THÈSE de DOCTORAT de l'UNIVERSITÉ PARIS 6

Spécialité :

INFORMATIQUE

présentée

par Monsieur Denis CONAN

pour obtenir le grade de DOCTEUR de l'UNIVERSITÉ PARIS 6

Sujet de la thèse :

TOLÉRANCE AUX FAUTES PAR RECOUVREMENT ARRIÈRE DANS LES SYSTÈMES INFORMATIQUES RÉPARTIS

soutenue le 09 Septembre 1996

devant le jury composé de :

Monsieur	Claude	GIRAULT	Président
Monsieur	Madjid	BOUABDALLAH	Rapporteur
Monsieur	Jean-Marie	RIFFLET	Rapporteur
Monsieur	Guy	BERNARD	Examinateur
Monsieur	Yves	CROUZET	Examinateur
Monsieur	Gilles	MULLER	Examinateur

Remerciements

Je tiens à remercier très vivement :

Monsieur Claude Girault, Professeur à l'Université Pierre et Marie Curie, pour avoir accepté de présider ce jury.

Monsieur Guy BERNARD, Professeur à l'Institut National des Télécommunications, pour m'avoir accueilli dans son équipe et avoir dirigé cette thèse.

Monsieur Jean-Marie RIFFLET, Professeur à l'Université Denis Diderot, et Monsieur Madjid BOUABDALLAH, Professeur à L'Université Technologique de Compiègne, pour avoir accepté d'être rapporteurs de mes travaux et pour leurs remarques constructives qui ont permis d'améliorer et de compléter ce mémoire de thèse.

Monsieur Yves CROUZET, Chargé de Recherche CNRS au LAAS, et Monsieur Gilles MULLER, Chargé de Recherche INRIA à l'IRISA, qui me font l'honneur de participer à ce jury.

L'ensemble du Département Informatique de l'INT pour l'ambiance joyeuse qui y règne et pour les conditions de travail idéales qui m'ont permis de mener à bien ce travail.

Les membres de l'équipe Systèmes Répartis pour l'intérêt qu'ils ont porté à mes recherches, pour leur gentillesse et pour l'aide qu'ils n'ont jamais refusée.

Résumé

Cette thèse traite de la reprise sur erreur par recouvrement arrière. Les applications réparties cibles tolèrent des interruptions momentanées du service, pourvu que les rétablissements soient automatiques, et, s'exécutent sur les nœuds de réseaux de stations de travail faiblement couplés. Le mode de défaillance est le mode "silence sur défaillance".

Après une étude approfondie des problèmes, des objectifs, des mécanismes existants, et, des politiques possibles (efficaces) et souhaitables (efficientes) du recouvrement arrière, nous présentons un mécanisme de recouvrement arrière original bien adapté aux réseaux étendus. Un réseau étendu est vu comme une agglomération de réseaux locaux, chacun d'eux constituant une unité de reprise répartie indépendante et contrôlée par un gestionnaire.

À la base, le mécanisme de recouvrement arrière supporte les applications *presque*-déterministes et est transparent pour l'utilisateur. Il comprend les mécanismes de constitution de points de reprise, de journalisation des messages et de recouvrement arrière composés de leurs optimisations les plus efficientes. En outre, il supporte de façon optionnelle les applications composées de processus *presque*-déterministes et de processus indéterministes, ceci en contre-partie de l'intervention de l'utilisateur.

Deux prototypes ont été réalisés. Le premier, au dessus du noyau `SunOS`, est entièrement portable. Il a permis l'évaluation des limites de la portabilité : la non-transparence (complète) du recouvrement arrière, l'impossibilité d'optimiser les constitutions de points de reprise. Le deuxième prototype est réalisé dans `Chorus/MiX`. Il a permis l'évaluation des apports de la technologie micro-noyau : l'accès à toutes les ressources d'un processus par la connaissance du nom de l'acteur correspondant, la manipulation aisée de la mémoire virtuelle, la transparence du nommage et de la localisation, et, le parallélisme. Enfin, pour les deux types de système d'exploitation, nous soulignons la difficulté actuelle de construire des coupures cohérentes.

Mots-clefs : Tolérance aux fautes, recouvrement arrière, systèmes répartis, point de reprise, journalisation de messages, systèmes d'exploitation monolithiques, systèmes d'exploitation micro-noyau.

Abstract

This dissertation deals with rollback recovery of distributed applications running on wide area networks. Workstations are loosely connected and fail silently. Applications allow interruptions of the service, provided that recoveries are automatic.

The dissertation starts with an in-depth study of the problems, the objectives, the existing mechanisms and the possible (effective) and desirable (efficient) politics of rollback recovery. Then, we present a new mechanism well-suited to wide area networks. A wide area network is defined as a set of interconnected local area networks. Each one forms a distributed recovery unit managed by a special entity.

The basic mechanism allows *nearly*-deterministic applications and is user-transparent. It comprises the checkpointing, the message logging and the rollback mechanisms with their most efficient optimizations. Moreover, it may be adapted to support distributed applications composed of *nearly*-deterministic and undeterministic processes, in return to the programming of the fault tolerance by the user.

Two prototypes have been developed. The first one, on top of the `SunOS` kernel, is completely portable. It permitted to evaluate the limits of portability : nearly but not full transparency of the rollback recovery, the impossibility of optimizing checkpointing. The second prototype was implemented into `Chorus/MiX`. It permitted to evaluate micro-kernel technology contributions to the building of fault-tolerant systems : full access to the process resources given the name of the corresponding actor, easy virtual memory management, transparency of location and naming, and, parallelism. Finally, we emphasize the difficulty in building consistent cuts with the two kinds of operating system.

Keywords : Fault tolerance, rollback recovery, distributed systems, checkpoint, message logging, monolithic operating systems, micro-kernel operating systems.

Table des matières

Table des figures

Liste des tableaux

Introduction

Le rapport ARAGO 15 [OFTA94] préparé par l'Observatoire Français des Techniques Avancés indique que "le coût annuel des défaillances informatiques en France excède 10 milliards de francs". La cause de ces défaillances est attribuée à "un manque de maîtrise de la complexité, et c'est certainement là que la tolérance aux fautes est appelée à se révéler la plus précieuse". En guise de recommandation, les auteurs estiment notamment que "les recherches [...] devraient s'accompagner de, ou s'intégrer dans des travaux visant à la structuration des systèmes" car "les modèles actuels de développement des systèmes sont loin d'intégrer [...] la tolérance aux fautes".

Cette thèse est une contribution à l'intégration de la tolérance aux fautes dans les systèmes répartis.

Un système réparti est un ensemble de nœuds inter-connectés par un réseau de communication et communiquant par échange de messages. Le choix d'un tel système est justifié par le fait qu'il permet notamment :

- la réalisation de services non supportés par les plus grands ordinateurs,
- une sûreté de fonctionnement accrue par l'isolation physique des différentes parties du système.

Par ailleurs, distribution et tolérance aux fautes (et par extension, sûreté de fonctionnement) se motivent mutuellement. Dans un système réparti, une application s'exécutant sur un nœud peut se diffuser, et c'est souhaitable, sur plusieurs autres nœuds. Naturellement, la défaillance d'un de ses composants fait défaillir l'ensemble de l'application répartie. La répartition apparaît alors comme une entrave à la sûreté de fonctionnement.

Les composants d'un système informatique sont naturellement divisés en deux catégories : les composants matériels et les composants logiciels. La tolérance aux fautes s'intéresse bien entendu à chacun d'eux. La tolérance aux fautes au niveau matériel s'intéresse aux composants matériels afin de les rendre intrinsèquement plus fiables et extérieurement mieux contrôlables. La tolérance aux fautes au niveau logiciel (celle qui nous intéresse) met en œuvre des techniques de génie logiciel dans le but de tolérer les fautes logicielles et matérielles. Citons à titre d'exemple la détection concomitante d'erreurs, la structuration des applications en blocs de reprise et la conception diversifiée. Pour appliquer ces techniques de génie logiciel, les concepteurs d'application disposent de plusieurs paradigmes de programmation. Les principaux modèles de programmation de la tolérance aux fautes sont le modèle conversationnel, le modèle transactionnel, le modèle des machines à états, et, le modèle du primaire et des secondaires. Pour implanter ces paradigmes de programmation, le système d'exploitation fournit un certain nombre d'abstractions. Il s'agit des actions atomiques, des appels de procédure à distance, des processus fiables, de la gestion de groupe, de la diffusion, de l'horloge globale commune, et, de la mémoire stable.

Par conséquent, il est possible de tolérer les fautes matérielles au niveau matériel. À l'autre extrémité, il faut utiliser des techniques de génie logiciel pour tolérer les fautes logicielles. Entre les deux, les fautes matérielles peuvent être tolérées en utilisant les abstractions que les systèmes d'exploitation peuvent supporter.

Notre travail s'inscrit dans ce dernier axe. Plus précisément, nous étudions le concept des processus fiables dans les systèmes répartis. La méthode de tolérance aux fautes est la reprise sur erreur par recouvrement arrière automatique. Comme son nom l'indique, le recouvrement arrière automatique vise à la transparence de la tolérance aux fautes pour l'application. Nous pensons montrer dans ce travail qu'avec les techniques actuelles, il est faisable et relativement peu coûteux de réaliser un mécanisme de recouvrement arrière automatique, et ceci de façon (presque) transparente pour les applications réparties.

Le plan de l'ouvrage est organisé comme suit :

- Le chapitre 1 situe la reprise sur erreur par recouvrement arrière automatique dans la sûreté de fonctionnement, puis dans la tolérance aux fautes. Les applications cibles sont les applications commerciales à usage général - tolérant des interruptions momentanées du service, pourvu que les rétablissements soient automatiques. Le mode de défaillance est le mode "silence sur défaillance".

- Le chapitre 2 est un état de l'art du domaine. Le recouvrement arrière automatique met en œuvre trois mécanismes. La constitution de points de reprise sauvegarde en mémoire stable des états de l'application. La journalisation sauvegarde en mémoire stable l'exécution répartie. Le mécanisme de recouvrement arrière (proprement dit) organise le retour en arrière de l'application lorsque surviennent des défaillances. Le modèle du système réparti et des programmes répartis est décrit formellement. Les problèmes du recouvrement arrière (le déterminisme d'exécution et la cohérence d'état global) sont analysés tant du point de vue théorique que de celui des difficultés de réalisation qu'ils entraînent. Les objectifs du recouvrement arrière (le degré de tolérance aux fautes, les surcoûts, l'inhibition, et, la quantité de travail à défaire ou à refaire) permettent de définir les politiques possibles (efficaces[1]) et souhaitables (efficientes[2]). Le chapitre se termine avec l'exposé des solutions algorithmiques et des implantations existantes pour chacun des mécanismes de base.

- Le chapitre 3 présente le travail algorithmique. Nous choisissons de construire un mécanisme global efficace et efficient supportant les applications *presque*-déterministes et transparent pour l'utilisateur. Ce mécanisme global simple comprend les mécanismes de base composés de leurs optimisations les plus efficientes dans le modèle de système réparti. En outre, il est assez souple pour permettre l'ajout d'extensions. Ces extensions autorisent le support d'applications composées de processus *presque*-déterministes et de processus indéterministes, ceci en contre-partie de la programmation par l'utilisateur de la tolérance aux fautes. La première optimisation est la programmation de la sémantique des transmissions de messages. L'idée est de laisser l'utilisateur choisir les messages devant être rejoués lors d'une éventuelle ré-exécution. La deuxième optimisation est la prise en compte de l'indéterminisme. L'utilisateur regroupe les actions indéterministes dans certains processus qu'il déclare indéterministes et isole ceux-ci en programmant la sémantique des transmissions des

1. qui font bien le travail demandé.
2. qui le font de manière performante.

messages échangés avec eux. Le résultat est une application composée d'un ensemble de processus *presque*-déterministes et d'un ensemble de processus indéterministes.

– Le chapitre 4 présente deux implantations. Le premier prototype est réalisé au dessus du noyau `SunOS`. Le résultat est un logiciel portable. En revanche, la deuxième implantation est réalisée au dessus du micro-noyau `Chorus`, dans `Chorus/MiX`. L'évaluation des implantations est effectuée au niveau conceptuel sous deux angles : les limites de la portabilité, les apports et les limites de la "technologie micro-noyau" pour le recouvrement arrière.

Chapitre 1

La situation de la reprise sur erreur par recouvrement arrière dans la sûreté de fonctionnement

Ce chapitre situe la reprise sur erreur par recouvrement arrière dans sa discipline englobante : la sûreté de fonctionnement.

"La sûreté de fonctionnement d'un système informatique est la qualité du service qu'il délivre, qualité telle que les utilisateurs puissent lui accorder une confiance justifiée" [Lapr85]. Le but de cette discipline est de *"concevoir, réaliser et utiliser des systèmes informatiques où la faute est naturelle, prévue et tolérable"* [Surf76].

La démarche de situation de la reprise sur erreur par recouvrement arrière dans la sûreté de fonctionnement est effectuée en deux temps. La section 1.1 introduit la tolérance aux fautes comme un des moyens de la sûreté de fonctionnement. La section 1.2 définit les processus fiables comme une des abstractions du système d'exploitation pour la tolérance aux fautes. Enfin, la section 1.3 termine le chapitre en résumant les hypothèses générales de l'étude.

1.1 De la sûreté de fonctionnement à la tolérance aux fautes

Cette section introduit les notions de base de la sûreté de fonctionnement. Tout d'abord, la sous-section 1.1.1 définit les concepts de faute, erreur et défaillance[1]. Ensuite, la sous-section 1.1.2 développe les attributs de la sûreté de fonctionnement. Enfin, la sous-section 1.1.3 précise la situation de la tolérance aux fautes par rapport à l'évitement, à l'élimination et à la prévision des fautes.

Toutes ces notions sont représentées dans la figure 1.1 par ce que Laprie nomme "l'arbre de la sûreté de fonctionnement" [Lapr94]. Au cours de cette présentation, nous précisons les fautes, les erreurs, et, les modes de défaillance que nous désirons respectivement tolérer, détecter et "couvrir". En résumé, nous supposons que les applications défaillent à cause des processeurs des stations de travail et que celles-ci suivent le mode "silence sur défaillance". Ces hypothèses sont directement impliquées par le fait que les applications supportées sont des applications commerciales à usage général[2]. En effet, les applications commerciales à usage général sont exécutées sur des réseaux de stations de travail faiblement couplées. Elles nécessitent une fiabilité modérée. En ce qui concerne la disponibilité, la sécurité et l'intégrité, nous considérons que ce sont des problèmes d'actualité complexes, indépendants et devant faire l'objet d'études spécifiques.

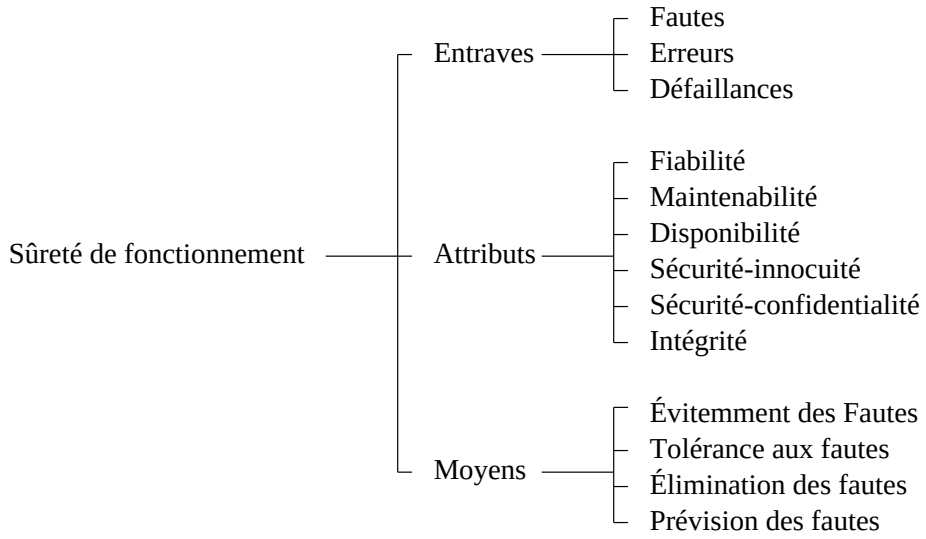


FIG. 1.1 – L'arbre de la sûreté de fonctionnement.

1. "*Le terme "panne", bien que d'un usage courant en français, est d'une utilisation délicate car il peut désigner tantôt une défaillance (comme dans "tomber en panne"), tantôt une faute (comme dans "trouver la panne") ; son utilisation devient plus que délicate dès que l'on s'écarte des fautes physiques accidentelles : peut-on raisonnablement parler de "panne de conception", voire de "panne intentionnelle" ? L'utilisation de "faute" dans ses diverses acceptations permet de fournir un terme générique qui s'accorde à la généricité de la notion de sûreté de fonctionnement*" [OFTA94].

2. Comme nous le verrons en sous-section 1.1.2, ce sont des applications tolérant des interruptions momentanées du service, pourvu que les rétablissements soient automatiques.

1.1.1 Les entraves à la sûreté de fonctionnement

Une *faute* est caractérisée par sa nature, sa durée, son origine [Lapr94] et son étendue [Nels90].

- La nature d'une faute donne le caractère intentionnel ou accidentel d'une faute. Les fautes intentionnelles sont créées délibérément dans le dessein de nuire tandis que les fautes accidentelles apparaissent de manière fortuite. L'effacement prémédité d'un fichier et la destruction d'un disque dur suite à une coupure d'électricité sont des exemples de fautes respectivement intentionnelle et accidentelle.

- La durée d'une faute exprime la dépendance vis-à-vis de conditions ponctuelles. Une faute temporaire ou molle est présente pendant une durée limitée et disparaît si telle ou telle condition interne ou externe du système apparaît. Dans le cas contraire où aucune condition interne ou externe ne peut l'éliminer, une faute est dite permanente ou dure [3]. La perte d'un message pendant une surcharge passagère d'un réseau de communication et la destruction définitive d'un processeur sont des exemples de fautes respectivement temporaire et permanente.

- L'origine d'une faute est la source d'une faute. Une faute est due à des phénomènes soit physiques soit humains. Elle est provoquée soit par une partie de l'état du système soit par l'environnement. Elle appartient soit à la phase de conception soit à la phase d'exploitation du système. La coupure d'un câble d'un réseau de communication et la mauvaise définition des données d'un programme sont des exemples de fautes respectivement physique et humaine.

- L'étendue d'une faute précise la portion du système affectée. Une faute locale affecte une partie du système alors qu'une faute globale en affecte plusieurs. Dans le cadre d'un réseau local, la destruction d'un processeur d'une station de travail multi-processeur et la coupure de l'électricité dans un bâtiment sont des exemples de fautes respectivement locale et globale.

Dans notre étude, nous écartons les fautes intentionnelles, ceci afin de laisser de côté l'aspect sécurité des systèmes informatiques. De même, un composant est enlevé du système dès la détection d'une faute. Les fautes sont donc permanentes. En outre, nous nous intéressons uniquement aux fautes d'exploitation : ce point est argumenté dans la sous-section 1.2.1. Enfin, les fautes peuvent être globales.

Une *erreur* est une partie fausse de l'état du système. Tant que le service n'utilise pas la partie fausse, l'erreur est dite latente. Activée, l'erreur est dite effective et une défaillance apparaît si le service rendu n'est plus correct. Une erreur est donc susceptible de provoquer une défaillance si elle n'est pas détectée.

Powell propose une classification des erreurs selon leur type - valeur ou temps - et pour un seul utilisateur du système [Powe92]. Premièrement, le service ne correspond pas en valeur à celui spécifié. Si la valeur rendue n'est pas interprétable alors l'erreur est dite non-codée, sinon elle est dite arbitraire. Il est plus facile de rejeter une valeur non-codée qu'une valeur arbitraire. Deuxièmement, le service n'est pas délivré dans l'intervalle de temps spécifié. L'auteur distingue six catégories. Le service rendu peut être toujours en avance ou toujours en retard ou encore arbitrairement en avance ou en retard. Lorsque le système omet ponctuellement de rendre le service, c'est une omission. Lorsque l'omission

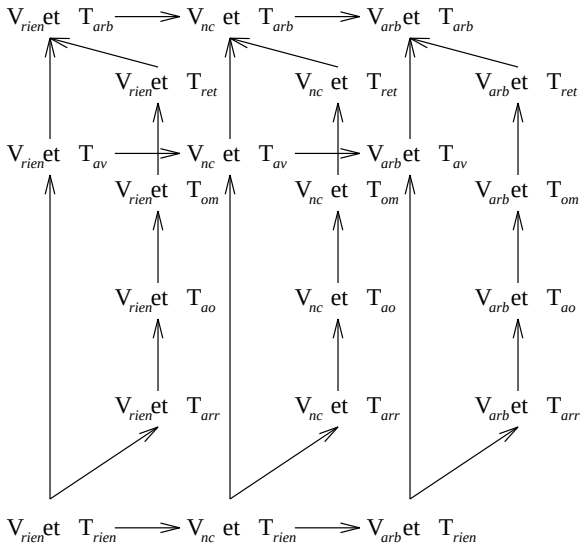
3. Gray nomme ces fautes respectivement les *"Heisenbugs" et "Bohrbugs"* [Gray86].

est définitive, c'est un arrêt franc (en anglais, crash). Si le système est sujet à des omissions, suite à k omissions successives, il est déclaré en arrêt sur omissions. En résumé, les types d'erreur du domaine des valeurs sont au nombre de deux : non-codée et arbitraire, et ceux du domaine du temps au nombre de six : arrêt franc, arrêt sur omissions, omissions, toujours en retard, toujours en avance, arbitraire.

Une *défaillance* dénote l'incapacité d'un élément du système à assurer le service spécifié par l'utilisateur. Une défaillance est caractérisée par son domaine, sa perception par les utilisateurs et ses conséquences sur l'environnement [Lapr94].

- Le domaine de défaillance est le domaine des erreurs activées.
- Une défaillance est cohérente si tous les utilisateurs en ont la même perception. Sinon, c'est une défaillance incohérente ou byzantine.
- Les conséquences sur l'environnement sont appréciées de mineures ou bénignes à majeures ou catastrophiques suivant les dommages subis.

À la suite de la classification des erreurs, Powell définit le graphe d'implication des modes de défaillance (*cf.* figure 1.2).



V_{rien}	: sans défaillance par valeur
V_{nc}	: défaillance par valeur non-codée
V_{arb}	: défaillance par valeur arbitraire
T_{rien}	: sans défaillance temporelle
T_{arr}	: arrêt franc
T_{ao}	: arrêt sur omissions
T_{om}	: omission
T_{ret}	: défaillance temporelle par retard
T_{av}	: défaillance temporelle par avance
T_{arb}	: défaillance temporelle arbitraire

FIG. 1.2 – Le graphe d'implication des modes de défaillance.

Ce graphe est construit en ordonnant par ordre croissant de sévérité les erreurs des deux catégories (valeur et temps) et en composant ces deux graphes. La racine *"V_{rien} et T_{rien}"*

désigne l'assertion la plus facile à respecter : pas de défaillance. La feuille *"V_{arb} et T_{arb}"* définit l'hypothèse la plus difficile : n'importe quelles défaillances. Par ailleurs, la "couverture" du mode de défaillance *"V_x et T_y"* est définie comme la probabilité que l'assertion *"V_x et T_y"* définissant le comportement du système est vraie, sachant que le système est défaillant. L'hypothèse la plus forte consistant à dire que le système ne défaille jamais ne peut pas être vraie ; sa couverture est donc 0. Toutes les autres hypothèses possèdent une couverture comprise entre 0 et 1. Dans la suite de l'étude, l'expression "couvrir un mode de défaillance" exprime le fait que l'on cherche à amener la couverture du mode de défaillance la plus proche possible de 1. *A contrario*, "ne pas couvrir un mode défaillance" signifie que l'on ne s'intéresse pas aux fautes provoquant les types d'erreurs amenant aux défaillances concernées.

