en panne peut utiliser une infrastructure pair-à-pair sous-jacente. Donc dans ce contexte, le ou les serveurs de ressources ne maintiennent plus explicitement des listes de ressources, mais reposent uniquement sur cette infrastructure.

Une particularité de l'infrastructure pair-à-pair de ProActive est qu'elle fonctionne en mode *best-effort*, c'est-à-dire qu'elle fournit des nœuds sans critères de choix : lorsqu'on demande un nœud, le nœud retourné par l'infrastructure est le premier nœud à répondre à la demande. Par conséquent, lorsqu'un nœud est acquis au déploiement de l'application ou bien récupéré par le serveur de ressources pour redémarrer une activité, les propriétés de ce nœud en terme de MTBF, de CPU ou encore de mémoire ne sont pas prévisibles. En effet, l'infrastructure pair-à-pair est utilisée pour administrer de larges ensembles de machines *hétérogènes*. Elle a par exemple permis d'administrer un ensemble de 1007 processeurs constitué de machines de bureau et de grappes de machines dédiées, et de déployer sur cet ensemble une application résolvant le problème des N-Reines [CAR 06a].

La problématique est ici que la configuration du mécanisme de tolérance aux pannes dépend fortement des propriétés des nœuds d'exécution. Par exemple, la

valeur du TTC doit être adaptée au MTBF du nœud, ou encore les optimisations possibles telles que l'envoi de point de reprise asynchrone (section 4.2.3) dépendent de la mémoire disponible.

Nous voulons donc proposer une solution qui permette de tenir compte de ces propriétés dans le choix de la configuration du mécanisme de tolérance aux pannes *sans modifier les mécanismes de gestion des nœuds de l'infrastructure pair-à-pair*. Nous voulons utiliser pour cela le mécanisme des groupes de recouvrement pour ProActive proposé dans le chapitre 6 pour pouvoir appliquer sur un nœud une configuration adaptée *au moment où ce nœud est fourni par l'infrastructure pair-à-pair*.

Nous supposons tout d'abord que l'administrateur d'une infrastructure pair-à-pair (celui qui déploie les nœuds d'exécution) classe les différents nœuds *en familles* en fonction de leurs caractéristiques. Par exemple, il peut définir la famille `desktop` pour des machines de bureau de faible puissance, avec un MTBF faible. Ce nom de famille est associé à chaque nœud déployé sur ces machines de bureau sous la forme d'une variable d'environnement et est donc disponible grâce à la méthode `Node.getProperty(property)` (section 4.2.1). En particulier, l'infrastructure pair-à-pair n'a pas à gérer explicitement les familles comme des groupes : la seule information concernant la famille est le nom associé au nœud via la variable d'environnement.

La classification en famille doit être accessible aux utilisateurs de l'infrastructure (ceux qui déploient des applications). Elle va permettre à l'utilisateur de définir des configurations de tolérance aux pannes adaptées à chacune des familles constituant l'infrastructure, sous la forme de différents *technical services* dans le descripteur de déploiement de son application (section 4.2.1).

Nous avons vu dans la section 6.4.1 que les groupes de recouvrement se forment automatiquement au déploiement par configuration commune, donc par *technical service* commun. Il va donc se former à l'exécution un groupe de recouvrement par famille. Ce regroupement permet d'appliquer la configuration adaptée à chaque nœud d'exécution, en isolant les effets des nœuds les moins stables sur le reste du système.

Nous prévoyons donc d'adapter le mécanisme de déploiement ainsi que le serveur de ressources pour qu'ils tiennent compte *automatiquement* de la famille des nœuds (donnée par une variable d'environnement du nœud), et appliquent la configuration adaptée. Pour cela, il faut tout d'abord que l'utilisateur puisse spécifier dans le descripteur de déploiement une association [famille,*technical service*]. Nous proposons donc d'ajouter à la balise `<service>` un paramètre `family` qui spécifie un nom de famille :

```
...
<virtualNodesDefinition>
  <virtualNode name="groupeRec1" serviceRefid="servDesktop"/>
  ...
</virtualNodesDefinition>
...
<technicalServiceDefinitions>
```

```
  <service id="servDesktop" class="FaultTolerance" family="desktop">
    <arg name="globalServer" value="rmi://host1/globalServer"/>
    <arg name="checkpointingMode" value="synchronous"/>
    <arg name="TTC" value="30"/>
  </service>
  ...
</technicalServiceDefinitions>
...
```

Ainsi, dans cet exemple, tous les nœuds acquis au déploiement ou récupérés par le serveur de ressources après une panne dont le nom de famille est `desktop` seront configurés par le *technical service* `servDesktop`. Ce *technical service* définit une configuration avec un TTC faible pour supporter un MTBF faible et un envoi de point de reprise synchrone pour limiter l'utilisation de la mémoire. De plus, les effets des pannes potentiellement fréquentes dans ce groupe seront confinés grâce aux groupes de recouvrement.

La modification la plus importante à apporter à l'implémentation existante est la gestion *dynamique* des groupes de recouvrements. En effet, le serveur de ressources peut fournir un nœud de n'importe quelle famille pour recouvrir une activité après une panne, et potentiellement un nœud dans une famille différente de celle du nœud tombé en panne. Ce nouveau nœud sera automatiquement configuré avec un *technical service* différent : l'activité redémarrée va donc *changer* de groupe de recouvrement.

Il va nous falloir prendre en compte ce dynamisme dans les groupes, en adaptant tout d'abord les mécanismes de gestion des groupes et de localisation inter-groupe proposés dans les sections 6.2.1 et 6.2.2. De plus, lors d'un changement de groupe, une activité va devoir atteindre un état cohérent avec l'état global du groupe de destination avant de pouvoir être intégrée dans ce groupe. Nous prévoyons pour cela d'utiliser des *groupes de transition*, créés et associés au groupe de destination au moment du changement de groupe, et ne contenant *que* l'activité changeant de groupe. Lorsqu'une activité se trouve dans un groupe de transition, elle doit rester dans ce groupe tant qu'elle n'a pas pris un point de reprise faisant partie d'une ligne de recouvrement du groupe de destination. Il faudra donc appliquer entre un groupe de recouvrement et les groupes de transition qui lui sont associés un protocole qui combine le protocole interne du chapitre 4 et la journalisation pessimiste des messages.

Cette approche adaptative en fonction des caractéristiques des ressources est une direction de recherche prometteuse. Elle permet en particulier une gestion complète et transparente du problème de l'hétérogénéité des ressources. Elle pourrait de plus inclure la possibilité pour l'utilisateur de spécifier une *qualité de service* attendue qui serait dynamiquement prise en compte au moment de la configuration du service non fonctionnel. Une telle approche à la fois automatique et adaptative permettrait aux applications distribuées de se comporter de façon entièrement *autonome* face aux pannes, et de garantir une qualité de service applicative.

Bibliographie

[AGB 00] A. AGBARIA, and R. FRIEDMAN. Virtual machine based heterogeneous checkpointing, 2000.

[A.L 91] J. RUSSELL A. LOWRY, and A. GLODBERG. "Optimistic failure recovery for very large networks". pages 66–75, Septembre 1991.

[ALV 98] L. ALVISI, and K. MARZULLO. Message logging : Pessimistic, optimistic, causal, and optimal. *Software Engineering*, 24(2) :149–159, 1998.

[ALV 99] L. ALVISI, S. RAO, S. HUSAIN, A. MEL, and E. ELNOZAHY. "An analysis of communication-induced checkpointing". In *FTCS '99 : Proceedings of the Twenty-Ninth Annual International Symposium on Fault-Tolerant Computing*, page 242, Washington, DC, USA, 1999. IEEE Computer Society.

[ALV 02] L. ALVISI, K. BHATIA, and K. MARZULLO. Causality tracking in causal message-logging protocols. *Distributed Computing*, 15(1) :1–15, 2002.

[AVI 04] A. AVIZIENIS, J.C. LAPRIE, B. RANDELL, and C. LANDWEHR. Basic concepts and taxonomy of dependable and secure computing. *IEEE Transactions on Dependable and Secure Computing*, 01(1) :11–33, 2004.

[BAD 02] L. BADUEL, F. BAUDE, and D. CAROMEL. "Efficient, Flexible, and Typed Group Communications in Java". In *Joint ACM Java Grande - ISCOPE Conference*, pages 28–36, Seattle, 2002. ACM Press.

[BAI 95] D. BAILEY, T. HARRIS, W. SAPHIR, R. WIJNGAART, A. WOO, and M. YARROW. "The nas parallel benchmarks 2.0". Technical report Report NAS-95-020, NASA Ames Research Center, December 1995.

[BAL 95] R. BALDONI, A. MOSTEFAOUI, J. BRZEZINSKI, J. HÉLARY, and M. RAYNAL. "Characterization of consistent global checkpoints in large-scale distributed systems". In *FTDCS '95 : Proceedings of the 5th IEEE Workshop on Future Trends of Distributed Computing Systems*, page 314, Washington, DC, USA, 1995. IEEE Computer Society.

[BAL 98] R. BALDONI, F. QUAGLIA, and B. CICIANI. "A vp-accordant checkpointing protocol preventing useless checkpoints". In *SRDS '98 : Proceedings of the The 17th IEEE Symposium on Reliable Distributed Systems*, page 61, Washington, DC, USA, 1998. IEEE Computer Society.

[BAL 99] R. BALDONI, F. QUAGLIA, and P. FORNARA. An index-based checkpointing algorithm for autonomous distributed systems. *IEEE Transaction on Parallel Distributed Systems*, 10(2) :181–192, 1999.

[BAL 02] R. BALDONI, and M. RAYNAL. Fundamentals of distributed computing : A practical tour of vector clock systems. *IEEE Distributed Systems Online*, 3(2), 2002.

[BAU 05a] F. BAUDE, D. CAROMEL, C. DELBÉ, and L. HENRIO. "A hybrid message logging-cic protocol for constrained checkpointability.". In *Euro-Par*, pages 644–653, 2005.

[BAU 05b] F. BAUDE, D. CAROMEL, C. DELBÉ, and L. HENRIO. Un protocole de tolérance aux pannes pour objets actifs non préemptifs. *Technique et Science Informatiques*, 24, 2005.

[BOL 00] W. BOLOSKY, J. DOUCEUR, D. ELY, and M. THEIMER. "Faisability of a serverless distributed file system deployed on an existing set of desktop pcs". In *ACM international conference on Measurement and modeling of computer systems, SIGMETRICS*, pages 34–43, 2000.

[BOS 02] G. BOSILCA, A. BOUTEILLER, F. CAPPELLO, S. DJAILALI, G. FEDAK, C. GERMAIN, T. HERAULT, P. LEMARINIER, O. LODYGENSKY, F. MAGNIETTE, V. NERI, and A. SELIKHOV. "Mpich-v : toward a scalable fault tolerant mpi for volatile nodes". In *Supercomputing '02 : Proceedings of the 2002 ACM/IEEE conference on Supercomputing*, pages 1–18, Los Alamitos, CA, USA, 2002. IEEE Computer Society Press.

[BOU 99] S. BOUCHENAK, and D. HAGIMONT. "Pickling threads state in the java system". In *Third European Research Seminar on Advances in Distributed Systems (ERSADS'99), Madeira Island, Portugal.*, 1999.

[BOU 03a] 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 Supercomputing*, page 25, Washington, DC, USA, 2003. IEEE Computer Society.

[BOU 03b] A. BOUTEILLER, G. KRAWEZIK, P. LEMARINIER, and F. CAPPELLO. "Mpich-v3 : A hierarchical fault tolerant mpi for multi-cluster grids". In *IEEE/ACM SC 2003, Phoenix USA*, November 2003.

