La reprise sur erreur par recouvrement arrière automatique dans les systèmes répartis

Denis Conan

Université de Technologie de Compiègne, Lab. Heudiasyc UMR CNRS 6599, 60605 Compiègne cedex
conan@utc.fr

Guy Bernard

Institut National des Télécommunications, 91011 Évry cedex
bernard@int-evry.fr

RÉSUMÉ. *La reprise sur erreur par recouvrement arrière automatique a fait l'objet d'une littérature très abondante. Cet article présente une étude approfondie des problèmes (le déterminisme d'exécution, la cohérence forte d'état global), des objectifs (le degré de tolérance aux fautes, les surcoûts, l'inhibition et la quantité de travail à défaire ou à refaire), des politiques possibles (efficaces) et souhaitables (performantes), et, des mécanismes existants du recouvrement arrière automatique.*

ABSTRACT. *Rollback error recovery is well documented in the litterature. This article presents a detailed study of the problems (the execution determinism, the global state strong consistenccy), the objectives (the fault tolerance degree, overheads, the inhibition and the quantity of work to undo or to redo), the possible (efficient) and désirable (performant) politics, and, the mechanisms of rollback recovery.*

1. Introduction

Bien que les systèmes répartis soient devenus de plus en plus populaires, ils sont très vulnérables aux défaillances de nœuds. La tolérance aux fautes à l'aide de matériels spécialisés s'avère très coûteuse. Dans cet article, nous faisons la synthèse d'une méthode de tolérance logicielle aux fautes matérielles transparente et économique pour l'utilisateur : la reprise sur erreur par recouvrement arrière automatique.

La reprise sur erreur par recouvrement arrière automatique est fondée sur le concept du processus fiable. Un processus fiable est un processus continuant de s'exécuter correctement même s'il est interrompu par une faute, et ensuite, récupéré et repris. Pendant l'exécution, des images du processus sont sauvegardées en mémoire stable. Et lorsque survient une faute, le processus reprend son exécution à partir d'une des images précédemment sauvegardées.

La reprise sur erreur par recouvrement arrière automatique met en œuvre les trois mécanismes de base suivants. Le mécanisme de constitution de points de reprise sauvegarde en mémoire stable certains états de l'application répartie pendant l'exécution sans occurrence de fautes. Le mécanisme de journalisation, parfois facultatif, sauvegarde en mémoire stable les messages de l'application répartie. Le mécanisme de recouvrement arrière organise le retour en arrière de l'application répartie lorsque surviennent des défaillances de nœuds.

Cet article modélise et analyse la problématique du recouvrement arrière et présente une étude critique des différentes politiques envisageables. C'est sur ce dernier aspect qu'il se distingue le plus de la synthèse de Elnozahy, Johnson et Wang [EJW96]. La section 2. décrit le modèle du système réparti et des applications réparties. La section 3. introduit les problèmes et les concepts de base de la reprise sur erreur par recouvrement arrière automatique. La section 4. considère les objectifs des politiques en termes d'efficacité et de performance. La section 5. présente les politiques de reprise possibles et souhaitables, et la section 6. développe les mécanismes de base. Enfin, la section 7. présente des voies de recherche nouvelles. Dans la suite, l'expression "reprise sur erreur par recouvrement arrière automatique" est abrégée en "recouvrement arrière".

2. Le modèle

Avant d'aborder les politiques et les mécanismes du recouvrement arrière, nous décrivons le modèle du système réparti puis le modèle des applications réparties. Nous nous limitons volontairement aux fautes matérielles ; cependant, nous signalons les endroits où cette hypothèse est utilisée.

Le système réparti est composé de nœuds reliés entre eux par un réseau de communication. Le système réparti est dit *faiblement couplé*. Chaque **nœud** est équipé d'un processeur, de blocs de mémoire, d'interfaces de communication avec le réseau et d'une console. Les nœuds suivent le modèle *"silence sur défaillance"* [PBS+88] : la détection d'une erreur provoque l'arrêt franc du processeur [1]. L'état du processeur ainsi que

1. Précédemment, Schlichting et Schneider avaient défini le mode *"arrêt sur défaillance"* [SS83]. Ce

le contenu de la mémoire du nœud défaillant sont perdus à la suite d'occurrences de fautes. Le modèle de mémoire est du type *"NORMA" (No-Remote-Memory-Access)* : un processeur accède uniquement à la mémoire de son nœud. Tous les nœuds accèdent à une même mémoire stable survivant à la défaillance globale du système réparti. Le **réseau de communication** véhicule les messages entre nœuds de manière *asynchrone* : le processeur du nœud émetteur initialise la transmission puis continue son exécution. Les délais de transmission sont finis mais non bornés et imprévisibles. Par ailleurs, les communications sont fiables : sans perte, sans duplication et sans altération des messages.

Une exécution assigne des *valeurs* à des *variables*. Un **état** est l'affectation de valeurs à des variables. L'ensemble des états est dénoté $\mathbf{S}$. Une **action** (atomique) $\mathcal{A}$ représente la relation entre un ancien état s et un nouvel état t notée $s\mathcal{A}t$. L'ensemble des actions est dénoté $\mathbf{A}$. Une **exécution séquentielle** d'un processus P à partir de l'état initial $s^0 \in \mathbf{S}$, est la succession d'un nombre fini d'actions sur des états notée $s^0\mathcal{A}^1 s^1 \mathcal{A}^2 s^2 \mathcal{A}^3 \ldots \mathcal{A}^f s^f$. Les états s^0 et s^f sont respectivement appelés l'état initial et l'état final. La séquence d'actions $\mathcal{A}^1\mathcal{A}^2\mathcal{A}^3 \ldots \mathcal{A}^f$ à laquelle est ajoutée l'action initiale fictive $\mathcal{A}^0$ initialisant s^0 est appelée l'**histoire séquentielle** de l'exécution séquentielle.

Pour des raisons de simplicité évidentes, l'étude se limite à la tolérance des fautes matérielles. Toutefois, la faute du processeur d'un nœud peut résulter de la défaillance d'un composant logiciel du nœud. En témoignent les mécanismes de nombreux systèmes : par exemple, les systèmes Tandem [Gra90]. Pour qu'une faute logicielle affecte le processeur d'un nœud, l'erreur engendrée doit être détectée. Généralement, un mécanisme de détection d'erreurs s'exprime par un mécanisme d'exception. Si ce dernier est efficace, le processus est qualifié de robuste [Cri89]. Notons dès maintenant que la maîtrise des fautes logicielles est importante. Nous reviendrons sur ce point au cours de la section 7.. Dans la suite de l'étude, tout processus est supposé robuste. Par conséquent, nous redéfinissons le mode **"silence sur défaillance"** comme suit : défaillance matérielle du processeur, ou, détection d'une erreur dans l'exécution d'un processus robuste provoquant une action exceptionnelle arrêtant le processeur du nœud.

Une application répartie [2] $\widehat{P}$ est composée de $p = \mid \widehat{\mathbf{P}} \mid$ processus communiquant par échange de messages. L'action d'**émission** `émettre`(P_j, m) d'un message m par P_i vers P_j ajoute m au canal c_{ij}. Pratiquement, m est transmis de P_i vers P_j par le réseau de communication et est gardé dans la mémoire du nœud où s'exécute P_j. La mémoire du nœud récepteur est supposée suffisante pour contenir les nouveaux messages arrivant. L'action de **réception** `recevoir`(l) d'un message l par P_j sur l'ensemble des canaux c_{ij} assigne à l le premier message arrivé par l'un des canaux. Si aucun message n'est arrivé alors P_j attend jusqu'à l'arrivée d'un message par l'un des canaux. Par

mode est une extension du mode "silence sur défaillance" en ce qu'il suppose en plus que tous les autres nœuds du système ont connaissance de la défaillance. C'est pour cette raison que le mode "silence sur défaillance" est préféré.

2. "ˆ" pour répartie.

convention, les messages sont numérotés dans leur ordre d'arrivée. Toute action autre qu'une action d'émission ou de réception d'un message est appelée **action interne**. L'**état d'un canal** c_{ij} est constitué de l'ensemble des messages émis et non encore reçus. Un **état global** $\widehat{s}$ est composé d'un état local [3] $s_i^{x_i}$ par processus $P_i \in \widehat{\mathbf{P}}$ [CL85]. L'ensemble des états globaux est noté $\widehat{\mathbf{S}}$.

L'**exécution répartie** d'une application à partir de l'état global $\widehat{s}$ est constituée de la fusion suivant un temps absolu des exécutions des processus. Par déduction, l'**histoire répartie** d'une exécution répartie est constituée de la fusion suivant un temps absolu des histoires séquentielles. Par définition, l'**histoire des messages** d'une exécution répartie est égale à l'histoire répartie de laquelle toutes les actions internes sont retranchées. Dans la pratique, les actions d'émission et de réception sont repérées respectivement par des numéros d'ordre d'émission et de réception. L'*histoire d'un message* est alors constitué du quadruplet {*Identité de l'émetteur, Numéro d'ordre d'émission, Identité du récepteur, Numéro d'ordre de réception*}.

Pour la compréhension de la suite de l'article, nous présentons deux relations de précédence. Lamport définit la relation binaire $\prec_{\mathbf{A}}$ sur les actions appelée **précédence causale** [Lam78] :

$$\mathcal{A}_i^x \prec_{\mathbf{A}} \mathcal{A}_j^y \stackrel{\text{def}}{=} \begin{cases} & (i=j) \wedge (y = x+1) \\ \vee & \exists m : (\mathcal{A}_i^x = \texttt{émettre}(P_j, m)) \wedge (\mathcal{A}_j^y = \texttt{recevoir}(m)) \\ \vee & \exists \mathcal{A}_k^z : (\mathcal{A}_i^x \prec_{\mathbf{A}} \mathcal{A}_k^z) \wedge (\mathcal{A}_k^z \prec_{\mathbf{A}} \mathcal{A}_j^y) \end{cases}$$

Fromentin et Raynal définissent une relation d'ordre strict sur l'ensemble des états locaux produits par une exécution répartie. Cet ordre noté $\prec_F$ et appelé **précédence forte**, indique qu'un état s_i^x a cessé d'exister lorsqu'un autre état s_j^y a commencé d'exister [FR94] :

$$s_i^x \prec_F s_j^y \stackrel{\text{def}}{=} (\mathcal{A}_i^{x+1} \prec_{\mathbf{A}} \mathcal{A}_j^y) \vee (s_j^y = s_i^{x+1}).$$

Une exécution répartie est représentée par un diagramme espace-temps. La figure 1 montre un tel diagramme pour une application composée de trois processus. Le déroulement du temps est décrit par une ligne continue pour chaque processus. Les actions sont symbolisées par des tirets sur les processus. Les messages sont matérialisés par des flèches connectant une action `émettre` à une action `recevoir`. L'exécution tracée dans cette figure montre par exemple les relations suivantes : $\mathcal{A}_1^2 \prec_{\mathbf{A}} \mathcal{A}_2^1$, $s_1^2 \not\prec_F s_2^1$ et $s_1^1 \prec_F s_2^1$.

Lors de l'exécution sans occurrence de fautes d'un processus [4] P_{TF}, certains de ses états sont sauvegardés en mémoire stable. Ce sont des **points de reprise**. L'ensemble des points de reprise d'un processus P_{TF} est noté $\mathbf{R}_{P_{TF}}$. La constitution d'un point de reprise peut être vue comme l'insertion d'une action $\mathcal{C}$ dans l'exécution : $s^r \mathcal{C} s^r$. Une **action défaillante** $\mathcal{A}^d$ est une action interrompue par la défaillance du processeur du nœud sur lequel elle s'exécute, ou l'action identique [5]. L'**état défaillant** est l'état

3. le $x_i$ème état local de P_i.
4. "TF"pour "Tolérant aux Fautes".
5. Identique = sans effet. Le processus n'était pas en train de s'exécuter quand survient la défaillance.