D'une manière générale, la défaillance d'un composant du système peut affecter l'état d'autres composants du système. Pour les autres composants, cette défaillance est une faute qui amène des erreurs. Si les erreurs sont activées, de nouvelles défaillances s'ensuivent. Le cycle "faute - erreur - défaillance" est amorcé. Pris dans l'autre sens, le cycle *"faute-erreur-défaillance"* explique la complexité de l'analyse d'une défaillance.

Prenons l'exemple d'un programme réparti réalisant un calcul mathématique. Supposons que l'un des programmes séquentiels utilise une variable entière comme le dénominateur d'un calcul de fractions et que le programmeur ne teste pas si la variable est nulle. Si à un endroit donné le programmeur affecte la valeur nulle à cette variable, nous sommes en présence d'une faute de humaine de conception. À l'exécution, l'état du programme séquentiel est erroné dès que la variable est nulle. Si cette affectation est suivie d'une division, le processeur détecte l'erreur et génère une exception. À supposer qu'un mécanisme d'exception soit utilisé (*cf.* paragraphe 2.1.2.2), il provoque la défaillance du processeur. Pour le programme réparti, la défaillance du processeur correspond à la faute d'un des programmes séquentiels. Un programme séquentiel non défaillant ne peut pas récupérer les résultats du calcul effectué par celui qui est défaillant. L'état du deuxième programme séquentiel est donc erroné.

L'un des objectifs des concepteurs de systèmes sûrs de fonctionnement est la limitation de la longueur de cette chaîne. Pour commencer, il faut connaître *a priori* les chemins de propagation des fautes. Pour cela, la relation *"dépend de"* [Cris90] permet de construire le graphe de dépendance des éléments du système. Un premier composant dépend d'un deuxième composant si, et seulement si, le service rendu par le premier composant n'est correct que si le deuxième composant délivre aussi un service correct. Ensuite, il faut essayer de confiner les fautes dans un seul composant.

Le projet Delta-4 [Powe88] place un contrôleur d'attachement au réseau (NAC, Network Attachment Controller) entre chaque station de travail et le réseau. Ces contrôleurs transforment des stations de travail, qui possèdent un mode de défaillance arbitraire, en des nœuds, qui possèdent un mode de défaillance par arrêt franc sans défaillance par valeur. Ce mode de défaillance des stations de travail est appelé *"silence sur défaillance"* [4]. Nous adoptons ce mode. Ce choix est impliqué par la catégorie d'applications supportées : applications réparties à usage général (*cf.* sous-section 1.1.2).

4. Précédemment, Schlichting et Schneider avaient défini le mode *"arrêt sur défaillance"* [Schl83]. Ce mode est une extension du mode *"silence sur défaillance"* en ce qu'il suppose en plus que tous les autres composants du système ont connaissance de la faute. C'est pour cette raison que le mode *"silence sur défaillance"* est préféré.

1.1.2 Les attributs de la sûreté de fonctionnement

De nombreux attributs expriment les propriétés attendues du système et évaluent la qualité du service rendu. La liste de ces attributs s'allonge au fur et à mesure de la prise en compte des nouveaux besoins en services informatiques. À titre d'indication, Laprie définit quatre attributs dans [Lapr85] puis six dans [Lapr94].

- La *fiabilité* évalue la continuité du service. C'est la probabilité conditionnelle qu'un système n'ait pas défailli pendant l'intervalle de temps $[0, t]$, sachant qu'il était opérationnel à l'instant 0.
- La *maintenabilité* apprécie l'aptitude du système aux réparations et aux évolutions pendant tout son cycle de vie. C'est le temps d'interruption de fonctionnement due à des opérations de maintenance préventive (avant qu'une défaillance ne survienne), corrective (à la suite d'une défaillance), et perfective (pour faire évoluer le système).
- La *disponibilité* exprime la fraction du temps pendant lequel le système accepte de rendre un service. C'est la probabilité que le système soit opérationnel à l'instant t.
- La *sécurité-innocuité* évalue la continuité du service avant l'arrivée d'une défaillance catastrophique. Elle utilise donc la même expression mathématique que la fiabilité appliquée uniquement aux défaillances catastrophiques.
- La *sécurité-confidentialité* apprécie la capacité du système à résister aux fautes intentionnelles.
- L'*intégrité* estime les erreurs qui correspondent à de l'information altérée.

Siewiorek définit quatre classes d'applications selon les besoins en sûreté de fonctionnement [Siew91]. L'auteur ordonne les applications suivant leur demande croissante en fiabilité. Il estime qu'*"en général, il est plus difficile de construire un système hautement fiable qu'un système hautement disponible, parce que la fiabilité impose des exigences plus rigoureuses"*.

- Les applications *commerciales à usage général* tolèrent des interruptions momentanées du service, pourvu que les rétablissements soient automatiques. Leur demande en sûreté de fonctionnement concerne modérément la fiabilité. Les applications de conception de circuits intégrés sont des exemples d'applications commerciales à usage général.
- Les applications *hautement disponibles* tolèrent les pertes occasionnelles d'utilisateurs, mais ne tolèrent pas les défaillances catastrophiques. Leur demande en sûreté de fonctionnement concerne surtout la disponibilité. Les applications de réservations de places d'avion ou de train sont des exemples d'applications hautement disponibles.
- Les applications *à longue durée de vie* ne peuvent bénéficier d'aucune maintenance. Leur demande en sûreté de fonctionnement concerne essentiellement la maintenabilité et la fiabilité. Les applications de transmission de données des satellites de télécommunications sont des exemples d'applications à longue durée de vie.
- Les applications dites *critiques* sont les applications qui fonctionnent en temps réel et pour lesquelles une faute peut mettre en danger des vies humaines. Leur demande en sûreté de fonctionnement concerne fortement tous les attributs. Les applications de contrôle de trajectoire d'avion ou de train sont des exemples d'applications critiques.

L'auteur donne à titre d'exemple une liste de systèmes informatiques conçus spécialement pour chaque classe d'applications : UnivacI, IBM 3090 et DEC VAX pour les applications commerciales à usage général, Tandem, Stratus, Sequoia et Intel 432 pour les applications hautement disponibles, Star, Voyager et Galileo pour les applications à longue durée de vie, C.vmp et Sift pour les applications critiques.

Dans notre étude, nous nous proposons de supporter les applications commerciales à usage général. La demande en sûreté de fonctionnement est donc modérée. Comme nous le verrons au chapitre 2, le système est composé de stations de travail reliées entre elles par un réseau de communication. Ce dernier est supposé ne pas se partitionner.

Les stations de travail ne possèdent pas de matériel spécifiquement ajouté pour la sûreté de fonctionnement. Les processeurs et tout autre matériel sont supposés adopter le mode *"silence sur défaillance"*. En outre, nous ne nous intéressons pas aux fautes générées par le logiciel placé au dessus du matériel : ce point est discuté en sous-section 1.2.1. En conséquence, une station de travail défaille seulement lorsque son processeur ou tout autre matériel défaille. Par conséquent, les défaillances couvertes sont du mode *"silence sur défaillance"*.

1.1.3 Les moyens de la sûreté de fonctionnement

Avant de développer la tolérance aux fautes dans la deuxième partie de ce chapitre, nous donnons sa définition et la situons par rapport aux autres moyens de la sûreté de fonctionnement [Lapr85].

La première phase de l'existence d'un système sûr de fonctionnement est la conception et la réalisation. Afin de minimiser l'apparition de fautes de conception, les constructeurs possèdent un ensemble de propriétés à vérifier et de règles à appliquer. Ce savoir-faire est regroupé dans la discipline intitulée l'*évitement des fautes*.

Des fautes passent inévitablement à travers l'évitement des fautes. C'est pourquoi les constructeurs prévoient des mécanismes pour les tolérer. L'étude de ces mécanismes constitue la *tolérance aux fautes*. La tolérance aux fautes est mise en oeuvre par le traitement des erreurs et par le traitement des fautes. Le traitement d'erreur est destiné à éliminer les erreurs et le traitement de faute à eviter que les fautes ne soient activées à nouveau.

Avant de mettre en production le système, il doit être validé. Les constructeurs vérifient les propriétés de la spécification statiquement (par preuves ou analyses) ou dynamiquement (par simulations ou tests). Cette étude compose l'*élimination des fautes*.

Enfin, comme il restera toujours des fautes, les constructeurs estiment la présence des erreurs et leurs conséquences. Pour ce faire, des fautes sont volontairement injectées dans le système et l'on étudie la couverture de la tolérance aux fautes. C'est la *prévision des fautes*.

1.2 De la tolérance aux fautes à la reprise sur erreur

Cette section présente un panorama de la tolérance aux fautes. Tout d'abord, la sous-section 1.2.1 définit les deux niveaux de tolérance aux fautes des systèmes informatiques : matériel et logiciel. Les sous-sections qui suivent développent la tolérance logicielle aux fautes, en allant du niveau le plus élevé au niveau le plus proche du matériel. La sous-section 1.2.2 donne l'exemple de trois techniques de génie logiciel. La sous-section 1.2.3 détaille les quatre paradigmes de programmation les plus utilisés pour la tolérance aux

fautes et la sous-section 1.2.4 introduit les abstractions du système d'exploitation nécessaires pour les implanter. Enfin, la sous-section 1.2.5 introduit les étapes de la tolérance aux fautes logicielles.

Toutes ces méthodes et tous ces outils sont présentés dans la figure 1.3. Les techniques de génie logiciel sont placées dans la partie supérieure du schéma. En dessous, figurent les quatre modèles de programmation. Les flèches en pointillés précisent que les conversations et les machines à états modélisent le plus naturellement les "blocs de recouvrement" et la "conception diversifiée", respectivement. Dans la partie inférieure du schéma, figurent les abstractions du système d'exploitation. Les flèches en tirets indiquent les abstractions du système d'exploitation demandées par les modèles de programmation. Enfin, les flèches pleines tracent les dépendances des abstractions les unes par rapport aux autres.

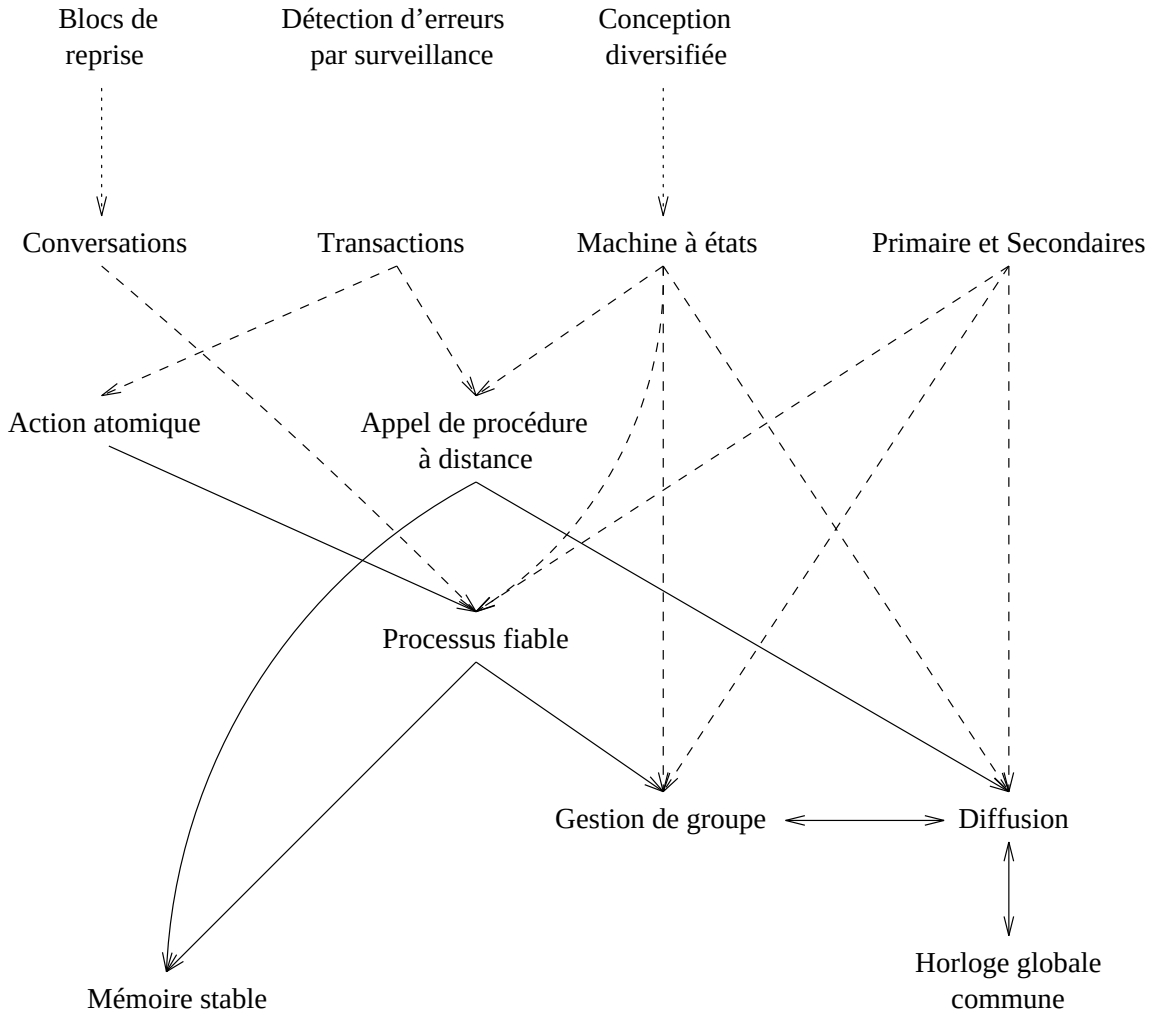


FIG. 1.3 – Le graphe des dépendances des abstractions du système d'exploitation.

Au cours de cette présentation, nous justifions le fait que nous tolérons les fautes matérielles au niveau logiciel. Il s'ensuit que nous n'utiliserons pas de techniques de génie logiciel. Puisque les applications sont des applications commerciales à usage général, l'utilisateur ne souhaite pas programmer la tolérance aux fautes de ses applications. En conséquence, la technique de tolérance aux fautes choisie est transparente à l'utilisateur ; c'est la reprise sur erreur par recouvrement arrière. Elle dérive du modèle du primaire et des secondaires et est fondée sur les processus fiables, la gestion de groupe, la diffusion, l'horloge globale commune et la mémoire stable.

1.2.1 La tolérance aux fautes aux niveaux matériel et logiciel

Les composants d'un système informatique sont naturellement divisés en deux catégories : les composants matériels et les composants logiciels. La tolérance aux fautes s'intéresse bien entendu à chacun d'eux. Les fautes sont dites matérielles (*resp.* logicielles) si elles apparaissent dans un composant matériel (*resp.* logiciel). La tolérance aux fautes est dite matérielle (*resp.* logicielle) si elle intervient au niveau matériel (*resp.* logiciel). Par abus de langage, la tolérance aux fautes au niveau matériel (*resp.* logiciel) sera appelée tolérance matérielle aux fautes (*resp.* tolérance logicielle aux fautes). La tolérance matérielle aux fautes s'applique à tolérer les fautes matérielles, la tolérance logicielle aux fautes à tolérer les fautes tant matérielles que logicielles.

La tolérance matérielle aux fautes s'intéresse à tous les composants matériels du système afin de les rendre intrinsèquement plus fiables et extérieurement mieux contrôlables. Parmi les techniques spécifiques et les matériels spécialisés, se trouvent pêle-mêle [Siew90, Cris91a]: les codes détecteurs et correcteurs d'erreurs, les comparateurs, les mémoires stables, les disques miroir, les batteries de secours, les accès aux composants par doubles bus et doubles réseaux de communication, la réplication de tous les composants, les contrôleurs de diagnostics à distance, le diagnostic automatique, les composants à déconnexion rapide, les unités fonctionnelles remplaçables sans arrêter le système, la reconfiguration automatique, etc.

Une analyse plus détaillée de ces mécanismes comprendrait notamment l'étude des systèmes VAXft 3000 ou VAX Cluster [Kron86], Tandem NonStop [Bart81] ou Integrity S2 [Jewe91], Stratus S/32 ou IBM System/88 [Webb91], Sequoia [Bern88], ainsi que l'étude de FTM [Bana91a, Bana91b] et TSM [Horn91] pour la réalisation matérielle du concept de mémoire stable [Lamp81].

La tolérance logicielle aux fautes met en œuvre des techniques de génie logiciel dans le but de tolérer les fautes logicielles et matérielles. Pour appliquer ces techniques de génie logiciel, les concepteurs d'applications disposent de plusieurs paradigmes de programmation (*cf.* sous-section 1.2.3). Pour implanter les paradigmes de programmation, le système d'exploitation fournit un certain nombre d'abstractions (*cf.* sous-section 1.2.4).

Afin de mesurer l'impact respectif de la tolérance matérielle et de la tolérance logicielle aux fautes, Gray analyse les causes des dommages répertoriés par les utilisateurs des systèmes Tandem entre 1985 et 1989 [Gray90]. Le tableau 1.1 donne le pourcentage des dommages attribués à chacune des cinq catégories de sources, pour les années 1985 et 1989, c.à-d. pour les années de début et de fin de l'enquête. Clairement, la principale source de défaillance devient le logiciel, suivie par les opérations et procédures, c.à-d. les interactions homme-machine. L'auteur explique cette évolution par les progrès tant au niveau matériel qu'au niveau logiciel. La conclusion qui s'impose est que les efforts

doivent dorénavant s'orienter vers la tolérance aux fautes logicielles, donc vers la tolérance logicielle aux fautes.

Dans le système Tandem, toutes les fautes logicielles détectées ont conduit à des erreurs provoquant une exception[5]. Le concepteur fournit un gestionnaire d'exception pour un certain nombre d'entre-elles. Si le gestionnaire masque la faute et remet le système dans un état cohérent alors l'exception est dite "récupérable". Les exceptions non récupérables provoquent l'arrêt du processeur. Le processeur pair exécutant la même tâche teste périodiquement si le premier processeur est en arrêt sur défaillance. Avec des bus de données, des périphériques, des disques durs et des alimentations doublés, le système tolère donc les défaillances d'un processeur, d'un bus, d'un périphérique, d'un disque dur et d'une alimentation. Le résultat est que le temps moyen jusqu'à défaillance est évalué à 21 ans ; un système informatique tolérant aux fautes ne défaillira pas avant son obsolescence.

Sources des dommages	**Année 1985**	**Année 1989**
Logiciel	33 %	*62 %*
Matériel	*29 %*	7 %
Maintenance	*19 %*	5 %
Opérations - Procédures	9 %	*15 %*
Environnement	5 %	6 %
Autres	5 %	5 %

TAB. 1.1 – Les sources des dommages dans le système Tandem en 1985 et en 1989.

Lee et Iyer analysent 200 rapports de défaillances de processeurs semblant être dues au logiciel du système Tandem [Lee95]. Les rapports sont classés en trois catégories : "cause identifiée", "cause non identifiée" et "problème non logiciel". Un incident est confirmé être un problème logiciel uniquement si la faute dans le logiciel est localisée, puis l'incident reproduit, et ensuite, la faute fixée. Le rapport correspondant est classé "cause identifiée". Si l'incident n'était originellement pas dû à une faute logicielle (par exemple, une faute matérielle) alors il est classé "problème non logiciel". Si les analystes croient que le problème est logiciel et que la faute n'a pas encore été localisée alors l'incident correspondant est classé "cause non identifiée". Les pourcentages pour chaque classe sont : 153 rapports de type "cause identifiée", 26 de type "cause non identifiée" et 21 de type "problème non logiciel". Les 153 rapports pour lesquels le problème logiciel est confirmé sont ventilés par catégories de fautes logicielles dans le tableau 1.2. La plupart des catégories sont évidentes. "Faute de données" correspond à l'utilisation de variables ou de constantes incorrectes. "Faute dans la définition des données" correspond à une faute dans la déclaration ou la définition de variables. "Opération manquante" inclut l'utilisation de pointeurs ou autres variables non initialisés (13 sur 27), la non mise à jour de variables (9 sur 27) et la non notification d'événements aux autres processus (5 sur 27). "Situation inattendue" comprend les problèmes de compétition et de temps (18 sur 46), l'absence de procédures pour le traitement d'un type d'erreurs (8 sur 46), le paramétrage incorrect ou le lancement incorrect d'une procédure par l'utilisateur (7 sur 46), et enfin, la non prise en compte de scénarios qui n'arrivent que rarement (13 sur 46).

Ensuite, Lee et Iyer analysent le comportement du système lorsqu'il subit des fautes confirmées ou supposées logicielles, c.à-d. lors des incidents classés "cause identifiée" ou "cause non identifiée" (174 rapports). La sévérité des défaillances logicielles est de quatre

5. *cf.* [Cris89] pour une définition précise de la gestion des exceptions.

Catégories de fautes	Nombre de rapports
Calcul incorrect	3
Faute de données	*21*
Faute dans la définition des données	7
Opération manquante	*27*
Effet de bord dû à une mise à jour du code	5
Situation inattendue	*46*
Défaut du microcode	8
Autres n'appartenant pas l'une des catégories précitées	12
Impossibles à classer à cause d'informations insuffisantes	24

TAB. 1.2 – Les catégories des fautes logicielles rencontrées dans le système Tandem.

ordres : "arrêt d'un processeur", "arrêt de plusieurs processeurs", "arrêt pendant le redémarrage" et "impossible à classer par manque d'informations". Les nombres de rapports de chaque type sont respectivement 138, 31, 1 et 4. Or, le système tolère les défaillances d'un seul processeur. D'où, la tolérance aux fautes logicielles est évaluée à 82% (138 sur 169, en excluant les 5 cas spéciaux).

Pour les 138 rapports indiquant la défaillance d'un seul processeur, les auteurs classent les raisons de la tolérance aux fautes logicielles (*cf.* tableau 1.3). Dans la première catégorie (29% des rapports), le processeur pair ré-exécute la tâche défaillante, mais ne possède pas le même état mémoire, le même ordonnancement des événements, les mêmes erreurs matérielles, etc. Dans 20% des rapports (28 sur 138), le processeur pair ne ré-exécute même pas la tâche défaillante. Cela arrive lorsque le système exécute un utilitaire important (par exemple, gestionnaire de statistiques sur l'utilisation de la mémoire), mais spécifique à un processeur. L'utilitaire n'est pas exécuté sur le processeur pair, ce dernier n'expérimente donc pas la faute causée par l'utilitaire. Selon les auteurs, cette situation n'est pas strictement de la tolérance aux fautes logicielles. La tolérance aux fautes logicielles est alors corrigée : 78% (138-28 sur 169-28). Une autre raison de la tolérance aux fautes logicielles est que les fautes sont détectées après que la tâche se soit terminée correctement (5% des rapports). Enfin, pour 22 rapports, c'est le processeur pair qui est défaillant. Cela montre que la programmation de la tolérance aux fautes implique un coût. Les auteurs ajustent donc à nouveau la tolérance aux fautes logicielles à 74% (138-28-22 sur 169-38-22).