[BOU 04] S. BOUCHENAK, D. HAGIMONT, S. KRAKOWIAK, N. DE PALMA, and F. BOYER. Experiences implementing efficient java thread serialization, mobility and persistence. *Softw. Pract. Exper.*, 34(4) :355–393, 2004.

[BOU 05] A. BOUTEILLER, B. COLLIN, T. HERAULT, P. LEMARINIER, and F. CAPPELLO. "Impact of event logger on causal message logging protocols for fault tolerant mpi". In *IPDPS '05 : Proceedings of the 19th IEEE International Parallel and Distributed Processing Symposium (IPDPS'05) - Papers*, page 97, Washington, DC, USA, 2005. IEEE Computer Society.

[BRI 84] D. BRIATICO, A. CIUFFOLETTI, and L. SIMONCINI. "A distributed domino-effect free recovery algorithm". In *IEEE International Symposium on Reliability, Distributed Software, and Databases*, 1984.

[BRO 03] G. BRONEVETSKY, D. MARQUES, K. PINGALI, and P. STODGHILL. "C3 : A system for automating application-level checkpointing of mpi programs". In *The 16th International Workshop on Languages and Compilers for Parallel Computers (LCPC'03)*, October 2003.

[BRO 06] G. BRONEVETSKY, R. FERNANDES, D. MARQUES, K. PINGALI, and P. STODGHILL. "Recent advances in checkpoint/recovery systems". In *Parallel and Distributed Processing Symposium, 2006. IPDPS 2006*, volume 8, pages 25–29, Avril 2006.

[BUN 02] P. BUNGALE, S. SRIDHAR, and V. KRISHNAMURTHY. "A poll-free, low-latency approach to process state capture/recovery in heterogeneous computing systems". In *Networks, Parallel and Distributed Processing, and Applications*, Tsukuba, Japan, Avril 2002.

[BUS 05a] J. BUSCA, F. PICCONI, and P. SENS. "Pastis : A highly-scalable multi-user peer-to-peer file system.". In *Euro-Par*, pages 1173–1182, 2005.

[BUS 05b] J. BUSTOS-JIMENEZ, D. CAROMEL, A. COSTANZO, and J. PIQUER. "Balancing active objects on a peer to peer infrastructure". In *SCCC '05 : Proceedings of the XXV International Conference on The Chilean Computer Science Society*, page 109, Washington, DC, USA, 2005. IEEE Computer Society.

[CAP 05] F. CAPPELLO, E. CARON, M. DAYDE, F. DESPREZ, E. JEANNOT, Y. JEGOUN, S. LANTERI, J .LEDUC, N. MELAB, G. MORNET, R. NAMYST, P. PRIMET, and O. RICHARD. "Grid'5000 : a large scale, reconfigurable, controlable and monitorable Grid platform". In *Grid'2005 Workshop*, Seattle, USA, November 13-14 2005. IEEE/ACM.

[CAR 01] D. CAROMEL, F. HUET, and J. VAYSSIÈRE. "A simple security-aware mop for java". In *REFLECTION '01 : Proceedings of the Third International Conference on Metalevel Architectures and Separation of Crosscutting Concerns*, pages 118–125, London, UK, 2001. Springer-Verlag.

[CAR 04a] D. CAROMEL, C. DELBÉ, and L. HENRIO. "A fault-tolerance protocol for asp calculus : Design and proof". Technical report, INRIA #5246, Juin 2004.

[CAR 04b] D. CAROMEL, L. HENRIO, and B. SERPETTE. "Asynchronous and deterministic objects". In *POPL '04 : Proceedings of the 31st ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 123–134, New York, NY, USA, 2004. ACM Press.

[CAR 05] D. CAROMEL, and L. HENRIO. "*A Theory of Distributed Object : Asynchrony - Mobility - Groups - Components*". Springer Verlag, 2005.

[CAR 06a] D. CAROMEL, A. COSTANZO, and C. MATHIEU. Peer-to-peer for computational grids : Mixing clusters and desktop machines. *submitted to Special Issue of Parallel Computing Journal on Large Scale Grid*, 2006.

[CAR 06b] D. CAROMEL, C. DELBÉ, and A. COSTANZO. "Peer-to-peer and fault-tolerance : Towards deployment based technical services". In *Second*

CoreGRID Workshop on Grid and Peer to Peer Systems Architecture, Paris, France, Janvier 2006.

[CAR 06c] D. CAROMEL, C. DELBÉ, A. COSTANZO, and M. LEYTON. Proactive : an integrated platform for programming and running applications on grids and p2p systems. *Computational Methods in Science and Technology*, 12(1), 2006.

[CAR 06d] D. CAROMEL, C. DELBÉ, and L. HENRIO. "Promised consistency for rollback recovery". Technical report, INRIA #5902, Mai 2006.

[CHA 85] K. CHANDY, and L. LAMPORT. Distributed snapshots : determining global states of distributed systems. *ACM Trans. Comput. Syst.*, 3(1) :63–75, 1985.

[CHA 96a] T. CHANDRA, and S. TOUEG. Unreliable failure detectors for reliable distributed systems. *J. ACM*, 43(2) :225–267, 1996.

[CHA 96b] B. CHARRON-BOST, F. MATTERN, and G. TEL. Synchronous, asynchronous, and causally ordered communication. *Distributed Computing*, 9(4) :173–191, 1996.

[CHI 95] S. CHIBA. "A metaobject protocol for C++". In *ACM Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'95)*, SIGPLAN Notices 30(10), pages 285–299, Austin, Texas, USA, October 1995.

[CHO 02] S. CHOI, and S. DEITZ. "Compiler support for automatic checkpointing". In *HPCS '02 : Proceedings of the 16th Annual International Symposium on High Performance Computing Systems and Applications*, page 213, Washington, DC, USA, 2002. IEEE Computer Society.

[CON 96] D. CONAN. "*Tolérance aux fautes par recouvrement arrière dans les systèmes informatiques répartis*". PhD thesis, Université Paris 6, Septembre 1996.

[CON 98] D. CONAN, and G. BERNARD. "La reprise sur erreur par recouvrement arrière automatique dans les systèmes répartis". In *Parallélisme et répartitions (coll. Parallélisme, réseaux et répartition) Ed. J.F. Myoupo, Hermès*, april 1998.

[COT 06] C. COTI, T. HERAULT, P. LEMARINIER, L. PILARD, A. REZMERITA, E. RODRIGUEZ, and F. CAPPELLO. "Blocking vs. non-blocking coordinated checkpointing for large-scale fault tolerant mpi". In *IEEE/ACM SC 2006, Tempa, USA*, Novembre 2006.

[DAM 99] O. DAMANI, A. TARAFDAR, and V. GARG. "Optimistic recovery in multi-threaded distributed systems". In *SRDS '99 : Proceedings of the 18th IEEE Symposium on Reliable Distributed Systems*, page 234, Washington, DC, USA, 1999. IEEE Computer Society.

[DEL 04] C. DELBÉ. "Causal ordering of asynchronous request services". In *DSN '04 : Proceedings of the 2004 International Conference on Dependable Systems and Networks (DSN'04) - student forum*, pages 154–157, Washington, DC, USA, 2004. IEEE Computer Society.

[ELN 92a] E. ELNOZAHY, D. JOHNSON, and W. ZWAENPOEL. "The Performance of Consistent Checkpointing". In *Proceedings of the 11th IEEE Symposium on Reliable Distributed Systems*, Houston, Texas, 1992.

[ELN 92b] E. ELNOZAHY, and W. ZWAENEPOEL. Manetho : Transparent rollback-recovery with low overhead, limited rollback and fast output. *IEEE Transactions on Computers, Special Issue on Fault-Tolerant Computing*, pages 526–531, 1992.

[ELN 02] E. ELNOZAHY, L. ALVISI, Y. WANG, and D. JOHNSON. A survey of rollback-recovery protocols in message-passing systems. *ACM Comput. Surv.*, 34(3) :375–408, 2002.

[ELN 04] E. ELNOZAHY, and J. PLANK. Checkpointing for peta-scale systems : A look into the future of practical rollback-recovery. *IEEE Transactions on Dependable and Secure Computing*, 1(2) :97–108, April-June 2004.

[ENO 98] T. ENOKIDO, H. HIGAKI, and M. TAKIZAWA. "Significant message precedence in object-based systems". In *ICPADS '98 : Proceedings of the 1998 International Conference on Parallel and Distributed Systems*, page 284, Washington, DC, USA, 1998. IEEE Computer Society.

[FOL 94] B. FOLLIOT, and P. SENS. "Gatostar : A fault tolerant load sharing facility for parallel applications". In *EDCC-1 : Proceedings of the First European Dependable Computing Conference on Dependable Computing*, pages 581–598, London, UK, 1994. Springer-Verlag.

[FOS 99] I. FOSTER, and C. KESSELMAN. "*The Grid : Blueprint for a new Computing Infrastructure*". Morgan Kaufman Publishers, 1999.

[GAN 03] A. GANESH, A. KERMARREC, and L MASSOULI. Peer-to-peer membership management for gossip-based protocols. *IEEE Trans. Comput.*, 52(2) :139–149, 2003.

[GIO 05] R. GIOIOSA, J. SANCHO, S. JIANG, and F. PETRINI. "Transparent, incremental checkpointing at kernel level : a foundation for fault tolerance for parallel computers". In *SC '05 : Proceedings of the 2005 ACM/IEEE conference on Supercomputing*, page 9, Washington, DC, USA, 2005. IEEE Computer Society.

[HOW 99] J. HOWELL. "Straightforward java persistence through checkpointing". In *Proceedings of the 8th International Workshop on Persistent Object Systems (POS8) and Proceedings of the 3rd International Workshop on Persistence and Java (PJW3)*, pages 322–334, San Francisco, CA, USA, 1999. Morgan Kaufmann Publishers Inc.

[HéL 97a] J. HÉLARY, A. MOSTÉFAOUI, and M. RAYNAL. "Preventing useless checkpoints in distributed computations". In *SRDS '97 : Proceedings of the 16th Symposium on Reliable Distributed Systems (SRDS '97)*, page 183, Washington, DC, USA, 1997. IEEE Computer Society.

[HéL 97b] J. HÉLARY, A. MOSTÉFAOUI, and M. RAYNAL. "Virtual precedence in asynchronous systems : Cencept and applications". In *WDAG '97 : Proceedings of the 11th International Workshop on Distributed Algorithms*, pages 170–184, London, UK, 1997. Springer-Verlag.

[HéL 99a] J. HÉLARY, A. MOSTEFAOUI, and M. RAYNAL. Communication-induced determination of consistent snapshots. *IEEE Trans. Parallel Distrib. Syst.*, 10(9) :865–877, 1999.

[HéL 99b] J. HÉLARY, R. NETZER, and M. RAYNAL. Consistency issues in distributed checkpoints. *IEEE Trans. Softw. Eng.*, 25(2) :274–281, 1999.

[HéL 00] J. HÉLARY, A. MOSTEFAOUI, R. NETZER, and M. RAYNAL. Communication-based prevention of useless checkpoints in distributed computations. *Distrib. Comput.*, 13(1) :29–43, 2000.

[JOH 87] D. JOHNSON, and W. ZWAENEPOEL. "Sender-based message logging". In *The 7th annual international symposium on fault-tolerant computing*. IEEE Computer Society, 1987.

[JOH 88] D. JOHNSON, and W. ZWAENEPOEL. "Recovery in distributed systems using optimistic message logging and checkpointing". In *7th Annual ACM Symposium on Principles of Distributed Computing*, pages 171–181, Toronto (Canada), 1988.