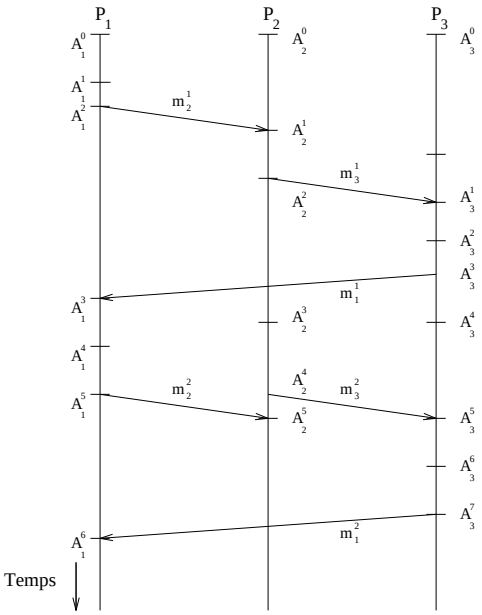

FIG. 1 - *Le diagramme espace-temps d'une exécution répartie*

s^{d-1} précédant la défaillance du processeur, et une **exécution défaillante** du processus P_{TF} est une exécution de P_{TF} durant laquelle une faute survient pendant l'action défaillante $\mathcal{A}^d$. Puisque l'état défaillant est perdu, l'exécution de P_{TF} doit reprendre à partir d'un des points de reprise précédemment sauvegardés. Par définition, la **réexécution** est l'exécution de P_{TF} à partir du point de reprise s^r. La **nouvelle exécution** est l'exécution à partir de l'état initial s^0 jusqu'à l'état de reprise s^r, suivie de la réexécution. Nous verrons ci-dessous que si le processus P est déterministe, la nouvelle exécution est équivalente à l'exécution. Par abus de langage, une réexécution telle que la nouvelle exécution est équivalente à l'exécution est dite équivalente (à l'exécution).

3. Les problèmes

Cette section examine les deux problèmes du recouvrement arrière : le déterminisme d'exécution et la cohérence forte d'état global. Tout d'abord, le déterminisme d'exécution exprime la faculté de répéter l'exécution d'un processus. Cette propriété détermine le choix de l'utilisation ou non de la journalisation de l'exécution. En résumé, il est inutile de sauvegarder en mémoire stable l'histoire de l'exécution si la réexécution peut ne pas être équivalente. Ensuite, la répartition introduit le concept de cohérence d'état global. Le concept en lui-même est simple : la réception d'un message ne peut pas être observée avant son émission. Autrement dit, un état global cohérent est un état qui a existé ou a pu réellement exister durant l'exécution. La notion duale de la cohérence est l'absence de message en transit. L'idée est qu'un message émis doit être

rejoué lors de la réexécution si son action de réception a été perdue. Ces deux notions dont nous allons montrer la dualité sont regroupées sous la notion de cohérence forte d'état global.

3.1. *Le Déterminisme d'exécution*

Une **action** est **déterministe** ssi toute exécution de $\mathcal{A}$ à partir du même état initial donne le même état final. Deux **exécutions séquentielles** à partir d'un même état initial sont **équivalentes** lorsque leurs histoires sont égales et toutes les actions sont déterministes. En pratique, des exemples d'actions indéterministes sont les appels système dont l'exécution dépend de la valeur de l'horloge physique. Nous supposons que les actions indéterministes sont automatiquement repérées. Ce repérage est un problème d'implantation.

Pour une application, les actions d'émission et de réception de messages compliquent nettement la tâche. Puisque le temps de transfert d'un message varie, deux messages reçus consécutivement lors d'une exécution peuvent être reçus dans un ordre différent lors d'une autre exécution. Ces messages sont appelés des *"messages en concurrence"* (en anglais, *racing messages*) [NM94]. Par définition, deux **exécutions réparties** à partir d'un même état global initial sont **équivalentes** lorsque, pour tout processus $P_i \in \widehat{\mathbf{P}}$, les deux exécutions séquentielles de P_i sont équivalentes [6].

Un **processus** est **déterministe** ssi, à partir d'un état initial, une seule exécution séquentielle est possible, ou autrement dit, l'exécution complètement conditionnée (déterminée) par l'état initial. Une **application** est **déterministe** ssi, à partir d'un état global initial, toutes les exécutions réparties sont équivalentes. À cause de l'indéterminisme des actions de réception, très peu d'applications sont déterministes. Par définition, une **application** est dit **déterministe par morceaux** (noté *pm*-déterministe) ssi, à partir d'un état global initial et d'une histoire des messages, toute exécution répartie est équivalente [SY85]. En pratique, l'histoire des messages est obtenue par le mécanisme de journalisation lors de l'exécution : cela consiste à enregistrer en mémoire stable les histoires des messages reçus. Dans la suite, nous considérons qu'une application est soit *pm*-déterministe soit indéterministe. Les résultats pour une application *pm*-déterministe sont valables pour une application déterministe.

3.2. *La cohérence forte d'état global*

Le deuxième problème du recouvrement arrière est la cohérence forte de l'état global [HNR97]. Ce terme nouvellement introduit englobe les notions de cohérence d'état global et d'absence de message en transit (en anglais, transitlessness consistency). Respecter la cohérence forte signifie respecter la cohérence et l'absence de message en transit. Un protocole de constitution de points de reprise qui respecte la cohérence et garantit l'absence de message en transit évite l'utilisation d'un mécanisme de journalisation. Nous terminons cette sous-section par l'étude de deux autres propriétés qui

6. Notons que cette définition évite le problème de l'indéterminisme de la datation des événements par rapport à un temps absolu.

couvrent de façons différentes la cohérence forte : la stabilité et la recouvrabilité. L'objectif de cette sous-section est de fournir les concepts de base pour la présentation des politiques possibles du recouvrement arrière.

3.2.1. *La Cohérence*

Lorsqu'une faute survient, l'exécution répartie doit reprendre à partir d'un état global par lequel l'exécution "est passée ou aurait pu passer". Pour une application, cette condition s'exprime par la **cohérence** sur un état global [CL85] :

$$\texttt{cohérent}(\widehat{s}) \stackrel{\text{def}}{=} \forall i \neq j : \neg(s_i \prec_F s_j).$$

Le prédicat `cohérent` signifie que l'action de réception d'un message ne peut pas être observée si son action d'émission ne l'est pas aussi. Un **message** dont la réception est observée alors que l'émission ne l'est pas, est, par définition, dit **orphelin** (par rapport à l'état d'observation). Par extension, un **processus** dépendant d'un message orphelin est dit **orphelin**. Dans l'exemple de la figure 1, l'état global $\{s_1^2, s_2^1, s_3^0\}$ est cohérent tandis que l'état global $\{s_1^1, s_2^1, s_3^0\}$ ne l'est pas à cause de la relation $s_1^1 \prec_F s_2^1$ causée par le message orphelin m_2^1.

Netzer et Xu ont, les premiers, donné la condition nécessaire et suffisante pour qu'un ensemble quelconque d'états locaux (de taille inférieure à p) puissent former un état global cohérent (de taille égale à p). Nous exprimons cette condition par la notion d'*utilité* d'un point de reprise [BHMR95a]. Auparavant, il faut introduire la relation de précédence sur les intervalles de points de reprise. Un **intervalle de points de reprise** I_i^x d'un processus P_i est l'ensemble des actions produites par P_i entre les points de reprise R_i^x et R_i^{x+1} (y compris l'action menant à l'état de reprise). L'ensemble des intervalles de points de reprise est noté $\mathbf{I}$. La relation de précédence sur les intervalles de points de reprise notée $\prec_{\mathbf{I}}$ n'est pas un ordre partiel :

$$I_i^x \prec_{\mathbf{I}} I_j^y \stackrel{\text{def}}{=} \begin{cases} & (i=j) \wedge (y=x) \\ \vee & (i=j) \wedge (y=x+1) \\ \vee & \exists m : (\texttt{émettre}(P_j, m) \in I_i^x) \wedge (\texttt{recevoir}(m) \in I_j^y) \\ \vee & \exists I_k^z \in \mathbf{I} : (I_i^x \prec_{\mathbf{I}} I_k^z) \wedge (I_k^z \prec_{\mathbf{I}} I_j^y) \end{cases}$$

Il s'ensuit la condition nécessaire et suffisante pour déterminer si un ensemble quelconque de points de reprise $\mathbf{QcQ} = \{R_i^{x_i}\}_{i \in \mathbf{I}}$ peut appartenir à un état global cohérent :

$$\texttt{cohérent}(\mathbf{QcQ}) \stackrel{\text{def}}{=} \forall (P_i, P_j) \in \widehat{\mathbf{P}}_{TF} \times \widehat{\mathbf{P}}_{TF}, \neg(I_i^{x_i} \prec_{\mathbf{I}} I_j^{x_j - 1}).$$

Considérant le cas particulier $\mathbf{QcQ} = \{R_i^x\}$, un **point de reprise** R_i^x est **utile** ssi il peut appartenir à un état global cohérent :

$$\texttt{utile}(R_i^x) \stackrel{\text{def}}{=} \neg(I_i^x \prec_{\mathbf{I}} I_i^{x-1}).$$

Reprenons la figure 1 et supposons que toutes les actions internes sont des constitutions de points de reprise. La figure 2 représente les relations de précédence entre les

intervalles de points de reprise. Sur la figure, seules les deuxième et troisième termes de la relation $\prec_{\mathbf{I}}$ sont représentées. L'état global $\{\mathcal{A}_1^4, \mathcal{A}_2^3, \mathcal{A}_3^4\}$ est cohérent alors que l'état global $\{\mathcal{A}_1^1, \mathcal{A}_2^0, \mathcal{A}_3^1\}$ ne l'est pas à cause de la relation $(I_3^1 \prec_{\mathbf{I}} I_1^1) \wedge (I_1^1 \prec_{\mathbf{I}} I_2^0) \wedge (I_2^0 \prec_{\mathbf{I}} I_3^0)$.

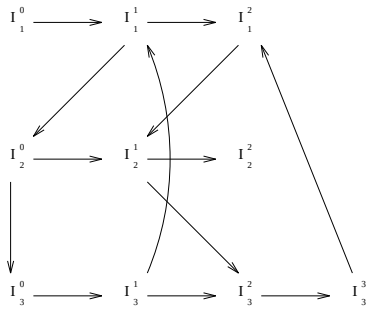

FIG. 2 - *Le graphe de précédence des intervalles de constitutions de points de reprise*

Si rien n'est fait pour que les points de reprise soient utiles ou pour que les messages soient journalisés afin de reconstituer un état global cohérent à la suite de défaillances, l'application peut être sujette à l'effet domino [Rus80] : l'application doit commencer sa réexécution à partir de son état global initial. C'est ce que nous appelons le **premier effet domino**.

3.2.2. *L'absence de message en transit*

Même si les points de reprise sont tous utiles, des messages peuvent traverser les états globaux cohérents. Ces messages dits en transit seront nécessaires lors de la réexécution. Le message m_1^1 de la figure 1 est en transit dans l'état global $\{s_1^1, s_2^3, s_3^4\}$.

Dans [HNR97], les auteurs montrent que ce problème est le dual du premier (la cohérence) et en déduisent la condition nécessaire et suffisante pour qu'un ensemble quelconque de points de reprise puisse appartenir à un état global sans message en transit. Les auteurs définissent une relation de précédence sur les actions qui consiste visuellement sur le diagramme espace-temps à retourner les flèches des messages. De manière non formelle, comme pour la cohérence, s'il existe un chemin (causal ou non) entre deux points de reprise, cela signifie que ces deux points de reprise ne peuvent pas appartenir à un même état global absent de message en transit. Nous laissons le lecteur se reporter à l'article pour le développement formel de la condition.

Lorsqu'aucun message n'est sauvegardé en mémoire stable pendant l'exécution, il peut être possible de remonter loin dans le temps l'exécution de l'application et de toujours trouver des états globaux présentant des messages en transit. L'application doit alors se réexécuter à partir de son état global initial. C'est le **deuxième effet domino**.