Raisons de la tolérance aux fautes logicielles	Nombre de rapports
Processeur pair ré-exécutant la tâche défaillante mais ne ré-expérimentant pas la faute	40
Processeur pair ne ré-exécutant pas la tâche défaillante	28
Effet de la latence d'erreur	7
Processeur pair arrêté	22
Cause du problème non identifiée	26
Impossible à classer	15

TAB. 1.3 – Les raisons de la tolérance aux fautes logicielles dans les systèmes Tandem.

Nous verrons plus loin que le système Tandem met en œuvre le modèle du primaire et du secondaire. Une autre réalisation de ce modèle est la reprise sur erreur par recouvrement arrière. Les conclusions des études de Gray et de Lee et Iyer nous confortent dans le choix de la reprise sur erreur par recouvrement arrière pour la tolérance aux fautes des applications réparties à usage général. Nous verrons de même que la reprise sur erreur par recouvrement arrière prise seule ne peut tolérer que les fautes des processeurs. Par analogie avec le système Tandem, ce mécanisme de base pourra être couplé avec une technique de génie logiciel. Aussi, la sous-section 1.2.2 présente succinctement les techniques de génie logiciel. Ensuite, la sous-section 1.2.3 développe quatre paradigmes de programmation qui permettent d'appliquer ces techniques de génie logiciel. Enfin, la sous-section 1.2.4 montre quelles abstractions le système d'exploitation doit fournir pour la réalisation de ces paradigmes de programmation. En sous-section 1.2.5, nous ouvrons une parenthèse sur les étapes de la tolérance aux fautes.

1.2.2 Les techniques de génie logiciel de la tolérance logicielle aux fautes

Les techniques de génie logiciel les plus documentées sont la détection concomitante[6] d'erreurs par surveillance [Mahm88], la structuration des applications en blocs de recouvrement [Rand75] et la conception diversifiée [Aviz85].

La *détection concomitante d'erreurs par surveillance* consiste à ajouter un processus appelé contrôleur de séquences. Tout programme peut être représenté par des nœuds et des arcs, c.à-d. par un organigramme, donnant ainsi le flot de contrôle. À la compilation, on associe une signature à chaque nœud, et pendant l'exécution, le contrôleur de séquences calcule les signatures et les vérifie. De même, une capacité d'accès peut être affectée à chaque objet. À la compilation, on construit le graphe des ensembles d'objets accessibles aux différents nœuds. Pendant l'exécution, le contrôleur de séquences vérifie les accès à la mémoire. Enfin, le programmeur peut insérer dans le programme des assertions. Ces tests sont effectués par le contrôleur de séquences pendant l'exécution.

Dans la *structuration des applications en blocs de recouvrement*, un système est vu comme un empilement de niveaux d'abstraction ou machines virtuelles. Une opération du niveau inférieur est considérée comme atomique : entièrement exécutée ou non commencée. Chaque opération est définie dans un bloc de recouvrement. L'entrée et la sortie du bloc de recouvrement sont explicites. Un bloc de recouvrement est constitué de plusieurs solutions programmées et d'une assertion vérifiant l'exécution. Les solutions sont essayées successivement jusqu'à ce que l'une d'entre elles vérifie l'assertion ou jusqu'à ce qu'elles soient toutes essayées sans succès.

Dans l'approche de la *conception diversifiée*, N versions différentes de l'application sont créées à partir d'une même spécification. Ces N versions peuvent être exécutées sur N sites différents, N fois ; l'exécution est établie correcte par un voteur. La spécification initiale détermine :

1. la fonction réalisée,
2. les points de rencontre ou de synchronisation pour les votes,
3. le contenu et le format des informations de vote,

6. "*Le terme concomitance est préféré au terme concurrence car il exprime mieux la simultanéité reflétée dans le terme anglais "concurrency"*" [OFTA94].

4. l'algorithme de vote,

5. la réponse de l'application en l'absence d'informations de vote.

Ces trois techniques font l'objet d'une étude comparative dans [Huda93][7]. Une même application est implantée par plusieurs groupes d'étudiants qui chacun applique une des techniques précédemment exposées. Les applications obtenues sont testées par injection de fautes. Les résultats montrent la supériorité de la conception diversifiée sur les deux autres techniques.

Dans notre étude, puisque nous ne désirons pas tolérer les fautes logicielles (de conception), nous n'employons pas de technique de génie logiciel. En outre, les utilisateurs concevant des applications dites commerciales à usage général ne souhaitent pas utiliser de techniques de génie logiciel spécialement pour la tolérance aux fautes. La tolérance aux fautes est vue comme un service du système informatique. En revanche, par analogie avec le système Tandem, on peut très bien imaginer que le système d'exploitation intègre un mécanisme de surveillance. Ainsi, la détection d'une erreur causée par une faute logicielle génère une exception. Cette exception provoque l'arrêt du processus.

1.2.3 Les paradigmes de programmation modélisant la tolérance logicielle aux fautes

Les techniques de génie logiciel s'appliquent à la conception des applications par l'utilisation de modèles de programmation. Comme dans [Mish92], quatre de ces modèles sont exposés : les conversations [Rand75], les objets et les actions [Gray81, Lisk83], les machines à états [Schn90], et, le primaire et les secondaires [Borg83].

Le premier paradigme de programmation est le *modèle des conversations* [Rand75]. Ce paradigme de programmation dérive directement de la technique de structuration des applications en blocs de recouvrement (*cf.* sous-section 1.2.2). Une application est constituée de processus communiquant à l'aide de messages. Les instructions exécutées par les processus peuvent être regroupées en blocs de recouvrement. Au début du bloc, les processus constituent un point de reprise qui est une image de leur état. À la fin du bloc, les calculs effectués depuis le début du bloc sont vérifiés par le test d'une assertion. La conclusion de ce test permet soit de sortir du bloc, soit de revenir à l'état d'exécution du début du bloc. Si le processus communique avec d'autres processus durant l'exécution du bloc, alors il doit lier sa sortie avec ceux-ci. Inversement, dans [Rama93], les assertions sont testées périodiquement durant l'exécution de façon à ce qu'il n'y ait qu'une constitution de point de reprise entre chaque test. Les processus s'attendent à la suite des constitutions des points de reprise.

La difficulté majeure de l'approche est le repérage des processus impliqués dans une conversation. Dans [Kim82, Jalo86], le problème est résolu en nommant les blocs de recouvrement et leurs participants. Différemment, dans [Kim86, Kim90], les processus traquent les dépendances interprocessus, ne coordonnent pas les sorties de blocs et gardent en mémoire stable les points de reprise. Lorsque survient une faute, ils peuvent revenir en arrière de plusieurs blocs. Enfin, dans [Tyrr86], l'application écrite en C.S.P. ("Communicating

7. L'article fait aussi mention d'une quatrième technique de génie logiciel : la tolérance aux fautes algorithmique [Tayl90]. Cette technique semble moins utilisée et développée ; elle n'est d'ailleurs pas citée dans [Lapr90].

Sequential Processes") est automatiquement modélisée en un réseau de Pétri puis en un ensemble de conversations.

Le deuxième paradigme de programmation est le *modèle des transactions* [Gray81] aussi appelé *modèle des objets et des actions* [Lisk83]. Plusieurs opérations transformant de manière cohérente un ensemble d'objets sont regroupées dans une transaction. Les opérations d'une même transaction sont soit validées soit annulées. Une transaction possède quatre propriétés [Haer83] :

1. L'*atomicité* : une transformation est complètement terminée ou non commencée.
2. La *cohérence* : toute transformation validée doit préserver la cohérence de la base de données en respectant les spécifications.
3. L'*isolement* : les actions d'une transaction doivent être cachées des autres transactions.
4. La *permanence* : une fois qu'une transaction est validée, le système doit garantir que les résultats survivrons à toute future faute.

Afin d'améliorer la souplesse du modèle, Moss introduit les transactions imbriquées [Moss82]. Une transaction qui est longue est décomposée en sous-transactions. Seules les transactions les plus éloignées de la transaction racine peuvent modifier l'état d'un objet. D'autres structures de transactions existent, avec toujours comme objectif la souplesse : à titre d'exemple, les actions colorées [Shri90].

Dans [Lisk83, Lisk87], un gardien implante un certain nombre de procédures qui s'exécutent en réponse à des requêtes. Il encapsule des ressources accessibles seulement par ses procédures. Une action commence au niveau haut dans un gardien et se diffuse dans les autres gardiens par les procédures. De la même manière, QuickSilver se compose de serveurs interagissant par transactions [Hask88]. Arjuna se compose d'objets délivrant les abstractions nécessaires au modèle des actions et des objets [Shri91].

Avant de présenter les deux derniers paradigmes, notons la dualité des deux premiers [Shri87, Manc89, Stri91]. Une conversation comprend un ensemble de processus et une transaction transforme un ensemble d'objets. Les points de reprise des processus (*resp.* des objets) sont constitués en début de conversation (*resp.* d'action). Le tableau 1.4 montre la dualité des caractéristiques des deux modèles.

Le troisième paradigme de programmation est le *modèle des machines à états* [Schn90]. Ce modèle de programmation est directement dérivé de la technique de conception diversifiée. Une application est structurée en un ensemble de serveurs implantés par plusieurs processus. Un serveur ou machine à états gère un ensemble de ressources modifiées par des commandes. L'exécution d'une commande est atomique et déterministe : la suite des commandes en entrée du serveur suffit pour connaître la suite des commandes en sortie. La tolérance aux fautes est obtenue en répliquant les serveurs. Le nombre de répliques dépend des modes de défaillance couverts et du nombre de fautes concomitantes tolérées. Pour k fautes concomitantes, il faut $k+1$ répliques pour tolérer des défaillances par arrêts francs, et $2k+1$ répliques pour tolérer des défaillances arbitraires.

Les principaux problèmes à résoudre pour ce modèle sont :

1. l'acceptation par les serveurs répliqués de la même séquence de commandes,

Modèle des conversations	Modèle des transactions
Processus	Objet
Conversation	Transaction
Règles interdisant les communications hors de la conversation	Isolement de la transaction
Point de reprise des processus	Point de reprise des objets
Synchronisation de la sortie de la conversation	Synchronisation de la sortie de la transaction

TAB. 1.4 – La dualité des modèles des conversations et des transactions.

2. le vote des réponses délivrées,

3. l'intégration de nouvelles répliques lorsque le nombre de répliques est insuffisant.

Cette approche est documentée par de nombreux projets : entre autres, Sift [Wens78], Circus [Coop85], Isis [Birm85], AAS [Cris90], Consul [Mish91], Delta-4 [Cher92], et, Voltan [Shri92].

Guerraoui et Schiper établissent un lien entre le modèle des transactions et celui des machines à états [Guer95]. Les auteurs définissent un mécanisme générique (DTM, Dynamic-Terminating Multicast) qui est paramétré pour réaliser les différents mécanismes de base requis par les deux modèles. Le problème générique consiste en une prise de décision, parmi un groupe, à partir d'un ensemble de réponses générées à la suite d'une requête. Les paramètres du mécanisme sont la sémantique du message de requête, le groupe, la syntaxe du message de réponse et la règle d'interprétation. Le tableau 1.5 montre les valeurs des paramètres pour les deux modèles.

Paramètres	Modèle des transactions	Modèle des machines a états
Requête	Demande de vote	Message à diffuser
Groupe	Gestionnaires d'objets (GO)	Groupe de diffusion
Réponse	"Oui" ou "Non"	Horloge Logique (HL) + Numéro du récepteur (NR)
Interprétation	Validé si tous les GOs répondent "Oui"	Numéro requête = maximum des (HL+NR)

TAB. 1.5 – Les liens entre les modèles des transactions et des machines à états.

Le quatrième paradigme de programmation modélisant la tolérance aux fautes est le *modèle du primaire et des secondaires*. Ce modèle de programmation est dérivé du modèle

des machines à états. La différence principale réside dans le fait que les répliques ne sont pas actives pendant l'exécution sans occurrence de faute. Les commandes sont uniquement traitées par un serveur (le primaire), les autres (les secondaires) ne sont présents qu'en cas de fautes du primaire. Ce modèle - appelé réplication *passive* - se distingue de celui des machines à états - appelé réplication *active* - par une plus grande simplicité des mécanismes mis en œuvre. En effet, ici, l'ordonnancement des requêtes est simplifié car centralisé chez le primaire et le vote n'existe pas. Il est à noter que certaines implantations des machines à états supportent en plus le modèle du primaire et des secondaires Delta-4 [Spei89]. Parmi les réalisations existantes, citons Auragen 4000 [Borg83], Targon/32 [Borg89] et Manetho [Elno92b].

Entre la réplication passive et la réplication active, la réplication *semi-active* [Barr90, Powe91] permet de combiner l'avantage de la réplication passive (synchronisation aisée) avec l'avantage de la réplication active (faible interruption du service). Dans ce modèle, le primaire fait suivre les commandes aux secondaires. Ceux-ci les exécutent sans produire de commandes en sortie.

La réplication passive est employée aussi bien pour modéliser la tolérance aux fautes que pour améliorer les temps de réponse. Le point commun à ces deux objectifs est l'augmentation de la disponibilité. Cette démarche double se remarque surtout dans la conception des serveurs de fichiers. Par exemple, HA-NFS [Bhid90], Harp [Lisk91] et Echo [Hisg93] utilisent la réplication passive principalement pour améliorer la tolérance aux fautes, Locus [Walk83], Ficus [Pope90], Coda [Saty90b] surtout pour améliorer les temps de réponse.

Les secondaires peuvent être uniquement des images du primaire présentes en mémoire stable et automatiquement récupérées lorsqu'une faute survient. Cette méthode est communément appelée la *reprise sur erreur par recouvrement arrière*. Si les sauvegardes des images du primaire en mémoire stable s'effectuent automatiquement alors aucun paradigme de programmation n'est nécessaire. D'où, la tolérance aux fautes devient transparente à l'utilisateur. Par analogie avec le modèle des conversations, l'application peut être vue comme une conversation incluant tous les processus de l'application. Lorsqu'un processus défaille, son exécution reprend à partir d'une image présente en mémoire stable. Cette image doit vérifier l'assertion : "l'exécution de l'application est cohérente". Les interactions de l'application avec le monde extérieur doivent vérifier l'assertion : "l'interaction n'aura jamais besoin d'être rejouée".

Pour conclure, de nombreux articles prônent le développement des paradigmes de programmation pour modéliser la tolérance aux fautes, voire l'unification de tous ces paradigmes en un seul [Schl91]. Dans une autre optique, des recherches étudient les abstractions du système d'exploitation permettant la transparence de la tolérance aux fautes pour les utilisateurs [Baco91]. C'est cette dernière voie que nous décidons de suivre. L'argument principal est le même que pour le choix de ne pas utiliser de techniques de génie logiciel. En effet, les utilisateurs concevant des applications commerciales à usage général ne souhaitent pas utiliser de paradigmes de programmation spécialement pour la tolérance des fautes. La tolérance aux fautes est vue comme un service du système. Par conséquent, dans la suite de l'étude, nous nous intéresserons aux abstractions du système d'exploitation pour la reprise sur erreur par recouvrement arrière.

NB: Contrairement à la plupart des publications, nous employons l'expression *"reprise sur erreur par recouvrement arrière"* dans son intégralité, ceci pour la distinguer de la *"reprise sur erreur par poursuite"* [Vaid93a, Prad94a, Prad94b]. Dans cette dernière, les serveurs sont

répliqués et sauvegardent périodiquement et au même instant logique leur image en mémoire stable. Ces images sont comparées pour détecter des erreurs. Le terme "poursuite" signifie que les images des serveurs établis en erreur récupèrent l'image correcte. Cette méthode est donc une réalisation de la réplication active - modèle des machines à états.

1.2.4 Les abstractions du système d'exploitation pour la tolérance logicielle aux fautes

Pour réaliser les modèles de programmation, le système d'exploitation doit fournir certaines abstractions [Mish92] : l'horloge globale commune, la diffusion, la gestion de groupe de processus, la mémoire stable, le processus fiable, l'appel de procédure à distance (RPC, *"Remote Procedure Call"*) et l'action atomique. Chaque abstraction pourrait être longuement développée ; le lecteur peut se reporter aux publications [Cris91a, Mish92, LL94] pour un développement plus complet. À la suite de la présentation des abstractions, nous indiquons celles que chaque paradigme de programmation utilise. Les relations de dépendances des abstractions du système d'exploitation les unes par rapport aux autres ainsi qu'avec les paradigmes de programmation sont représentées dans la figure 1.3.

Dans un système informatique réparti, chaque unité du système possède sa propre notion du temps. Or, la détection des erreurs et le diagnostic doivent insérer l'instant d'occurrence d'une faute dans l'exécution du système. Pour cela, une *horloge globale commune* est nécessaire. Pratiquement, il existe deux approches à l'implantation d'une horloge globale commune. Dans la première approche, les horloges physiques des unités du système sont synchronisées à des intervalles réguliers. Les horloges physiques prises deux à deux ne doivent pas s'écarter plus d'une certaine distance. Par conséquent, l'horloge globale commune dépend de la diffusion. Dans la seconde approche, une horloge logique est construite. Elle ordonne les différentes actions du système en formant un treillis de ces actions.

À de nombreuses reprises, une unité du système communique avec toutes les autres unités. C'est la *diffusion*. Le message diffusé est généralement délivré soit à toutes les unités non fautives d'un groupe soit à aucune d'elles (*cf.* l'alinéa qui suit pour la notion de groupe). Le message diffusé est placé de manière cohérente dans le treillis des actions du système. Toutes les unités placent telle faute soit avant soit après la réception de tel message diffusé. Un mécanisme de diffusion fiable garantit que toutes les unités non-défaillantes reçoivent le message diffusé dans un intervalle de temps connu ou que l'émettrice est avertie des fautes. Un mécanisme de diffusion atomique garantit en plus que toutes les unités ou aucune reçoivent le message. En conclusion, la diffusion dépend de l'horloge globale commune pour la cohérence et de la gestion de groupe pour la dissémination des messages dans un groupe.

Certains services du système doivent être délivrés par un groupe de processus pour être corrects. Pour ceux-là, un mécanisme de *gestion de groupe* permet de connaître l'ensemble des membres à tout instant. Le même mécanisme prévoit les procédures de reconfiguration lors des entrées et des sorties de processus du groupe. La gestion de groupe dépend de la diffusion pour l'atomicité des modifications.

Du point de vue de la tolérance aux fautes, les mémoires du système sont divisées en deux types. La *mémoire volatile* perd son contenu lorsque survient une faute, la *mémoire stable* le garde. Les accès à la mémoire stable doivent être atomiques. La mémoire stable ne dépend d'aucune autre abstraction.

Un *processus fiable* est un processus qui continue de s'exécuter correctement même s'il est interrompu par une faute. Le mécanisme se divise en deux temps. Pendant l'exécution, des images du processus sont sauvegardées en mémoire stable. Lorsque survient une faute,

le processus rétablit son exécution à partir d'une des images précédemment sauvegardées. Le processus fiable dépend de la mémoire stable pour la sauvegarde des images et de la gestion de groupe pour la reprise et la reconfiguration.

L'*appel de procédure à distance* est un mécanisme de communication interprocessus. L'appel est exécuté par le processus appelant (le client) et le corps de la procédure par le processus appelé (le serveur). L'exécution du client est bloquée jusqu'au retour de la réponse du serveur. Suivant les implantations, l'appel de procédure est exécuté au moins une fois ou exactement une fois ou au plus une fois. L'appel de procédure dépend de la mémoire stable pour les ré-essais et de la diffusion pour la réplication de l'appel à toutes les répliques du groupe.

L'*action atomique* est le regroupement d'opérations exécutées par plusieurs unités qui apparaissent comme une transformation d'états indivisible pour les autres exécutions. L'indivisibilité s'exprime par quatre propriétés : l'atomicité, la cohérence, l'isolement et la persistence [Haer83]. L'action atomique est réalisée dans un processus fiable.

Maintenant, nous construisons le graphe des dépendances des paradigmes de programmation vis-à-vis des abstractions du système d'exploitation. Nous ne montrons que les dépendances directes.

Une conversation est un groupe de processus fiables qui communiquent par le biais de messages. Les conversations dépendent donc des processus fiables et de la gestion de groupe.

Les actions du modèle des objets et actions sont construites sur le principe des actions atomiques. Elles utilisent les appels de procédures à distance pour accéder aux objets. Le modèle des transactions dépend donc des actions atomiques et de l'appel de procédure à distance.

Dans le modèle des machines à états, les serveurs sont des processus fiables. Ils sont invoqués par des appels de procédures à distance. Ils sont répliqués et leurs répliques sont gérées par le mécanisme de gestion de groupe. Le modèle des machines à états dépend donc des processus fiables, de l'appel de procédures à distance, de la diffusion et de la gestion de groupe.

Le primaire du modèle du primaire et des secondaires est choisi parmi un groupe de processus. Il diffuse les requêtes qu'il reçoit aux secondaires. Lorsque le nombre de secondaires est insuffisant, un nouveau secondaire est construit à partir d'une image du primaire. Le modèle du primaire et des secondaires dépend donc de la gestion de groupe, de la diffusion et des processus fiables.

En conclusion, puisque la reprise sur erreur par recouvrement arrière est une réalisation du modèle du primaire et des secondaires, le mécanisme à construire dépend des abstractions suivantes :

1. processus fiable,
2. gestion de groupe,
3. diffusion,
4. horloge globale commune,
5. mémoire stable.

1.2.5 Les étapes de la tolérance logicielle aux fautes

Un système peut traverser jusqu'à onze étapes différentes lorsque survient une faute [Siew91, Lapr94] : le confinement, la détection, le masquage, le ré-essai, le diagnostic, la passivation, la reconfiguration, la récupération, le rétablissement, la réparation et la réintégration. L'ordre d'apparition des étapes n'est pas toujours celui indiqué.

1. Le *confinement* est l'acte de limiter l'étendue des erreurs induites pour que celles-ci ne se propagent pas à tous les éléments du système. Cela consiste notamment, pour un composant non-défaillant, à éviter d'activer un composant défaillant, et en conséquence, de dépendre de ce dernier.

2. La *détection* s'applique à révéler la présence d'une erreur dans un composant.

3. Le *masquage* est l'action de cacher les effets d'une faute en prenant le résultat d'un traitement à la sortie d'autres composants redondants.

4. Le *ré-essai* consiste à tenter une nouvelle fois le traitement. Cela fonctionne particulièrement bien pour les fautes temporaires d'origine logicielle : celles que Gray nomme les Heisenbugs [Gray86].

5. Le *diagnostic* s'avère nécessaire si l'étape de détection ne précise pas assez les caractéristiques d'une faute.

6. La *passivation* empêche une nouvelle activation des fautes en retirant les composants considérés comme fautifs.

7. La *reconfiguration* est nécessaire si le système ne peut plus délivrer le même service qu'auparavant.

8. La *récupération* permet de remplacer l'état erroné par un état antérieur ou par un nouvel état construit à partir de l'état erroné.

9. Le *rétablissement* fait repartir le composant (nouveau ou non) à partir du nouvel état formé par la récupération.

10. La *réparation* n'existe qu'en cas de défaillance d'un composant matériel. Elle s'effectue soit en ligne (sans retirer le composant) soit hors ligne (en isolant le composant).

11. La *réintégration* suit la réparation et consiste à rendre opérationnel le composant pour le système.

1.3 Le résumé du positionnement de l'étude

Pour conclure ce premier chapitre, nous résumons les hypothèses concernant les applications supportées, les fautes tolérées, les erreurs détectées, les modes de défaillance couverts et les abstractions du système d'exploitation nécessaires à la reprise sur erreur par recouvrement arrière.

Les applications cibles sont les applications commerciales à usage général. Donc, l'attribut qui mesure la qualité du service délivré, est la fiabilité.