[JOH 93] D. JOHNSON. "Efficient transparent optimistic rollback recovery for distributed application programs". Technical report, Pittsburgh, PA, USA, 1993.

[KAL 93] L. KALE, and S. KRISHNAN. "Charm++ : a portable concurrent object oriented system based on c++". In *OOPSLA '93 : Proceedings of the eighth annual conference on Object-oriented programming systems, languages, and applications*, pages 91–108, New York, NY, USA, 1993. ACM Press.

[KIC 01] G. KICZALES, and E. HILSDALE. "Aspect-oriented programming". In *ESEC/FSE-9 : Proceedings of the 8th European software engineering conference held jointly with 9th ACM SIGSOFT international symposium on Foundations of software engineering*, page 313, New York, NY, USA, 2001. ACM Press.

[KIL 99] M. KILLIJIAN, J. RUIZ-GARCIA, and J. FABRE. Using compile-time reflection for objects'state capture. *Lecture Notes in Computer Science*, 1616 :150–160, 1999.

[LAM 78] L. LAMPORT. "Time, clocks, and the ordering of events in a distributed system". In *Communications of the ACM*, volume 21, pages 558–565, Juillet 1978.

[LAM 04] L. LAMPORT, and M. MASSA. "Cheap paxos". In *DSN '04 : Proceedings of the 2004 International Conference on Dependable Systems and Networks (DSN'04)*, page 307, Washington, DC, USA, 2004. IEEE Computer Society.

[LEM 04] P. LEMARINIER, A. BOUTEILLER, T. HERAULT, G. KRAWEZIK, and F. CAPPELLO. "Improved message logging versus improved coordinated checkpointing for fault tolerant mpi". In *CLUSTER '04 : Proceedings of the 2004 IEEE International Conference on Cluster Computing*, pages 115–124, Washington, DC, USA, 2004. IEEE Computer Society.

[LEO 94] H. LEONG, and D. AGRAWAL. "Using message semantics to reduce rollback in optimistic message logging recovery schemes". In *14th International Conference on Distributed Computing Systems*, pages 227–234, Juin 1994.

[LEW 03] M. LEWIS, A. FERRARI, M. HUMPHREY, J. KARPOVICH, M. MORGAN, A. NATRAJAN, A. NGUYEN-TUONG, G. WASSON, and A. GRIMSHAW. Support for extensibility and site autonomy in the legion grid system object model. *Journal of Parallel Distributed Computing*, 63(5) :525–538, 2003.

[LIN 03a] C. LIN, S. WANG, and S. KUO. An efficient time-based checkpointing protocol for mobile computing systems over mobile ip. *Mob. Netw. Appl.*, 8(6) :687–697, 2003.

[LIN 03b] LINUXNETWORX. http ://www.linuxnetworx.com/, 2003.

[MAN 96] D. MANIVANNAN, and M. SINGHAL. "A low-overhead recovery technique using quasi-synchronous checkpointing". In *ICDCS '96 : Proceedings of the 16th International Conference on Distributed Computing Systems (ICDCS '96)*, page 100, Washington, DC, USA, 1996. IEEE Computer Society.

[MAN 99] D. MANIVANNAN, and MUKESH SINGHAL. Quasi-synchronous checkpointing : Models, characterization, and classification. *IEEE Trans. Parallel Distrib. Syst.*, 10(7) :703–713, 1999.

[MAN 02] D. MANIVANNAN, and M. SINGHAL. Asynchronous recovery without using vector timestamps. *J. Parallel Distributed Computing*, 62(12) :1695–1728, 2002.

[MAR 03] O. MARIN, M. BERTIER, and P. SENS. Darx - a framework for the fault-tolerant support of agent software. *issre*, 00 :406, 2003.

[MAT 89] F. MATTERN. "Virtual time and global states of distributed systems". In *Parallel and Distributed Algorithms : proceedings of the International Workshop on Parallel and Distributed Algorithms*. 1989.

[MAT 92] F. MATTERN. On the relativistic structure of logical time in distributed systems, 1992.

[MIT 00] N. MITTAL, and V. GARG. "Debugging distributed programs using controlled re-execution". In *PODC '00 : Proceedings of the nineteenth annual ACM symposium on Principles of distributed computing*, pages 239–248, New York, NY, USA, 2000. ACM Press.

[MUT 02] A. MUTHITACHAROEN, R. MORRIS, T. GIL, and B. CHEN. "Ivy : A read/write peer-to-peer file system". In *Proceedings of 5th Symposium on Operating Systems Design and Implementation*, 2002.

[NAK 04] H. NAKAMURA, T. HAYASHIDA, M. KONDO, Y. TAJIMA, M. IMAI, and T. NANYA. "Skewed checkpointing for tolerating multi-node failures.". In *23rd International Symposium on Reliable Distributed Systems (SRDS 2004), 18-20 Octobre 2004, Florianpolis, Brazil*, pages 116–125. IEEE Computer Society, 2004.

[NET 95] R. NETZER, and J/ XU. Necessary and sufficient conditions for consistent global snapshots. *IEEE Trans. Parallel Distrib. Syst.*, 6(2) :165–169, 1995.

[NIE 05] R. NIEUWPOORT, J. MAASSEN, G. WRZESISKA, R. HOFMAN, C. JACOBS, T. KIELMANN, and H. BAL. Ibis : a flexible and efficient java-based grid programming environment : Research articles. *Concurr. Comput. : Pract. Exper.*, 17(7-8) :1079–1107, 2005.

[PLA 93] J. PLANCK. "*Efficient Checkpointing on MIMD Architecures*". PhD thesis, Princeton University, Juin 1993.

[PLA 95a] J. PLANK, M. BECK, G. KINGSLEY, and K. LI. "Libckpt : Transparent Checkpointing under Unix". In *Proceedings of USENIX Winter1995 Technical Conference*, pages 213–224, New Orleans, Louisiana/U.S.A., Janvier 1995.

[PLA 95b] J. PLANK, J. XU, and R. NETZER. "Compressed differences : An algorithm for fast incremental checkpointing". Technical Report CS-94-302, University of Tennessee, 1995.

[PLA 98] J. S. PLANK, K. LI, and M. A. PUENING. Diskless checkpointing. *IEEE Transactions on Parallel and Distributed Systems*, 9(10) :972–986, Octobre 1998.

[RAN 75] B. RANDELL. "System structure for software fault tolerance". In *Proceedings of the international conference on Reliable software*, pages 437–449, New York, NY, USA, 1975. ACM Press.

[RAO 00] S. RAO, L. ALVISI, and H. VIN. The cost of recovery in message logging protocols. *IEEE Transactions on Knowledge and Data Engineering*, 12(2) :160–173, 2000.

[SAN 05] S. SANKARAN, J. SQUYRES, B. BARRETT, A. LUMSDAINE, J. DUELL, P. HARGROVE, and E. ROMAN. The LAM/MPI checkpoint/restart framework : System-initiated checkpointing. *International Journal of High Performance Computing Applications*, 19(4) :479–493, Winter 2005.

[SCH 04] M. SCHULZ, G. BRONEVETSKY, R. FERNANDES, D. MARQUES, K. PINGALI, and P. STODGHILL. "Implementation and evaluation of a scalable application-level checkpoint-recovery scheme for mpi programs". In *SC '04 : Proceedings of the 2004 ACM/IEEE conference on Supercomputing*, page 38, Washington, DC, USA, 2004. IEEE Computer Society.

[SEK 99] T. SEKIGUCHI, H. MASUHARA, and A. YONEZAWA. "A simple extension of java language for controllable transparent migration and its portable implementation". In *COORDINATION '99 : Proceedings of the Third International Conference on Coordination Languages and Models*, pages 211–226, London, UK, 1999. Springer-Verlag.

[SEN 95] P. SENS. "The performance of independent checkpointing in distributed systems". In *HICSS '95 : Proceedings of the 28th Hawaii International Conference on System Sciences (HICSS'95)*, page 525, Washington, DC, USA, 1995. IEEE Computer Society.

[SEN 00] P. SENS. Contribution à l'intégration de la tolérance aux fautes dans les environnements répartis, 2000. Thèse d'Habilitation de l'Université Pierre et Marie Curie.

[SIL 98] L. SILVA, and J. SILVA. "System-level versus user-defined checkpointing". In *SRDS '98 : Proceedings of the The 17th IEEE Symposium on Reliable Distributed Systems*, page 68, Washington, DC, USA, 1998. IEEE Computer Society.

[SIL 99a] L. SILVA, and J. SILVA. "The performance of coordinated and independent checkpointing". In *IPPS '99/SPDP '99 : Proceedings of the 13th International Symposium on Parallel Processing and the 10th Symposium on Parallel and Distributed Processing*, pages 280–284, Washington, DC, USA, 1999. IEEE Computer Society.

[SIL 99b] L. SILVA, and J. SILVA. Using message semantics for fast-output commit in checkpointing-and-rollback recovery, 1999.

[SIS 89] A. SISTLA, and J. WELCH. "Efficient distributed recovery using message logging". In *PODC '89 : Proceedings of the eighth annual ACM Symposium on Principles of distributed computing*, pages 223–238, New York, NY, USA, 1989. ACM Press.

[SOU 96] C. SOUZA, and S. THEROUDE. "Persistent java". In *Proceedings of the First International Workshop on Persistence and Java*, September 1996.

[STE 96] G. STELLNER. "Cocheck : Checkpointing and process migration for mpi". In *IPPS '96 : Proceedings of the 10th International Parallel Processing Symposium*, pages 526–531, Washington, DC, USA, 1996. IEEE Computer Society.

[STR 85] R. STROM, and S. YEMINI. Optimistic recovery in distributed systems. *ACM Trans. Comput. Syst.*, 3(3) :204–226, 1985.

[STR 88] R. STROM, D. BACON, and S. YEMINI. "Volatile logging in n-fault-tolerant distributed systems". In *Proc IEEE Fault-tolerant Computing Symposium*, pages 44–49, 1988.

[STR 98] V. STRUMPEN. Compiler technology for portable checkpoints, 1998.

[TAN 95] TODD TANNENBAUM, and MICHAEL LITZKOW. Checkpointing and migration of unix processes in the Condor distributed processing system. *Dr Dobbs Journal*, February 1995.

[TAR 98a] A. TARAFDAR, and V. GARG. "Adressing false causality while detecting predicates in distributed programs". In *18th International Conference on Distributed Computing Systems*, pages 94–101, Mai 1998.

[TAR 98b] A. TARAFDAR, and V. GARG. "Happened before is the wrong model for potential causality". Technical report TR-PDS-1998-006, Parallel and Dsitributed Systems Group, University of Texas, Juillet 1998.

[TAT 99] M. TATSUBORI, S. CHIBA, K. ITANO, and M. KILLIJIAN. "OpenJava : A class-based macro system for java". In *Reflection and Software Engineering*, Lecture Notes in Computer Science, pages 117–133, 1999.

[THA 05] DOUGLAS THAIN, TODD TANNENBAUM, and MIRON LIVNY. Distributed computing in practice : the condor experience. *Concurrency - Practice and Experience*, 17(2-4) :323–356, 2005.

[TRU 00] E. TRUYEN, B. ROBBEN, B. VANHAUTE, T. CONINX, W. JOOSEN, and P. VERBAETEN. "Portable support for transparent thread migration in java". In *ASA/MA*, pages 29–43, 2000.

[VAI 93] N. VAIDYA. Distributed recovery units : An approach for hybrid and adaptive distributed recovery, 1993.