3.2.3. *La cohérence forte*

Un état global fortement cohérent est un état global cohérent absent de message en transit. Il est important de noter que l'utilisation d'un mécanisme de constitution

de points de reprise qui respecte les cohérence et absence de message en transit évite l'utilisation d'un mécanisme de journalisation. Autrement dit, si l'absence de message orphelin et l'absence de message en transit ne sont pas fournies par le mécanisme de constitution de points de reprise, c'est le mécanisme de journalisation qui la fournit.

À ce jour, seuls les protocoles de constitution de points de reprise à base de barrières de synchronisation arrêtant tous les processus de l'application, respectent la cohérence forte.

3.2.4. *À propos de la stabilité et de la recouvrabilité*

Les notions de stabilité et de recouvrabilité sont abondamment utilisées dans l'étude des mécanismes de journalisation. La stabilité est une notion particulière aux mécanismes de journalisation et la recouvrabilité correspond à la cohérence plus la stabilité. Pratiquement, la distinction vient du fait que la stabilité et la recouvrabilité incluent la persistance des informations de reprise. Par exemple, l'état global précédant l'émission d'un message vers le monde extérieur doit être cohérent et enregistré en mémoire stable, ou, de manière équivalente, il doit être recouvrable.

Nous avons préféré organiser la présentation autour de la cohérence forte pour insister sur le fait que les notions de message orphelin et de message en transit sont duales et minimales. En outre, cela nous permet de souligner l'importance de la remarque suivante : si un protocole de constitution de points de reprise respecte l'absence de message orphelin ou en transit, aucun mécanisme de journalisation n'est obligatoire. Nous étudions maintenant la stabilité et la recouvrabilité.

Soit $RE_i^{x_i}$ le dernier point de reprise de P_i, un **message** m reçu par P_i est dit **stable** ssi l'histoire de tous les messages reçus depuis $RE_i^{x_i}$ est sauvegardée en mémoire stable [JZ90b] :

$$\texttt{stable}(m) \stackrel{\text{def}}{=} \left\{ \begin{array}{ll} & \texttt{jms}(m) \\ \wedge & (\exists l, \exists x : (\mathcal{A}_i^x = \texttt{recevoir}(l)) \wedge (\mathcal{C}_{RE_i^{x_i}} \prec_{\mathbf{A}} \mathcal{A}_i^x) \\ & \wedge (\mathcal{A}_i^x \prec_{\mathbf{A}} \texttt{recevoir}(m))) \Rightarrow \texttt{jms}(l)) \end{array} \right.$$

avec le prédicat $\texttt{jms}(m)$ [7] évalué à "vrai" ssi l'histoire de m est sauvegardée en mémoire stable, et avec $\mathcal{C}_{RE_i^{x_i}}$ repérant le dernier point de reprise constitué par P_i.

Soit un état local s_i^x d'un processus P_i et soit l le dernier message reçu par P_i avant s_i^x, l'**état local** s_i^x est dit **stable** ssi l est stable. Un **état global** est **recouvrable** ssi tous les états locaux sont stables et l'état global ainsi formé est cohérent.

Avant qu'un message soit émis à destination du monde extérieur à l'application, l'état global doit être recouvrable. L'opération qui rend recouvrable un état global est appelée la **validation**.

Si l'on n'y prend pas garde, la quantité d'information sauvegardée en mémoire stable peut dépasser la capacité de la mémoire. Aussi, au cours de l'exécution, les messages journalisés et les points de reprise constitués devenant non nécessaires pour un

7. "jms" pour "Journalisé en Mémoire Stable".

recouvrement arrière sont effacés. Un **message** m reçu par le processus P_i est **demandé** s'il n'est pas suivi par un état de reprise stable, ou si, entre m et le prochain point de reprise, P_i a émis un message n demandé par P_j [SBY88] :

$$\texttt{demandé}(m) \stackrel{\text{def}}{=} \left\{ \begin{array}{ll} & \nexists R_i^x \in \mathbf{R}_{P_i} : (\texttt{stable}(R_i^x)) \wedge (\texttt{recevoir}(m) \prec_{\mathbf{A}} R_i^x) \\ \vee & \exists \mathcal{A}_i^y = \texttt{émettre}(P_j, n), \exists R_i^x \in \mathbf{R}_{P_i} : \\ & (\texttt{recevoir}(m) \prec_{\mathbf{A}} \mathcal{A}_i^y) \wedge (\mathcal{A}_i^y \prec_{\mathbf{A}} \mathcal{C}_{R_i^x}) \wedge \texttt{demandé}(n) \end{array} \right.$$

Un **point de reprise** R_i^x est **demandé** s'il n'est pas suivi par un état de reprise stable, ou si un message reçu entre R_i^x et R_i^{x+1} est demandé [SBY88] :

$$\texttt{demandé}(R_i^x) \stackrel{\text{def}}{=} \left\{ \begin{array}{ll} & \neg\texttt{stable}(R_i^{x+1}) \\ \vee & \exists \mathcal{A}_i^y = \texttt{recevoir}(m) \in H_{\widehat{P}}^{s,t} : \\ & (\mathcal{C}_{R_i^x} \prec_{\mathbf{A}} \mathcal{A}_i^y) \wedge (\mathcal{A}_i^y \prec_{\mathbf{A}} \mathcal{C}_{R_i^{x+1}}) \wedge \texttt{demandé}(m) \end{array} \right.$$

Les messages et les points de reprise qui ne sont plus demandés, sont effacés de la mémoire stable.

4. Les objectifs

Les premiers critères de choix pour définir une politique de recouvrement arrière sont le déterminisme d'exécution et l'interdépendance entre les mécanismes de base : constitution de points de reprise, journalisation de l'exécution répartie et recouvrement arrière. Dans cette section, nous étudions d'autres critères de choix en mesurant l'efficacité et la performance des mécanismes. Ces critères sont rassemblés sous le terme "objectifs". Nous identifions quatre objectifs du recouvrement arrière : le degré de tolérance aux fautes, les surcoûts pendant l'exécution, l'inhibition pendant l'exécution et la quantité de travail à défaire ou à refaire.

4.1. *Le degré de tolérance aux fautes*

Le **degré de tolérance aux fautes** est le nombre maximum de fautes simultanées tolérées. Les fautes comprennent aussi bien les fautes de l'application que celles des mécanismes de base. C'est l'objectif le plus important du recouvrement arrière. Une politique est efficace si les mécanismes choisis tolèrent la faute globale du système réparti sans effet domino.

Par déduction, cela exclut les politiques choisissant un mécanisme de constitution de points de reprise non coordonnés sans calcul du prédicat `utile` et sans mécanisme de journalisation (par exemple, [BL88]).

En outre, à cause de la synchronisation explicite - une entité contrôle le lancement et la terminaison de l'algorithme de constitution -, les mécanismes de construction d'état globaux cohérents sont sensibles à la défaillance du nœud supportant le coordinateur. Ces mécanismes doivent donc indiquer explicitement comment sont tolérées les fautes du coordinateur. Il en est de même pour les mécanismes de journalisation et de recouvrement arrière centralisés.

4.2. *Les surcoûts pendant l'exécution*

Les **surcoûts** pendant l'exécution sont de trois ordres [DVCL93] : **de charge du réseau**, **de stockage** et **d'exécution**.

Tout d'abord, tous les mécanismes du recouvrement arrière génèrent un trafic de messages sur le réseau de communication.

Ensuite, les mécanismes de constitution de points de reprise et de journalisation sauvegardent des informations en mémoire stable. Nous estimons que les capacités de stockage ne sont pas limitées. Toute opération sur la mémoire stable - distante, donc accessible uniquement à travers le réseau de communication - est synchrone par rapport à l'exécution. Actuellement, c'est le surcoût d'exécution dû à ces accès à la mémoire stable qui est le plus important. Appliqué au mécanisme de constitution de points de reprise, l'objectif est de constituer des points de reprise utiles, de minimiser leur taille et d'échelonner leur enregistrement. Pour la journalisation de l'exécution, l'objectif est de minimiser la taille du journal et la fréquence des sauvegardes. Pour le recouvrement arrière, il est de minimiser le nombre de processus devant se réexécuter.

Tous les mécanismes de base possèdent un temps d'exécution propre. Les mécanismes seront d'autant plus performants qu'ils interfèrent moins avec l'application. En d'autres termes, l'objectif est qu'ils utilisent les ressources aux moments où elles sont disponibles. La *concomitance* évalue le parallélisme d'exécution d'un mécanisme de base avec l'application [Pla93] :

$$\text{Concomitance} = (1 - \frac{\text{Surcoût du mécanisme}}{\text{Temps d'exécution du mécanisme}}) * 100.$$

4.3. *L'inhibition pendant l'exécution*

La concomitance mesure l'intrusion des mécanismes de base dans l'exécution. Dans le cas des systèmes multi-processeurs, cette intrusion peut être atténuée en mettant à profit le parallélisme. Cependant, pour un algorithme donné, certaines intrusions ne pourront jamais être enlevées. Ces dernières sont regroupées sous le terme "**inhibition**".

Les interactions avec le monde extérieur sont une source d'inhibition très forte. En effet, l'état global précédant une émission vers le monde extérieur doit être recouvrable. Cette inhibition est encore plus forte si l'application doit construire un état global cohérent à chaque fois. En conséquence, pour les applications *pm*-déterministes communiquant "fréquemment" avec le monde extérieur, les politiques "souhaitables" incluent un mécanisme de journalisation.

Par ailleurs, les mécanismes de constitution de points de reprise respectant la cohérence forte inhibent de façon très importante l'exécution, puisqu'on ne connaît pas à ce jour de protocoles n'imposant pas l'arrêt complet de l'application pour respecter la cohérence forte.

4.4. *La quantité de travail à défaire ou à refaire*

La performance des mécanismes de base peut également être évaluée en terme de quantité de travail à défaire ou à refaire. Elle se décline selon trois métriques : la dis-

tance de recouvrement arrière, le nombre de processus se réexécutant et le nombre de recouvrements arrière d'un même processus pour une même défaillance.

Premièrement, la distance de recouvrement arrière mesure l'effet domino - ou quantité de travail à défaire. En résumé, les politiques pour lesquelles l'effet domino peut survenir avec une forte probabilité sont considérées non performantes.

Deuxièmement, le nombre de processus devant se réexécuter (c'est le premier paramètre de la quantité de travail à refaire) varie selon les applications et les politiques de recouvrement arrière. Lorsqu'un processus indéterministe est défaillant, tous les processus dépendant causalement des messages qu'il a émis après l'état global calculé, doivent se ré-exécuter. C'est pourquoi les processus des applications indéterministes se réexécutent généralement tous à chaque défaillance. Pour les applications *pm*-déterministes, les mécanismes de journalisation optimistes peuvent laisser apparaître des processus orphelins (*cf.* section 6.). Certains préfèrent donc choisir une journalisation pessimiste, limitant ainsi le nombre des réexécutions aux processus des nœuds défaillants.

Troisièmement, certaines politiques peuvent obliger un même processus à se réexécuter plusieurs fois pour une même défaillance (c'est le deuxième paramètre de la quantité de travail à refaire). C'est le cas des politiques choisissant des constitutions de points de reprise non coordonnés, une journalisation optimiste et des recouvrements arrière ne calculant pas d'état global recouvrable [SY85]. Ces politiques non performantes sont par conséquent déconseillées.

5. Les politiques

Dans cette section, nous faisons la synthèse des sections précédentes. Le but est de proposer une panoplie de politiques de recouvrement arrière ainsi que des règles pour les choisir. Nous écartons explicitement les politiques qui nous semblent inefficaces ou non performantes. Les règles de choix sont divisées en trois catégories. Premièrement, certains des mécanismes de base s'excluent mutuellement. La sous-section 5.1. présente les combinaisons permises (politiques "possibles"). Deuxièmement, le type de l'application (*pm*-déterministe ou indéterministe) conditionne la politique de recouvrement arrière. La sous-section 5.2. étudie plus particulièrement la prise en compte de l'indéterminisme. Troisièmement, certaines politiques possibles ne sont pas "souhaitables", ceci au vu des objectifs énoncés dans la section 4.. La sous-section 5.3. effectue donc un tri parmi les politiques "possibles" pour ne garder que celles qui sont "souhaitables". Ce dernier point concerne principalement les applications *pm*-déterministes.