Les fautes considérées sont accidentelles, permanentes et d'origine matérielle ou humaine (procédurale ou de maintenance), ou encore environnemental. Les erreurs détectées

sont des arrêts francs (de processeurs) de stations de travail. Le mode de défaillance est le mode “silence sur défaillance”. De plus, les défaillances sont perçues de manière cohérente et les conséquences sont bénignes.

Le moyen de la sûreté de fonctionnement employé est la tolérance aux fautes.

La tolérance aux fautes est logicielle, mais n'emploie ni de technique de génie logiciel ni de paradigme de programmation. La méthode de tolérance aux fautes utilisée est la reprise sur erreur par recouvrement arrière.

La reprise sur erreur par recouvrement arrière est fondée sur les processus fiables, la gestion de groupe, la diffusion, l'horloge globale commune et la mémoire stable.

Chapitre 2

La reprise sur erreur par recouvrement arrière automatique

La reprise sur erreur par recouvrement arrière automatique met en œuvre les trois mécanismes 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, certaines fois facultatif, sauvegarde en mémoire stable l'exécution de l'application répartie,
- le mécanisme de recouvrement arrière organise le retour en arrière de l'application répartie lorsque surviennent des défaillances de nœuds.

La section 2.1 décrit le modèle du système réparti et des programmes répartis. La section 2.2 introduit les problèmes et ensuite les concepts de base de la reprise sur erreur par recouvrement arrière automatique. La section 2.3 donne les objectifs des politiques en termes d'efficacité et d'efficience. La section 2.4 présente les politiques possibles et souhaitables de reprise, et, la section 2.5 développe les mécanismes de base. Enfin, la section 2.6 présente les voies de recherche prometteuses.

Dans la suite, l'expression “reprise sur erreur par recouvrement arrière automatique” est abrégée en “recouvrement arrière”.

2.1 Le modèle

Avant d'aborder les politiques et les mécanismes du recouvrement arrière, nous décrivons le modèle du système réparti puis le modèle des programmes (ou applications) répartis.

Le système réparti est un ensemble de stations de travail connectées entre elles via un réseau local. Le choix de ce système s'explique naturellement parce qu'il est de loin le plus répandu. Les premiers mécanismes ont d'ailleurs été réalisés sur ce modèle de système réparti. Depuis peu, de nombreuses études s'intéressent aux systèmes fortement couplés ; c'est l'une des voies de recherche citées en section 2.6.

Les programmes répartis sont composés d'un ensemble de programmes séquentiels. Nous nous limitons volontairement aux fautes matérielles : stations de travail à processeur "silence sur défaillances". Nous n'ignorons pas que le traitement des seules fautes matérielles est aujourd'hui insuffisant. Aussi, à titre de généralisation, nous rappelons la définition d'un programme robuste. Puis, nous montrons comment utiliser les mécanismes d'exceptions pour ajouter de la tolérance aux fautes logicielles.

La partie la plus importante de cette section comprend les définitions des concepts de base du recouvrement arrière. Nous donnons les définitions d'une exécution répartie, d'une histoire répartie, d'une exécution équivalente, du déterminisme d'exécution et de la cohérence d'état global.

2.1.1 Le système réparti

Il 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"* [Powe88] : la détection d'une erreur provoque l'arrêt franc du processeur[1]. L'état du processeur ainsi que 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)* [Hwan93a] : 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*[2] [Cris96]. Le processeur du nœud émetteur initialise la transmission puis continue son exécution. Ce sont les interfaces de communication avec le réseau qui s'occupent de la transmission des messages. Le modèle des délais de transmission est du type *"DBI"(Délais Bornés mais Inconnus)*. Par ailleurs, les communications sont fiables : sans perte, sans duplication et sans altération des messages.

2.1.2 Les programmes répartis

Le développement du modèle des programmes répartis est organisé comme suit. Le paragraphe 2.1.2.1 définit les termes "état", "action", "exécution" et "histoire". Ces définitions sont reprises et appliquées aux programmes répartis dans le paragraphe 2.1.2.3.

1. Précédemment, Schlichting et Schneider avaient défini le mode *"arrêt sur défaillance"* [Schl83]. Ce 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 faute. C'est pour cette raison que le mode "silence sur défaillance" est préféré.

2. Suite à une suggestion de Powell, Cristian parle de "*timed* asynchronous system model".

Comme nous l'avons déjà indiqué, le paragraphe 2.1.2.2 ouvre une parenthèse sur la tolérance aux fautes logicielles. Ce paragraphe donne la définition d'un programme robuste et montre comment les exceptions peuvent être utilisées conjointement au recouvrement arrière.

2.1.2.1 Programme séquentiel

Un programme assigne des *valeurs* à des *variables*. L'ensemble des valeurs est dénoté **Val** et l'ensemble des variables **Var**. Un **état** est l'affectation de valeurs à des variables - une correspondance entre **Var** et **Val**. Par conséquent, un état s assigne la valeur $s(x)$ à la variable x. L'ensemble des états est dénoté $\mathbf{S}$.

Une **action** $\mathcal{A}$ représente la relation entre un ancien état s et un nouvel état t notée $s\mathcal{A}t$. Les actions sont atomiques : l'exécution de l'action $\mathcal{A}$ dans l'état s peut produire l'état t. Par déduction, une action $\mathcal{A}$ provoque le changement "instantané" de l'état de s à t. L'ensemble des actions est dénoté $\mathbf{A}$.

Un **programme séquentiel** P est un objet syntaxique construit selon la grammaire d'un langage de programmation. Une exécution $EX_P^{s^0,t}$ du programme séquentiel P débutant à l'instant t à partir de l'état initial $s^0 \in \mathbf{S}$, est la succession d'un nombre fini d'actions sur des états notée $s^0\mathcal{A}^1s^1\mathcal{A}^2s^2\mathcal{A}^3 \ldots \mathcal{A}^f s^f$. Les états s^0 et s^f sont respectivement appelés les états initial et final de $EX_P^{s^0,t}$. Par la suite, l'exécution d'un programme séquentiel est appelée une **exécution séquentielle**. L'ensemble des exécutions possibles de P à partir d'un état initial $s \in \mathbf{S}$ est notée $\mathbf{EX}_P^s$ et l'ensemble des exécutions possibles de P notée $\mathbf{EX}_P$. L'exécution d'un programme séquentiel produit une séquence d'actions. La séquence d'actions de $EX_P^{s^0,t}$ à laquelle est ajoutée l'action initiale fictive $\mathcal{A}^0$ initialisant s^0 est appelée l'*histoire* $H_P^{s^0,t}$ du programme séquentiel. Par la suite, l'histoire d'un programme séquentiel est appelée une **histoire séquentielle**.

2.1.2.2 Fautes, erreurs et défaillances d'un nœud

Pour des raisons de simplicité évidentes, l'étude se limite à la tolérance des fautes matérielles (physiques). 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 [Gray90]. 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 programme séquentiel est qualifié de robuste [Cris89].

Aussi, à titre de généralisation, nous rappelons la définition formelle d'un programme robuste. Puis, nous montrons avec quelles précautions les mécanismes d'exception peuvent être utilisés conjointement au recouvrement arrière. En outre, cela nous amène à reformuler l'expression du mode "silence sur défaillance".

La *spécification normale* (en anglais, *standard*) $\mathbf{B}_\sigma$ [3] d'un programme P est une relation entre les états initiaux et finaux :

$$\mathbf{B}_\sigma \subseteq \mathbf{S} \times \mathbf{S}.$$

Le domaine $\mathbf{dom}(\mathbf{B}_\sigma)$ d'une relation $\mathbf{B}_\sigma$ est l'ensemble des états initiaux $s \in \mathbf{S}$ pour lesquels il existe des états finaux $t \in \mathbf{S}$:

$$\mathbf{dom}(\mathbf{B}_\sigma) \stackrel{\text{def}}{=} \{s \in \mathbf{S} \mid \exists t \in \mathbf{S} : (s,t) \in \mathbf{B}_\sigma\}.$$

3. "B" pour but, "σ" pour normal.

La *sémantique normale* $[P]_\sigma$ est la fonction que le programme P calcule lorsqu'il *se termine normalement* - la terminaison normale en langage C signifie que la "prochaine" action, séparée par un point virgule de l'action courante, sera la prochaine à s'exécuter :

$$[P]_\sigma \subseteq \mathbf{S} \times \mathbf{S}.$$

Le *domaine d'un programme* est l'ensemble des états en entrée du programme. L'ensemble des états initiaux $s \in \mathbf{S}$ tel que le programme P se termine normalement dans un état $t \in \mathbf{S}$ et qui satisfait la spécification normale $\mathbf{B}_\sigma$ est le *domaine normal* $\mathbf{DN}$ de P :

$$\mathbf{DN} \stackrel{\text{def}}{=} \{s \in \mathbf{S} \mid \exists t \in \mathbf{S} : ((s,t) \in [P]_\sigma) \wedge ((s,t) \in \mathbf{B}_\sigma)\}.$$

Si un programme P est invoqué dans un état initial n'appartenant pas au domaine normal $\mathbf{DN}$, le service normal $\mathbf{B}_\sigma$ spécifié pour P ne peut pas être fourni par P. L'ensemble des états initiaux n'appartenant pas au domaine normal, est appelé le *domaine exceptionnel* $\mathbf{DE}$:

$$\mathbf{DE} \stackrel{\text{def}}{=} \mathbf{S} - \mathbf{DN}.$$

L'exécution d'un programme dans son domaine exceptionnel est une *exécution exceptionnelle*. Un **mécanisme d'exception** est une structure de contrôle d'un langage permettant au programmeur d'exprimer le remplacement de la continuation d'un programme par une continuation exceptionnelle lors de la détection d'une exception. Soit $\mathbf{E}$ l'ensemble des exceptions que le programmeur déclare pour P afin de fournir un service exceptionnel. Une *spécification exceptionnelle* $\mathbf{B}_e$ pour un programme P indique les actions à exécuter lorsqu'une exception $e \in \mathbf{B}_e$ est détectée :

$$\mathbf{B}_e \subseteq \mathbf{S} \times \mathbf{S}.$$

Par conséquent, la *spécification* $\mathbf{B}$ de P est l'ensemble des spécifications normales et exceptionnelles définit pour P :

$$\mathbf{B} \stackrel{\text{def}}{=} \{\mathbf{B}_x \mid x \in \mathbf{E} \cup \{\sigma\}\},$$

où σ est le point de sortie normal de P. Soit $\mathbf{EA}$ (Entrée Anticipée) l'ensemble des entrées pour lesquelles le comportement de P est spécifié :

$$\mathbf{EA} \stackrel{\text{def}}{=} \bigcup_{x \in \mathbf{E} \cup \{\sigma\}} \mathbf{dom}(\mathbf{B}_x).$$

Une spécification *complète* indique le comportement de P pour tous les états en entrée $s \in \mathbf{S}$:

$$\mathbf{S} \subseteq \mathbf{EA}.$$

Un programme P est dit *totalement correct* respectivement à une spécification $\mathbf{B}$ si sa sémantique $[P]$ "correspond" à la sémantique de $\mathbf{B}$:

$$\texttt{totalcorrect}(P) \stackrel{\text{def}}{=} \begin{cases} & \forall x \in \mathbf{E} \cup \{\sigma\} : \mathbf{dom}(\mathbf{B}_x) \subseteq \mathbf{dom}([P]_x) \\ \wedge & \forall s,t \in \mathbf{dom}(\mathbf{B}_x), \mathbf{S} : (s,t) \in [P]_x \Rightarrow (s,t) \in \mathbf{B}_x \end{cases}$$

avec le prédicat $\texttt{totalcorrect}(P)$ évalué à "vrai" ssi (si et seulement si) P est totalement correct. Ainsi, P est totalement correct s'il se termine à un point x à chaque fois que la spécification $\mathbf{B}$ demande qu'il se termine à x. En outre, tout état final t à partir d'un état initial $s \in \mathbf{dom}(\mathbf{B}_x)$, pour tout $x \in \mathbf{E} \cup \{\sigma\}$, est toujours conforme à la spécification $\mathbf{B}_x$.

Il est à noter que cela n'implique pas que $[P]$ et $\mathbf{B}$ sont égales. Un programme totalement correct respectivement à une spécification complète est **robuste** - son comportement est prévisible pour toutes les entrées possibles.

Par définition, tout mécanisme d'exception modifie le comportement d'un programme P dans des situations exceptionnelles. La spécification exceptionnelle peut prévoir de :

1. recouvrir l'erreur en ramenant l'exécution à un état antérieur,
2. recouvrir l'erreur en transformant l'état erroné à partir duquel l'exécution de P peut se poursuivre,
3. compenser l'erreur en utilisant les redondances dans l'état erroné pour fournir un service conforme à la spécification, ou,
4. terminer l'exécution de P en arrêtant le processeur du nœud.

Le premier cas (recouvrement par reprise) réalise une fonction identique à celle que nous nous proposons de réaliser. Donc, il risque d'y avoir un conflit entre les deux mécanismes de tolérance aux fautes.

Les deuxième et troisième cas (respectivement, recouvrement par poursuite et par compensation) contredisent l'hypothèse de déterminisme d'exécution que nous énoncerons plus loin. L'idée est que le déroutement de l'exécution n'est pas obligatoirement automatiquement "rejoué". Par exemple, une exécution peut ne pas rencontrer un manque d'espace mémoire lorsqu'elle est "rejouée" sur un autre nœud. Dorénavant, nous considérons que le programmeur optant pour ce type de solution, sait que son programme séquentiel devient indéterministe. Il devra donc en tenir compte pour le choix de la méthode de recouvrement arrière.

Quant au dernier cas, il correspond au cas "silence sur défaillance". L'arrêt du processeur du nœud est nécessaire car les états du processeur ou de la mémoire ou encore des interfaces de communication peuvent être corrompus. En effet, la latence du mécanisme de détection est au moins égal au temps de calcul de l'assertion vérifiant l'état de l'exécution de P.

Par conséquent, nous redéfinissons de façon plus précise le mode **"silence sur défaillance"** comme suit : défaillance matérielle du processeur, ou, détection d'une erreur dans l'exécution d'un programme robuste provoquant une action exceptionnelle arrêtant le processeur du nœud. Dans la suite de l'étude, tout programme est supposé robuste.

2.1.2.3 Programme réparti

Un **programme réparti** $\widehat{P}$[4] est composé de $p =\mid \widehat{\mathbf{P}} \mid$ programmes séquentiels communiquant par échange de messages asynchrones. L'ensemble des programmes séquentiels est noté $\widehat{\mathbf{P}}$. Les programmes séquentiels sont supposés *connectés* :

$$\forall (P_i, P_j) \in \widehat{\mathbf{P}} \times \widehat{\mathbf{P}}, c_{ij},$$

avec c_{ij} une fonction booléenne évaluée à "vrai" ssi P_i et P_j sont reliés par un canal bi-directionnel.

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

4. "$\widehat{\ }$" pour réparti.

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 convention, les messages sont numérotés dans leur ordre d'arrivée par le programme séquentiel. On obtient la séquence finie suivante : $m_j^1 m_j^2 m_j^3 \ldots m_j^f$, où m_j^f est le dernier message reçu avant la terminaison du programme séquentiel P_j.

Toute action autre qu'une action d'émission ou de réception d'un message est appelée une **action interne**.

L'**état d'un canal** c_{ij} à l'instant physique t est constitué de l'ensemble des messages émis et non encore reçus. L'**état global** $\widehat{s}$ d'un programme réparti $\widehat{P}$ à l'instant physique t est composé des états $s_i^{x_i}$ [5] de tous les programmes séquentiels $P_i \in \widehat{\mathbf{P}}$ à l'instant t [Chan85]. L'ensemble des états globaux est noté $\widehat{\mathbf{S}}$.

L'exécution $EX_{\widehat{P}}^{\widehat{s},t}$ d'un programme réparti $\widehat{P}$ débutant à l'instant physique t à partir de l'état global $\widehat{s}$, est constituée de la juxtaposition des exécutions des programmes séquentiels. Par la suite, l'exécution du programme réparti est appelée une **exécution répartie**. Par déduction, l'histoire $H_{\widehat{P}}^{\widehat{s},t}$ d'un programme réparti $\widehat{P}$ est constituée de la juxtaposition des histoires des programmes séquentiels. Par la suite, l'histoire du programme réparti est appelée une **histoire répartie**. Par définition, l'histoire des messages $HM_{\widehat{P}}^{\widehat{s},t}$ de l'exécution répartie $EX_{\widehat{P}}^{\widehat{s},t}$ est égale à l'histoire de l'exécution répartie à laquelle toutes les actions internes sont retranchées :

$$HM_{\widehat{P}}^{\widehat{s},t} \stackrel{\text{def}}{=} \{\mathcal{A} \in H_{\widehat{P}}^{\widehat{s},t} \mid \exists m : (\mathcal{A} = \texttt{émettre}(P_j, m)) \vee (\mathcal{A} = \texttt{recevoir}(m))\}.$$

Lamport définit la relation binaire $\prec_{\mathbf{A}}$ sur les actions appelée **précédence causale** [Lamp78] :

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

Deux relations de précédence peuvent être définies sur l'ensemble des états locaux produits par une exécution répartie. L'ordre de précédence causale induit un ordre strict sur les états des programmes séquentiels. Cet ordre noté $\prec_f$ et appelé **précédence faible**, indique qu'un état s_i^x a commencé d'exister avant un autre état s_j^y [From94] :

$$s_i^x \prec_f s_j^y \stackrel{\text{def}}{=} \mathcal{A}_i^x \prec_{\mathbf{A}} \mathcal{A}_j^y.$$

Fromentin et Raynal définissent une autre relation d'ordre strict. 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 [From94] :

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

L'exécution d'un programme réparti est classiquement représentée par un diagramme espace-temps. La figure 2.1 trace un tel diagramme pour un programme réparti composé de

5. le $x_i$ème état local de P_i.

trois programmes séquentiels. Le déroulement du temps est décrit par une ligne continue pour chaque programme séquentiel. Les actions sont symbolisées par des tirets sur les programmes séquentiels. 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 : $\mathcal{A}_1^1 \prec_{\mathbf{A}} \mathcal{A}_2^1$, $s_1^1 \prec_f s_2^1$, $s_1^1 \not\prec_F s_2^1$ et $s_1^1 \prec_F s_2^3$.

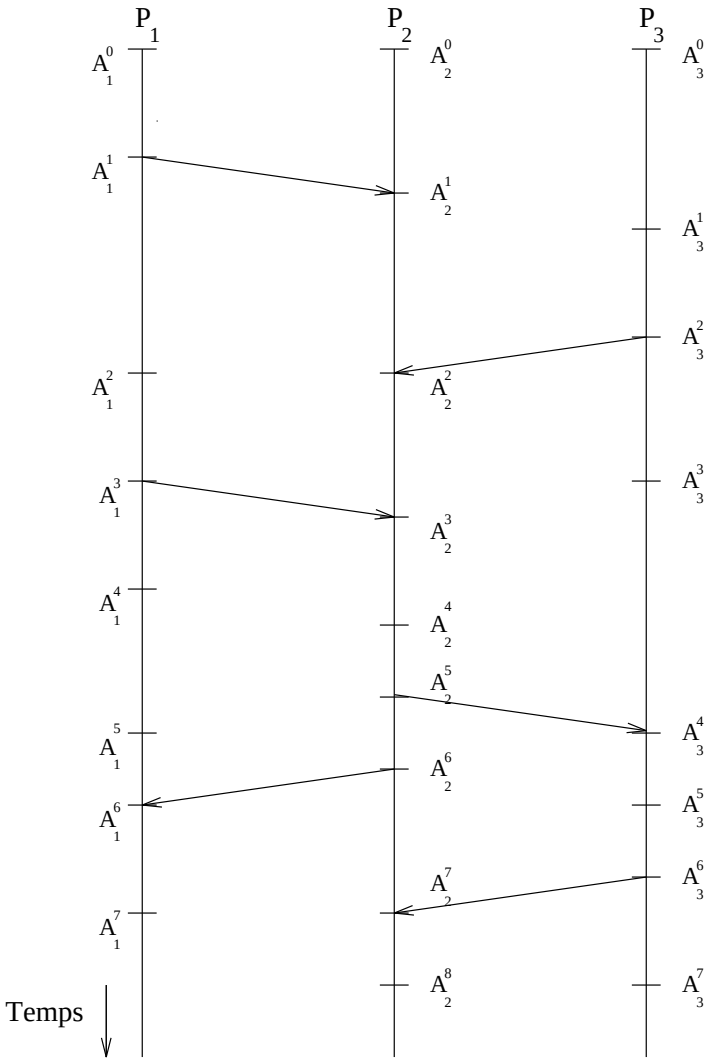


FIG. 2.1 – Le diagramme espace-temps d'une exécution répartie.

2.2 Les problèmes

Cette section examine les deux problèmes du recouvrement en arrière : le déterminisme d'exécution et la cohérence d'état global.

Tout d'abord, le déterminisme d'exécution exprime la faculté de répéter l'exécution d'un programme. Cette propriété détermine le choix de l'utilisation ou non de la journalisation de l'exécution. En résumé, rien ne sert de sauvegarder en mémoire stable l'histoire de l'exécution, si la ré-exécution 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 tel qu'il a pu "réellement

exister" durant l'exécution. À partir de la définition d'un état global, nous introduisons les concepts de point de reprise utile, de programme orphelin, de message stable, d'état recouvrable et d'effet domino.

2.2.1 Déterminisme d'exécution

Dans la réalité, un programme s'exécute au dessus d'un système d'exploitation et fait régulièrement appel à des fonctionnalités fournies par ce dernier. En outre, il accède au contenu de la mémoire stable. Enfin, certaines de ses actions peuvent directement dépendre de la valeur de l'horloge physique du nœud au moment où elles s'exécutent. Les variables d'un programme séquentiel sont donc divisées en quatre parties distinctes :

1. variables de calculs - celles qui n'appartiennent pas aux trois autres catégories -,
2. variables "représentantes" [6] de variables de la mémoire stable,
3. variables "représentantes" de variables du système d'exploitation,
4. variables "représentantes" de l'horloge physique.

Prenons l'exemple du système d'exploitation UNIX. Les incréments de boucles sont des variables de calculs. Les tampons d'entrée/sortie des fichiers de la mémoire stable sont des variables "représentantes" de variables de la mémoire stable. Les descripteurs des fichiers ouverts sont des variables "représentantes" de variables du système d'exploitation. Les valeurs des temporisations sont des variables "représentantes" de l'horloge physique.

Étant donné un état initial s et une action $\mathcal{A}$ d'un programme séquentiel P, cherchons à obtenir le même état final pour deux exécutions. Si l'état final est uniquement fonction des variables de calcul de l'état initial, les deux exécutions de $\mathcal{A}$ aboutissent naturellement au même état final. Si l'état final est fonction des variables de la dernière catégorie, deux exécutions de $\mathcal{A}$ ne se déroulant pas à la même valeur de l'horloge physique ne donnent évidemment pas le même état final. En ce qui concerne les variables "représentantes" de la mémoire stable, selon que le contenu de la mémoire stable concerné est modifié ou non entre deux exécutions de $\mathcal{A}$, le même état final ou deux états finaux différents sont obtenus. Il reste les variables "représentantes" des variables du système d'exploitation. Par soucis de simplicité, dans ce qui suit, nous considérons, à tort, qu'elles servent toutes à nommer un "service" du système d'exploitation. En outre, les noms globaux (uniques) de ces services ne changent pas d'une exécution à une autre. Par exemple, le descripteur de fichier est un nom global autorisant entre autre l'exécution des fonctions "lire" et "écrire". Le descripteur de ce fichier est alors supposé être un nom global unique du système réparti. Ce point dépend trop du système d'exploitation pour qu'il soit complètement élucidé à ce stade de la discussion (*cf.* sous-section 2.5.1). Par conséquent, deux exécutions de $\mathcal{A}$ utilisant les variables "représentantes" de variables du système d'exploitation donnent un même état final.