[VAI 99] N. VAIDYA. Staggered consistent checkpointing. *IEEE Trans. Parallel Distrib. Syst.*, 10(7) :694–702, 1999.

[WAL 94] J. WALDO, G. WYANT, A. WOLLRATH, and S. KENDALL. "A note on distributed computing". Technical report SMLI TR-94-29, Sun Microsystems Labs, Novembre 1994.

[WAN 93] Y. WANG, Y. HUANG, and W.K. FUCHS. "Progressive retry for software error recovery in distributed systems". In *Proceedings of the IEEE Fault-Tolerant Computing Symposium (FTCS-23)*, pages 138–144, June 1993.

[WAN 97] Y. WANG. Consistent global checkpoints that contain a given set of local checkpoints. *IEEE Trans. Comput.*, 46(4) :456–468, 1997.

[ZAM 98] F. ZAMBONELLI. "On the effectiveness of distributed checkpoint algorithms for domino-free recovery". In *HPDC '98 : Proceedings of the The Seventh IEEE International Symposium on High Performance Distributed Computing*, page 124, Washington, DC, USA, 1998. IEEE Computer Society.

[ZHE 04] G. ZHENG, L. SHI, and L. KALE. "Ftc-charm++ : an in-memory checkpoint-based fault tolerant runtime for charm++ and mpi.". In *CLUSTER*, pages 93–103, 2004.

Annexe A

Preuves

Cette appendice présente les preuves des différents lemmes, propriétés et théorèmes définies dans le chapitre 5. Ces preuves ont été rédigées dans le cadre d'une collaboration avec Ludovic Henrio de l'équipe OASIS.

A.1 Preuves de la section 5.2 : $\mathcal{P}$-cohérence

A.1.1 Définition 5.2.4 : $\mathcal{C}_p^{\sqcup}$ et $\mathcal{C}_p^{\sqcap}$

Nous montrons ici que les définitions de $\mathcal{C}_p^{\sqcup}$ et $\mathcal{C}_p^{\sqcap}$ ci-dessous définissent bien respectivement la plus petite coupe cohérente de l'exécution $(E, \prec)$ qui contient tous les évènements e tels que $e \in_{\mathcal{P}} \mathcal{C}_p$ et la plus grande coupe cohérente de l'exécution $(E, \prec)$ contenue dans $\mathcal{C}_p$.

$$\mathcal{C}_p^{\sqcup} = \{e | \exists e' \in_{\mathcal{P}} \mathcal{C}_p,\, e \preceq e'\}$$
$$\mathcal{C}_p^{\sqcap} = \{e | \forall e', e' \preceq e \Rightarrow e' \in \mathcal{C}_p\}$$

Preuve : D'abord, comme l'ensemble des coupes cohérentes forme un treillis, $\mathcal{C}_p^{\sqcap}$ et $\mathcal{C}_p^{\sqcup}$ peuvent bien être définies comme la plus petite coupe cohérente de $(E, \prec)$ contenant tous les e tels que $e \in_{\mathcal{P}} \mathcal{C}_p$ et la plus grande coupe cohérente plus petite que $\mathcal{C}_p$.

- Définition $\mathcal{C}_p^{\sqcup} = \{e | \exists e' \in_{\mathcal{P}} \mathcal{C}_p,\, e \preceq e'\} : \{e | \exists e' \in_{\mathcal{P}} \mathcal{C}_p,\, e \preceq e'\}$ est une coupe cohérente, et contient tous les évènements $e \in_{\mathcal{P}} \mathcal{C}_p$; la minialité de la coupe est triviale.
- Définition $\mathcal{C}_p^{\sqcap} = \{e | \forall e', e' \preceq e \Rightarrow e' \in \mathcal{C}_p\} : \{e | \forall e', e' \preceq e \Rightarrow e' \in \mathcal{C}_p\}$ est cohérente par transitivité de $\prec$. Elle est inclue dans $\mathcal{C}_p$. Concernant la maximalité de la coupe, si $\{e | \forall e', e' \preceq e \Rightarrow e' \in \mathcal{C}_p\} \subset \mathcal{C}$ avec $\mathcal{C}$ cohérente, alors considérons un évènement $e'' \in \mathcal{C} \backslash \{e | \forall e', e' \preceq e \Rightarrow e' \in \mathcal{C}_p\}$, il existe $e_0, e_0 \preceq e'' \wedge e_0 \notin \mathcal{C}_p$, et à cause de la cohérence de $\mathcal{C}$, $e_o \in \mathcal{C}$. Par conséquent $\mathcal{C}$ ne peut pas être plus petite que $\mathcal{C}_p$.

□

A.1.2 Propriété 5.2.2 : $\mathcal{C}_p^{\sqcup}$ et $\mathcal{C}_p^{\sqcap}$

Pour toute coupe $\mathcal{P}$-cohérente $\mathcal{C}_p$, on a :
- $e \in \mathcal{C}_p^{\sqcup} \wedge Det(e) \Rightarrow e \in_{\mathcal{P}} \mathcal{C}_p$
- $\mathcal{C}_p^{\sqcap} \xrightarrow{*} \mathcal{C}_p^{\sqcup}$

Preuve :
- Propriété $e \in \mathcal{C}_p^{\sqcup} \wedge Det(e) \Rightarrow e \in_{\mathcal{P}} \mathcal{C}_p$: Soit $e \in \mathcal{C}_p^{\sqcup}$, avec $Det(e)$, on a $\exists e' \in_{\mathcal{P}} \mathcal{C}_p$, $e \preceq e'$, donc d'après la définition 5.2.3, on a $e \in_{\mathcal{P}} \mathcal{C}_p$.
- Propriété $\mathcal{C}_p^{\sqcap} \xrightarrow{*} \mathcal{C}_p^{\sqcup}$: Triviale puisque les deux coupes sont deux coupes cohérentes d'une seule et même exécution et que $\mathcal{C}_p^{\sqcap} \subseteq \mathcal{C}_p^{\sqcup}$.

□

A.1.3 Propriété 5.2.3 : La réduction conserve la $\mathcal{P}$-cohérence

Nous prouvons ici que si $\mathcal{C}_p$ est une coupe $\mathcal{P}$-cohérente de l'exécution $(E, \prec)$, alors

$$\mathcal{C}_p \xrightarrow{e} \mathcal{C}'_p \Rightarrow \exists (E', \prec') \text{ telle que } \mathcal{C}'_p \text{ est une coupe } \mathcal{P}\text{-cohérente de } (E', \prec')$$

Preuve : Cette preuve développe les cas non-triviaux. En particulier, les cas mettant en jeu la clôture transitive de la relation de causalité ne sont pas développés ici : ils se prouvent par récurrence sur la longueur de l'inférence nécessaire pour retrouver directement $e \prec e'$.
- Si $Det(e)$, $\mathcal{C}_p^{\sqcap} \xrightarrow{e} \mathcal{C}'$, alors il existe une exécution équivalente à celle qui a généré E telle que $S \xrightarrow{*} \mathcal{C}_p^{\sqcap} \xrightarrow{e} \mathcal{C}'$. $\mathcal{C}_p^{\sqcap}$ est une coupe de E et e est minimal dans $(E, \prec^0) \backslash \mathcal{C}_p^{\sqcap})$. $\mathcal{C}'$ est cohérente.
  Nous prouvons d'abord que $\mathcal{C}'_p$ est une $\mathcal{P}$-coupe. Comme on a $Det(e)$, supposons e' tel que $e' \prec_i^0 e$. Alors $e' \in \mathcal{C}' \Rightarrow e' \in \mathcal{C}'_p$ ($\mathcal{C}'$ est cohérente et $\mathcal{C}' \subseteq \mathcal{C}'_p$ par construction). Si $e \prec_i^0 e'$ avec $e' \in \mathcal{C}'_p$, alors $Det(e')$ (sinon $\mathcal{C}_p$ ne serait pas une $\mathcal{P}$-coupe), et $Det(e)$ est suffisant pour conclure que $\mathcal{C}'_p$ est une $\mathcal{P}$-coupe.
  Ensuite, nous prouvons que $\prec'$ est l'ordre induit par l'ordre de causalité potentielle $\prec^0$ sur $\mathcal{C}'_p$. Supposons que $e' \prec^0 e$. Alors $e' \in \mathcal{C}' \backslash \{e\} = \mathcal{C}_p^{\sqcap}$ donc $e' \in \mathcal{C}_p$, et $e' \prec e$ ou $e' \bowtie e$ par définition de la réduction. De plus, $e' \prec^0 e \Rightarrow (e' \bowtie e)$ (Définition 5.1.4), donc finalement $e' \prec' e$. Si on suppose que $e \prec^0 e'$, alors $e' \notin \mathcal{C}_p^{\sqcap}$ donc $e \in_{\mathcal{P}} \mathcal{C}_p$ implique que $e \prec' e'$ et $\prec^0 \Rightarrow \prec'$.
  Réciproquement, supposons que $e' \prec' e$, alors $e' \in \mathcal{C}_p^{\sqcap}$, donc il existe une exécution $(E, \prec^0)$ telle que $S^0 \xrightarrow{*e'*} \mathcal{C}_p^{\sqcap} \xrightarrow{e} \mathcal{C}'$. La définition 5.1.4 implique soit $e \bowtie e'$ ou $e' \prec^0 e$. Sinon, si $e \prec' e'$ alors $e' \notin \mathcal{C}_p^{\sqcap}$, donc, comme e est minimal dans $E \backslash \mathcal{C}_p^{\sqcap}$ (Définition 5.1.5), une des exécutions E est $\mathcal{C}_p^{\sqcap} \xrightarrow{e*e'}$ donc soit $e \bowtie e'$ ou $e \prec^0 e'$.
  Finalement, $\mathcal{C}'_p$ est une coupe $\mathcal{P}$-cohérente de $(E, \prec^0)$ puisque $e \prec^0 e' \Rightarrow Det(e)$ et $e' \prec^0 e \Rightarrow e' \in \mathcal{C}' \Rightarrow e' \in \mathcal{C}'_p$ ($\mathcal{C}'$ est cohérente et $\mathcal{C}' \subseteq \mathcal{C}'_p$ par construction).
- Si $Det(e) \wedge \exists \mathcal{P}(e_0) \in \mathcal{C}_p$ tel que $\mathcal{P}(e_0) \lhd e$, alors premièrement $\mathcal{C}'_p$ est une $\mathcal{P}$-coupe de $(E, \prec)$, définition 5.2.2 impose les mêmes restrictions sur e_0 et $\mathcal{P}(e_0)$. Deuxièmement, $e_0 \prec^0 e' \Leftrightarrow \mathcal{P}(e_0) \prec e' \Leftrightarrow e \prec' e'$, et même chose pour $e' \prec \mathcal{P}(e_0)$. Donc $\prec'$ est l'ordre induit par l'ordre de causalité potentielle $\prec^0$ sur $\mathcal{C}'_p$ (c'est le même ordre que $\prec$ avec $\mathcal{P}(e_0)$ remplacé par e_0).

Finalement, $\mathcal{C}'_p$ est une coupe $\mathcal{P}$-cohérente de $(E, \prec)$ car la définition 5.2.3 impose les mêmes restrictions sur e_0 et $\mathcal{P}(e_0)$.