Nous résumons les résultats par le tableau 1 où ne figurent que les politiques "possibles" et "souhaitables".

Enfin, la sous-section 5.4. montre comment structurer une application en ensemble de processus avec chacun une politique différente.

5.1. *Les politiques possibles*

Tout d'abord, nous classons les mécanismes de base existant dans la littérature et donnons une présentation en une phrase de chaque classe. La présentation succincte

repose sur les concepts de base introduits en section 3.. Le lecteur est invité à se reporter à la section 6. pour une présentation plus complète des algorithmes.

Les classes de mécanismes de constitution de points de reprise sont au nombre de quatre :

- C1 : *constitution d'un état global cohérent* : construction explicitement synchronisée d'un état global cohérent.
- C2 : *constitution d'une barrière de synchronisation* : constitution synchronisée par arrêt global de l'application avec vidage des canaux de communication pour qu'il n'y ait pas de message en transit et pour que les points de reprise soient tous utiles.
- C3 : *constitution de points de reprise non coordonnés* : constitution non synchronisée d'un point de reprise ne cherchant pas à former un état global cohérent.
- C4 : *constitution de points de reprise induite par les communications* : constitution implicitement synchronisée de points de reprise utiles déclenchée par le calcul de prédicats sur les estampilles des messages inter-processus reçus.

Les mécanismes de journalisation sont divisés en quatre classes :

- J1 : *journalisation des messages en transit d'un état global* : journalisation en mémoire stable des messages en transit, donc limitée à la durée de construction de l'état global.
- J2 : *journalisation pessimiste* : journalisation en mémoire stable de tous les messages avant que les processus récepteurs exécutent l'action `recevoir`.
- J3 : *journalisation optimiste* : journalisation en mémoire stable de tous les messages [8], mais non synchronisée avec les réceptions.
- J4 : *journalisation causale* : jounalisation de tous les messages [9], non synchronisée avec les réceptions, par les processus en dépendant causalement.

Nous notons J0 lorsqu'aucun mécanisme de journalisation n'est utilisé.

Quant aux mécanismes de recouvrement arrière, il n'en existe que deux classes :

- R1 : *recouvrement arrière avec calcul de l'état global recouvrable maximal* : recouvrement arrière faisant précéder le début de la réexécution par un algorithme, centralisé ou décentralisé, de calcul de l'état global recouvrable maximal [10].
- R2 : *recouvrement arrière sans calcul de l'état global recouvrable maximal* : recouvrement arrière ne faisant pas précéder le début de la réexécution par un algorithme de calcul de l'état global recouvrable maximal.

8. Cela n'implique pas que le contenu de tous les messages soit enregistré en mémoire stable. En général, seuls les histoires sont enregistrées.

9. Idem.

10. Intuitivement, l'état global recouvrable maximal est l'état tel qu'une quantité minimum de travail soit à défaire.

Les définitions succinctes des différentes classes induisent des relations de dépendances. Nous raisonnons en partant des mécanismes de constitution de points de reprise puis des mécanismes de journalisation et enfin des mécanismes de recouvrement arrière.

Le mécanisme de constitution d'état global cohérent et le mécanisme de constitution induite par les communications nécessitent un mécanisme de journalisation des messages en transit pour éviter l'effet domino. La journalisation n'est demandée que pour les messages traversant l'état global :

$$\mathsf{C1} \vee \mathsf{C4} \Rightarrow \mathsf{J1}.$$

Le mécanisme de barrière de synchronisation ne demande aucune journalisation :

$$\mathsf{C2} \Rightarrow \mathsf{J0}.$$

Le mécanisme de constitution de points de reprise non coordonnés impose la journalisation de tous les messages :

$$\mathsf{C3} \Rightarrow \mathsf{J2} \vee \mathsf{J3} \vee \mathsf{J4}.$$

Pour toutes les politiques de recouvrement arrière, le calcul de l'état global recouvrable maximal est facultatif. Si ces politiques laissent apparaître des messages orphelins, les processus s'apercevant après coup qu'ils sont orphelins, se réexécutent. Nous avons donc la relation suivante :

$$\mathsf{C1} \vee \mathsf{C2} \vee \mathsf{C3} \vee \mathsf{C4} \Rightarrow \mathsf{R1} \vee \mathsf{R2}.$$

5.2. *La prise en compte de l'indéterminisme*

La réexécution d'une application dépend de l'état global cohérent de reprise et des messages en transit. Or, une application indéterministe ne garantit pas la réexécution des réceptions des messages en transit. Donc, aucun message en transit ne doit traverser les états globaux correspondant aux lignes de reprise. Par conséquent, seules des constitutions d'une barrière de synchronisation sont autorisées :

$$\neg \mathtt{déterministe}(\widehat{P}) \Rightarrow \mathsf{C2}.$$

Cette règle est plus stricte que la plupart de celles rencontrées dans la littérature. En effet, la grande majorité des études considèrent qu'une application indéterministe peut se satisfaire de points de reprise utiles. Cela signifie que l'application est supposée ne pas posséder un comportement indéterministe entre la sauvegarde de l'image du processus en mémoire stable et les réceptions des messages en transit. En supposant qu'il soit possible de définir l'ensemble des actions indéterministes, nous considérons donc que C1 et C4 (dans ce cas notées $\mathsf{C1}^*$ et $\mathsf{C4}^*$) sont aussi possibles :

$$\neg \mathtt{déterministe}(\widehat{P}) \Rightarrow \mathsf{C1}^* \vee \mathsf{C2} \vee \mathsf{C4}^*$$

La règle précédente limite le nombre de politiques possibles pour les applications indéterministes. Une application ayant un comportement tantôt déterministe tantôt indéterministe utilise les mécanismes $C1^*$ ou bien $C2$ ou encore $C4^*$ pour passer d'une "phase déterministe" à une "phase indéterministe" (et inversement). À l'heure actuelle, peu de recherches se sont intéressées au recouvrement arrière pour des applications composées de sous-ensembles de processus de types différents : *pm*-déterministes ou indéterministes (*cf.* section 7.) [JZ90a, JZ91, Con97].

En conclusion, puisqu'aucun mécanisme de journalisation n'est utilisé - excepté pour $C1^*$ et $C4^*$- et puisque les états de reprise constitués sont automatiquement recouvrables, nous obtenons la règle suivante :

$$\neg \texttt{déterministe}(\widehat{P}) \Rightarrow (((C1^* \vee C4^*) \wedge J1 \wedge R2) \vee (C2 \wedge J0 \wedge R2)).$$

Lorsque survient une défaillance, tous les processus dépendant des processus des nœuds défaillants se réexécutent. La réexécution n'est pas équivalente. Enfin, les politiques peuvent être classées par ordre décroissant de performance. La politique la plus performante est celle qui ne synchronise pas les constitutions de points de reprise : $C4^* \wedge J1 \wedge R2$. Ensuite, vient celle qui limite l'inhibition due à la synchronisation des points de reprise : $C1^* \wedge J1 \wedge R2$. Puis, arrive celle qui arrête les communications de l'application : $C2 \wedge J0 \wedge R2$. Pour les deux premières politiques, l'ordre indiqué n'est pas respecté pour toutes les applications. En effet, nous verrons dans la sous-section 6.2. que le calcul du prédicat `utile` s'effectuant à chaque réception de message est important. Par conséquent, cet ordonnancement suppose que les accès concomitants à la mémoire stable lors des "quelques" constitutions synchronisées inhibent l'application plus que le calcul du prédicat `utile` lors des "nombreuses" réceptions. Cette hypothèse n'a jamais été vérifiée par une comparaison d'implantations.

5.3. *Les politiques souhaitables*

Nous déterminons maintenant les politiques "souhaitables" comme étant celles qui sont performantes. Pour ce faire, nous analysons les politiques possibles au regard des objectifs du recouvrement arrière.

Pour des applications *pm*-déterministes, il est clair qu'il est inefficace de journaliser tous les messages pour ensuite privilégier une réexécution non équivalente. Si l'application est grande - en nombre de processus ou en espace mémoire -, les constitutions synchronisées de points de reprise créent un goulet d'étranglement au niveau de la mémoire stable [KMBT92]. Si l'application communique fréquemment avec le monde extérieur, la validation des messages est plus rapide que la constitution d'un point de reprise global. Toutefois, si les processus communiquent intensément, il peut être préférable de synchroniser les constitutions et de n'enregistrer que quelques messages. Ainsi, nous obtenons deux catégories d'applications. L'une choisit des constitutions de points de reprise non coordonnés avec la journalisation de tous les messages :

$$\texttt{appli\_grande}(\widehat{P}) \Rightarrow C3 \wedge (J2 \vee J3 \vee J4),$$

avec le prédicat `appli_grande` évalué à "vrai" ssi l'application est grande. L'autre catégorie privilégie les états globaux cohérents ou les constitutions de points de reprise

utiles avec la journalisation de quelques messages :

$$\texttt{appli\_comm\_intenses}(\widehat{P}) \Rightarrow (((\mathsf{C1} \vee \mathsf{C4}) \wedge \mathsf{J1}) \vee (\mathsf{C2} \wedge \mathsf{J0})),$$

avec le prédicat `appli_comm_intenses` évalué à "vrai" ssi les communications sont intenses. Pour les applications appartenant aux deux catégories à la fois, l'étude approfondie des implantations du mécanisme de recouvrement et la pratique sont nécessaires.

Ensuite, le mécanisme de journalisation détermine le choix d'une politique pour chaque catégorie. C'est un compromis entre la performance pendant l'exécution et la quantité de travail à défaire.

Pour la première catégorie ci-dessus, la journalisation pessimiste est préférée lorsque l'exécution le permet et lorsque l'on privilégie des réexécutions courtes [HW95]. Cette politique ne nécessite pas le calcul de l'état global recouvrable maximal avant le début de la réexécution : $\mathsf{C3} \wedge \mathsf{J2} \wedge \mathsf{R2}$. Au contraire, la journalisation optimiste parie sur une probabilité de défaillance faible. Elle favorise donc l'exécution au détriment de recouvrements qui risquent d'être plus longs. En outre, afin de limiter le nombre de réexécutions et donc la quantité de travail à refaire, cette politique demande le calcul de l'état global recouvrable maximal avant le début de la réexécution : $\mathsf{C3} \wedge \mathsf{J3} \wedge \mathsf{R1}$. La journalisation répartie sur l'application combine les avantages des deux journalisations précédentes. Alvisi et Marzullo montrent quelle est optimale [AM94]. Cette politique, non utilisable si l'on désire tolérer les fautes globales (tous les processus sont défaillants), ne demande pas de calcul de l'état global maximal recouvrable : $\mathsf{C3} \wedge \mathsf{J4} \wedge \mathsf{R2}$.

Pour la deuxième catégorie citée ci-dessus, le choix des politiques suit ce qui a été énoncé dans le paragraphe précédent pour les applications indéterministes.

En résumé, les politiques de recouvrement arrière se divisent en trois classes selon trois types d'application. Le tableau 1 présente, en fonction des types d'application, les politiques possibles et souhaitables et les types de réexécutions.