Plus formellement, une action $\mathcal{A}$ est **déterministe** ssi plusieurs exécutions de $\mathcal{A}$ à partir du même état initial donnent le même état final. Dans ce cas, le prédicat `déterministe`($\mathcal{A}$) est évalué à "vrai". Deux exécutions séquentielles EX_P^{s,t_1} et EX_P^{s,t_2} à partir d'un même état initial $s \in \mathbf{S}$ sont **équivalentes** lorsque leurs histoires séquentielles sont égales et

6. Par "représentantes", nous entendons que ces variables internes au programme sont des copies de données externes au programme.

toutes les actions sont déterministes :

$$EX_P^{s,t_1} \equiv EX_P^{s,t_2} \stackrel{\text{def}}{=} \begin{cases} & H_P^{s,t_1} = H_P^{s,t_2} \\ \wedge & \forall \mathcal{A} \in H_P^{s,t_1}, \texttt{déterministe}(\mathcal{A}) \end{cases}$$

avec $\equiv$ signifiant l'équivalence.

Un **programme séquentiel déterministe** P est un programme séquentiel tel qu'à partir de tout état initial $s \in \mathbf{S}$, une seule exécution séquentielle est possible :

$$\texttt{déterministe}(P) \stackrel{\text{def}}{=} \forall s \in \mathbf{S}, \mid \mathbf{EX}_P^s \mid = 1.$$

Autrement dit, un programme séquentiel déterministe permet des exécutions séquentielles équivalentes :

$$\texttt{déterministe}(P) \Rightarrow \forall i, j \in \mathbf{N} \times \mathbf{N}, EX_P^{s,t_i} \equiv EX_P^{s,t_j}.$$

Pour un programme réparti, 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 conflictuels"* (en anglais, *racing messages*) [Netz94], le prédicat $\texttt{conflit}(m_1, m_2)$ est alors évalué à "vrai" :

$$\texttt{conflit}(m_1, m_2) \stackrel{\text{def}}{=} \begin{cases} & (\mathcal{A}_i^x = \texttt{recevoir}(m_1)) \prec_{\mathbf{A}} (\mathcal{A}_i^z = \texttt{recevoir}(m_2)) \\ \wedge & \nexists \mathcal{A}_i^y = \texttt{émettre}(l, P_j) : (\mathcal{A}_i^x \prec_{\mathbf{A}} \mathcal{A}_i^y) \wedge (\mathcal{A}_i^y \prec_{\mathbf{A}} \mathcal{A}_i^z) \end{cases}$$

Deux exécutions réparties $EX_{\widehat{P}}^{\widehat{s},t_1}$ et $EX_{\widehat{P}}^{\widehat{s},t_2}$ à partir d'un même état global initial $\widehat{s} = (s_1, s_2, \ldots, s_{|\widehat{P}|}) \in \widehat{\mathbf{S}}$ sont équivalentes lorsque leurs histoires réparties sont égales et toutes les actions déterministes :

$$EX_{\widehat{P}}^{\widehat{s},t_1} \equiv EX_{\widehat{P}}^{\widehat{s},t_2} \stackrel{\text{def}}{=} \begin{cases} & H_{\widehat{P}}^{\widehat{s},t_1} = H_{\widehat{P}}^{\widehat{s},t_2} \\ \wedge & \forall P_i \in \widehat{\mathbf{P}}, (\forall \mathcal{A} \in H_{P_i}^{s_i,t_1}, \texttt{déterministe}(\mathcal{A})) \end{cases}$$

Un **programme réparti** $\widehat{P}$ est **déterministe** ssi, à partir d'un état global initial $\widehat{s} \in \widehat{\mathbf{S}}$, une seule exécution répartie est possible :

$$\texttt{déterministe}(\widehat{P}) \stackrel{\text{def}}{=} \forall \widehat{s} \in \widehat{\mathbf{S}}, \mid \mathbf{EX}_{\widehat{P}}^{\widehat{s}} \mid = 1.$$

Il est évident que très peu de programmes répartis sont déterministes. De la même façon que pour un programme séquentiel déterministe, un programme réparti déterministe permet des exécutions réparties équivalentes :

$$\texttt{déterministe}(\widehat{P}) \Rightarrow \forall i, j \in \mathbf{N} \times \mathbf{N}, EX_{\widehat{P}}^{\widehat{s},t_i} \equiv EX_{\widehat{P}}^{\widehat{s},t_j}.$$

Par définition, un **programme réparti** $\widehat{P}$ est défini **presque déterministe** (noté p-déterministe) ssi, à partir d'un même état global initial $\widehat{s} \in \widehat{\mathbf{S}}$ et d'une même histoire des messages $HM_{\widehat{P}}^{\widehat{s},t}$, deux exécutions sont équivalentes :

$$\texttt{p-déterministe}(\widehat{P}) \stackrel{\text{def}}{=} (HM_{\widehat{P}}^{\widehat{s},t_1} = HM_{\widehat{P}}^{\widehat{s},t_2}) \Rightarrow (EX_{\widehat{P}}^{\widehat{s},t_1} \equiv EX_{\widehat{P}}^{\widehat{s},t_2}).$$

En pratique, l'histoire des messages est obtenue par le mécanisme de journalisation lors de l'exécution. Dans la suite de l'étude, un programme réparti est soit p-déterministe soit indéterministe. Les résultats pour un programme réparti p-déterministe seront valables pour un programme réparti déterministe.

2.2.2 Cohérence d'état global

Le deuxième problème du recouvrement arrière est la cohérence de l'état global. Avant ce concept, nous définissons les notions d'exécution défaillante et de nouvelle exécution. Ensuite, nous étudions l'impact de la cohérence de l'état global sur chacun des mécanismes de base. L'objectif est ici de fournir les concepts de base pour la présentation des politiques possibles du recouvrement arrière.

2.2.2.1 Exécution défaillante et nouvelle exécution

Lors de l'exécution sans occurrence de fautes d'un programme séquentiel P_{TF} [7], certains de ces états sont sauvegardés en mémoire stable. Ce sont des **points de reprise**. L'ensemble des points de reprise d'un programme séquentiel P_{TF} est noté $\mathbf{R}_{P_{TF}}$. Remarquons que la constitution d'un point de reprise doit respecter la règle d'atomicité des actions du programme séquentiel. Pratiquement, cela signifie qu'un point de reprise ne peut pas être constitué au milieu d'un appel système. La constitution d'un point de reprise peut être vue comme l'insertion d'une action C dans l'exécution : $s^r \mathcal{C} s^r$. Mais, il est souvent plus simple de considérer qu'une entité extérieure est capable de saisir l'état du programme en l'interrompant entre deux actions :

$$EX^{s,t}_{P_{TF}} \equiv EX^{s,t}_{P}.$$

Nous utiliserons tantôt l'une tantôt l'autre des formulations.

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[8]. L'*état défaillant* est l'état s^{d-1} précédant la défaillance du processeur. Pratiquement, cet état est inconnu car perdu et ne peut pas être récupéré. Pour simplifier, considérons une seule faute pendant l'exécution $EX^{s^0,t^0}_{P_{TF}}$. Une **exécution défaillante** $EXD^{s^0,t^0}_{P_{TF}}$ du programme séquentiel P_{TF} est une exécution de P_{TF} durant laquelle une faute survient pendant l'action défaillante $\mathcal{A}^d$:

$$EXD^{s^0,t^0}_{P_{TF}} \stackrel{\text{def}}{=} s^0 \mathcal{A}^1 s^1 \mathcal{A}^2 \ldots s^{d-1}.$$

L'exécution du programme séquentiel doit reprendre à partir d'un des points de reprise précédemment sauvegardés. Par définition, la *ré-exécution* $REX^{s^r,t^r}_{P_{TF}}$ est l'exécution du programme séquentiel à partir du point de reprise $s^r \in \mathbf{S}$:

$$REX^{s^r,t^r}_{P_{TF}} \stackrel{\text{def}}{=} \exists s^r \in \mathbf{S}, (r \leq d) : (s^r \mathcal{A}^{r+1} s^{r+1} \mathcal{A}^{r+2} \ldots s^f).$$

La **nouvelle exécution** $NEX^{s^0,t^0}_{P_{TF}}$ est l'exécution à partir de l'état initial s^0 jusqu'à l'état de reprise s^r, plus la ré-exécution $REX^{s^r,t^r}_{P_{TF}}$:

$$NEX^{s^0,t^0}_{P_{TF}} \stackrel{\text{def}}{=} s^0 \mathcal{A}^1 s^1 \mathcal{A}^2 \ldots s^r \mathcal{A}^{r+1} s^{r+1} \mathcal{A}^{r+2} \ldots s^f.$$

La ré-exécution de P_{TF} peut être vue comme l'exécution d'un nouveau programme séquentiel P' dont la sémantique est incluse dans celle de P_{TF}. D'où, intuitivement, pour être robuste, l'état initial s^r de P' doit avoir "réellement pu exister" durant l'exécution. Or, il est évident que cette règle est toujours vérifiée pour un programme séquentiel. Donc, le comportement de P' est conforme à la spécification $\mathbf{B}$. Finalement, nous établissons que le

7. "TF" pour "Tolérant aux Fautes".

8. Identique = sans effet. Le programme n'était pas en train de s'exécuter quand survient la défaillance.

programme séquentiel tolérant aux fautes P_{TF} est robuste. En outre, P' possède le même comportement que P_{TF} ssi P est déterministe :

$$EX_{P_{TF}}^{s^0,t^0} \equiv NEX_{P_{TF}}^{s^0,t^0} \Leftrightarrow \texttt{déterministe}(P).$$

Par abus de langage, une ré-exécution telle que la nouvelle exécution est équivalente à l'exécution, est dite équivalente (à l'exécution).

2.2.2.2 Cohérence d'état global

Une **ligne de reprise** $\widehat{s^r}$ d'un programme réparti $\widehat{P}_{TF}$ est définie comme un ensemble de points de reprise : $\widehat{s^r} = \{s_1^r, s_2^r, \ldots, s_{|\widehat{P}_{TF}|}^r\} \in \widehat{\mathbf{S}}$. Par analogie avec un programme séquentiel, lorsqu'une faute survient, l'exécution du programme réparti doit reprendre à partir d'un état global ayant "réellement pu exister" durant l'exécution. Pour un programme réparti, cette condition s'exprime par la propriété de **cohérence** sur un état global $\widehat{s} = \{s_1, s_2, \ldots, s_{|\widehat{P}_{TF}|}\} \in \widehat{\mathbf{S}}$ [Chan85, Rayn94] :

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

avec $\texttt{cohérent}(\widehat{s})$ un prédicat évalué à "vrai" ssi $\widehat{s}$ est un état global cohérent. Ce prédicat signifie simplement que l'action de réception d'un message ne peut pas être observée si son action d'émission ne l'est pas aussi. Dans l'exemple de la figure 2.1, l'état global $\{s_1^1, s_2^1, s_3^1\}$ est cohérent tandis que l'état global $\{s_1^1, s_2^3, s_3^3\}$ ne l'est pas à cause de la relation $s_1^1 \prec_F s_2^3$.

À la suite du paragraphe 2.1.2.2, nous en déduisons que, pour que $\widehat{P}_{TF}$ soit robuste, il faut que les lignes de reprise soient cohérentes.

2.2.2.3 Cohérence et constitution de points de reprise

Lors de l'exécution d'un programme réparti, des points de reprise sont constitués. L'ensemble des points de reprise d'un programme réparti est noté $\mathbf{R}_{\widehat{P}_{TF}}$. Sans entrer dès maintenant dans les détails, il existe trois façons de constituer des points de reprise en environnement réparti : construire une "coupure" cohérente, ne pas coordonner les constitutions, et, constituer des points de reprise "utiles".

La première solution est la construction d'une coupure cohérente. Une *coupure* $\widehat{\mathbf{CO}}$ est un ensemble d'actions définissant un état global de l'exécution répartie $EX_{\widehat{P}_{TF}}^{\widehat{s^0},t^0}$. C'est en fait l'histoire de l'exécution jusqu'à un certain état global. Une **coupure** est dite **cohérente** ssi l'état global correspondant à la coupure est cohérent :

$$\texttt{cohérent}(\widehat{\mathbf{CO}}) \stackrel{\text{def}}{=} \forall \mathcal{A}_i \in \widehat{\mathbf{CO}}(\forall \mathcal{A}_j \in H_{\widehat{P}_{TF}}^{\widehat{s^0},t^0}, (\mathcal{A}_j \prec_{\mathbf{A}} \mathcal{A}_i) \Rightarrow (\mathcal{A}_j \in \widehat{\mathbf{CO}})).$$

Le mécanisme de constitution de points de reprise peut coordonner les constitutions locales pour, à chaque fois, construire une coupure cohérente.

La deuxième solution construit des points de reprise sans former explicitement des états globaux cohérents. Nous verrons plus loin que la journalisation de l'exécution répartie permet de reconstruire un état global dit recouvrable qui est cohérent (*cf.* § 2.2.2.6). Par abus de langage, les **points de reprise** ainsi constitués sont dits **non coordonnés**.

La dernière solution est basée sur la définition (donnée ci-dessous) d'un point de reprise "utile". Contrairement à la première solution qui utilise la condition nécessaire de

cohérence d'un état global, ici, c'est la condition nécessaire et suffisante qui est exprimée. L'objectif est de ne constituer que des points de reprise utiles. Intuitivement, un point de reprise est utile soit parce que le programme séquentiel le demande soit pour éviter que d'autres points de reprise deviennent inutiles. Netzer et Xu ont, les premiers, donné une définition de l'utilité d'un point de reprise [Netz95]. Cependant, nous préférons la formulation de Baldoni et *al.*, car elle est plus expressive et les auteurs y joignent un algorithme de calcul [Bald95a].

Auparavant, il faut introduire la relation de précédence causale sur les intervalles de points de reprise. Un *intervalle de points de reprise* I_i^x d'un programme séquentiel 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) :

$$I_i^x \stackrel{\text{def}}{=} \{\mathcal{A}_i^a \in H_{P_i}^{s_i,t_i} \mid (\mathcal{A}_i^a = \mathcal{C}_{R_i^x}) \vee ((\mathcal{C}_{R_i^x} \prec_{\mathbf{A}} \mathcal{A}_i^a) \wedge (\mathcal{A}_i^a \prec_{\mathbf{A}} \mathcal{C}_{R_i^{x+1}}))\}.$$

L'ensemble des intervalles de points de reprise est noté $\mathbf{I}$. La relation de précédence causale 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 une ligne de reprise cohérente :

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

Un point de reprise R_i^x est **utile** ssi il peut appartenir à une ligne de reprise cohérente :

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

avec le prédicat $\texttt{utile}(R_i^x)$ évalué à "vrai" ssi R_i^x est utile.

Reprenons la figure 2.1 et supposons que toutes les actions internes sont des constitutions de points de reprise. La figure 2.2 trace les relations de précédence causale entre les intervalles de points de reprise. Sur la figure, seules les deuxième et troisième termes de la relation $\prec_{\mathbf{I}}$ sont représentées. Les intervalles de reprise I_1^0, I_1^1, I_1^2, I_1^3, I_1^4, I_2^0, I_2^1, I_2^2, I_3^0, I_3^1, I_3^2, I_3^3 et I_3^4 sont respectivement consécutifs aux actions $\mathcal{A}_1^0$, $\mathcal{A}_1^2$, $\mathcal{A}_1^4$, $\mathcal{A}_1^5$, $\mathcal{A}_1^7$, $\mathcal{A}_2^0$, $\mathcal{A}_2^4$, $\mathcal{A}_2^8$, $\mathcal{A}_3^0$, $\mathcal{A}_3^1$, $\mathcal{A}_3^3$, $\mathcal{A}_3^5$ et $\mathcal{A}_3^7$. La ligne de reprise $\{\mathcal{A}_1^4, \mathcal{A}_2^4, \mathcal{A}_3^3\}$ est cohérente alors que la ligne de reprise $\{\mathcal{A}_1^2, \mathcal{A}_2^4, \mathcal{A}_3^3\}$ ne l'est pas à cause de la relation $(I_3^3 \prec_{\mathbf{I}} I_2^1) \wedge (I_2^1 \prec_{\mathbf{I}} I_3^2)$.

2.2.2.4 Cohérence et journalisation de l'exécution répartie

Le rôle du mécanisme de journalisation est de sauvegarder de manière stable l'exécution répartie. Afin de minimiser la quantité d'information à sauvegarder, seules les actions concernant les messages sont journalisées. Pour étudier l'impact de la cohérence sur la journalisation, trois propriétés sur les messages sont définies : *orphelin*, *manquant* et *stable*.

Un **message** m émis par P_i vers P_j est dit **orphelin** ssi son émission n'appartient pas à l'histoire du programme réparti $H_{\widehat{P}_{TF}}^{\widehat{s,t}}$ tandis que sa réception y appartient [Brze95]:

$$\texttt{orphelin}(m) \stackrel{\text{def}}{=} (\texttt{émettre}(P_j, m) \notin H_P^{s,t}) \wedge (\texttt{recevoir}(m) \in H_P^{s,t}),$$

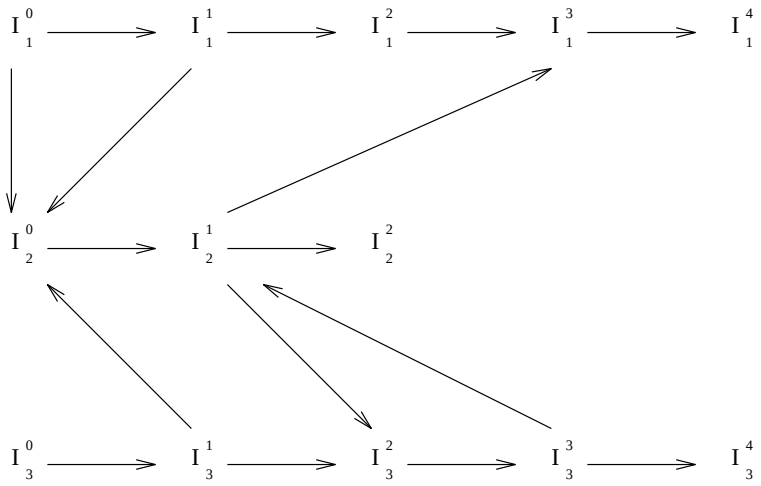


FIG. 2.2 – Le graphe de précédences causales des intervalles de constitutions de points de reprise.

avec le prédicat $\texttt{orphelin}(m)$ évalué à "vrai" ssi m est un message orphelin. Par définition, un état local s_i^x du programme séquentiel P_i est *orphelin* ssi s_i^x dépend d'un message orphelin :

$$\texttt{orphelin}(s_i^x) \stackrel{\text{def}}{=} \begin{cases} & \exists m : s_i^x = \texttt{recevoir}(m) \wedge \texttt{orphelin}(m) \\ \vee & \exists m, \exists y : \mathcal{A}_i^y = \texttt{recevoir}(m) \wedge (\mathcal{A}_i^y \prec_{\mathbf{A}} \mathcal{A}_i^x) \wedge \texttt{orphelin}(m) \end{cases}$$

Un programme séquentiel ayant reçu un message orphelin est, par extension, dit orphelin.

Un **message** m émis par P_i vers P_j est dit **manquant** ssi son émission appartient à l'histoire du programme réparti $H^{\widehat{s,t}}_{\widehat{P}_{TF}}$ tandis que sa réception n'y appartient pas [Brze95]:

$$\texttt{manquant}(m) \stackrel{\text{def}}{=} (\texttt{émettre}(P_j, m) \in H_P^{s,t}) \wedge (\texttt{recevoir}(m) \notin H_P^{s,t}),$$

avec le prédicat $\texttt{manquant}(m)$ évalué à "vrai" ssi m est un message manquant.

Par définition, le *point de reprise effectif* $RE_i^{x_i}$ d'un programme séquentiel P_i est le dernier point de reprise de P_i. Un **message** m est dit **stable** ssi l'histoire de tous les messages reçus depuis le point de reprise effectif est sauvegardée en mémoire stable [John90b] :

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

avec le prédicat $\texttt{stable}(m)$ évalué à "vrai" ssi m est un message stable, le prédicat $\texttt{jms}(m)$ [9] évalué à "vrai"ssi l'histoire de m est sauvegardée en mémoire stable. Il est évident qu'un message stable ne peut pas devenir orphelin ou manquant :

$$\texttt{stable}(m) \Rightarrow \neg\texttt{orphelin}(m) \wedge \neg\texttt{manquant}(m).$$

Soit un état local s_i^x d'un programme séquentiel P_i et soit l le dernier message reçu par P_i avant s_i^x, s_i^x est dit *stable* ssi l est stable :

$$\texttt{stable}(s_i^x) \stackrel{\text{def}}{=} \exists l, \exists y : (\mathcal{A}_i^y = \texttt{recevoir}(l)) \wedge (((\mathcal{A}_i^y \prec_{\mathbf{A}} \mathcal{A}_i^x) \vee (\mathcal{A}_i^y = \mathcal{A}_i^x)) \Rightarrow \texttt{stable}(l)).$$

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

2.2.2.5 Cohérence et monde extérieur du programme réparti

Rappelons qu'un programme réparti est défini comme une composition de programmes séquentiels. L'état d'un programme séquentiel est la composition de variables de calcul ainsi que de variables "représentantes" de la mémoire stable ou du système d'exploitation ou encore de l'horloge physique. Nous pouvons donc dire que l'état du programme réparti comprend un nombre fini de variables formant un monde clos. Heureusement, rien n'empêche un programme séquentiel de communiquer avec le monde extérieur au programme réparti. Seulement, le monde extérieur au programme réparti n'est pas censé être tolérant aux fautes, ni même pouvoir se recouvrir en arrière. Par conséquent, les messages émis à destination du monde extérieur ne doivent jamais être défaits. En d'autres termes, l'état global précédant l'émission doit être "recouvrable" (*cf.* sous-section 2.2.2.6). Par ailleurs, pour pouvoir être rejoués, les messages reçus en provenance du monde extérieur doivent être sauvegardés en mémoire stable.

2.2.2.6 Cohérence et recouvrement arrière

Dans les paragraphes précédents, la cohérence d'une ligne de reprise a été définie par rapport aux messages orphelins. Cependant, si au lancement du programme réparti, l'exécution dépend uniquement des états des programmes séquentiels, en cours d'exécution, il n'en est pas de même. En effet, à tout instant "logique" où l'état global est cohérent, l'exécution dépend aussi de l'état des canaux : des messages émis qui sont à recevoir - des messages manquants. D'où, la ré-exécution dépend de l'état global du programme réparti plus des messages orphelins ou manquants. Lorsque survient une défaillance, le programme réparti doit recouvrir un état global cohérent sans message manquant. Le recouvrement arrière dépend donc de l'utilisation ou non d'un mécanisme de journalisation.