– Enfin si $\mathcal{C}_p^{\sqcap} \xrightarrow{e} \mathcal{C}' \wedge Det(e) \wedge \nexists \mathcal{P}(e_0) \in \mathcal{C}_p$ tel que $\mathcal{P}(e_0) \lhd e$, alors les propriétés 5.2.1 et 5.2.2 assurent que $S \xrightarrow{*} \mathcal{C}_p^{\sqcup} \xrightarrow{e} \mathcal{C}''$, $\mathcal{C}''$ est une coupe d'une exécution possible $(E', \prec^1)$.
D'abord, $\mathcal{C}'_p$ est une $\mathcal{P}$-coupe de $(E', \prec^1)$. En effet, $e' \prec^1 e \Rightarrow e' \in \mathcal{C}_p^{\sqcup}$, alors soit $Det(e')$ ou $e' \in_{\mathcal{P}} \mathcal{C}_p$ (propriété 5.2.2), et si $e \prec^1 e'$ alors $e' \notin \mathcal{C}'_p$.
De plus, $\prec'$ est l'ordre induit par l'ordre de causalité potentielle $\prec^1$ sur $\mathcal{C}'_p$. Supposons que $e' \prec^1 e$, avec $e' \in_{\mathcal{P}} \mathcal{C}'_p$, alors $e' \in_{\mathcal{P}} \mathcal{C}_p$. La définition 5.1.4 implique $e' \bowtie e$ et donc $e' \prec' e$. Supposons $e \prec^1 e'$, alors $e' \notin \mathcal{C}_p^{\sqcup}$ et donc $e' \notin_{\mathcal{P}} \mathcal{C}_p$, ce qui est contradictoire. Réciproquement, si $e' \prec' e$ (on ne peut pas avoir $e \prec' e'$) alors $e' \bowtie e$ et $e' \in \mathcal{C}_p^{\sqcup}$. La définition 5.1.4 implique $e' \prec^1 e$.
Finalement, nous prouvons la $\mathcal{P}$-cohérence. si $e' \prec^1 e$ alors $e' \in \mathcal{C}_p^{\sqcup}$ et la propriété 5.2.2 implique $Det(e') \vee e' \in_{\mathcal{P}} \mathcal{C}_p \subseteq \mathcal{C}'_p$. Si $e \prec^1 e'$ alors $e' \notin \mathcal{C}_p^{\sqcup}$, et $e' \notin \mathcal{C}_p$, et $e' \notin \mathcal{C}'_p$ car $\mathcal{C}'_p = \mathcal{C}_p \cup \{e\}$. Donc $\mathcal{C}'_p$ est une coupe $\mathcal{P}$-cohérente de l'exécution $(E', \prec'')$ (définition 5.2.3).

□

A.1.4 Théorème 5.2.1 : Exécution possible

Il existe une exécution depuis la coupe $\mathcal{C}_p$ vers la coupe cohérente $\mathcal{C}_p^{\sqcup}$:

$$\mathcal{C}_p \xrightarrow{*} \mathcal{C}_p^{\sqcup}$$

Preuve : Nous allons prouver que pour toute coupe $\mathcal{C}_p$ $\mathcal{P}$-cohérent, si $\mathcal{C}_p$ n'est pas cohérente alors :

$$\exists \mathcal{C}'_p \exists e \in \mathcal{C}_p^{\sqcup} \backslash \mathcal{C}_p, \mathcal{C}_p \xrightarrow{e} \mathcal{C}'_p$$

Donc par récurrence, il existe une façon de réduire $\mathcal{C}_p$ vers $\mathcal{C}_p^{\sqcup}$.

$\mathcal{C}_p^{\sqcup} \neq \mathcal{C}_p^{\sqcap}$, et $\mathcal{C}_p^{\sqcup}$ et $\mathcal{C}_p^{\sqcap}$ étant des coupes de la même exécution, $\exists e \in \mathcal{C}_p^{\sqcup} \backslash \mathcal{C}_p^{\sqcap}$, $\mathcal{C}_p^{\sqcap} \xrightarrow{e*} \mathcal{C}_p^{\sqcup}$. De plus, e est soit déterministe, soit correspond à une promesse minimale dans $\mathcal{C}_p$, et $\mathcal{C}_p \xrightarrow{e} \mathcal{C}'_p$. En effet, $\mathcal{C}_p^{\sqcap} \xrightarrow{e*} \mathcal{C}_p^{\sqcup}$ implique que e est minimal dans $\mathcal{C}_p^{\sqcup} \backslash \mathcal{C}_p^{\sqcap}$ (propriété 5.1.1) alors $e \notin \mathcal{C}_p$ (sinon nous aurions $e \in \mathcal{C}_p^{\sqcap}$), donc $\mathcal{C}_p \xrightarrow{e} \mathcal{C}'_p$. De plus, si $Det(e)$ alors $\mathcal{P}(e) \in \mathcal{C}_p$ (propriété 5.2.2) et $\mathcal{P}(e) \lhd e$. □

Notons de plus que la réduction ci-dessus réduit toujours un évènement soit déterministe, soit correspondant à une promesse minimale dans $\mathcal{C}_p$.

A.1.5 Théorème 5.2.2 : Exécution contrôlée

$$\mathcal{C}_p \xrightarrow{*} \mathcal{C}'_p \Rightarrow \exists \mathcal{C}' \text{ cohérente}, \mathcal{C}_p^{\sqcup} \xrightarrow{*} \mathcal{C}' \wedge \mathcal{C}'_p \xrightarrow{*} \mathcal{C}'$$

Plus précisément, nous allons prouver que :

$$\mathcal{C}_p \xrightarrow{*} \mathcal{C}'_p \Rightarrow \mathcal{C}_p^{\sqcup} \xrightarrow{*} \mathcal{C}_p'^{\sqcup}$$

En effet, le théorème 5.2.1 assure que $\mathcal{C}_p \xrightarrow{*} \mathcal{C}_p^{\sqcup}$, et $\mathcal{C}_p \xrightarrow{*} \mathcal{C}_p'^{\sqcup}$, donc $\mathcal{C}_p'^{\sqcup}$ est une $\mathcal{C}'$ possible.

Preuve : Nous prouvons par récurrence sur la longueur de la réduction $\mathcal{C}_p \xrightarrow{*} \mathcal{C}'_p$. Si la longueur est zéro alors $\mathcal{C}_p = \mathcal{C}'_p$. Supposons $\mathcal{C}_p \xrightarrow{*} \mathcal{C}'_p \wedge \mathcal{C}_p^{\sqcup} \xrightarrow{*} \mathcal{C}_p'^{\sqcup}$, soit $\mathcal{C}'_p \xrightarrow{e} \mathcal{C}''_p$, $\mathcal{C}''_p$ une coupe $\mathcal{P}$-cohérente d'une exécution $(E'', \prec'')$ (propriété 5.2.3) :

- Si $e \in \mathcal{C}_p'^{\sqcup}$, alors $Det(e) \vee \mathcal{P}(e) \in \mathcal{C}'_p$ (propriété 5.2.2), et donc $\mathcal{C}_p'^{\sqcup} = \mathcal{C}_p''^{\sqcup}$. En effet, les deux sont des coupes cohérentes de la même exécution (si $Det(e)$ ou $\mathcal{P}(e) \in \mathcal{C}'_p$, la preuve de la propriété 5.2.3 assure que l'exécution de référence reste la même). Finalement $\mathcal{C}_p^{\sqcup} \xrightarrow{*} \mathcal{C}_p''^{\sqcup}$ par hypothèse de récurrence.
- Si $e \notin \mathcal{C}_p'^{\sqcup}$, alors il existe une coupe $\mathcal{C}_1$ de l'exécution $(E'', \prec'')$ telle que $\mathcal{C}_p'^{\sqcup} \xrightarrow{e} \mathcal{C}_1$:
  - Si $Det(e)$: e est compatible avec tous les autres évènements $\mathcal{C}_p'^{\sqcup} \backslash \mathcal{C}_p'^{\sqcap}$: $\mathcal{C}'_p \xrightarrow{e} \mathcal{C}''_p \Rightarrow \exists \mathcal{C}_2, \mathcal{C}_p'^{\sqcap} \xrightarrow{e} \mathcal{C}_2$, et, comme, $\mathcal{C}_p'^{\sqcap} \xrightarrow{*} \mathcal{C}_p'^{\sqcup}$, on a $\exists \mathcal{C}_1, \mathcal{C}_p'^{\sqcup} \xrightarrow{e} \mathcal{C}_1$, avec $(E'', \prec'') = (E', \prec')$.
  - Sinon $Det(e)$: $\exists \mathcal{C}_2, \mathcal{C}_p'^{\sqcap} \xrightarrow{e} \mathcal{C}_2$ et $\mathcal{C}_p'^{\sqcap} \xrightarrow{*} \mathcal{C}_p'^{\sqcup}$, la propriété 5.2.1 assure que $\exists \mathcal{C}_1, \mathcal{C}_p'^{\sqcup} \xrightarrow{e} \mathcal{C}_1$ (puisqu'il n'existe pas de e' dans $\mathcal{C}_p^{\sqcup}$ tel que $\mathcal{P}(e') \lhd e$).

  Finalement $\mathcal{C}_1 = \mathcal{C}_p''^{\sqcup}$ car $\mathcal{C}''_p \subseteq \mathcal{C}_1$, $\mathcal{C}_1$ est cohérente, et $\mathcal{C}_1$ diffère seulement de $\mathcal{C}_p'^{\sqcup}$ par l'ajout de e (si il y avait une coupe cohérente contenant $\mathcal{C}''_p$ plus petite que $\mathcal{C}_1$, on aurait une contradiction avec la minialité de $\mathcal{C}_p'^{\sqcup}$).

□

A.1.6 Propriété 5.2.4 : Réduction de coupes $\mathcal{P}$-cohérentes incluses

Soient $\mathcal{C}_p$ et $\mathcal{C}'_p$ deux coupes $\mathcal{P}$-cohérentes *d'une même exécution* $(E, \prec)$, alors on a :

$$\mathcal{C}_p \subseteq \mathcal{C}'_p \Rightarrow \exists \mathcal{C}''_p \text{ coupe } \mathcal{P}\text{-cohérente de } (E, \prec),\ \mathcal{C}_p \xrightarrow{*} \mathcal{C}''_p \wedge \mathcal{C}'_p \xrightarrow{*} \mathcal{C}''_p$$

Preuve : Notons tout d'abord que $\mathcal{C}_p^{\sqcup} \subseteq \mathcal{C}_p'^{\sqcup}$. On a $\mathcal{C}_p \xrightarrow{*} \mathcal{C}_p^{\sqcup}$ (théorème 5.2.1), donc $\mathcal{C}_p \xrightarrow{*} \mathcal{C}_p^{\sqcup} \xrightarrow{*} \mathcal{C}_p'^{\sqcup}$. Le théorème 5.2.1 assure aussi que $\mathcal{C}_p'^{\sqcup} \xrightarrow{*} \mathcal{C}_p'^{\sqcup}$. Finalement, $\mathcal{C}_p'^{\sqcup}$ est une coupe $\mathcal{C}''_p$ possible. □

Notons de plus que la réduction menant à $\mathcal{C}''_p$ peut suivre celle définie en théorème 5.2.1 : on réduit toujours un évènement soit déterministe, soit correspondant à une promesse minimale dans $\mathcal{C}_p$.