Types de l'application $\widehat{P}$	**Politiques possibles et souhaitables**	**Types de réexécution**
$\neg$ `déterministe`$(\widehat{P})$	$\mathsf{C4^*} \wedge \mathsf{J1} \wedge \mathsf{R2}$	$\neg$ `équivalente`
	$\mathsf{C1^*} \wedge \mathsf{J1} \wedge \mathsf{R2}$	
	$\mathsf{C2} \wedge \mathsf{J0} \wedge \mathsf{R2}$	
`pm-déterministe`$(\widehat{P}) \wedge$ `appli_comm_intenses`$(\widehat{P})$	$\mathsf{C4} \wedge \mathsf{J0} \wedge \mathsf{R2}$	$\neg$ `équivalente`
	$\mathsf{C1} \wedge \mathsf{J1} \wedge \mathsf{R2}$	
	$\mathsf{C2} \wedge \mathsf{J0} \wedge \mathsf{R2}$	
`pm-déterministe`$(\widehat{P}) \wedge$ `appli_grande`$(\widehat{P})$	$\mathsf{C3} \wedge \mathsf{J4} \wedge \mathsf{R2}$	`équivalente`
	$\mathsf{C3} \wedge \mathsf{J3} \wedge \mathsf{R1}$	
	$\mathsf{C3} \wedge \mathsf{J2} \wedge \mathsf{R2}$	

TAB. 1 - *Les politiques de recouvrement arrière possibles et souhaitables*

5.4. *La structuration de l'application en groupes de processus*

L'application répartie peut être divisée en groupes de processus, chacun possédant sa propre politique. L'objectif est d'adapter le recouvrement arrière aux très grandes

applications. En voici les principales raisons :

- Le nombre de messages transmis sur le réseau est proportionnel au nombre de processus et certains mécanismes utilisent des diffusions.
- La taille des vecteurs de dépendance est proportionnelle au nombre de processus. Les vecteurs de dépendance repèrent les relations de précédence causale. Lorsqu'ils sont grands, les messages inter-processus risquent de générer plusieurs messages transmis sur le réseau.
- L'application s'étendant sur plusieurs réseaux locaux peut utiliser plusieurs protocoles de communication.
- La probabilité de recouvrement arrière est proportionnelle au nombre de processus.
- La probabilité d'apparition de fautes augmente avec le nombre de nœuds dans le système réparti.

Sistla et Welch regroupent les processus s'exécutant sur les nœuds d'un même réseau local [SW89]. Les messages entre groupes sont traités comme des messages à destination du monde extérieur.

Lowry et *al*. construisent des passerelles s'occupant des communications entre groupes [LRG91]. Une passerelle est responsable de la transmission des messages entre deux groupes dans une direction. La passerelle jumelle s'occupe des messages transmis dans le sens inverse. En conséquence, l'histoire séquentielle d'une passerelle est composée de réceptions de messages en provenance d'un groupe et d'émission de messages à destination d'un autre groupe. Le mécanisme de journalisation des passerelles peut être pessimiste ou optimiste. Dans ce dernier cas, la validation d'un état nécessite un protocole entre passerelles.

6. Les mécanismes

Dans cette section, les trois mécanismes de base du recouvrement arrière sont présentés en détail.

6.1. *La constitution de points de reprise*

Pendant l'exécution, le mécanisme de constitution de points de reprise enregistre en mémoire stable des états locaux des processus. À la fin de la section, le tableau 2 rassemble les références bibliographiques regroupées par classes.

C1 : La constitution d'un état global cohérent

Le principe des algorithmes de la première classe est de construire un état global cohérent et d'enregistrer les messages en transit. [HMR93] est une synthèse des algorithmes de constitution d'états globaux cohérents.

Le premier algorithme a été donné par Chandy et Lamport [CL85]. Il suppose que les messages sont reçus dans l'ordre de leur émission - les communications sont du

type FIFO (en anglais, First In First Out). Le processus initiateur enregistre son état en mémoire stable. Ensuite, il lance l'établissement de l'état global par l'émission sur tous ses canaux d'un message de contrôle appelé marqueur. À la réception d'un marqueur, tout processus sauvegarde son état en mémoire stable et émet un marqueur sur tous ses canaux. Pour un processus, pour un canal c, les messages en transit sont ceux reçus par c entre l'enregistrement de l'état et la réception du marqueur sur c. Pour un processus, l'algorithme est terminé lorsqu'un marqueur a été reçu sur tous ses canaux. Le processus prévient ensuite l'initiateur de la constitution effective du point de reprise. Une deuxième phase est nécessaire pour tolérer les fautes de processus - exceptées celles de l'initiateur - pendant l'exécution de l'algorithme. Elle consiste à transformer les points de reprise effectifs en points de reprise permanents [KT87]. Pour tolérer les fautes de l'initiateur, l'algorithme doit posséder une troisième phase.

En conclusion, dans cet algorithme, les messages ne possèdent aucune estampille. D'autres algorithmes éliminent les marqueurs et ajoutent une estampille - généralement une marque de couleur - pour distinguer les messages émis avant ou après l'état global. Enfin, les algorithmes de cette classe qui tolèrent le non-ordonnancement des transmissions de messages sont nombreux [LY87, LRV87, Ahu93].

C2 : La constitution d'une barrière de synchronisation

Le principe de ces algorithmes est d'abord d'arrêter les communications, puis d'attendre que tous les messages soient reçus, et enfin de sauvegarder les états locaux [BS83]. Puisque toutes les communications interprocessus sont arrêtées et tous les messages reçus, tous les canaux sont vides. Donc, l'état global est cohérent et aucun message en transit ne le traverse. Pour l'arrêt des processus et le vidage des canaux, les algorithmes s'appuient sur un mécanisme de diffusion.

Les mécanismes sont peu nombreux dans cette première classe, principalement parce que l'inhibition est très forte. Cette solution est toutefois acceptable pour des applications indéterministes.

C3 : La constitution de points de reprise non coordonnés

Le principe de base de ces algorithmes est la non synchronisation de la constitution du point de reprise d'un processus avec les autres processus. La constitution comprend l'enregistrement en mémoire stable de l'état du processus et des messages demandés émis avant la constitution [SBY88]. Parallèlement, le mécanisme de journalisation sauvegarde en mémoire stable l'histoire répartie. Les messages sont estampillés pour la journalisation de l'histoire répartie. Un point de reprise est entièrement constitué lorsque l'état du processus et les messages demandés sont sauvegardés, et, lorsque l'état de reprise est stable. Si le mécanisme de journalisation enregistre déjà en mémoire stable le contenu des messages - en plus de leur histoire -, le calcul et l'enregistrement des messages demandés ne sont pas nécessaires [SY85]. Le calcul et l'enregistrement du contenu des messages demandés au moment des constitutions est plus performant que l'enregistrement du contenu de tous les messages.

La tolérance aux fautes des constitutions de points de reprise non coordonnés dépend de la tolérance aux fautes du mécanisme de journalisation associé.

C4 **: La constitution de points de reprise induite par les communications**

Le principe de ces algorithmes est le calcul du prédicat `utile` "au vol" (c.-à-d., pendant l'exécution, à la réception des messages et uniquement à l'aide des informations contenues dans les estampilles des messages), pour la détermination des instants de constitution des points de reprise. [HMR97] est une synthèse des algorithmes de cette classe.

Les processus décident spontanément de constituer des points de reprise. L'algorithme peut forcer la constitution de points de reprise supplémentaires afin de préserver l'utilité des points de reprise spontanés. L'objectif est alors de constituer le moins possible de points de reprise forcés. Le calcul de cette dernière condition est effectuée de deux manières différentes.

La première approche consiste à repérer les chemins zigzagants causaux et surtout non causaux [XN93] pour éviter qu'une dépendance apparaisse entre l'intervalle de reprise précédant le point de reprise spontané et celui commençant avec ce point de reprise. Reprenons les figures 1 et 2. À cause de la relation $(I_3^1 \prec_{\mathbf{I}} I_1^1) \wedge (I_1^1 \prec_{\mathbf{I}} I_2^0) \wedge (I_2^0 \prec_{\mathbf{I}} I_3^0)$, le point de reprise $\mathcal{A}_3^2$ est inutile. Le chemin zigzagant (m_2^1, m_3^1, m_1^1) est causal : $(\texttt{recevoir}(m_2^1) \prec_{\mathbf{A}} \texttt{émettre}(m_3^1)) \wedge (\texttt{recevoir}(m_3^1) \prec_{\mathbf{A}} \texttt{émettre}(m_1^1))$. Ce chemin zigzagant est "cassé" par la constitution d'un point de reprise de P_1 avant $\mathcal{A}_1^3$. Le point de reprise $\mathcal{A}_3^2$ reste alors utile. De même, à cause de la relation $(I_3^3 \prec_{\mathbf{I}} I_1^2) \wedge (I_1^2 \prec_{\mathbf{I}} I_2^1) \wedge (I_2^1 \prec_{\mathbf{I}} I_3^2)$, le point de reprise $\mathcal{A}_3^6$ est inutile. Le chemin zigzagant (m_2^2, m_3^2, m_1^2) est lui non causal : $\texttt{recevoir}(m_2^2) \not\prec_{\mathbf{A}} \texttt{émettre}(m_3^2)$. Ce chemin zigzagant est "cassé" par la constitution d'un point de reprise de P_2 avant $\mathcal{A}_2^5$. Le point de reprise $\mathcal{A}_3^6$ reste alors utile. À ce jour, le protocole forçant le moins de points de reprise est décrit dans [HMNR97]. Cependant, l'existence et la recherche d'un protocole optimal restent des problèmes ouverts.

La deuxième approche résout un problème un peu différent du nôtre appelé RDT (en anglais, Rollback-Dependency Trackability) [Wan97]. Il s'agit de calculer "au vol" le premier état global cohérent auquel appartient un point de reprise donné et de prendre les points de reprise forcés nécessaires afin de préserver l'utilité du point de reprise initial. La condition exprimant le RDT est plus conservative que celle exprimant le traquage des chemins zigzagants : le nombre de points de reprise forcés est plus important. Pour cette deuxième approche, le protocole optimal est connu [BHMR97].

Les références bibliographiques des mécanismes de constitution de points de reprise sont regroupées selon la classe de leur algorithme dans le tableau 2.

6.2. *La journalisation de l'exécution répartie*

Le mécanisme de journalisation sauvegarde l'histoire de l'application pendant l'exécution. Par ailleurs, la journalisation optimiste nécessite l'exécution d'un algorithme de validation des messages à destination du monde extérieur. Ces algorithmes sont présentés avec la classe optimiste. À la fin de la section, le tableau 3 rassemble les références bibliographiques regroupées par classes.

J1 **: La journalisation des messages en transit d'un état global**

Classes	Références bibliographiques
C1	[BCS84], [CL85], [KT87], [SK86], [LY87], [LB88], [LRV87], [VRL87], [Ven89], [TKT89], [CJ91], [SS92], [Ahu93], [Eln93], [Vai94], [Con96].
C2	[BS83], [KMBT92], [LFS93].
C3	[BBG83] [PP83], [SY85], [JZ87], [SBY88], [BBG+89], [Joh89], [SW89], [JV91], [AHM93], [Sen95], [WHV+95], [Con96].
C4	[BHMR95a], [MS96], [BHMR97], [BHR97], [HMNR97], [HMNR97], [QBC97], [Wan97].

TAB. 2 - *Les mécanismes de constitution de points de reprise - références bibliographiques*

Lorsque l'état global est construit à partir d'un mécanisme de constitution de points de reprise à synchronisation explicite (classe C1), le calcul de l'ensemble des messages en transit est trivial pour les canaux FIFO. Ce sont les messages émis avant la réception du marqueur du processus émetteur et reçus avant la fin de l'algorithme chez le processus récepteur. Si les canaux ne sont pas FIFO, la terminaison de l'algorithme est moins évidente et nécessite l'exécution d'un algorithme de terminaison qui visite tous les nœuds de la coupe [Mat89]. Dans tous les cas, la journalisation ne commence qu'à la réception du premier marqueur et se termine avec l'algorithme de constitution de points de reprise.

La synchronisation implicite des processus (classe C4) et donc l'absence de marqueur oblige la sauvegarde en mémoire volatile de tous les messages. Ces messages n'ont besoin d'être enregistré en mémoire stable que s'ils sont en transit dans un ou plusieurs états globaux cohérents. Par conséquent, étant donné un ensemble quelconque de points de reprise, le problème est le calcul de l'ensemble des états globaux cohérents auquel appartient cet ensemble. Jusqu'à maintenant, cette question a été peu abordée. [Wan97] décrit deux algorithmes qui donnent les points de reprise repérant les états globaux minimal et maximal ; [MNS97] décrit un algorithme énumérant tous les états globaux cohérents. Une solution serait donc d'utiliser les états globaux cohérents fournis par ces algorithmes pour calculer l'ensemble des messages en transit à enregistrer en mémoire stable. [WF92], [XNM95], [MR96] définissent, lors des réceptions, des classes de messages qui ne peuvent être en transit dans aucun état global cohérent et réduisent ainsi la quantité d'information à enregistrer en mémoire stable.