Lorsqu'un mécanisme de journalisation est utilisé, la propriété de recouvrabilité s'exprime en fonction de la stabilité des états locaux [John90b]. Un **état global** $\widehat{s} = \{s_1^{x_1}, s_2^{x_2}, \ldots, s_{|\widehat{\mathbf{P}}_{TF}|}^{x_{|\widehat{\mathbf{P}}_{TF}|}}\}$ d'un programme réparti $\widehat{\mathbf{P}}_{TF}$ est **recouvrable** ssi tous les états locaux sont stables et $\widehat{s}$ est cohérent :

$$\texttt{recouvrable}(\widehat{s}) \stackrel{\text{def}}{=} \texttt{cohérent}(\widehat{s}) \wedge (\forall P_i \in \widehat{\mathbf{P}}_{TF}, \texttt{stable}(s_i^{x_i})).$$

Si les réceptions de messages ne sont pas synchronisées avec les sauvegardes en mémoire stable, les états locaux de certains programmes séquentiels peuvent être orphelins. Si cet état persiste à cause d'une défaillance alors ces programmes séquentiels orphelins doivent eux-aussi reprendre leur exécution.

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

Lorsqu'aucun message n'est sauvegardé en mémoire stable pendant l'exécution, le recouvrement arrière est sujet à l'effet domino [Russ80, Koo87]. Intuitivement, il peut être possible de remonter loin dans le temps l'exécution d'un programme réparti et de trouver des coupures cohérentes présentant des messages manquants. C'est pour éviter cet effet que les algorithmes de coupures cohérentes calculent et sauvegardent les messages manquants [Chan85]. Remarquons que l'ensemble des messages manquants correspondant à la coupure peut être vide. Plus formellement, pour une exécution répartie $H_{\widehat{P}_{TF}}^{\widehat{s},t}$, l'**effet**

domino est α-**borné** ($\alpha \geq 0$) dans $\mathbf{R}_{\widehat{P}_{TF}}$ ssi l'effet domino ne peut pas faire revenir l'exécution d'un programme séquentiel plus de α points de reprise en arrière [Bald95b]:

$$\forall P_i \in \widehat{\mathbf{P}}_{TF}, ((I_i^x \prec_{\mathbf{I}} I_i^y) \wedge (y < x)) \Rightarrow (x - y \leq \alpha).$$

Par ailleurs, Baldoni et *al.* concluent que les mécanismes ne constituant que des points de reprise utiles évitent l'effet domino, ceci quelque soit α. Les états des canaux correspondant aux états globaux cohérents sont toujours vides - l'ensemble des messages manquants est vide.

2.2.2.7 Cohérence et effacement des informations de recouvrement

La quantité d'information sauvegardée en mémoire stable peut devenir trop importante. Au cours de l'exécution, les informations devenant non nécessaires pour un recouvrement arrière sont effacées. Les informations à effacer sont les messages journalisés et les points de reprise constitués. Un **message** m reçu par le programme séquentiel 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 [Stro88]:

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

avec le prédicat $\texttt{demandé}(m)$ évalué à "vrai" ssi m est demandé. 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 [Stro88]:

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

Strom et *al.* démontrent qu'un message ou un point de reprise qui ne sont plus demandés ne le seront plus jamais. Ils peuvent être effacés.

2.3 Les objectifs

Les premiers critères de choix d'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 l'efficience 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.

Maintenant, le recouvrement arrière est étudié à partir de réalisations concrètes sur réseaux de stations de travail utilisant le système d'exploitation `UNIX`. Aussi, les expressions "applications (réparties)" et "processus (mono-activité)" sont préférées à "programmes répartis" et "programmes séquentiels".

2.3.1 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 avec un effet domino borné.

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 [Bhar88].

En outre, à cause de la synchronisation (une entité contrôle le lancement et la terminaison de l'algorithme de constitution), les mécanismes de coupures cohérentes 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.

2.3.2 Les surcoûts pendant l'exécution

Les **surcoûts** pendant l'exécution sont de trois ordres [Deco93] : **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. Depuis longtemps, le nombre de messages sert à mesurer l'efficience des algorithmes répartis. Avec les nouvelles technologies de réseaux - `ATM`, etc -, certains pensent que cette métrique perd de son importance [Elno95]. Toutefois, avec l'utilisation grandissante des réseaux étendus, nous estimons que le coût des messages [Gray88] reste important pour l'évaluation des algorithmes répartis. L'étude de politiques de recouvrement arrière adaptées aux réseaux étendus est d'ailleurs une voie de recherche prometteuse (*cf.* section 2.6).

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 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 ré-exécuter. Remarquons que les politiques inefficaces sont en plus inefficientes.

Tous les mécanismes de base possèdent un temps d'exécution propre. Dans les systèmes répartis, les processeurs, le réseau de communication et les mémoires sont partagés. 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 [Plan93] :

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

2.3.3 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. Le recouvrement arrière pour les systèmes multi-processeurs est d'ailleurs sujet à de nombreuses études (*cf.* section 2.6). Cependant, rappelons que notre modèle de système réparti est constitué de nœuds mono-processeurs. Inéluctablement, 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 stable. Cette inhibition est encore plus forte si l'application doit construire une ligne de reprise cohérente à chaque fois. En conséquence, pour les applications p-déterministes communiquant "fréquemment" avec le monde extérieur, les politiques "souhaitables" incluent un mécanisme de journalisation.

En conclusion, jusqu'à présent, la tolérance aux fautes par recouvrement arrière ne garantit pas un temps de réponse de l'application borné. Cette affirmation limite évidemment considérablement le champ d'application de la méthode.

2.3.4 La quantité de travail à défaire ou à refaire

En dernier lieu, l'efficience des mécanismes de base est évaluée en terme de quantité de travail à refaire. La **quantité de travail à refaire** se décline selon trois métriques : la distance de recouvrement arrière, le nombre de processus se ré-exécutant et le nombre de recouvrements arrière d'un 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 ne bornant pas l'effet domino sont considérées inefficientes.

Deuxièmement, le nombre de processus devant se ré-exécuter varie selon les applications et les politiques de recouvrement arrière. Les processus des applications indéterministes se ré-exécutent généralement tous à chaque défaillance. Pour les applications p-déterministes, les mécanismes de journalisation optimistes peuvent laisser apparaître des processus orphelins (*cf.* section 2.5). Certains préfèrent donc choisir une journalisation pessimiste, limitant ainsi le nombre des ré-exécutions aux processus des nœuds défaillants.

Troisièmement, certaines politiques peuvent obliger un même processus à se ré-exécuter plusieurs fois pour une même défaillance. 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 [Stro85]. Ces politiques inefficientes sont par conséquent déconseillées.

2.4 Les politiques

Dans cette section, nous faisons la synthèse des sections précédentes. Le but est de proposer une panoplie de politiques de recouvrement en arrière ainsi que des règles pour les choisir. Nous écartons explicitement les politiques qui nous semblent inefficaces ou inefficientes. Les règles de choix sont divisées en trois catégories.

Premièrement, certains des mécanismes de base s'excluent mutuellement. La sous-section 2.4.1 présente les agencements permis ou politiques "possibles".

Deuxièmement, le type de l'application (p-déterministe ou indéterministe) conditionne la politique de recouvrement arrière. La sous-section 2.4.2 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 2.3. La sous-section 2.4.3 effectue donc un tri parmi les politiques "possibles" pour ne garder que celles qui sont "souhaitables". Ce dernier point concerne principalement les applications p-déterministes.

Nous résumons les résultats par le tableau 2.1 où ne figurent que les politiques "possibles" et "souhaitables".

Enfin, la sous-section 2.4.4 montre comment structurer une application en ensemble de processus avec chacun une politique différente.

2.4.1 Les politiques possibles

Tout d'abord, nous classons les mécanismes de base existant dans la littérature et donnons une présentation en une ligne de chacune. La présentation succincte repose sur les concepts de base introduits en section 2.2. Le lecteur est invité à se reporter à la section 2.5 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'une coupure cohérente* : construction synchronisée d'une coupure cohérente.
- C2 : *constitution d'une barrière de synchronisation* : constitution synchronisée par arrêt global de l'application avec vidage des canaux de communication pour que les points de reprise soient tous utiles.
- C3 : *constitution de points de reprise non coordonnés* : constitution non synchronisée de points de reprise et sans relation particulière avec celles des autres processus.
- C4 : *constitution de points de reprise adaptatif* : constitution non synchronisée d'un point de reprise utile lorsque cela est estimé nécessaire.

Les mécanismes de journalisation sont divisés en quatre classes :

- J1 : *journalisation des messages manquants d'une coupure* : journalisation en mémoire stable des messages manquants traversant la coupure, donc limitée à la durée de construction de la coupure.
- J2 : *journalisation pessimiste* : journalisation en mémoire stable de tous les messages avant que les processus récepteurs effectuent l'action `recevoir`.
- J3 : *journalisation optimiste* : journalisation en mémoire stable de tous les messages, mais non synchronisée avec les réceptions.
- J4 : *journalisation répartie sur l'application* (en anglais, *family-based*) : jounalisation de tous les messages, non synchronisée avec les réceptions, répartie sur toute l'application, voire uniquement en mémoire volatile.

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 ré-exécution par un algorithme, centralisé ou réparti, 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 ré-exécution par un algorithme de calcul de l'état global recouvrable maximal.

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 coupure cohérente nécessite un mécanisme de journalisation des messages manquants pour limiter l'effet domino :

$$\mathsf{C1} \Rightarrow \mathsf{J1} \vee \mathsf{J2} \vee \mathsf{J3} \vee \mathsf{J4}.$$

Le mécanisme de barrière de synchronisation ne demande aucune journalisation :

$$\mathsf{C2} \Rightarrow \mathsf{J0} \vee \mathsf{J1} \vee \mathsf{J2} \vee \mathsf{J3} \vee \mathsf{J4}.$$

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

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

Le mécanisme de constitution de points de reprise adaptatif oblige la journalisation de certains messages pour limiter l'effet domino. En outre, les instants de journalisation ne sont pas les mêmes pour tous les processus, contrairement à ce qui est supposé par la journalisation des messages manquants d'une coupure :

$$\mathsf{C4} \Rightarrow \mathsf{J2} \vee \mathsf{J3} \vee \mathsf{J4}.$$

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 ré-exécutent. Nous avons donc la relation suivante :

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

2.4.2 La prise en compte de l'indéterminisme

La ré-exécution d'une application dépend de l'état global cohérent de reprise plus des messages manquants traversant la coupure correspondant à la ligne de reprise. Or, une application indéterministe ne garantit pas la ré-exécution des réceptions des messages manquants. D'où, aucun message manquant ne doit traverser les coupures correspondant aux lignes de reprise. Par conséquent, seules des constitutions de points de reprise utiles sont autorisées :

$$\neg\texttt{déterministe}(\widehat{P}) \Rightarrow \mathsf{C2} \vee \mathsf{C4}.$$

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

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 aussi construire des coupures cohérentes. Cela signifie que l'application est supposée ne pas posséder un comportement indéterministe pendant la durée de la constitution des points de reprise. En supposant qu'il soit possible de définir l'ensemble des actions indéterministes, nous considérons donc que C1 (dans ce cas noté C1*) est aussi possible :

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

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 : p-déterministes ou indéterministes (*cf.* section 2.6) [John90a, John91].

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

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

Lorsque survient une défaillance, tous les processus dépendant des processus des nœuds défaillants, se ré-exécutent. La ré-exécution n'est pas équivalente. Enfin, les politiques peuvent être classées par ordre décroissant d'efficience. La politique la plus efficiente est celle qui ne synchronise pas les constitutions de points de reprise : C4 ∧ J0 ∧ R2. Ensuite, vient celle qui limite l'inhibition due à la synchronisation des points de reprise : C1*∧J1∧R2. Puis, arrive celle qui arrête les communications de l'application : C2∧J0∧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 2.5.3 que le calcul du prédicat `utile` s'effectuant à chaque réception de messages 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.

2.4.3 Les politiques souhaitables

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

Pour des applications p-déterministes, il est clair qu'il est inefficace de journaliser tous les messages pour ensuite privilégier une ré-exécution 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 [Kaas92]. 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

de politiques. L'une choisit des constitutions de points de reprise non coordonnés avec la journalisation de tous les messages :

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

avec le prédicat `appli_grande` évalué à "vrai" ssi l'application est grande. L'autre catégorie privilégie les coupures cohérentes ou les constitutions de points de reprise utiles avec la journalisation de quelques messages :

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

avec le prédicat `appli_comm_intenses` évalué à "vrai" ssi les communications sont intenses. Il est clair que ces deux catégories sont différentes. 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 dans la catégorie. C'est un compromis entre l'efficience 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 ré-exécutions courtes [Huan95]. Cette politique ne nécessite pas le calcul de l'état global recouvrable maximal avant le début de la ré-exécution : $\mathsf{C3} \wedge \mathsf{J2} \wedge \mathsf{R2}$. 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 ré-exécutions, cette politique demande le calcul de l'état global recouvrable maximal avant le début de la ré-exécution : $\mathsf{C3} \wedge \mathsf{J3} \wedge \mathsf{R1}$. La journalisation répartie sur l'application combine les avantages des deux journalisations précédentes. Alvisi et Marzullo montrent quelle est optimale [Alvi94]. Cette politique 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 2.1 présente en première colonne les types d'application, ensuite, les politiques possibles et souhaitables, puis, les types de ré-exécutions.

2.4.4 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.
- 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.

Types de l'application $\widehat{P}$	**Politiques possibles et souhaitables**	**Types de ré-exécution**
$\neg$ déterministe($\widehat{P}$)	C4 $\wedge$ J0 $\wedge$ R2	$EX^{\widehat{s},t}_{\widehat{P}_{TF}} \not\equiv NEX^{\widehat{s},t}_{\widehat{P}_{TF}}$
	C1* $\wedge$ J1 $\wedge$ R2	
	C2 $\wedge$ J0 $\wedge$ R2	
déterministe($\widehat{P}$) $\wedge$ appli_comm_intenses($\widehat{P}$)	C4 $\wedge$ J0 $\wedge$ R2	$EX^{\widehat{s},t}_{\widehat{P}_{TF}} \equiv NEX^{\widehat{s},t}_{\widehat{P}_{TF}}$
	C1 $\wedge$ J1 $\wedge$ R2	
	C2 $\wedge$ J0 $\wedge$ R2	
déterministe($\widehat{P}$) $\wedge$ appli_grande($\widehat{P}$)	C3 $\wedge$ J4 $\wedge$ R2	$EX^{\widehat{s},t}_{\widehat{P}_{TF}} \not\equiv NEX^{\widehat{s},t}_{\widehat{P}_{TF}}$
	C3 $\wedge$ J3 $\wedge$ R1	
	C3 $\wedge$ J2 $\wedge$ R2	

TAB. 2.1 – Les politiques de recouvrement arrière possibles et souhaitables.

– 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 [Sist89]. 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 [Lowr91]. 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. Si leur mécanisme de journalisation est optimiste, la validation d'un état nécessite un protocole entre passerelles.

2.5 Les mécanismes

Dans cette section, les trois mécanismes de base du recouvrement arrière sont présentés en détail. Auparavant, nous ouvrons une parenthèse sur les pré-requis du recouvrement arrière.

Chacune des sous-sections se divise en deux parties. La première partie présente l'aspect algorithmique et la deuxième l'aspect réalisation. Pour chaque mécanisme de base, nous terminons la présentation des algorithmes avec un tableau rassemblant les références bibliographiques regroupées par classes. Nous n'hésitons pas à donner des détails d'implantation afin de sensibiliser le lecteur aux difficultés de réalisation du recouvrement arrière.

2.5.1 Les pré-requis

Nous ouvrons une parenthèse pour, d'une part, introduire la terminologie des mécanismes d'horloge globale commune, et d'autre part, mettre l'accent sur deux pré-requis importants qui ne sont en général pas réalisés par le système d'exploitation.

2.5.1.1 Les pré-requis pour l'algorithmique

Les pré-requis pour l'algorithmique sont ceux développés dans la sous-section 1.2.5 : la mémoire stable, la gestion de groupe, la diffusion et l'horloge globale commune. Parmi ces abstractions, une d'entre-elle (l'horloge globale commune) demande que nous nous y attardions un peu.

Dans la plupart des algorithmes, l'horloge globale commune est logique. Seules quelques algorithmes de coupures cohérentes reposent sur une horloge globale commune physique [Tong89, Cris91b]. L'horloge globale logique est réalisée en estampillant les messages interprocessus. Pratiquement, l'action `émettre` insère une horloge logique au message et l'action `recevoir` met à jour sa connaissance de l'horloge globale commune à partir de l'estampille du message reçu.

Le temps logique peut être exprimé par un scalaire ou bien par un vecteur ou encore par une matrice. Les estampilles sont alors des scalaires ou bien des vecteurs ou encore des matrices. Plus généralement, les mécanismes d'horlogerie servent à exprimer la causalité

entre différentes actions du système réparti. Ils sont donc utilisés pour la réalisation de toutes les relations de précédence causale : $\prec_{\mathbf{A}}$, $\prec_f$, $\prec_F$ et $\prec_{\mathbf{I}}$. Pour un développement plus complet de ces mécanismes, le lecteur est invité à se référer aux publications suivantes : [Fidg91, Schw92, From94, Jard94, Rayn95].

Pour les estampilles contenant un scalaire, leur taille est très petite et fixe, le calcul de l'horloge rapide, le nombre maximum d'actions repérables petit et la précision de l'horloge très faible. À l'opposé, pour les estampilles contenant une matrice, leur taille est grande et fonction du nombre de processus, le calcul de l'horloge long, le nombre maximum d'actions repérables grand et la précision de l'horloge très grande. La méthode la plus employée utilise un vecteur pour repérer le temps logique. Si l'estampille contient le vecteur, les dépendances sont dites *transitives* :

$$\texttt{recevoir}(m) \Rightarrow \forall i \leq \mid \widehat{\mathbf{P}}_{TF} \mid, VDT[i] = \texttt{max}(VDT[i], V[i]),$$

avec VDT le vecteur des dépendances transitives du processus récepteur et V le vecteur de l'estampille de m. Si l'estampille contient un scalaire, les dépendances sont dites *directes* :

$$\texttt{recevoir}(m) \Rightarrow VDD[i] = \texttt{max}(VDD[i], V),$$

avec P_i le processus émetteur du message m, VDD le vecteur des dépendances directes du processus récepteur et V le scalaire de l'estampille de m.

2.5.1.2 Les pré-requis pour l'implantation

Le modèle du système réparti a déjà introduit deux pré-requis particuliers à la tolérance aux fautes : une mémoire stable et un service de communication fiable. Il faut y ajouter d'autres pré-requis, communs aux mécanismes de migration. Ces derniers sont rassemblés sous la propriété de "transparence". Le but est de fournir un "système à image unique" [Thei85, Doug91, Thie91]. Nous citons ici les deux pré-requis les moins bien réalisés par les systèmes d'exploitation actuels:

1. *transparence du nommage* : c'est l'une des conditions du déterminisme d'exécution (*cf.* sous-section 2.2.1).

2. *transparence de la localisation* : le nom d'un objet ne contient pas le nom du nœud qui le contient ou bien, si l'objet migre, un service de localisation repère automatiquement le nom du nouveau nœud.

L'implantation est d'autant plus compliquée que le système d'exploitation ne réalise pas la transparence. Par exemple, dans le système d'exploitation `Unix`, un canal de communication (`socket`) porte un nom local. Il est donc nécessaire de garder les noms des canaux ouverts lors de l'exécution pour les ré-ouvrir avant le début de la ré-exécution. Une première solution relativement simple n'est que partiellement satisfaisante. Les informations concernant un `socket` sont gérées dans un tableau de taille fixe. À chaque reconnexion, la case du tableau utilisée par l'ancien `socket` n'est pas récupérée. Elle est ignorée et le nouveau `socket` utilise une autre entrée. La conséquence est que le nombre de ré-exécutions devient limité. La solution "puriste" consistant à récupérer l'ancien socket est nettement plus compliquée.

Pour le système d'exploitation `Chorus/MiX`, les noms des canaux (`ports`) sont uniques et un service de localisation automatique est disponible. Ainsi, la reconnexion est effectuée dès que le `port` est reconstruit et re-déclaré au système d'exploitation du nœud de ré-exécution.

2.5.2 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. Le premier objectif est de faire en sorte que les points de reprise participent à une ligne de reprise cohérente ou précèdent un état global recouvrable, afin d'éviter l'effet domino.

Dans la deuxième partie de cette sous-section, nous présentons deux problèmes et une liste d'optimisations d'implantation. Les problèmes sont la définition et la détermination de l'instant de saisie de l'état d'un processus. Quant aux optimisations, elles sont importantes car les constitutions de points de reprise ralentissent l'exécution.

2.5.2.1 L'algorithmique

C1 : La constitution d'une coupure cohérente

Le principe des algorithmes de la première classe est de construire une coupure globale cohérente et d'enregistrer les messages traversant la coupure. [Hela93] est une synthèse des algorithmes de coupures cohérentes.

Le premier algorithme de coupure cohérente a été donné par Chandy et Lamport [Chan85]. 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 la coupure 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 manquants 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 - excepté l'initiateur - pendant l'exécution de l'algorithme. Elle consiste à transformer les points de reprise effectifs en points de reprise permanents [Koo87]. Pour tolérer les fautes de l'initiateur, l'algorithme est à trois phases.

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 la coupure.

Les algorithmes de cette classe qui tolèrent le non-ordonnancement des transmissions de messages sont nombreux [Lai87, Li87, Ahuj93].

C2 : La constitution d'une barrière de synchronisation

Le principe de ces algorithmes est d'arrêter les communications, ensuite, d'attendre que tous les messages soient reçus, puis, de sauvegarder les états locaux [Bari83]. Puisque toutes les communications interprocessus sont arrêtées et tous les messages reçus, tous les canaux sont vides. Donc, la coupure est cohérente et aucun message manquant ne la traverse. Les points de reprise sont donc utiles. 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 plus des messages demandés émis avant la constitution [Stro88]. 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 [Stro85]. Le calcul et l'enregistrement du contenu des messages demandés au moment des constitutions est plus efficient 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 adaptatifs

Le principe de ces algorithmes est le calcul exact ou l'approximation du prédicat `utile`, pour la détermination des instants des constitutions de points de reprise.

Le seul algorithme de calcul exact du prédicat `utile` que nous connaissons est donné par Baldoni et *al.* [Bald95a]. Il suppose que les communications sont FIFO. Le principe est le repérage des chemins zigzagants [Xu93]. Baldoni et *al.* distinguent deux types de chemins zigzagants : causaux et non causaux.

Dans la figure 2.3, nous traçons l'exécution d'une application composée de trois processus. Toutes les actions internes sont des constitutions de points de reprise. La figure 2.4 représente le graphe correspondant des précédences causales des intervalles de constitution de points de reprise.

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 une constitution de point de reprise de P_1 avant $\mathcal{A}_1^3$. Le point de reprise $\mathcal{A}_3^2$ reste alors utile.

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 une constitution de point de reprise de P_2 avant $\mathcal{A}_2^5$. Le point de reprise $\mathcal{A}_3^6$ reste alors utile.