A.2 Preuves de la section 5.3 : Cadre pratique

A.2.1 Propriété 5.3.1 : Équivalence entre réduction locale et globale

Sous les hypothèses sur les communications 5.3.3 et 5.3.4, les hypothèses 5.3.2 and 5.3.7 sont équivalentes à la propriété suivante :

Soient les états globaux $S = \{s_0 \dots s_i \dots s_n\}$ et $S' = \{s_0 \dots s'_i \dots s_n\}$, alors

$$s_i \xrightarrow{e}_L s'_i \Leftrightarrow S \xrightarrow{e} S'$$

Preuve : Nous prouvons d'abord que les hypothèses 5.3.2 et 5.3.7 implique $s_i \xrightarrow{e}_L s'_i \Leftrightarrow S \xrightarrow{e} S'$ pour tout les évènements sauf les réceptions de message en transit :

– $s_i \xrightarrow{e}_L s'_i \Leftarrow S \xrightarrow{e} S'$. Supposons $S \xrightarrow{e} S'$:
Si e n'est pas une réception de message ($\nexists e', (e', e) \in \Gamma$) alors l'hypothèse 5.3.7 implique $s_i \xrightarrow{e}_L s'_i$.
Sinon, e est une réception de message ($\exists e', (e', e) \in \Gamma$). Nous avons par hypothèse $S \xrightarrow{e} S'$ et $e \notin S$. De plus, $S \xrightarrow{e} S'$ implique (définition 5.3.1) $Odt_{\prec}(e') \subseteq s_i$. En effet, la définition 5.3.1 rend la réduction $S \xrightarrow{e} S'$ impossible, alors qu'il existe $e_0 \in Odt_{\prec}(e')$ qui n'a pas encore été réduit.
Finalement, on a $s_i \xrightarrow{e}_L s'_i$.

– $s_i \xrightarrow{e}_L s'_i \Rightarrow S \xrightarrow{e} S'$. Supposons $s_i \xrightarrow{e} s'_i$:
Si e n'est pas une réception de message ($\nexists e', (e', e) \in \Gamma$), l'hypothèse 5.3.7 implique $\exists S_0,\ S_0 \xrightarrow{e} S'_0$ avec $S_0\big|_i = s_i$, et l'hypothèse 5.3.2, comme $S'\big|_i = s_i$ implique $S \xrightarrow{e} S'$ avec $S'_0\big|_i = S'\big|_i = s'_i$ et $i \neq j \Rightarrow S'\big|_j = S\big|_j$.
Sinon, e est une réception de message ($\exists e', (e', e) \in \Gamma$). Alors $Odt_{\prec}(e') \subseteq s_i$. De plus, $s_i \xrightarrow{e}_L s'_i$ implique (hypothèse 5.3.4) $\exists S,\ S \xrightarrow{*} S' \wedge s_j \xrightarrow{e'}_L s'_j$, avec $s'_j = S'\big|_j$, ce qui implique en utilisant le cas où e n'est pas une réception appliqué à e' : $\exists S_0, S_1\ S_0 \xrightarrow{e'} S_1$ avec $S_1\big|_j = s'_j$. Alors, l'hypothèse 5.3.3 et le fait que $e \notin S$ (car $e \notin s_i$) implique $\exists S_2 \supseteq S, S_3\ S_2 \xrightarrow{e} S_3$, donc il existe un n tel que $S \xrightarrow{e_0} S'_1 \xrightarrow{e_1} \dots S'_n \xrightarrow{e_n} S_2 \xrightarrow{e} S_3$. Supposons que n est minimal, c'est-à-dire $\forall k \leq n,\ S'_k \xrightarrow{e}$, dans ce cas, de par la définition de $\prec$ (définition 5.1.2), si $n > 0$ alors nous considérons le premier e_k exécuté sur A_i, ce qui implique $S'_k\big|_i = S\big|_i\ e_k \prec e$. On a $S_k \xrightarrow{e}$, $S'_k\big|_i = S\big|_i$, $\exists S_0,\ S_0 \xrightarrow{e} S'_0$ avec $S_0\big|_i = S\big|_i$ donc par définition de $Odt_{\prec}(e)$ (définition 5.3.1), $e_k \in Odt_{\prec}(e)$ ce qui est contradictoire avec $Odt_{\prec}(e) \subseteq s_i$.
Finalement, $n = 0$ et $S \xrightarrow{e} S'$.

Deuxièmement, nous prouvons que la relation $s_i \xrightarrow{e}_L s'_i \Leftrightarrow S \xrightarrow{e} S'$ pour tout les évènements sauf les réceptions de message en transit implique les hypothèses 5.3.2 et 5.3.7 :

– hypothèse 5.3.2 : Considérons un évènement e qui n'est pas une réception de message($\nexists e', (e', e) \in \Gamma$). Supposons $S \xrightarrow{e} S_1$ et $S\big|_i = S'\big|_i$, alors $s_i \xrightarrow{e}_L s'_i$ avec $s'_i = S_1\big|_i$. S_1 est de la forme $S_1 = \{s_0 \dots s_i \dots s_n\}$. Soit $S'_1 = \{s_0 \dots s'_i \dots s_n\}$ alors $S_1 \xrightarrow{e} S'_1$.

– hypothèse 5.3.7 : D'abord, supposons $s_i \xrightarrow{e}_L s'_i$ alors on a, pour tout S tel que $S = \{s_0 \dots s_i \dots s_n\}$, $S \xrightarrow{e} S'$, ce qui est suffisant pour les évènements qui ne sont pas des réceptions. Si e est une réception ($(e', e) \in \Gamma$) alors on a en plus $Odt_{\prec}(e') \subseteq s_i$ car $S \xrightarrow{e} S'$, ce qui est suffisant.
Supposons maintenant

$$\exists S, S',\ S\big|_i = s_i \wedge S \xrightarrow{e} S' \quad \wedge \quad (e', e) \in \Gamma \Rightarrow Odt_{\prec}(e') \subseteq s_i$$

Si e n'est pas une réception ($\nexists e', (e', e) \in \Gamma$), $S \xrightarrow{e} S'$ implique $s_i \xrightarrow{e} s'_i$ (s_i peut appartenir à n'importe quel S tel que $S\big|_i = s_i$).
Sinon, on a $\exists e',\ (e', e) \in \Gamma$; tous les S tels que $S\big|_i = s_i$ ne sont plus équivalents. Supposons un état global cohérent S_0 avec $S_0\big|_i = s_i$. L'hypothèse 5.3.3 et $e \notin S$ impliquent $\exists S_2 \supseteq S, S'_2,\ S_2 \xrightarrow{e} S'_2$. Donc il existe un n tel que $S \xrightarrow{e_0} S'_1 \xrightarrow{e_1} \dots S_n \xrightarrow{e_n} S_2 \xrightarrow{e} S'_2$. Comme de plus $Odt_{\prec}(e') \subseteq s_i \subseteq S$, alors le

même raisonnement que le cas précédent amène à $S \xrightarrow{e} S'$. Finalement nous pouvons conclure : $s_i \xrightarrow{e}_L s'_i \Leftrightarrow S \xrightarrow{e} S'$.

□

A.2.2 Propriété 5.3.5 :La réduction locale maintient la $\mathcal{P}$-cohérence

Nous prouvons ici que si $\mathcal{C}_p$ est une coupe $\mathcal{P}$-cohérente de l'exécution $(E, \prec)$, alors

$$\mathcal{C}_p \xrightarrow{e}_L \mathcal{C}'_p \Rightarrow \exists (E', \prec') \text{ tel que } \mathcal{C}'_p \text{ est une coupe } \mathcal{P}\text{-cohérente de } (E', \prec')$$

Pour prouver que la réduction locale maintient la $\mathcal{P}$-cohérence, nous posons le lemme suivant :

Lemme A.2.1 *Tout évènement qui précéde causalement la réception déterministe d'un message soit précéde l'envoi du message, soit précede un évènement de $Odt_{\prec}(e')$, soit appartient à tout état dans lequel e' peut être réduit :*

$$(e, e') \in \Gamma \wedge e_1 \prec e' \wedge Det(e') \Rightarrow e_1 \prec e \vee (\exists e_2 \in Odt_{\prec}(e),\ e_1 \prec e_2) \vee (\forall c_i,\ c_i^{\sqcap} \xrightarrow{e'} s'_i \Rightarrow e_1 \in, c_i^{\sqcap})$$

Ce lemme est nécessaire pour pouvoir recevoir à n'importe quel moment de la réexécution une réception déterministe. La preuve se base sur le fait que tous les évènements qui pourraient avoir lieu à la place d'une réception sont dans $Odt_{\prec}$ ou avant l'envoi. Donc tout évènement qui précède une réception déterministe, s'il pouvait être réduit depuis l'état local, appartient à $Odt_{\prec}$ ou précède l'envoi.

Nous prouvons ci-dessous la propriété 5.3.5.

Preuve : Supposons $\mathcal{C}_p \xrightarrow{e}_L \mathcal{C}'_p$, $c_i = \mathcal{C}_p\big|_i$, et que la réduction a lieu sur l'activité A_i. Nous supposons que $\mathcal{C}_p$ est une coupe cohérente de l'exécution $(E, \prec)$ et nous prouvons que $\mathcal{C}'_p$ est une coupe cohérente de l'exécution $(E', \prec')$.

Notons qu'on ne peut pas utiliser la propriété 5.2.3 car la réduction d'une coupe $\mathcal{P}$-cohérente repose sur la coupe cohérente $\mathcal{C}_p^{\sqcap}$ qui n'est pas la même que $\{c_1^{\sqcap}, \ldots, c_n^{\sqcap}\}$, et que donc, $\mathcal{C}_p \xrightarrow{e}_L \mathcal{C}'_p$ *n'implique pas* $\mathcal{C}_p \xrightarrow{e} \mathcal{C}'_p$.

- Si $Det(e)$: D'abord, l'hypothèse 5.3.9 implique $\forall e_0 \prec e,\ e_0 \in A_i \Rightarrow e_0 \in \mathcal{C}_p$, ce qui assure que $\mathcal{C}'_p$ est une $\mathcal{P}$-coupe.
  De plus, si e n'est pas une réception, $e' \prec e \Rightarrow \exists e_1,\ e' \prec e_1 \prec_i e$, avec $e_1 \in \mathcal{C}_p$. Comme $\mathcal{C}_p$ est $\mathcal{P}$-cohérente , $Det(e') \vee e' \in_{\mathcal{P}} \mathcal{C}_p$, ce qui assure que $\mathcal{C}'_p$ est $\mathcal{P}$-cohérente car $\mathcal{C}_p \subseteq \mathcal{C}'_p$.
  Si e est une réception ($(e', e) \in \Gamma$) alors e' appartient à $\mathcal{C}_p$: $e' \in \mathcal{C}_p$. D'abord, la propriété 5.3.4 et l'hypothèse 5.3.9 impliquent que $Odt_{\prec}(e') \subseteq \mathcal{C}_p$. Par le lemme A.2.1, si $e'' \prec e$ alors soit $e'' \prec e'$ ou $e'' \prec e_1$ avec $e_1 \in Odt_{\prec}(e') \subseteq \mathcal{C}_p$. Donc comme $\mathcal{C}_p$ est $\mathcal{P}$-cohérente et comme e' et e_1 appartiennent à $\mathcal{C}_p$, alors $\forall e'' \prec e Det(e'') \vee e'' \in_{\mathcal{P}} \mathcal{C}_p \subseteq \mathcal{C}'_p$.
  Finalement, $\mathcal{C}'_p$ est une coupe $\mathcal{P}$-cohérente de l'exécution $(E, \prec)$.
- Sinon, si $Det(e)$: D'abord, si $\mathcal{P}(e_0) \in \mathcal{C}_p$ tel que $\mathcal{P}(e_0) \lhd e$ alors la $\mathcal{P}$-cohérence de $\mathcal{C}_p$ et le fait que $\mathcal{C}_p$ est une $\mathcal{P}$-coupe sont suffisant pour conclure la $\mathcal{P}$-cohérence de $\mathcal{C}'_p$.