J2 : La journalisation pessimiste

Le principe de la journalisation pessimiste est de sauvegarder le contenu et l'estampille des messages avant leur réception. Le premier algorithme est proposé par Powell et Presotto [PP83]. Les messages sont dirigés vers un nœud dédié qui estampille les messages, les enregistre en mémoire stable et ensuite les fait suivre aux processus récepteurs. Remarquons que si le nœud d'estampillage est celui du système de gestion de la mémoire stable, cela équivaut à dire que la journalisation est effectuée par les processus émetteurs. L'autre alternative rencontrée est la journalisation par les processus récepteurs. Dans ce cas, les processus effectuent eux-mêmes l'estampillage des mes-

sages. La journalisation par le récepteur est plus performante que la journalisation par l'émetteur [EZ94, Con96].

Puisque la mémoire stable est supposée sûre de fonctionnement, la journalisation est tolérante aux fautes.

J3 : La journalisation optimiste

Le principe de la journalisation optimiste est de recevoir immédiatement les messages et de sauvegarder ultérieurement en mémoire stable l'histoire des derniers messages reçus - voire aussi leur contenu. Des messages orphelins risquent donc d'apparaître lors des défaillances. Le premier algorithme de cette classe enregistre l'histoire et le contenu des messages [SY85]. Plus tard, les auteurs montrent que seule l'histoire des messages a besoin d'être enregistrée en mémoire stable [SBY88]. Dans ce cas, les messages demandés sont enregistrés au moment des constitutions de points de reprise. En outre, le contenu des messages demandés non encore enregistrés en mémoire stable est gardé en mémoire volatile par les émetteurs. Juang et Venkatesan démontrent que, pour les processus défaillants, les messages demandés qu'ils ont émis depuis leur dernier point de reprise stable sont automatiquement réémis lors de la réexécution [JV91]. Enfin, tous les algorithmes de cette classe effectuent la journalisation en mémoire stable au niveau des processus récepteurs.

La journalisation optimiste impose l'exécution d'un algorithme de validation des messages à destination du monde extérieur. Il existe deux catégories d'algorithmes de validation : centralisés et décentralisés.

Dans [Joh89], les processus désignent un nœud coordinateur récupérant toute l'histoire répartie. À chaque sauvegarde de leur histoire, les processus fournissent leur vecteur des dépendances directes. Un algorithme met à jour une matrice contenant, pour chaque processus, l'état stable cohérent connu [JZ90b]. Donc, lorsqu'un processus a besoin de valider son état, il s'adresse au coordinateur. Notons que les estampilles des messages possèdent uniquement un scalaire car ce sont les vecteurs des dépendances directes qui sont calculées. Le nœud coordinateur est le nœud gestionnaire de la mémoire stable. Dans l'implantation décrite, ce nœud est répliqué. La journalisation est donc tolérante aux fautes.

Strom et *al.* présentent le principe des algorithmes décentralisés dans [SY85]. Chaque processus maintient un vecteur de connaissance du dernier état de journalisation des processus et un vecteur des dépendances transitives. Premièrement, cela implique que les vecteurs des dépendances sont ajoutés en estampille des messages pour le calcul de la transitivité. Deuxièmement, il s'ensuit que la validation d'un message consiste à demander les vecteurs de journalisation aux autres processus. La complexité de l'algorithme par rapport au nombre de messages est au pire en $\mathcal{O}(\mid \widehat{\mathbf{P}}_{TF} \mid)$. Si les processus maintiennent seulement un vecteur des dépendances directes, Sistla et Welch montrent que la complexité de l'algorithme est au mieux en $\mathcal{O}(\mid \widehat{\mathbf{P}}_{TF} \mid)$ et au pire en $\mathcal{O}(\mid \widehat{\mathbf{P}}_{TF} \mid^3)$ messages [SW89]. Les algorithmes répartis sont naturellement tolérants aux fautes, car la faute d'un processus entraîne la réexécution de l'algorithme pour le calcul de l'état global recouvrable maximal (*cf.* sous-section 6.3.).

J4 **: La journalisation causale**

Le principe de la journalisation causale est de diffuser l'histoire séquentielle d'un processus sur les nœuds où s'exécutent les autres processus de l'application. L'idée est que seuls les processus dépendant du message m reçu par P_i ont besoin de connaître son histoire. Donc, P_i transmet l'histoire de m en estampille des messages émis après la réception de m.

Le premier algorithme de cette classe a été proposé par Elnozahy et Zwaenepoel [EZ92]. Les processus construisent un "graphe d'antécédence". Localement, pour un processus, ce graphe contient l'histoire répartie dont le processus dépend depuis la dernière sauvegarde en mémoire stable du graphe. Régulièrement - de façon optimiste -, le processus enregistre en mémoire stable son graphe local non encore enregistré. Les estampilles des messages contiennent les graphes locaux non encore enregistrés en mémoire stable. À la réception d'un message, le processus ajoute le graphe contenu dans l'estampille à son graphe local. Les auteurs montrent que les messages ne peuvent pas devenir orphelins. Pour valider son état, un processus enregistre en mémoire stable son graphe local.

Par la suite, Alvisi et Marzullo montrent que l'algorithme de Elnozahy et Zwaenepoel est une instance d'un algorithme générique prenant en paramètre le degré de tolérance aux fautes f. L'idée générale consiste à garder l'histoire d'un message uniquement en mémoire volatile et d'arrêter sa propagation dès que l'on sait qu'elle est présente sur au moins f nœuds. Les auteurs dérivent plusieurs algorithmes de l'algorithme générique. Les canaux de communication sont supposés de type FIFO. La différence des algorithmes tient dans l'approximation de l'ensemble des sites possédant déjà l'histoire de m et donc dans la taille des estampilles. Meilleure est l'approximation, plus grande est l'estampille.

Les algorithmes parus dans [JZ87, AHM93] sont vus comme des instances de l'algorithme générique avec f égal à 1. f est égal à $\mid \widehat{\mathbf{P}}_{TF} \mid$ pour l'algorithme d'Elnozahy et Zwaenepoel.

En conclusion, les algorithmes de cette classe combinent les avantages des journalisations pessimiste - pas de message orphelin - et optimiste - faible inhibition des réceptions.

Les références bibliographiques des mécanismes de journalisation sont regroupées selon la classe de leur algorithme dans le tableau 3.

Classes	**Références bibliographiques**
J1	[WF92], [XNM95], [MR96].
J2	[BBG83], [PP83], [BBG$^+$89], [Sen95], [WHV$^+$95], [LY87], [Con96].
J3	[SY85], [Joh89], [SW89], [SBY88], [Joh93], [JV91], [Con96].
J4	[AHM93], [EZ92], [Eln93], [AM94].

TAB. 3 - *Les mécanismes de journalisation - références bibliographiques*

6.3. *Le recouvrement arrière*

À la suite de défaillances, le mécanisme de recouvrement arrière détermine l'ensemble des processus devant se réexécuter, puis récupère les informations de reprise, et enfin, lance la réexécution. Les algorithmes de recouvrement arrière ne présentent pas de difficulté particulière, excepté la connaissance des processus non défaillants. Ce dernier point dépasse le cadre de notre étude, nous le laissons donc de côté.

Mises à part les politiques avec journalisation optimiste, les algorithmes de recouvrement arrière sont relativement simples. Tout d'abord, un processus détectant une défaillance diffuse un message d'avertissement. Ensuite, un processus non défaillant est élu. Celui-ci détermine les processus - défaillants ou non - devant se réexécuter, et enfin les redémarre. Ces derniers reconnectent leurs canaux de communication et la reprise commence. Pendant tout ce temps, les autres processus continuent leur exécution ainsi que la détection des fautes - notamment celle du coordinateur.

En ce qui concerne les algorithmes de calcul de l'état global recouvrable maximal, ils sont très proches des algorithmes de validation. Aussi, nous laissons le lecteur se reporter aux références bibliographiques citées à la sous-section 6.2..

Koo et Toueg démontrent que des messages orphelins peuvent provoquer un bouclage infini des recouvrements arrière [KT87]. Le modèle de système réparti asynchrone autorise qu'un message émis avant l'exécution de l'algorithme de recouvrement arrière ne soit reçu qu'après le début de la réexécution. Ce message est alors orphelin si l'émetteur est parmi les processus défaillants. Par suite, le processus récepteur est orphelin et se réexécute. Rien n'empêche que ce dernier processus ait émis un message devenant lui aussi orphelin, et ainsi de suite.

Contrairement à Koo et Toueg, nous considérons que ceci est un problème d'implantation plutôt qu'algorithmique. En effet, le bouclage infini ne peut apparaître que si les canaux de communication survivent aux défaillances. Pour ce faire, un numéro d'incarnation [SY85] est ajouté en estampille des messages. Le principe est de marquer les messages avec le nombre de réexécutions commencées. Dès lors, un processus connaissant les nouveaux numéros de réexécutions évite la réception des messages de l'ancienne époque. Il faut seulement que les échanges de messages soient arrêtés entre la détection des défaillances et la réception du message précisant les processus devant se réexécuter [KT87].

Dans la réalité, les canaux de communication sont le plus souvent physiques. Ils servent alors pour la détection des défaillances. Par conséquent, nombreux sont les mécanismes ne gérant pas les messages traversant les époques.

7. De nouvelles voies de recherche

Les nouvelles voies de recherche portent principalement sur l'extension du champ d'application du recouvrement arrière.

Architecture : Le recouvrement arrière est également appliqué aux multiprocesseurs et aux multicalculateurs. Des mécanismes ont déjà été conçus pour ces systèmes et pour

différents modèles de mémoire [11] : UMA (en anglais, Uniform-Memory-Access) [Ber88], NUMA (en anglais, Non-Uniform-Memory-Access) [AFM90, WFP90, BGJ+93, Pla93, JF94], COMA (en anglais, Cache-Only-Memory-Architecture) [BGM94] et NORMA (en anglais, NO-Remote-Memory-Access) [WF90, Pla93]. Ces mécanismes de recouvrement arrière sont très différents. Dans le but de choisir le système d'exécution le plus approprié, les systèmes répartis tendent à intégrer ces systèmes fortement couplés dans un même réseau. En conséquence, les mécanismes des systèmes faiblement couplés doivent être revus pour coopérer avec ceux des systèmes fortement couplés et les intégrer.

Système d'exploitation : Parmi les avancées récentes dans le domaine des systèmes d'exploitation, le principe des acteurs multi-activités occupe une place prépondérante. Le problème avec ces acteurs est que les activités partagent un même espace mémoire. Donc, en plus des interactions par échanges de messages, les algorithmes doivent prendre en compte les interactions par mémoire partagée. Un autre problème lié aux systèmes d'exploitation est la saisie et le redémarrage des processus. Le problème consiste en la définition de l'état du processus par rapport à l'état du système d'exploitation. À ce sujet, tout prône à évoluer vers des systèmes dits à "image unique" assurant une transparence totale [Tan95].

Réseaux de communication : Le domaine des réseaux est sans doute le domaine le plus en mutation. Avec les réseaux à hauts débits, le coût des communications dans un même réseau local est grossièrement divisé par un facteur de dix. Par conséquent, le compromis entre le temps de calcul et le nombre de messages est à ré-évaluer : par exemple, l'utilisation plus abondante des diffusions est peut-être possible. Avec les systèmes mobiles, le premier problème à résoudre est la définition de ce qu'est une faute. En outre, les systèmes mobiles changent de lieu de connexion. Ainsi, la reconfiguration et la récupération en sont compliquées : en plus des problèmes de localisation, les changements de configuration sont fréquents et les données de récupération (elles aussi pouvant être mobiles) doivent être accessibles lors de la reprise. Par conséquent, l'utilisation de la mémoire stable et la récupération d'information doivent être repensées [AB94, NF97]. Enfin, les connexions entre réseaux locaux puis nationaux et ensuite continentaux deviennent de plus en plus performantes. Naturellement, les utilisateurs conçoivent des applications se diffusant sur toujours plus de nœuds toujours plus éloignés les uns des autres. Les questions qui sont d'actualité pour les réseaux à hauts débits et les systèmes mobiles, le sont aussi pour les réseaux étendus.