Pour le calcul des chemins zigzagants causaux, les auteurs introduisent deux vecteurs $\texttt{pr\_courant}_i$ et $\texttt{simple}_i$ maintenus par tout $P_i \in \hat{\mathbf{P}}_{TF}$. $\texttt{pr\_courant}_i$ repère les derniers points de reprise connus de P_i. Ce vecteur est ajouté en estampille de chaque message émis par P_i. Le vecteur de l'estampille est noté $\texttt{pr\_courant}_m$. $\texttt{simple}_i$ permet de savoir si le chemin zigzagant comprend des points de reprise. En d'autres termes, un point de reprise est inclus entre la réception d'un des messages et l'émission du message suivant dans le chemin. Un chemin $I_i^x \prec_{\mathbf{I}} I_j^y$ est dit simple ssi la relation suivante n'est pas vérifiée : $\exists (k, z)$: $(I_i^x \prec_{\mathbf{I}} I_k^z) \wedge I_k^z \prec_{\mathbf{I}} I_j^y)$. P_i maintient le vecteur de booléens $\texttt{simple}_i$ tel que $\texttt{simple}_i[j]$ est évalué à vrai ssi, à la connaissance de P_i, le chemin zigzagant $I_j^{\texttt{pr\_courant}_i[j]} \prec_{\mathbf{I}} I_i^{\texttt{pr\_courant}_i[i]}$ est simple. Autrement dit, le chemin zigzagant issu de $I_j^{\texttt{pr\_courant}_i[j]}$ et se terminant en $I_i^{\texttt{pr\_courant}_i[i]}$ n'inclut pas de points de reprise. $\texttt{simple}_i$ est ajouté en estampille de chaque message émis par P_i. Le vecteur de l'estampille est noté $\texttt{simple}_m$.

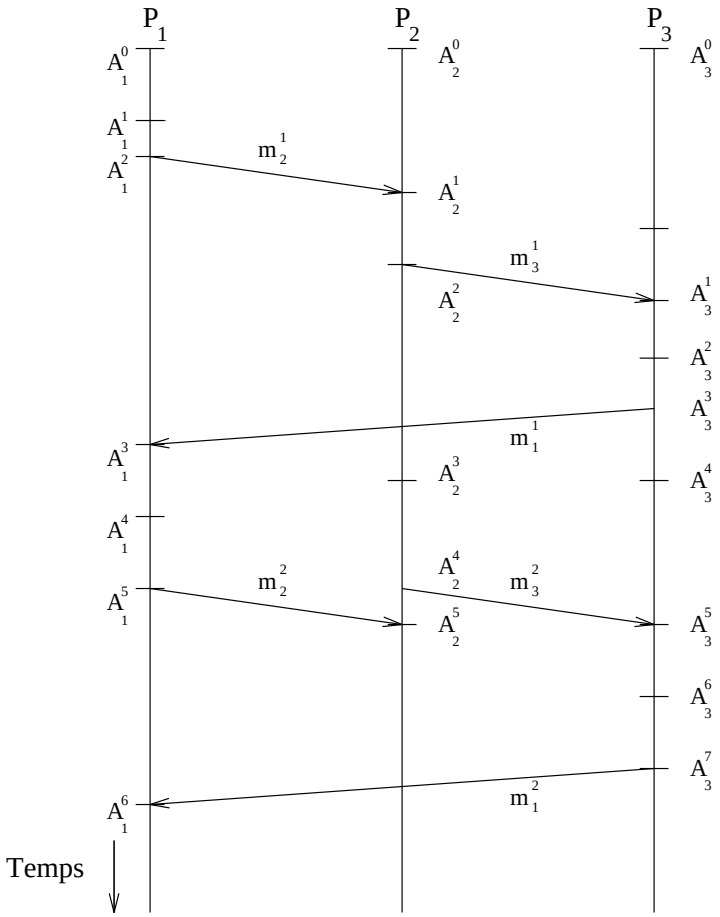


FIG. 2.3 – Les chemins zigzagants causal et non causal.

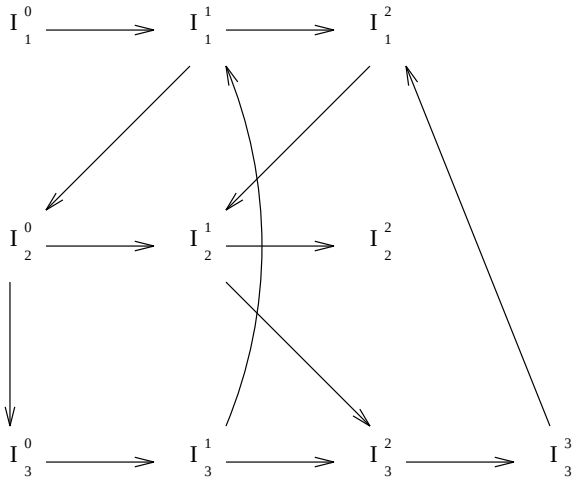


FIG. 2.4 – Le graphe de précédences causales correspondant à l'exécution de la figure 2.3.

Lorsqu'un message m arrive à P_i, P_i calcule le prédicat suivant :

$$\texttt{zigzag\_causal}(m) \stackrel{\text{def}}{=} ((\texttt{pr\_courant}_m[i] = \texttt{pr\_courant}_i[i]) \wedge \neg \texttt{simple}_m[i]),$$

avec le prédicat $\texttt{zigzag\_causal}(m)$ évalué à "vrai" ssi m provoque un chemin zigzagant causal. Par conséquent, P_i constitue un point de reprise à chaque fois que $\texttt{zigzag\_causal}(m)$ est évalué à "vrai".

Pour le calcul des chemins zigzagants non causaux, chaque processus maintient un autre vecteur $\texttt{émis\_vers}_i$ et une matrice $\texttt{causal}_i$. $\texttt{émis\_vers}_i$ est un vecteur de booléens indiquant si P_i a émis un message vers les autres processus depuis son dernier point de reprise. $\texttt{émis\_vers}_i$ est ajouté en estampille de chaque message émis par P_i. Le vecteur de l'estampille est noté $\texttt{émis\_vers}_m$. La matrice de booléens $\texttt{causal}_i$ est telle que $\texttt{causal}_i[x, y]$ est évalué à "vrai" ssi, à la connaissance de P_i, il existe un chemin zigzagant causal de P_x vers P_y depuis leur dernier point de reprise. $\texttt{causal}_i$ est ajouté en estampille de chaque message émis par P_i. Le vecteur de l'estampille est noté $\texttt{causal}_m$.

Lorsqu'un message m arrive à P_i, P_i calcule le prédicat suivant :

$\texttt{zigzag\_non\_causal}(m) \stackrel{\text{def}}{=}$

$\exists x : \texttt{émis\_vers}_i[x] \wedge (\exists y : (\texttt{pr\_courant}_m[y] > \texttt{pr\_courant}_i[y]) \wedge \neg \texttt{causal}_m[y, x])$,

avec le prédicat $\texttt{zigzag\_non\_causal}(m)$ évalué à "vrai" ssi m provoque un chemin zigzagant non causal non doublé par un chemin zigzagant causal. Par conséquent, P_i constitue un point de reprise à chaque fois que $\texttt{zigzag\_non\_causal}(m)$ est évalué à "vrai".

En conclusion, l'estampille des messages contient 3 vecteurs de taille $\mid \widehat{\mathbf{P}}_{TF} \mid$ et une matrice de taille $\mid \widehat{\mathbf{P}}_{TF} \mid \times \mid \widehat{\mathbf{P}}_{TF} \mid$. L'algorithme de calcul du prédicat $\texttt{utile}$ implique le parcours d'une matrice et l'algorithme de mise à jour des variables locales celui d'une matrice et d'un vecteur. Malheureusement, nous ne connaissons aucune implantation du calcul du prédicat $\texttt{utile}$. Nous ne pouvons donc pas conclure sur l'inhibition engendrée par ce calcul lors des réceptions de messages. La méthode est dite "adaptative" au sens où un processus peut choisir de retarder la constitution et laisser apparaître des points de reprise inutiles.

Dans [Xu93], Xu et Netzer proposent un algorithme avec uniquement l'évaluation du prédicat $\texttt{zigzag\_causal}(m)$. L'idée est de constituer des points de reprise en maximisant le nombre de points de reprise utiles. Le but est de minimiser le risque d'apparition de l'effet domino. Cette méthode est aussi utilisée par d'autres mécanismes [Wang92a]. Mais, parce qu'elles constituent beaucoup de points de reprise, nous estimons que ces méthodes sont inefficientes. Nous les avions d'ailleurs déjà qualifiées d'inefficaces.

Les références bibliographiques des mécanismes de constitution de points de reprise sont regroupées selon la classe de leur algorithme dans le tableau 2.2.

Classes	**Références bibliographiques**
C1	[Chan85], [Koo87], [Spez86], [Lai87], [Leu88], [Li87], [Venk87], [Venk89], [Tong89], [Cris91b], [Silv92], [Ahuj93], [Elno93], [Vaid94].
C2	[Bari83], [Kaas92], [Leon93].
C3	[Borg83] [Powe83], [Stro85], [John87], [Stro88], [Borg89], [John89], [Sist89], [Juan91], [Alvi93], [Sens95], [Wang95].
C4	[Bald95a].

TAB. 2.2 – Les mécanismes de constitution de points de reprise - références bibliographiques.

2.5.2.2 L'implantation

L'état d'un processus est composé de trois parties distinctes [Wang95] :

1. *l'état volatile* : ce sont les variables du processus présentes en mémoire virtuelle (*cf.* sous-section 2.2.1),
2. *l'état persistant* : fichiers en mémoire stable,
3. *l'état du système d'exploitation* : état d'exécution (registres,...), ensemble de fichiers ouverts (identificateur interne, position courante, blocs du disque en mémoire centrale du nœud,...), ensemble de messages en attente de réception et autres états du noyau (identificateurs des processus, répertoire courant, gestionnaires de signaux, statistiques, parenté,...) [Doug91].

Dans la plupart des cas, l'état persistant du processus ne fait pas partie de l'état du processus. Cela signifie que la mémoire stable n'appartient pas au monde de l'application. D'une manière générale, les interactions avec la mémoire stable restent une question insuffisamment traitée (*cf.* section 2.6). Certains mécanismes essaient de faire entrer l'état persistant dans l'état du processus. Plank autorise les accès aux fichiers en mode lecture seule ou en mode écriture seule et séquentielle [Plan93]. Dans [Sens94, Wang95], tous les modes d'accès sont permis, mais deux processus ne doivent pas accéder à un même fichier. En effet, cela reviendrait à une communication par fichiers partagés. Pour chaque point de reprise, pour chaque fichier auquel le processus a accédé depuis le début de l'exécution, est associée la version du fichier correspondant à l'état de reprise.

Pour la constitution des points de reprise, deux cas se présentent. Soit les parties de l'état du processus sont transférables, c.à-d. qu'elles seront significatives sur le nœud de ré-exécution. Soit elles ne possèdent qu'une signification locale, auquel cas elles doivent être ignorées avec le risque d'une ré-exécution non équivalente [11].

Les signaux sont une autre possibilité de communication sous `Unix`. La plupart des implantations ne les traitent pas. Borg et *al.* résolvent le problème en implantant les signaux par des messages en provenance du monde extérieur [Borg89].

Enfin, les liens de parenté sont un mécanisme propre à `Unix`. Pour garder les liens de parenté suite à une défaillance, il faut redémarrer tous les processus d'une même famille sur le même nœud [Borg89]. Ces liens entre processus n'ont pas vraiment de sens dans un contexte réparti. C'est pourquoi ils sont le plus souvent ignorés.

Le deuxième problème d'implantation est la détermination de l'instant de saisie de l'état du processus. Afin de respecter la règle d'atomicité des actions d'un processus (*cf.* § 2.1.2.1), l'état du processus est saisissable uniquement lorsque le processus s'exécute hors du noyau - hors appel système. Certes, cela simplifie aussi considérablement l'état du système d'exploitation. Il faut donc implanter un mécanisme de sentinelles [Russ93] : mémorisation de la demande de constitution et attente du retour de l'appel système en cours. Se pose alors le problème des appels bloquants non interruptibles tels l'action `recevoir`. Pour une coupure cohérente ou un algorithme adaptatif, le problème est que le dernier message reçu ne doit pas faire partie de l'état du processus. En outre, l'ordre de constitution peut être différé de manière inacceptable. Par conséquent, ce problème d'implantation

11. Notons que la migration de processus peut utiliser une troisième possibilité appelée la *dépendance résiduelle*. Les parties de l'état non transférables sont laissées sur le nœud d'origine et les accès ultérieurs à ces informations sont redirigés vers le nœud d'origine [Doug91].

complexe demande une analyse minutieuse de l'implantation du système d'exploitation utilisé pour trouver des solutions acceptables.

Dans la littérature, trois optimisations sont proposées pour diminuer la quantité d'information à enregistrer en mémoire stable. Ce sont les points de reprise incrémentaux, l'exclusion de certaines zones de la mémoire virtuelle et la compression.

Le principe des constitutions de points de reprise incrémentaux est d'abord de sauvegarder en mémoire stable une copie complète de l'espace virtuel du processus. Ensuite, il suffit d'enregistrer uniquement les pages modifiées ou créées [John89, Elno93, Sens94]. Les points de reprise sont régulièrement fusionnés pour retrouver une copie complète de la mémoire virtuelle [Plan95].

Le principe de l'exclusion de certaines zones de la mémoire virtuelle consiste à laisser l'utilisateur spécifier des zones de mémoire virtuelle à ne pas sauvegarder [Plan95]. L'idée sous-jacente est de calculer ces zones de mémoire pendant la compilation, avec placement automatique d'ordre de constitutions (dites statiques) dans le code généré [Li90a, Long92]. Par conséquent, cette optimisation n'est pas intéressante pour les mécanismes que nous étudions, car nous ne désirons pas modifier le compilateur.

L'espace virtuel pourrait être compressé en mémoire centrale du nœud avant d'être enregistré en mémoire stable. Pour les systèmes répartis faiblement couplés, il est établi que la compression logicielle n'est pas avantageuse [Li90a, Elno93]. D'autres études indiquent qu'elle doit être effectuée par un composant matériel dédié [Burr92, Doug93].

Par ailleurs, la littérature répertorie trois optimisations améliorant la concomitance du mécanisme de sauvegarde de la mémoire virtuelle du processus avec l'exécution de l'application. Ce sont la précopie, la copie sur écriture et la copie en mémoire centrale.

La pré-copie consiste à sauvegarder toutes les pages de la mémoire virtuelle dans une première phase. Ensuite, itérativement, le mécanisme enregistre à nouveau en mémoire stable les pages modifiées depuis le début de la phase précédente. Enfin, lorsque le nombre de pages modifiées est en dessous d'un seuil fixé, le mécanisme les enregistre une dernière fois en une seule opération [John89].

La copie sur écriture consiste à protéger des écritures toutes les pages de l'espace virtuel du processus et à les rendre à nouveau accessibles une fois enregistrées en mémoire stable [Li90b, Elno93]. Cette optimisation est toujours associée à la suivante. Elnozahy et *al.* montrent que cette technique est toujours plus efficace que la pré-copie [Elno92a].

La copie en mémoire centrale consiste à introduire une copie des pages en mémoire centrale du nœud avant l'enregistrement en mémoire stable [Li90b]. Une zone de mémoire centrale est spécifiquement réservée pour ces copies. L'idée vient de la constatation que le temps d'accès à la mémoire stable est très supérieur au temps d'accès à la mémoire centrale. Une fois copiées, le processus peut continuer à accéder normalement à ces pages. La copie en mémoire centrale peut être employée par les techniques de pré-copie et de copie sur écriture.

2.5.3 La journalisation de l'exécution répartie

Le mécanisme de journalisation sauvegarde l'histoire de l'application pendant l'exécution. Seules les classes des mécanismes de journalisation pessimistes, optimistes et répartis sont détaillées. La journalisation des messages manquants a déjà été expliquée avec les algorithmes de coupures cohérentes (*cf.* section 2.5.2).

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 2.3 rassemble les références bibliographiques regroupées par classes.

2.5.3.1 L'algorithmique

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 [Powe83]. 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 messages.

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 [Stro85]. Plus tard, les auteurs montrent que seule l'histoire des messages a besoin d'être enregistrée en mémoire stable [Stro88]. 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 ré-émis lors de la ré-exécution [Juan91]. Enfin, tous les algorithmes de cette classe effectuent la journalisation 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 : répartis et centralisés.

Dans [John89], 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 [John90b]. D'où, 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 répartis dans [Stro85]. 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 transitives sont ajoutés en estampille des messages. Deuxièmement, il

s'ensuit que la validation d'un message consiste à demander les vecteurs de journalisation aux autres processus. 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 [Sist89]. Les algorithmes répartis sont naturellement tolérants aux fautes, car la faute d'un processus entraîne la ré-exécution de l'algorithme pour le calcul de l'état global recouvrable maximal (*cf.* sous-section 2.5.4).

J4 : répartie

Le principe de la journalisation répartie est de diffuser l'histoire séquentielle d'un processus sur les nœuds où s'exécutent 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. D'où, P_i transmet l'histoire de m en estampille des messages émis après la réception de m. C'est pourquoi ce type de journalisation est aussi appelé journalisation causale [Alvi94].

Le premier algorithme de cette classe a été proposé par Elnozahy et Zwaenepoel [Elno92c]. Les processus construisent un "graphe d'antécédence" - graphe de précédence. Localement, pour un processus, il contient l'histoire répartie dont il 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. Cet algorithme est donc naturellement tolérant aux fautes.

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. f est égal à $\mid \widehat{\mathbf{P}}_{TF} \mid$ pour l'algorithme précédemment décrit. 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 sites. Les auteurs dérivent plusieurs algorithmes de l'algorithme générique. Les processus sont supposés s'exécuter chacun sur un nœud différent et les communications 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 [John87, Alvi93] sont vus comme des instances de l'algorithme générique avec f égal à 1.

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 2.3.

Classes	**Références bibliographiques**
J2	[Borg83], [Powe83], [Borg89], [Sens95], [Wang95], [Lai87].
J3	[Stro85], [John89], [Sist89], [Stro88], [John93], [Juan91].
J4	[Alvi93], [Elno92c], [Elno93], [Alvi94].

TAB. 2.3 – Les mécanismes de journalisation - références bibliographiques.

2.5.4 Le recouvrement arrière

À la suite de défaillances, le mécanisme de recouvrement arrière détermine l'ensemble des processus devant se ré-exécuter. Puis, il récupère les informations de reprise, et enfin, lance la ré-exécution. Les algorithmes de recouvrement arrière ne présentent pas de difficulté particulière mise à part la connaissance des processus non défaillants. Ce dernier point dépasse le cadre de notre étude, nous le laissons donc de côté.

Nous terminons cette sous-section avec le problème du bouclage infini des recouvrements arrière. Nous considérons que c'est un problème dépendant de l'implantation.

2.5.4.1 L'algorithmique

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 ré-exécuter. Puis, il 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, il sont très proches des algorithmes de validation. Aussi, nous laissons le lecteur se reporter aux références bibliographiques citées au paragraphe 2.5.3.1.

2.5.4.2 L'implantation

Koo et Toueg démontrent que des messages orphelins peuvent provoquer un bouclage infini des recouvrements arrière [Koo87]. Le modèle DBI autorise qu'un message émis avant l'exécution de l'algorithme de recouvrement arrière ne soit reçu qu'après le début de la ré-exécution. Ce message est alors orphelin si l'émetteur est parmi les processus défaillants. D'où, le processus récepteur est orphelin et se ré-exécute. 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 c'est un problème d'implantation plutôt qu'algorithmique. Effectivement, cela est possible uniquement si les canaux de communication survivent aux défaillances. Dans ce cas, un numéro d'incarnation [Stro85] est ajouté en estampille des messages. Le principe est de marquer les messages avec le nombre de ré-exécutions commencées. Dès lors, un processus connaissant les nouveaux numéros de ré-exécutions 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 ré-exécuter [Koo87].

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.

2.6 Les voies de recherche prometteuses

Tout au long de ce chapitre, nous avons identifié des questions non résolues. Dans cette section, nous en faisons la synthèse. En outre, nous ouvrons le recouvrement arrière vers de nouvelles voies de recherche que nous estimons prometteuses.

La discussion s'organise autour des modèles de système réparti (*cf.* sous-section 2.6.1), d'application (*cf.* sous-section 2.6.2) et de faute (*cf.* sous-section 2.6.3).

2.6.1 Le modèle de système réparti

Architecture

Le recouvrement arrière est aussi examiné pour les multiprocesseurs et les multicalculateurs. Des mécanismes ont déjà été conçus pour ces systèmes et pour différents modèles de mémoire[12] :

- UMA (en anglais, Uniform-Memory-Access) [Bern88],
- NUMA (en anglais, Non-Uniform-Memory-Access) [Ahme90, Wu90b, Bana93, Plan93, Jans94],
- COMA (en anglais, Cache-Only-Memory-Architecture) [Bana94] et
- NORMA (en anglais, NO-Remote-Memory-Access). [Wu90a, Plan93].

Ces mécanismes de recouvrement arrière sont très différents les uns des autres. Cependant, 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, apparaissent les interactions par mémoire partagée. Il existe alors deux voies de recherche. Soit un acteur constitue une unité de reprise à comportement indéterministe. Soit chaque activité constitue une unité de reprise et les interactions par mémoire partagée sont traquées.

Parmi les systèmes supportant les acteurs multi-activités, les systèmes à micro-noyaux tels que `Mach` [Blac92] et `Chorus` [Rozi88, Rozi92] sont intéressants. En effet, ces systèmes sont constitués de serveurs communicants par échanges de messages. Rashid et *al.* montrent que, pour ces systèmes, les mécanismes de mémoire partagée sont réalisables par échanges de messages, et inversement [Rash87]. Il est alors envisageable d'implanter la mémoire partagée par des échanges de messages.

Un dernier 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.

Réseaux de communication

Le domaine des réseaux est sans doute le domaine le plus en mutation. Les mécanismes du recouvrement arrière sont à revoir pour les réseaux à hauts débits, les systèmes mobiles et les réseaux étendus.

12. *cf.* [Hwan93b] pour une définition précise des modèles de mémoire.

Avec les réseaux à hauts débits, le coût des communications dans un même réseau local est grossièrement divisé par un facteur de 10. Par conséquent, le compromis entre le temps de calcul et le nombre de messages est à ré-évaluer.

Parmi les nouveaux utilisateurs de réseaux de communication, il y a les systèmes mobiles. Le premier problème à résoudre est la définition de ce qu'est une faute. Ensuite, comme leur nom l'indique, les systèmes mobiles changent de lieu de connexion. Ainsi, la reconfiguration et la récupération en sont compliquées. L'utilisation de la mémoire stable et la récupération d'information chez les autres processus de l'application doivent être repensées.

Enfin, les connexions entre réseaux locaux puis nationaux et enfin continentaux deviennent de plus en plus performantes. Naturellement, les utilisateurs conçoivent des applications se diffusant sur toujours plus de nœuds toujours plus éloignés. Les questions qui sont d'actualité pour les réseaux à hauts débits et les systèmes mobiles, le sont aussi pour les réseaux étendus.

2.6.2 Le modèle d'application

Dans la littérature, le recouvrement arrière est aussi examiné pour le modèle des variables partagées [Wu90a, Jans93, Saty94] et le modèle orienté objets [Shri85, Dixo87, Lin90]. Chaque modèle cible un ensemble de catégories d'application. Il n'est donc pas forcément 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 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 [Manc89, Wang92a, Leon94, Bald95a, Brze95, Bald95b]. 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.

2.6.3 Le modèle de fautes

Comme nous l'avons déjà indiqué, 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.

Chapitre 3

Un mécanisme global efficace et efficient et ses extensions

Comme le montre le chapitre précédent, toutes les politiques de recouvrement arrière ne sont pas efficaces (possibles) ni efficientes (souhaitables). Nous choisissons de construire un mécanisme global[1] efficace et efficient supportant les applications p-déterministes et transparent (simple) pour l'utilisateur. Ce mécanisme global simple comprend les mécanismes de base, intégrant leurs optimisations les plus efficientes dans le modèle de système réparti (*cf.* section 2.1). En outre, il est assez souple pour permettre l'ajout d'extensions. Ces extensions autorisent le support d'applications composées de processus p-déterministes et de processus indéterministes, ceci en contrepartie de la programmation par l'utilisateur de la tolérance aux fautes.