Sinon, $\mathcal{C}_p^{\sqcup}$ est une coupe cohérente de $(E, \prec)$. De plus, $\mathcal{C}_p \xrightarrow{*} \mathcal{C}_p^{\sqcup}$ et (propriété 5.3.1) $c_i \xrightarrow{*}_L \mathcal{C}_p^{\sqcup}\big|_i$ en ne réduisant que des évènements déterministes ou promis. Pour tous ces évènements, nous pouvons appliquer l'hypothèse 5.3.9 avec e_0 déterministe ou promis et e la réception repoussée. Finalement, on a $\mathcal{C}_p^{\sqcup}\big|_i \xrightarrow{e}_L \mathcal{C}_p^{\sqcup}\big|_i \oplus \{e\}$. De plus, $\mathcal{C}_p^{\sqcup}$ étant une coupe cohérente, et avec la propriété 5.3.1, on a $\exists S', \mathcal{C}_p^{\sqcup} \xrightarrow{e} S'$. Finalement, $e_0 \prec e \Rightarrow e_0 \in \mathcal{C}_p^{\sqcup}$, et donc $e_0 \in_{\mathcal{P}} \mathcal{C}_p \vee Det(e_0)$ car $\mathcal{C}_p^{\sqcup}$ et $\mathcal{C}_p$ ne diffère que par des évènements déterministes ou promis (propriété 5.2.2). Notons que cela prouve aussi que $\mathcal{C}'_p$ est une $\mathcal{P}$-coupe. Dans ce cas, il est possible que l'exécution $(E', \prec')$ de laquelle $\mathcal{C}'_p$ est une coupe $\mathcal{P}$-cohérente diffère de l'exécution de référence $(E, \prec)$ *après* $\mathcal{C}_p^{\sqcup}$

□

A.2.3 Propriété 5.3.6 : $\mathcal{C}_p^{\sqcap}\big|_i$ inclus dans $c_i^{\sqcap}$

Preuve : Soit $e_0 \in \mathcal{C}_p^{\sqcap}$, alors $\forall e' \prec e_0, e' \in \mathcal{C}_p$. Pour tous les j (même si $j = i$), si $e_0 \in c_i$, on a $\forall e',\ e' \prec_i e_0 \Rightarrow e' \in \mathcal{C}_p$, donc $e_0 \in c_i'^{\sqcap}$. Donc, on a $\mathcal{C}_p^{\sqcap}\big|_i$ inclus dans $c_i^{\sqcap}$. □

A.2.4 Propriété 5.3.7 : Réduction sans blocage possible

L'exécution possible définie par le théorème 5.2.1 peut être faîte par la réduction locale $\rightarrow_L$:

$$\mathcal{C}_p \xrightarrow{e} \mathcal{C}'_p \wedge (Det(e) \vee \exists \mathcal{P}(e) \text{ minimale dans } \mathcal{C}'_p \backslash \mathcal{C}_p\ ) \Rightarrow \mathcal{C}_p \xrightarrow{e}_L \mathcal{C}'_p$$

Nous commençons par prouver le lemme suivant :

Lemme A.2.2 *Pour toute exécution, pouvant réduire des évènement positionnables en avance, il existe une réduction* équivalente *qui ne réduit aucun évènement en avance :*

$$\forall \sigma, c_i, e_i,\ c_i^0 \xrightarrow{e_1}_L \ldots \xrightarrow{e_n}_L c_i \wedge (e_{\sigma(i)} \prec e_{\sigma(j)} \Rightarrow i < j) \quad \Rightarrow \quad c_i, e_i,\ c_i^0 \xrightarrow{e_{\sigma(1)}}_L \ldots \xrightarrow{e_{\sigma(n)}}_L c_i$$

Preuve : Cette preuve repose sur l'échange d'évènements consécutifs. Supposons $c_i \xrightarrow{e}_L c'_i \xrightarrow{e'}_L c''_i$, si $e \prec e'$, nous devons avoir $c_i \xrightarrow{e'}_L c_i^1 \xrightarrow{e}_L c''_i$. On distingue deux cas :

- $e \succ e'$: alors, par hypothèse 5.3.9, $\exists c_i^1,\ c_i \xrightarrow{e'}_L c_i^1 \xrightarrow{e}_L c''_i$.
- $e \succ e'$: alors, par la propriété 5.3.1 et la définition 5.3.2), $S = \{s_1 \ldots c_i^{\sqcap} \ldots s_n\}$ et $S \xrightarrow{e} S'$, et de façon similaire $\exists S_2, S_2\big|_i = c_i'^{\sqcap} \wedge S_2 \xrightarrow{e'} S'_2$. Soit $e'_1 \ldots e'_n$ tel que $e'_k \prec e'$ et $e'_k \notin S$. Nous avons $e \succ e'_k \wedge e \prec e'_k$ et e'_k sur des activités différentes de celle qui ont exécuté e et e', donc $S \xrightarrow{e'_1 \ldots e'_n} S_2$ avec $S_2\big|_i = S\big|_i = c_i^{\sqcap}$. $S_2 \xrightarrow{e'} S'_2$, et $S'_2 \xrightarrow{e} S''_2$ car e et e' sont minimaux dans $E \backslash S_2$. Par conséquent, par la propriété 5.3.1 et la définition 5.3.2), on a $c_i^{\sqcap} \xrightarrow{e'}_L s'_i$ et $c_i \xrightarrow{e'}_L c_i^1$. Finalement, $c_i^{1\sqcap} = S'_2\big|_i$ et $S'_2 \xrightarrow{e} S''_2$ impliquent $c_i^1 \xrightarrow{e}_L c''_i$

□

Nous pouvons finalement prouver la propriété 5.3.7 :

Preuve : La réduction possible définie par le théorème 5.2.1 réduit toujours un évènement minimal de $\mathcal{C}_p^{\sqcup} \backslash \mathcal{C}_p^{\sqcap}$, soit déterministe soit correspondant à une promesse minimal dans $\mathcal{C}_p$. Soit e cet évènement : $\mathcal{C}_p^{\sqcap} \xrightarrow{e} S'$. Et soit A_i l'activité qui exécute e. Sans restriction, nous pouvons supposer que e est exécuté sur une activité sur laquelle il n'existe pas de conséquence d'un message orphelin. En effet, il existe toujours une réduction possible qui réduit toujours des évènements vérifiant cette hypothèse.

Avec cela, et avec l'hypothèse 5.3.8 nous avons :

$$\forall e',\, e' \in c_i^{\sqcap} \backslash \mathcal{C}_p^{\sqcap} \Rightarrow e' \notin c_i^0$$

En effet, si il existe $e \notin \mathcal{C}_p$ tel que $e \prec e'$, alors, comme $e' \in \mathcal{C}_p^0$, e et soit promis sur l'activité A_i (et est donc orphelin), soit est exécuté sur une autre activité, et par le lemme 5.3.1, il existe une chaîne de messages $(e_k, e'_k) \in \Gamma$ avec $k \in [1..l]$ telle que $e \preceq_i e_1 \prec e'_1 \prec_i e_2 \dots e'_{l-1} \prec e_l \prec e'_l \preceq_i e'$. L'hypothèse 5.3.8 implique que $e_1 \notin \mathcal{C}_p$ et $e_k \in \mathcal{C}_p \Rightarrow e'_{k-1} \in_{\mathcal{P}} \mathcal{C}_p$, puisque seules les réceptions orphelines peuvent être promise avant la coupe (hypothèse 5.3.8), il existe un message $e_k \notin \mathcal{C}_p$ et $e_k \in_{\mathcal{P}} \mathcal{C}_p$, ce message est orphelin.

Notons $s_i = \mathcal{C}_p^{\sqcap}\big|_i$, avec $s_i \subseteq c_i^{\sqcap}$ (propriété 5.3.6). Par l'hypothèse 5.3.1, $\exists s'_i,\, s_i \xrightarrow{e}_L s'_i$. Supposons que $c_i^0 \xrightarrow{e_1..e_n}_L c_i$. Alors $c_i^0 \subseteq \mathcal{C}_p^{\sqcap}$ implique $\forall e' \in c_i^{\sqcap} \backslash \mathcal{C}_p^{\sqcap},\, e' = e_i$. De plus, $\mathcal{C}_p^{\sqcap} \xrightarrow{e}$ implique $\forall e'' \prec e,\, e'' \in \mathcal{C}_p^{\sqcap} \subseteq c_i$.

Le lemme A.2.2 nous dit qu'il existe une réduction équivalente $c_i^0 \xrightarrow{e_1}_L c_i^1 .. c_i^{n-1} \xrightarrow{e_n}_L c_i$ telle que $e_i \prec e_j \Rightarrow i < j$. Il existe k tel que d'abord $\forall e'' \prec e, e'' \in c_i^0 \vee e'' \in c_i^k$ car $\forall e'' \prec e,\, e'' \in c_i$, puis $\forall e''' \succ e,\, e''' \notin c_i^k$ car $e''' \succ e \wedge e'' \prec e \Rightarrow e'' \prec e''' \Rightarrow e'' = e_i \wedge e''' = e_j$ avec $i < j$. De plus, nous pouvons choisir la réduction et pouvons choisir l tel que $\mathcal{C}_p^{\sqcap}\big|_i = c_i^l$ (tous les e tels que $\exists e_0 \prec e, e_0 \notin \mathcal{C}_p$ peuvent être repoussés après l).

Nous avons $c_i^{l\sqcap} \xrightarrow{e}_L s'_i$ car $c_i^{l\sqcap} = c_i^l = \mathcal{C}_p^{\sqcap}\big|_i$, et $\mathcal{C}_p \xrightarrow{e}$. Si e est une réception de message, alors $Odt_{\prec}(e') \subseteq \mathcal{C}_p^{\sqcap}\big|_i$ implique $Odt_{\prec'}(e') \subseteq c_i^l$. Nous pouvons conclure grâce à l'hypothèse 5.3.7 appliquée à $c_i^{l\sqcap}$ pendant la réexécution. Finalement, $c_i^0 \xrightarrow{e_1}_L c_i^1 .. c_i^{n-1} \xrightarrow{e_n}_L c_i$ et $c_i^l \xrightarrow{e}_L$ (définition 5.3.2).

Maintenant, nous prouvons par récurrence que $c_i \xrightarrow{e}_L$. Sachant que $c_i^l \xrightarrow{e}_L c'_i$ et $c_i^l \xrightarrow{e_1}_L c_i^1 .. c_i^{n-1} \xrightarrow{e_n}_L c_i$; pour tout $l' \geq l$ tel que $c_i^{l'} \xrightarrow{e}_L c'_i$ et $c_i^{l'} \xrightarrow{e_{l'}}_L c_i^{l'+1}$, nous prouvons que $c_i^{l'+1} \xrightarrow{e}_L c''_i \wedge c_i^{l'} \xrightarrow{e_{l'}}_L c''_i$. Deux cas sont possibles :

- $e_{l'} \succ e$: alors, comme $Det(e)$ ou $\mathcal{P}(()e) \in c_i^{l'}$ par l'hypothèse 5.3.9, $\exists c_i^1,\, c_i^{l'+1} \xrightarrow{e}_L c_i^1$.
- $e_{l'} \succ e \wedge e_{l'} \prec e$: alors les évènements sont compatibles et peuvent être exécutés dans n'importe quel ordre.

Pour conclure, chaque évènement déterministe ou promesse minimale peut être réduit localement depuis $\mathcal{C}_p$. On peut donc effectuer localement la réduction sans blocage.