Modèle d'application : Dans la littérature, le recouvrement arrière est aussi examiné pour le modèle des variables partagées [WF90, JF93, SMK+94] et le modèle orienté objets [SDP85, DS87, LA90]. Chaque modèle cible un ensemble de catégories d'application. Il ne semble donc pas intéressant de les unifier. En revanche, le modèle de passages de messages peut bénéficier des résultats des recherches sur les autres modèles. Citons deux exemples. Le traitement des interactions avec la mémoire stable est un problème encore mal traité pour notre modèle. Par analogie avec le modèle orienté

11. *cf.* [Hwa93] pour une définition précise des modèles de mémoire.

objets, l'utilisation de bases de données pour gérer la mémoire stable semble une voie de recherche prometteuse. Un autre exemple est le traitement des transmissions de messages. Par analogie avec les deux autres modèles, les messages n'ont pas tous le même impact sur l'exécution des processus récepteurs. Ils doivent par exemple être reçus "exactement une fois", "au moins une fois", "au plus une fois" ou "sans contrainte". Cette voie de recherche commence à être étudiée [MS89, WF92, LA94, BHMR95a, BHR95, BHMR95b]. Notre conviction est qu'elle peut jouer un rôle important dans la prise en compte de l'indéterminisme, et plus particulièrement, dans la cohabitation de processus de types différents [Con97].

Modèle de fautes : L'évolution des techniques de tolérance aux fautes montre que le défi de la sûreté de fonctionnement est actuellement la maîtrise des fautes logicielles. Dorénavant, toute méthode de tolérance aux fautes doit prendre en compte le traitement des fautes logicielles. À ce sujet, le recouvrement arrière conjointement utilisé avec un mécanisme d'exception peut donner de bons résultats. Néanmoins, il faut étudier plus précisément comment l'hypothèse du mode "silence sur défaillance" peut être relâchée.

8. Conclusion

La reprise sur erreur par recouvrement arrière automatique est une méthode de tolérance logicielle aux fautes matérielles transparente et économique pour l'utilisateur. À la base de cette méthode, le concept de processus fiable met en œuvre trois mécanismes : la constitution de points de reprise et la journalisation de messages pendant l'exécution, et le recouvrement arrière proprement dit lors d'occurrences de fautes.

La répartition introduit deux problèmes pour la reprise. Le premier : l'indéterminisme d'exécution, traduit le fait que deux messages reçus consécutivement lors de l'exécution peuvent être reçus dans un ordre différent lors de la réexécution. Il en résulte le recours à la journalisation des messages. Cependant, la journalisation ne peut pas être utilisée si certaines actions internes sont indéterministes : par exemple, les actions dépendantes de l'horloge physique du nœud ou de variables du système d'exploitation. Les processus ne possédant pas d'actions internes indéterministes sont dits "déterministes par morceaux". Le deuxième problème : la cohérence d'état global, reflète la relation de causalité entre les actions d'émission et de réception d'un même message : un message ne peut pas être vu reçu sans être vu émis. La conséquence principale est l'impossibilité de prendre une photo en "instantané" d'une application répartie.

Les premiers critères de choix pour définir une politique de recouvrement arrière sont le déterminisme d'exécution et l'interdépendance entre les mécanismes de base : constitution, journalisation et recouvrement. Ensuite, les critères de performance sont à prendre en compte. Il s'agit du degré de tolérance aux fautes, des surcoûts, de l'inhibition et de la quantité de travail à défaire ou à refaire. Nous avons ainsi identifié neuf politiques efficaces et performantes et trois catégories d'applications. Les applications indéterministes n'utiliseront pas de mécanisme de journalisation mais un mécanisme de constitution d'états globaux cohérents ; les réexécutions ne seront donc pas équivalentes à l'exécution. Les applications déterministes-par-morceaux à grand nombre

de processus préféreront échelonner les constitutions de points de reprise et utiliseront donc un mécanisme de constitution de points de reprise non coordonnés avec un mécanisme de journalisation des messages ; les réexécutions seront équivalentes à l'exécution. Enfin, les applications déterministes-par-morceaux échangeant de très nombreux messages préféreront ne pas journaliser ces derniers et utiliseront donc un mécanisme de constitution d'états globaux cohérents sans journalisation de messages ; les réexécutions ne seront pas équivalentes à l'exécution.

Les avancées en matière de systèmes d'exploitation et de réseaux ouvrent de nouvelles perspectives d'application aux techniques de reprise sur erreur par recouvrement arrière. Ces techniques devront cependant être adaptées pour prendre en compte l'hétérogénéité des modèles de mémoire, la mobilité des informations de reprise et les systèmes à grande échelle.

Références

[AB94] A. Acharya and B.R. Badrinath. Checkpointing Distributed Application on Mobile Computers. In *Proc. 3rd International Conference on Parallel and Distributed Information Systems*, Austin (USA), September 1994.

[AFM90] E. Ahmed, R, R.C. Frazier, and P.N. Marinos. Cache-Aided Rollback Error Recovery (CARER) Algorithms for Shared-Memory Multiprocessor Systems. In *Proc. 20th IEEE Symposium on Fault Tolerant Computing*, June 1990.

[AHM93] L. Alvisi, B. Hoppe, and K. Marzullo. Nonblocking and Orphan-Free Message Logging Protocols. In *Proc. 23th IEEE Symposium on Fault Tolerant Computing*, Toulouse (France), June 1993.

[Ahu93] M. Ahuja. Global Snapshots for Asynchronous Distributed Systems with Non-FIFO Channels. In Z. Yang and T.A. Marsland, editors, *Global States and Time in Distributed Systems*. IEEE Computer Society Press, 1993.

[AM94] L. Alvisi and K. Marzullo. Message Logging: Pessimistic, Optimistic, Causal and Optimal. In *Proc. 14th International Conference on Distributed Computing Systems*, Poznan (Poland), May 1994.

[BBG83] A. Borg, J. Baumbach, and S. Glazer. A Message System Supporting Fault Tolerance. In *Proc. 9th ACM Symposium on Operating Systems Principles*, Bretton Woods (USA), October 1983.

[BBG+89] A. Borg, W. Blau, W. Graetsch, F. Herrmann, and W. Oberle. Fault Tolerance Under UNIX. *ACM Transactions on Computer Systems*, 7(1), February 1989.

[BCS84] D. Briatico, A. Ciuffoletti, and L.A. Simoncini. Distibuted Domino-Effect Free Recovery Algorithm. In *Proc. 4th IEEE Symposium on Reliability in Distributed Software an Database Systems*, Maryland (USA), October 1984.

[Ber88] P.A. Bernstein. Sequoia: A Fault-Tolerant Tightly Coupled Multiprocessor for Transaction Processing. *IEEE Computer*, February 1988.

[BGJ+93] M. Banatre, A. Gefflaut, P. Joubert, P. Lee, and C. Morin. An Architecture For Tolerating Processor Failures in Shared-Memory Multiprocessors. Publication Interne PI-707, Institut de Recherche en Informatique et Systèmes Aléatoires, Rennes (France), Mars 1993.

[BGM94] M. Banatre, A. Gefflaut, and C. Morin. Tolerating Node Failures in Cache Only Memory Architectures. In *Proc. of Supercomputing '94*, Washington (USA), November 1994.

[BHMR95a] R. Baldoni, J.M. Hélary, A. Mostefaoui, and M. Raynal. Consistent Checkpointing in Message Passing Distributed Systems. Publication Interne PI-925, Institut de Recherche en Informatique et Systèmes Aléatoires, Rennes (France), Mai 1995.

[BHMR95b] R. Baldoni, J.M. Hélary, A. Mostefaoui, and M. Raynal. On Modeling Consistent Checkpoints and the Domino Effect in Distributed Systems. Publication Interne PI-933, Institut de Recherche en Informatique et Systèmes Aléatoires, Rennes (France), Mai 1995.

[BHMR97] R. Baldoni, J.-M. Hélary, A. Mostefaoui, and M. Raynal. A Communication-Induced Checkpointing Protocol that Ensures Rollback-Dependency Trackability. In *Proc. 27th IEEE Symposium on Fault Tolerant Computing*, June 1997.

[BHR95] J. Brzeziński, J.-M. H'elary, and M. Raynal. Semantics of recovery lines for backward recovery in distributed systems. *Annales des Télécommunications*, 50(11-12), Novembre/Décembre 1995.

[BHR97] R. Baldoni, J.-M. Hélary, and M. Raynal. Rollback-Dependency Trackability : An Optimal Characterization and its protocol. Publication Interne PI-1107, Institut de Recherche en Informatique et Systèmes Aléatoires, Rennes (France), Mai 1997.

[BL88] B. Bhargava and S.-R. Lian. Independent Checkpointing and Concurrent Rollback for Recovery in Distributed Systems - An Optimistic Approach. In *Proc. 7th IEEE Symposium on Reliable Distributed Systems*, 1988.

[BS83] G. Barigazzi and L. Strigini. Application-Transparent Setting of Recovery Points. In *Proc. 13th IEEE Symposium on Fault Tolerant Computing*, Milano (Italy), June 1983.

[CJ91] F. Cristian and F. Jahanian. A Timestamp-Based Checkpointing Protocol for Long-Lived Distributed Computations. In *Proc. 10th IEEE Symposium on Reliable Distributed Systems*, 1991.

[CL85] K.M. Chandy and L. Lamport. Distributed Snapshots: Determining Global States of Distributed Systems. *ACM Transactions on Computer Systems*, 3(1), February 1985.

[Con96] D. Conan. *Tolérance aux fautes par recouvrement arrière dans les systèmes informatiques répartis*. PhD thesis, Université Paris VI (France), Septembre 1996.

[Con97] D. Conan. Message delivery semantics for the rollback-recovery of undeterministic processes. In *2nd European Research Seminar in Distributed Systems*, Zinal (Switzerland), March 1997.

[Cri89] F. Cristian. Exception Handling and Tolerance of Software Faults. In T. Anderson, editor, *Dependability of resilient Computers*. BSP Professional Books, Blackwell Scientific Publications, (UK), 1989.

[DS87] G.N. Dixon and S.K. Shrivastava. Exploiting Type Inheritance Facilities to Implement Recoverability in Object Based Systems. In *Proc. 6th Symposium on Reliability in Distributed Software and Database Systems*, Williamsburg (USA), March 1987.

[DVCL93] G. Deconinck, J. Vounckx, R. Cuyers, and R. Lauwereins. Survey of Checkpointing and Rollback Techniques. Technical Reports of Esprit Project 6731 (FTMPS) 03.1.8 and 03.1.12, ESAT-ACCA Laboratory, Katholieke Universiteit Leuven (Belgique), June 1993.

[EJW96] E.N. Elnozahy, D.B. Johnson, and Y.M. Wang. A Survey of Rollback-Recovery Protocols in Message Passing Systems. Technical Report CMU-CS-96-181, Carnegie-Mellon University, 1996.

[Eln93] E.N. Elnozahy. *Manetho: Fault Tolerance in Distributed Systems Using Rollback-Recovery and Process Replication*. PhD thesis, Rice University (USA), October 1993.

[EZ92] E.N. Elnozahy and W. Zwaenepoel. Manetho : Transparent Rollback-Recovery with Low Overhead, Limited Rollback and Fast Output Commit. *IEEE Transactions on Computers*, 41(5), May 1992.

[EZ94] E.N. Elnozahy and W. Zwaenepoel. On the Use and Implementation of Message Logging. In *Proc. 24th IEEE Symposium on Fault Tolerant Computing*, Austin (USA), June 1994.

[FR94] E. Fromentin and M. Raynal. Local states in distributed computations: a few relations and formulas. *ACM Operating Systems Review*, 28(2), April 1994.

[Gra90] J. Gray. A Census of Tandem System Availability Between 1985 and 1990. *IEEE Transactions on Reliability*, 39(4), October 1990.

[HMNR97] J.-M. Hélary, A. Mostefaoui, R.H.B. Netzer, and M. Raynal. Communication-Based Prevention of Useless Checkpoints in Distribued Computations. In *Proc. 16th IEEE Symposium on Reliable Distributed Systems*, Durham (USA), October 1997.

[HMR93] J.-M. Helary, A. Mostefaoui, and M. Raynal. Déterminer un état global dans un système réparti. Publication Interne PI-769, Institut de Recherche en Informatique et Systèmes Aléatoires, Rennes (France), Octobre 1993.

[HMR97] J.-M. Hélary, A. Mostefaoui, and M. Raynal. Points de contrôle cohérents dans les systèmes répartis : concepts et protocoles. Publication Interne PI-1108, Institut de Recherche en Informatique et Systèmes Aléatoires, Rennes (France), Juin 1997.

[HNR97] J.-M. Hélary, R. Netzer, and M. Raynal. Consistency Issues in Distributed Checkpoints. Publication Interne PI-1106, Institut de Recherche en Informatique et Systèmes Aléatoires, Rennes (France), Mai 1997.

[HW95] Y. Huang and Y.-M. Wang. Why Optimistic Message Logging Has Not Been Used In Telecommunications Systems. In *Proc. 25th IEEE Symposium on Fault Tolerant Computing*, June 1995.

[Hwa93] K. Hwang. Parallel Models, Languages, and Compilers. In *Advanced Computer Architecture*, chapter 10. McGraw-Hill, 1993.

[JF93] B. Janssens and W.K. Fuchs. Relaxing Consistency in Recoverable Distributed Shared Memory. In *Proc. 23th IEEE Symposium on Fault Tolerant Computing*, Toulouse (France), June 1993.

[JF94] B. Janssens and W.K. Fuchs. The Performance of Cache-Based Error Recovery in Multiprocessors. *IEEE Transactions on Parallel and Distributed Systems*, 5(10), October 1994.

[Joh89] D.B. Johnson. *Distributed System Fault Tolerance Using Message Logging and Checkpointing*. PhD thesis, Rice University, December 1989.

[Joh93] D.B. Johnson. Efficient Transparent Optimistic Rollback Recovery for Distributed Applications Programs. In *Proc. 12th IEEE Symposium on Reliable Distributed Systems*, October 1993.

[JV91] T.T.-Y. Juang and S. Venkatesan. Crash Recovery With Little Overhead (Preliminary Version). In *Proc. 11th IEEE International Conference on Distributed Computing Systems*, 1991.

[JZ87] D.B. Johnson and W. Zwaenepoel. Sender-based Message Logging. In *Proc. 17th IEEE Symposium on Fault Tolerant Computing*, June 1987.

[JZ90a] D.B. Johnson and W. Zwaenepoel. Output-Driven Distributed Optimistic Message Logging and Checkpointing. Technical Report 90-118, Department of Computer Science, Rice University at Houston, Texas (USA), May 1990.

[JZ90b] D.B. Johnson and W. Zwaenepoel. Recovery in Distributed Systems Using Optimistic Message Logging and Checkpointing. *Journal of Algorithms*, 11, September 1990.

[JZ91] D.B. Johnson and W. Zwaenepoel. Transparent Optimistic Rollback Recovery. *ACM Operating Systems Review*, 25(2), April 1991.

[KMBT92] M.F. Kaashoek, R. Michiels, H.E. Bal, and A.S. Tanenbaum. Transparent Fault-Tolerance in Parallel ORCA Programs. In *Proc. 3rd USENIX Symposium on Experiences with Distributed and Multiprocessor Systems*, Newport Beach (USA), March 1992.

[KT87] R. Koo and S. Toueg. Checkpointing and Rollback Recovery for Distributed Systems. *IEEE Transactions on Software Engineering*, SE-13(1), January 1987.

[LA90] L. Lin and M. Ahamad. Checkpointing and Rollback-Recovery in Distributed Object Based Systems. In *Proc. 20th IEEE Symposium on Fault Tolerant Computing*, June 1990.

[LA94] H.V. Leong and D. Agrawal. Using Message Semantics to Reduce Rollback in Optimistic Message Logging Recovery Schemes. In *Proc. 14th International Conference on Distributed Computing Systems*, Poznan (Poland), May 1994.

[Lam78] L. Lamport. Time, clocks and the ordering of events in a distributed system. *Communications of the ACM*, 21(7), July 1978.

[LB88] P.-J. Leu and B. Bhargava. Concurrent Robust Checkpointing and Recovery in Distributed Systems. In *Proc. 4th IEEE International Conference on Data Engineering*, 1988.

[LFS93] J. Leon, A.L. Ficher, and P. Steenkiste. Fail-safe PVM: A portable package for distributed programming with transparent recovery. Technical Report CMU-CS-93-124, Carnegie Mellon University, February 1993.

[LRG91] A. Lowry, J.R. Russell, and A.P. Goldberg. Optimistic Failure Recovery for Very Large Networks. In *Proc. 10th IEEE Symposium on Reliable Distributed Systems*, 1991.

[LRV87] H.F. Li, T. Radhakrishnan, and k. Venkatesh. Global State Detection in Non-FIFO Networks. In *Proc. 7th IEEE International Conference on Distributed Computing Systems*, 1987.

[LY87] T.H. Lai and T.H. Yang. On Distributed Snapshots. *Information Processing Letters*, 25, May 1987.

[Mat89] F. Mattern. Virtual Time and Global States of Distributed Systems. In Cosnard et al., editor, *Proc. International Workshop on Parallel and Distributed Algorithms*, 1989.

[MNS97] D. Manivannan, R.H.B. Netzer, and M. Singhal. Finding Consistent Global Checkpoints in a Distributed Computation. *IEEE Transactions on Parallel and Distributed Systems*, 8(6), June 1997.

[MR96] A. Mostefaoui and M. Raynal. Efficient Message Logging for Uncoordinated Checkpointing Protocols. In *Proc. 2nd European Dependable Computing Conference*, 1996.

[MS89] L.V. Mancini and S.K. Shrivastava. Replication within Atomic Actions and Conversations: A Case Study in Fault-Tolerance Duality. In *Proc. 19th IEEE Symposium on Fault Tolerant Computing*, June 1989.

[MS96] D. Manivannan and M. Singhal. A Low Overhead Recovery Technique Using Quasi-Synchronous Checkpointing. In *Proc. 16th IEEE International Conference on Distributed Computing Systems*, Hong-Kong, May 1996.

[NF97] N. Neves and W.K. Fuchs. Adaptative Recovery for Mobile Environments. *Communications of the ACM*, 40(1), January 1997.

[NM94] R.H.B. Netzer and B.P. Miller. Optimal Tracing and Replay for Debugging Message-Passing Parallel Programs. *The Journal of Supercomputing*, 1994.

[PBS+88] D. Powell, G. Bonn, D. Seaton, P. Veríssimo, and F. Waeselynck. The Delta-4 Approach to Dependability in Open Distributed Computing Systems. In *Proc. 18th IEEE Symposium on Fault Tolerant Computing*, Los Alamitos (USA), June 1988.

[Pla93] J.S. Plank. *Efficient Checkpointing on MIMD Architectures*. PhD thesis, Princeton University, June 1993.

[PP83] M.L. Powell and D.L. Presotto. Publishing, A reliable broadcast communication mechanism. In *Proc. 9th ACM Symposium on Operating Systems Principles*, Bretton Woods (USA), October 1983.

[QBC97] F. Quaglia, R. Baldoni, and B. Ciciani. A Checkpointing-Recovery Scheme for Domino-Free Distributed Systems. In *Proc. IPPS Wokshop on Fault-Tolerant Parallel and Distributed Systems*, Geneva (Switzerland), April 1997.

[Rus80] D.L. Russell. State Restoration in Systems of Communicating Processes. *IEEE Transactions on Software Engineering*, SE-6(2), March 1980.

[SBY88] R.E. Strom, D.F. Bacon, and S.A. Yemini. Volatile Logging in N-Fault-Tolerant Distributed Systems. In *Proc. 18th IEEE Symposium on Fault Tolerant Computing*, Los Alamitos (USA), June 1988.

[SDP85] S.K. Shrivastava, G.N. Dixon, and G.D. Parrington. Objects and Actions in Reliable Distributed Systems. In T. Anderson, editor, *Robust Distributed Programs*, chapter 6. Collins, 1985. Revised version.

[Sen95] P. Sens. The Performance of Independent Checkpointing in Distributed Systems. In *3rd ACM Hawaii International Conference on System Sciences*, Hawaii (USA), 1995.

[SK86] M. Spezialetti and P. Kearns. Efficient Distributed Snapshots. In *Proc. 6th IEEE International Conference on Distributed Computing Systems*, 1986.

[SMK+94] M. Satyanarayanan, H.H. Mashburn, P. Kumar, D.C. Steere, and J.J. Kistler. Lightweight Recoverable Virtual Memory. *ACM Transactions on Computer Systems*, 12(1), February 1994.

[SS83] R.D. Schlichting and F.B. Schneider. Fail-Stop Processors: An Approach to Designing Fault-Tolerant Computing Systems. *ACM Transactions on Computer Systems*, 1(3), August 1983.

[SS92] L.M. Silva and J.G. Silva. Global Checkpointing for Distributed Programs. In *Proc. 11th IEEE Symposium on Reliable Distributed Systems*, 1992.

[SW89] A.P. Sistla and J.L. Welch. Efficient distributed recovery using message logging. In *Proc. 8th ACM Symposium on Principles of Distributed Computing*, 1989.

[SY85] R.E. Strom and S.A. Yemini. Optimistic Recovery in Distributed Systems. *ACM Transactions on Computer Systems*, 3(3), August 1985.

[Tan95] A.S. Tanenbaum. *Distributed Operating Systems*. Prentice Hall, New Jersey, 1995.

[TKT89] Z. Tong, R.Y. Kain, and W.T. Tsai. A Low Overhead Checkpointing and Rollback Recovery Scheme for Distributed Recovery. In *Proc. 8th IEEE Symposium on Reliable Distributed Systems*, 1989.

[Vai94] N. Vaidya. Consistent logical checkpointing. Technical Report 94-051, Texas A&M University, Texas (USA), July 1994.

[Ven89] S. Venkatesan. Message-Optimal Incremental Snapshots. In *Proc. 9th IEEE International Conference on Distributed Computing Systems*, 1989.

[VRL87] K. Venkatesh, T. Radhakrishnan, and H.F. Li. Optimal Checkpointing and local Recording for Domino-Free Rollback Recovery. *Information Processing Letters*, 25(5), July 1987.

[Wan97] Y.M. Wang. Consistent Global Checkpoints That Contain a Given Set of Local Checkpoints. *IEEE Transactions on Computers*, 46(4), April 1997.

[WF90] K.-L. Wu and W.K. Fuchs. Recoverable Distributed Shared Virtual Memory. *IEEE Transactions on Computers*, 39(4), April 1990.

[WF92] Y.-M. Wang and W.K. Fuchs. Optimistic Message Logging for Independent Checkpointing in Message-Passing Systems. In *Proc. 11th IEEE Symposium on Reliable Distributed Systems*, 1992.

[WFP90] K.-L. Wu, W.K. Fuchs, and J.H. Patel. Error Recovery in Shared Memory Multiprocessors Using Private Caches. *IEEE Transactions on Parallel and Distributed Systems*, 1(2), April 1990.

[WHV+95] Y.-M. Wang, Y. Huang, K.-P. Vo, P.-Y. Chung, and C. Kintala. Checkpointing and Its Applications. In *Proc. 25th IEEE Symposium on Fault Tolerant Computing*, June 1995.

[XN93] J. Xu and R.H.B. Netzer. Adaptive Independent Checkpointing for Reducing Rollback Propagation. In *Proc. 5th IEEE Symposium on Parallel and Distributed Processing*, Dallas (USA), December 1993.

[XNM95] J. Xu, R.H.B. Netzer, and M. Mackey. Sender-Based Message Logging for Reducing Rollback Propagation. In *Proc. 7th IEEE Symposium on Parallel and Distributed Processing*, San Antonio (USA), December 1995.