□

A.2.5 Propriété 5.3.8 : États globaux communs

Toute suite de réductions locales finit par atteindre un état atteignable par réduction globale :

$$\mathcal{C}_p \xrightarrow{*}_L \mathcal{C}'_p \;\Rightarrow\; \mathcal{C}'_p \xrightarrow{*}_L \mathcal{C}''_p \wedge \mathcal{C}_p \xrightarrow{*} \mathcal{C}''_p$$

Preuve : On a directement, des propriétés 5.3.5 et 5.2.4 :

$$\mathcal{C}_p \xrightarrow{*}_L \mathcal{C}'_p \;\Rightarrow\; \mathcal{C}'_p \xrightarrow{*} \mathcal{C}''_p \wedge \mathcal{C}_p \xrightarrow{*} \mathcal{C}''_p$$

où la réduction peut être la réduction possible du théorème 5.2.1. Donc, la propriété 5.3.1 appliquée à la réduction $\mathcal{C}'_p \xrightarrow{*} \mathcal{C}''_p$ nous permet de conclure :

$$\mathcal{C}_p \xrightarrow{*}_L \mathcal{C}'_p \;\Rightarrow\; \mathcal{C}'_p \xrightarrow{*}_L \mathcal{C}''_p \wedge \mathcal{C}_p \xrightarrow{*} \mathcal{C}''_p$$

□

A.2.6 Propriété 5.3.9 : $c_i^{\sqcap}$ dans le cas idéal

$$\mathcal{C}_p^{\sqcap} = \{c_1^{\sqcap}, \ldots, c_n^{\sqcap}\}$$

Nous montrons tout d'abord que la réduction locale conserve l'hypothèse 5.3.11 :

Propriété A.2.1 *$\rightarrow_L$ conserve l'hypothèse 5.3.11 :*

$$\forall (e, e') \in \Gamma,\, \forall \mathcal{C}_p,\, e \notin \mathcal{C}_p \Rightarrow e' \notin \mathcal{C}_p$$

La preuve de cette propriété est simple : la seule façon d'avoir une réception dans $\rightarrow_L$ est soit de recevoir un message en transit (qui ne peut pas être orphelin), soit d'avoir *déjà* réduit l'envoi, auquel cas rien ne doit être vérifié pour que l'hypothèse reste vraie.

Nous montrons ensuite qu'il existe toujours une promesse qui permet de déterminer localement qu'un évènement *après* cette promesse n'appartient pas à la coupe $\mathcal{C}_p^{\sqcap}$.

Propriété A.2.2 *L'hypothèse 5.3.11 implique que :*

$$\forall e \in \mathcal{C}_p, e \notin \mathcal{C}_p^{\sqcap} \Rightarrow \exists e' \prec_i e, e' \notin \mathcal{C}_p$$

Preuve : Supposons $e \in \mathcal{C}_p \backslash \mathcal{C}_p^{\sqcap}$, alors $\exists e' \prec e, e' \notin \mathcal{C}_p$. Supposons sans restriction que e' est maximal, c'est à dire que tous les évènements causalement entre e' et e appartiennent à $\mathcal{C}_p$.

Nous analysons les différents cas permettant d'inférer que $e' \prec e$:

- $e' \prec_i e$ alors la propriété est vraie pour e et e'.
- $(e', e) \in \Gamma$ alors (e', e) est orphelin, et donc $e \notin \mathcal{C}_p$ (propriété A.2.1), ce qui est contradictoire.

– $e' \prec e_0 \prec e$ avec $e_0 \in \mathcal{C}_p$, nous pouvons supposer sans restriction que soit $e' \prec_i e_0$ soit $(e', e_0) \in \Gamma$.
Si $(e', e_0) \in \Gamma$, alors par la propriété A.2.1, $(e', e_0) \in \Gamma \wedge e' \notin \mathcal{C}_p \Rightarrow e_0 \notin \mathcal{C}_p$, ce qui contredit la maximalité de e'.
Le seul cas encore possible est $e' \prec_i e_0 \prec e$: si $e_0 \prec_i e$, alors la propriété est vérifiée (premier cas). Sinon nécessairement, $\exists (e_1, e_2) \in \Gamma,\ e_0 \prec_i e_1 \prec e_2 \prec e$ (lemme 5.3.1) avec $e_1, e_2 \in \mathcal{C}_p$ comme e est maximal. e_1 ne peut pas avoir été réduit : si $e_1 \in c_i$ a été réduit, alors on a un c'_i entre c^0_i et c_i tel que $c'^{\sqcap}_i \xrightarrow{e_1} c''_i$ (définition 5.3.2) ce qui implique, comme $e' \prec_i e_1$, que $e' \in c'^{\sqcap}_i \subseteq c'_i \subseteq c_i$ et donc $e' \in \mathcal{C}_p$, ce qui est contradictoire.
Donc $e_1 \in \mathcal{C}^0_p$ et $Det(e_1)$ (hypothèse 5.3.5) et comme $\mathcal{C}^0_p$ est une $\mathcal{P}$-coupealors $e' \in_{\mathcal{P}} \mathcal{C}^0_p$. Comme $e' \notin \mathcal{C}_p\, \mathcal{P}(e') \in \mathcal{C}^0_p$, et par l'hypothèse 5.3.11, on a $\exists e'', (e'', e') \in \Gamma \wedge e'' \notin \mathcal{C}^0_p$.
Enfin, on a par l'hypothèse 5.3.8 $\nexists e_1 \in \mathcal{C}^0_p, e_1 \succ_i e' \wedge (e_1, e_2) \in \Gamma$, donc $e_1 \notin \mathcal{C}^0_p$, ce qui est contradictoire.

□

Enfin, la propriété A.2.2 nous permet de prouver la propriété 5.3.9 :

Preuve : D'abord, la propriété 5.3.6 implique que $\mathcal{C}^{\sqcap}_p \subseteq \{c^{\sqcap}_1, \dots c_i{}^{\sqcap}, \dots, c^{\sqcap}_n\}$. De plus, nous prouvons que $\mathcal{C}'^{\sqcap}_p \supseteq \{c^{\sqcap}_1, \dots c'^{\sqcap}_i, \dots, c^{\sqcap}_n\}$: soit $e_0 \in c^{\sqcap}_i$, alors $e_0 \in \mathcal{C}_p$. Montrons que $e_0 \notin \mathcal{C}^{\sqcap}_p$ est contradictoire. Par la propriété A.2.2, si $e_0 \notin \mathcal{C}^{\sqcap}_p$ alors $\exists e' \prec_i e,\ e' \notin \mathcal{C}_p$, et donc $e_0 \notin c^{\sqcap}_i$, ce qui est contradictoire. Donc $e_0 \in \mathcal{C}^{\sqcap}_p$. □

A.2.7 Théorème 5.3.3 : Équivalence des réductions dans le cas idéal

$$\mathcal{C}_p \xrightarrow{e}_L \mathcal{C}'_p \ \Rightarrow\ \mathcal{C}_p \xrightarrow{e} \mathcal{C}'_p$$

Preuve : Nous prouvons tout d'abord que $\mathcal{C}_p \xrightarrow{e}_L \mathcal{C}'_p \Rightarrow \exists \mathcal{C}''_p\, \mathcal{C}_p \xrightarrow{e} \mathcal{C}''_p$: Si $c^{\sqcap}_i \xrightarrow{e} s'_i$ alors par la propriété 5.3.1 et comme $\mathcal{C}^{\sqcap}_p = \{c^{\sqcap}_0 \dots c^{\sqcap}_i \dots c^{\sqcap}_n\}$ (propriété 5.3.9), on a $\mathcal{C}^{\sqcap}_p \xrightarrow{e} \mathcal{C}''_p$ (vérifié directement par la définition 5.2.5).

Sinon, $\exists e_0 \in \mathcal{C}^0_p, (e_0, e) \in \Gamma \wedge e \notin \mathcal{C}_p$. Dans ce cas, par l'hypothèse 5.3.10, on a $\mathcal{P}(e) \in \mathcal{C}_p$, et $\mathcal{C}^{\sqcap}_p \xrightarrow{e} \mathcal{C}''_p$.

Maintenant nous prouvons que $\mathcal{C}''_p = \mathcal{C}'_p$: Si $\exists \mathcal{P}(e') \in \mathcal{C}_p, \mathcal{P}(e') \lhd e$ et $c_i \xrightarrow{e'} c'_i$ alors nécessairement $\mathcal{C}''_p$ a été réduite avec le deuxième cas de la définition 5.2.5, $\mathcal{C}''_p = \mathcal{C}_p\{\mathcal{P}(e') \leftarrow e'\} = \mathcal{C}'_p = \{c_0 \dots c'_i \dots c_n\}$.

Sinon $c_i \xrightarrow{e} c_i \oplus \{e\}$. De façon similaire ($\mathcal{C}''_p$ ne peut plus être obtenu avec le deuxième cas de la définition 5.2.5), $\mathcal{C}''_p = \mathcal{C}_p \oplus \{e\} = \mathcal{C}'_p$. □

Abstract

The main goal of this thesis is to define a rollback-recovery fault tolerance protocol for the asynchronous communicating active objects model ASP (Asynchronous Sequential Processes), and its Java implementation ProActive. This work generalises the problem raised by the development of this protocol : we study the recovery of a distributed execution from an *inconsistent* global state.

We then propose a checkpointing protocol and its implementation that does not rely on consistent global states. We demonstrate the model efficiency through realistic experiments using communicating distributed applications that this solution is efficient in practice.

Another more general contribution to the problematic of recovering from a inconsistent global state by formally is the definition of the $\mathcal{P}$-consistency, a new recoverability condition based on the concept of promised event. This definition is part of an event-based formalism which can be applied to any system. In particular, by applying this formalism to the ASP model, we are able to prove the correctness of our protocol by showing that every global state created during the execution is a recoverable state.

Finally, we propose an extension of our protocol and an implementation adapted to the context of grid computing. This extension relies on the constitution of recovery groups during the deployment of the application. It allows to independently distribute stable storage and to limit the effects of a failure to the concerned group.

Keywords : Fault-tolerance, Checkpoiting, Consistency, Promised event

---

Résumé

L'objectif premier de cette thèse est de proposer un protocole de tolérance aux pannes par recouvrement arrière pour le modèle à objets actifs asynchrones communicants ASP (*Asynchronous Sequential Processes*) et son implémentation en Java ProActive. Cette thèse généralise la problématique soulevée par le développement de ce protocole : nous étudions le recouvrement d'une application répartie depuis un état global *non cohérent*.

Nous proposons donc dans un premier temps un protocole par points de reprise et son implémentation ne supposant pas que les états globaux soient cohérents. Nous montrons à travers des expérimentations réalistes utilisant des applications réparties communicantes que notre solution et son implémentation présentent de bonnes performances.

Nous contribuons aussi de manière plus générale à l'étude du recouvrement depuis un état global non cohérent en définissant formellement une nouvelle condition de recouvrabilité, la $\mathcal{P}$-cohérence, basée sur la notion de promesse d'évènement. Cette définition s'intègre dans un formalisme événementiel capable de prendre en compte la sémantique de n'importe quel système ; elle est donc applicable dans un cadre général. En particulier, en appliquant ce formalisme au modèle ASP, nous prouvons la correction de notre protocole en montrant que les états globaux formés durant l'exécution sont toujours recouvrables.

Enfin, nous contribuons plus spécifiquement au domaine des grilles de calcul en proposant une extension de notre protocole et son implémentation adaptée à ce contexte. Cette extension se base sur la constitution automatique de groupes de recouvrement au déploiement de l'application. Elle permet une répartition indépendante des mémoires stables et un confinement des effets d'une panne au seul groupe concerné.

Mots-clefs : Tolérance aux pannes, Points de reprise, Cohérence, Promesse d'évènement