La première section de ce chapitre motive la décomposition du mécanisme de recouvrement arrière en un mécanisme global et des extensions que nous ajoutons au mécanisme global. (*cf.* section 3.1). Ensuite, la section 3.2 développe le mécanisme global et la section 3.3 les extensions choisies. Enfin, la section 3.4 conclut le chapitre.

1. L'expression "mécanisme de base" serait ici peut-être plus appropriée, mais nous l'employons déjà pour parler des mécanismes de constitution de points de reprise, de journalisation et du recouvrement arrière.

3.1 La décomposition du mécanisme de recouvrement arrière en un mécanisme global et des extensions

Les utilisateurs sont classés en deux catégories : ceux qui interviennent dans la programmation de la tolérance aux fautes de leurs applications et ceux qui ne souhaitent pas s'en occuper. En conséquence, nous décomposons le mécanisme de recouvrement arrière en un mécanisme global plus des extensions.

Le mécanisme global est un mécanisme *transparent* (ou *simple*) pour l'utilisateur. Nous montrons comment le modèle de système réparti simplifie la conception de l'architecture du mécanisme global. Les applications utilisant uniquement le mécanisme global sont supposées p-déterministes. Parmi les optimisations détaillées dans le chapitre 2 lors de la présentation des mécanismes de base, certaines nécessitent l'intervention de l'utilisateur. Nous qualifions ici ces optimisations de *complexes* et choisissons de ne pas les intégrer dans le mécanisme global.

Les extensions concernent la prise en compte de l'indéterminisme d'exécution et ne sont pas transparentes à l'utilisateur. Tout d'abord, il s'agit de la programmation de la sémantique des transmissions de messages. L'idée est de laisser l'utilisateur choisir les messages devant être rejoués lors d'une éventuelle ré-exécution. Ensuite, l'utilisateur regroupe les actions indéterministes dans certains processus qu'il déclare indéterministes et isole ceux-ci en programmant la sémantique des transmissions des messages échangés avec eux. Le résultat est une application composée d'un ensemble de processus p-déterministes et d'un ensemble de processus indéterministes.

3.2 Le mécanisme global

Le mécanisme global est conçu pour tolérer les fautes d'applications s'exécutant sur des réseaux locaux de stations de travail. Cet environnement d'exécution a été choisi parce qu'il est le plus courant. L'architecture du mécanisme bénéficie des propriétés de ce modèle et construit des unités de reprise réparties dimensionnées à la taille d'un réseau local (*cf.* sous-section 3.2.1).

Le mécanisme de constitution des points de reprise est décomposé en trois phases, afin notamment d'optimiser l'enregistrement en mémoire stable des informations de recouvrement. En outre, il peut construire des coupures cohérentes ou des points de reprise non coordonnés (*cf.* sous-section 3.2.3).

La journalisation pessimiste ou optimiste est centralisée par le gestionnaire de la tolérance aux fautes de l'unité de reprise répartie, avant l'enregistrement en mémoire stable (*cf.* sous-section 3.2.4).

Le mécanisme de recouvrement arrière est bloquant et calcule l'état global recouvrable maximal (*cf.* sous-section 3.2.5).

Les notations pour les algorithmes sont données dans la sous-section 3.2.2.

Seuls trois algorithmes sont détaillés dans ce chapitre. Il s'agit de l'assurance des constitutions de points de reprise, de la validation des messages à destination du monde extérieur et du calcul de l'état global recouvrable maximal. Les algorithmes des actions du mécanisme de base sont présentés dans l'annexe A.

3.2.1 L'architecture du mécanisme global

Le mécanisme global est directement étudié pour s'adapter aux systèmes répartis construits comme une agglomération de réseaux locaux. Pour ce faire, les processus s'exécutant sur les nœuds d'un même réseau local sont groupés pour former une unité de reprise répartie. Ainsi, les mécanismes de base bénéficient des abstractions disponibles dans un réseau local (*cf.* § 3.2.1.1). En définissant de manière précise le rôle de l'unité de reprise répartie, sa gestion en est nettement simplifiée (*cf.* § 3.2.1.2). Ces choix sont discutés dans un dernier paragraphe (*cf.* § 3.2.1.3).

3.2.1.1 Un mécanisme global pour réseaux locaux

Le recouvrement arrière est fondé sur le concept de processus fiable. Le processus fiable est construit à partir de quatre abstractions du système d'exploitation : la mémoire stable, la gestion de groupe, la diffusion et l'horloge globale commune.

Or, les systèmes d'exploitation répartis pour réseaux locaux fournissent déjà une partie de ces abstractions. Les systèmes de gestion répartie des fichiers tels que AFS [Saty90a], LFS [Rose91], NFS [Sand85] implantent une mémoire secondaire accessible à tous les nœuds du réseau local. La fiabilité de cette mémoire secondaire peut être renforcée en répliquant les serveurs de fichiers. Un mécanisme de diffusion non fiable vers tous les nœuds du réseau local est généralement disponible. Si nécessaire, un protocole fiable peut être construit à partir du mécanisme de base.

En conséquence, pour gérer les processus s'exécutant sur les nœuds d'un même réseau local, le mécanisme de recouvrement arrière peut bénéficier des abstractions déjà fournies par le système d'exploitation réparti. Il y manque cependant la gestion de groupe et l'horloge globale commune.

En outre, la tolérance aux fautes d'une application peut se gérer en groupes de processus. Les processus sont alors appelés des *unités de reprise* et les groupes de processus des *unités de reprise réparties* (URR) [Vaid93b]. Deux processus appartenant à la même URR sont dits *voisins*. Deux processus n'appartenant pas à la même URR sont dits *distants*.

Par conséquent, nous considérons que le réseau local constitue une URR fondamentale intéressante. Tous les processus d'une application s'exécutant sur les nœuds d'un réseau local appartiennent à la même URR.

La gestion du groupe des processus d'une URR est assurée par un gestionnaire de la tolérance aux fautes (GTF). Il se comporte comme une passerelle pour les communications entre groupes [Lowr91]. Si les protocoles de communication entre réseaux locaux sont différents des protocoles internes aux réseaux locaux, les URR n'utilisent qu'un seul protocole et délèguent au GTF les communications. En outre, le mécanisme d'horloge globale logique se limite au réseau local. Pratiquement, un processus n'a pas besoin de maintenir des informations de dépendance concernant les processus distants.

Par ailleurs, nous verrons dans la suite du chapitre que les mécanismes de base s'appuient sur le GTF pour améliorer l'efficience.

Enfin, puisque le GTF est nécessaire à l'URR, il fait partie intégrante de l'URR. L'application est donc constituée des processus qu'elle crée directement plus des GTFs. Les GTFs doivent donc être eux-mêmes tolérants aux fautes. Ce point est étudié au fur et à mesure du chapitre, et plus particulièrement, dans la sous-section 3.2.5.

3.2.1.2 La gestion des unités de reprise réparties

Avant de détailler la gestion d'une URR, nous montrons que le GTF transforme le groupe de processus en une unité de confinement, de détection, de reconfiguration, de récupération et de reprise (*cf.* sous-section 1.2.4).

Les messages à destination de processus distants sont des messages à destination du monde extérieur à l'URR. Si les passerelles sont pessimistes, ces messages provoquent la validation de l'état des émetteurs au moment des émissions [Sist89]. Le GTF assure le confinement des fautes à l'URR. Le cas contraire - des passerelles optimistes - implique l'utilisation d'un algorithme de validation des états réparti entre les GTFs [Lowr91].

Si les fautes des processus distants n'ont pas besoin d'être détectées, le mécanisme de détection des fautes peut rester local à l'URR. L'autre alternative oblige l'utilisation d'un algorithme de détection réparti entre les GTFs.

Les communications entre groupes sont dirigées vers le GTF local, qui transmet aux GTFs distants, qui délivrent les messages aux processus récepteurs. Si les processus défaillants sont redémarrés dans les mêmes URRs, la reconfiguration est limitée au rétablissement des liens physiques avec les processus voisins et avec le GTF. Les liens entre processus distants ne sont alors que logiques : ils sont constitués de trois liens physiques [2]. En revanche, si les processus défaillants sont redémarrés dans d'autres URRs, les URRs de départ et d'arrivée sont reconfigurées ainsi que tous les GTFs. Par ailleurs, les messages en transit vers les processus défaillants déplacés doivent être redirigés.

Les messages en provenance de processus distants sont des messages en provenance du monde extérieur à l'URR. Ils provoquent l'enregistrement en mémoire stable du contenu du message avant sa réception. Aussi, si les processus défaillants sont redémarrés dans la même URR, la récupération des informations de recouvrement est immédiate. En effet, elles sont entièrement contenues dans le système de gestion répartie des fichiers du réseau local. Par contre, si les processus sont déplacés, le risque d'une faute globale du réseau local oblige la réplication des informations de recouvrement sur le système de gestion répartie des fichiers d'un réseau distant.

Si les passerelles sont pessimistes, le recouvrement ne concerne que les processus locaux. Dans le cas contraire, le recouvrement est réparti entre les GTFs.

Donc, par souci de simplicité, nous optons pour des passerelles pessimistes et pour une réintégration des processus défaillants dans les mêmes URRs. Par conséquent, le GTF transforme le groupe de processus en une unité de confinement, de détection, de reconfiguration, de récupération et de reprise. Une conséquence importante est qu'en aucun cas la gestion d'une URR ne dépend d'entités extérieures au réseau local.

Maintenant, nous détaillons la gestion d'une URR.

Chaque application possède un code d'identification unique. Ce code est composé du code d'identification unique du nœud [3] de création du premier processus plus de la valeur de l'horloge locale. Le GTF rend publique son adresse dans un fichier dont le nom est le code d'identification de l'application. Grâce au système de gestion répartie des fichiers, l'adresse est accessible à tous les nœuds.

Lorsqu'un processus de l'application se crée, celui-ci doit s'enregistrer auprès du GTF. Si le fichier de publication de l'adresse du GTF est déjà présent, cela signifie que des

2. (*processus - GTF*) + (*GTF - GTF*) + (*GTF - processus*).
3. Par exemple, le numéro Internet d'un site est unique.

processus voisins s'exécutent déjà. Si le fichier n'est pas encore présent, le processus le crée et démarre le GTF qui publie son adresse en l'insérant dans le fichier.

Lorsqu'un processus termine normalement son exécution, il avertit le GTF. S'il était le dernier représentant de l'application dans ce réseau local, le GTF termine lui aussi normalement son exécution. Il est à noter qu'un processus tentant de communiquer avec un processus qui a terminé normalement son exécution provoque une faute de programmation.

3.2.1.3 Discussion

Les hypothèses faites sur les abstractions fournies par les systèmes d'exploitation répartis des réseaux locaux actuels sont classiques. En effet, dans [Borg89, John89, Elno93], le mécanisme s'appuie sur un système de gestion répartie des fichiers répliqués que les auteurs considèrent alors comme fiable. Les algorithmes de diffusion fiable supposent au départ l'existence d'un mécanisme de diffusion non fiable [Chan84, Birm87, MS89, Kaas94].

Sistla et Welch étendent leur mécanisme de recouvrement arrière et divisent un grand réseau[4] en ajoutant une passerelle dans chaque partition [Sist89]. Lowry, Russell et Goldberg définissent les passerelles pessimiste et optimiste [Lowr91]. Ils montrent que l'utilisation de passerelles optimistes implique l'utilisation de protocoles entre passerelles pour le maintien de la tolérances aux fautes. Par déduction, il faut un mécanisme de gestion de groupe pour les passerelles elle-mêmes. Johnson insère un numéro d'application dans l'entête des messages interprocessus, le but est d'autoriser les communications entre applications[John93]. Les messages à destination d'une autre application sont considérés comme des messages à destination du monde extérieur. Vaidya étend la notion d'unité de reprise à un ensemble de processus et ainsi introduit la notion d'unité de reprise répartie[Vaid93b]. L'objectif est alors de structurer l'application en groupes de processus choisissant les mêmes mécanismes de base et communiquant intensément entre eux.

L'architecture que nous proposons s'inspire grandement de tous ces travaux. Cependant, nous précisons la définition d'une URR et justifions le dimensionnement des URRs à la taille des réseaux locaux. Par ailleurs, nous construisons une architecture globale évitant tout protocole entre GTFs. Enfin, nous montrons la difficulté qu'il y aurait à autoriser la migration des processus d'une URR à une autre.

3.2.2 Les notations algorithmiques

Chaque processus est associé avec un contrôleur. Le rôle du contrôleur est d'insérer les actions nécessaires aux mécanismes de base dans l'exécution du processus.

Les algorithmes utilisent des temporisations. Une temporisation est une horloge initialisée par un processus pour que le système d'exploitation le réveille. En initialisant la temporisation, le processus indique les actions à exécuter à la fin du temps écoulé. À un instant donné, pour un mécanisme et pour un processus, nous verrons qu'il ne peut y avoir qu'une seule temporisation initialisée. Les temporisations d'un processus P_i sont notées τ_i^x, avec x le nom de la temporisation. Une temporisation τ_i^x est initialisée par le processus P_i par l'action `init`(τ_i^x, n) et enlevée par l'action `enlever`(τ_i^x). n est un nombre entier associé à la temporisation τ_i^x, il est assimilé à un numéro de version. Par abus de notation, le nom de l'action indique aussi la durée de l'horloge. Au bout de τ_i^x secondes, la temporisation provoque l'action `fin`(τ_i^x, n). S'il existe déjà une temporisation τ_i^x avec un numéro de version inférieur, l'action `init`(τ_i^x, n) commence par effacer les temporisations

4. Grand en nombre de nœuds.

existantes. S'il existe déjà une temporisation τ_i^x avec un numéro de version supérieur ou égal à n, l'action `init`(τ_i^x, n) ne fait rien. De même, s'il n'existe plus de temporisation τ_i^x, l'action `enlever`(τ_i^x) ne fait rien. Conformément au modèle des applications, l'action `fin`(τ_i^x, n) est supposée être atomique et s'exécuter entre deux actions atomiques de P_i.

Les actions exécutées par un processus $P_i \in URR$ à la suite de la réception d'un message m_i^j en provenance d'un processus $P_j \in URR$ sont notées RECEVOIR(m). Remarquons que le GTF est un processus de l'URR. De même, les actions exécutées par un processus $P_i \in URR$ précédant l'émission d'un message m_j^i à destination d'un processus $P_j \in URR$ sont notées ÉMETTRE(m). Les messages des algorithmes - par exemple, $m\_pr\_sync_{GTF}^i$ - sont notés différemment des messages inter-processus - par exemple, m_i^j. Tout au long de ce chapitre, les actions `émettre`(m) et `recevoir`(m) sont modifiées en ÉMETTRE(m) et RECEVOIR(m). Enfin, conformément au modèle des applications, les actions ÉMETTRE(m) et RECEVOIR(m) sont supposées être atomiques et s'exécuter entre deux actions atomiques de P_i.

3.2.3 Le mécanisme de constitution des points de reprise

Le mécanisme de constitution de points de reprise construit aussi bien des coupures cohérentes que des constitutions de points de reprise non coordonnés.

Le premier paragraphe motive la prise en compte des deux types de constitution ainsi que l'hypothèse faite sur la durée de l'intervalle entre deux constitutions (*cf.* § 3.2.3.1). Ensuite, nous décomposons les constitutions de points de reprise en trois phases (lancement, enrgistrement des messages et assurance) et améliorons ainsi l'efficience du mécanisme (*cf.* § 3.2.3.3). Ces choix sont discutés dans un dernier paragraphe (*cf.* § 3.2.3.4).

3.2.3.1 Un mécanisme de constitutions de points de reprise synchronisées ou non

Les arguments pour construire des coupures cohérentes sont :

- le faible nombre de contenus de message à sauvegarder en mémoire stable,
- la prise en compte "partielle" de l'indéterminisme.

Les arguments pour constituer des points de reprise non coordonnés sont :

- le faible coût des interactions avec le monde extérieur,
- l'échelonnement des enregistrements des points de reprise en mémoire stable,
- la faible quantité de travail à refaire.

Notre mécanisme peut construire des coupures cohérentes ou des points de reprise non coordonnés. La méthode est de construire un mécanisme faiblement synchronisé ou virtuellement synchronisé, suivant les avantages recherchés.

D'une part, les applications possèdent une durée de vie longue - nettement supérieure à la dizaine de minutes. D'autre part, l'utilisateur n'est pas prêt à payer le prix fort pour rendre tolérante aux fautes son application. Par conséquent, la durée entre deux constitutions de points de reprise consécutives est de l'ordre de la dizaine de minutes. Notons que cet ordre de grandeur a été choisi pour l'évaluation d'implantations déjà existantes [John89, Alvi93, Elno93, Plan93, Leon94, Sens94]

La durée d'une constitution de points de reprise importe peu, pour autant qu'elle n'excède pas la dizaine de minutes. Nous mettons à profit la possibilité de faire durer une constitution de points de reprise pour minimiser les surcoûts dus au stockage. Pour cela, nous décomposons les constitutions de points de reprise en trois phases. Il s'ensuit l'optimisation du nombre de contenus de message à enregistrer en mémoire stable et l'échelonnement des accès à la mémoire stable.

Avant de présenter en détail le mécanisme de constitution de points de reprise, nous en donnons une description informelle.

1. Le lancement :
   (a) À la fin de l'intervalle de temps τ_i^0 [5], P_i transmet un ordre de constitution au GTF où à lui-même, selon que les constitutions de points de reprise doivent ou non être synchronisées.
   (b) Sur réception d'un ordre de constitution, si les constitutions de points de reprise doivent être synchronisées, le GTF diffuse de manière fiable l'ordre de constitution. Auparavant, le GTF vérifie que toutes les constitutions synchronisées précédentes sont terminées.
   (c) Sur réception d'un ordre de constitution, P_i sauvegarde son état. Dorénavant, tous les messages émis par P_i sont marqués. En outre, si les constitutions de points de reprise doivent être synchronisées, P_i sauvegarde son état sur réception d'un message marqué avant la réception de l'ordre de constitution venant du GTF.
2. L'enregistrement des messages :
   (a) La deuxième phase débute par l'initialisation d'une temporisation avant l'enregistrement des messages en mémoire stable. Pendant ce temps, P_i peut apprendre - par l'intermédiaire de messages marqués - que d'autres processus constituent aussi un point de reprise.
   (b) À la fin de cette temporisation, P_i sauvegarde en mémoire stable les messages demandés - calculés à l'aide des marques. Rappelons que l'ensemble des messages demandés est un sous-ensemble des messages émis par P_i. Observons que cette optimisation du nombre des messages sauvegardés en mémoire stable est valable que les constitutions de points de reprise soient synchronisées ou non.
3. L'assurance ou terminaison de la constitution des points de reprise :
   (a) P_i émet un message de notification de fin d'enregistrement des messages à destination du GTF. Si l'enregistrement des messages n'a pas été optimisé et si les constitutions ne sont pas synchronisées, la constitution du point de reprise de P_i est terminée.
   (b) P_i initialise une nouvelle temporisation.
   (c) Sur réception d'un message de notification de fin d'enregistrement des messages, le GTF exécute l'algorithme d'assurance du point de reprise de P_i (*cf.* figure 3.2) à la suite de quoi il émet un message d'assurance de points de reprise à destination de P_i.
   (d) Si la temporisation de P_i se déclenche avant la réception du message d'assurance, le mécanisme reprend en 2b en n'optimisant pas l'enregistrement des messages.

5. τ_i^0 est l'intervalle de temps entre deux constitutions de points de reprise.

3.2.3.2 Les structures de données, les messages et les temporisations

Les structures de données maintenues par chaque processus P_i

- pr_sync_i : booléen évalué à "faux" ssi P_i participe aux constitutions synchronisées de points de reprise. pr_sync_i est initialisé à "faux".
- $pr\_en\_cours_i$: vecteur de booléens tel que $pr\_en\_cours_i[j]$ est évalué à "vrai" ssi, à la connaissance de P_i, P_j est en cours de constitution d'un point de reprise. Toutes les composantes de ce vecteur sont initialisées à "faux".
- vdd_i : vecteur des dépendances directes construit par P_i [6]. Toutes les composantes de ce vecteur sont initialisées à 0.
- nra_i : numéro de recouvrement arrière ou numéro de l'exécution du mécanisme de recouvrement arrière (*cf.* sous-section 3.2.5). nra_i est initialisé à 0.

Les structures de données maintenues par le GTF

- pr_sync_{GTF} : vecteur de booléens tel que $pr\_sync_{GTF}[i]$ est évalué à "vrai" ssi P_i participe aux constitutions synchronisées de points de reprise. Toutes les composantes de ce vecteur sont initialisées à "faux".
- $pr\_en\_cours_{GTF}$: vecteur de booléens tel que $pr\_en\_cours_{GTF}[i]$ est évalué à "vrai" ssi P_i est en cours de constitution d'un point de reprise. Toutes les composantes de ce vecteur sont initialisées à "faux".
- **AS** : ensemble des processus ayant leur point de reprise en cours d'assurance - pour lesquels le message $m\_pr\_notif^i_{GTF}$ a été reçu et le message $m\_pr\_assur^{GTF}_i$ non émis. Cet ensemble est utilisé dans l'algorithme d'assurance (*cf.* figure 3.2). **AS** est initialisé à l'ensemble vide.
- $\mathbf{C}_{P_i}$: ensemble des processus qui, à la connaissance de P_i, appartiennent à la même "coupure concomitante" [7]. Il existe un ensemble par processus. Ces ensembles sont initialisés à l'ensemble vide.
- **Coupure** : ensemble des processus participant à la même "coupure concomitante". Cet ensemble est utilisé dans l'algorithme d'assurance (*cf.* figure 3.2). **Coupure** est initialisé à l'ensemble vide.
- nra_{GTF} : numéro de recouvrement arrière ou numéro de l'exécution du mécanisme de recouvrement arrière (*cf.* sous-section 3.2.5). nra_{GTF} est initialisé à 0.

Les messages

- $m\_pr\_sync^i_{GTF}$: transmission du résultat de la décision de participer ou non aux constitutions synchronisées.
- $m\_pr\_ordre^*_*$: ordre de constitution de points de reprise.

6. *cf.* § 3.2.4.3 pour la construction de ce vecteur.
7. *cf.* la définition d'une coupure concomitante dans ce qui suit.

- $m\_pr\_refus_i^{GTF}$: refus par le GTF d'une constitution synchronisée de points de reprise.
- $m\_pr\_notif_{GTF}^i$: notification par un processus de la fin de la constitution d'un point de reprise.
- $m\_pr\_assur_i^{GTF}$: assurance par le GTF de la constitution d'un point de reprise.

Les temporisations

- τ_i^0 : durée maximale entre deux constitutions de points de reprise.
- τ_i^1 : durée maximale entre la saisie de l'état et l'enregistrement en mémoire stable des messages.
- τ_i^2 : durée maximale d'attente du message d'assurance $m\_pr\_assur_i^{GTF}$.
- τ_i^3 : durée minimale entre deux constitutions de points de reprise.

3.2.3.3 Les trois phases des constitutions de points de reprise

La constitution d'un point de reprise comprend la sauvegarde de l'état du processus, l'enregistrement des messages demandés et l'assurance du point de reprise par le GTF. À la fin de ce paragraphe, nous étudions la tolérance aux fautes du mécanisme.

1re phase : Le lancement

Un processus peut décider de synchroniser ou non ses constitutions de points de reprise avec les autres processus de l'URR. Le résultat de cette décision est gardé dans la variable pr_sync_i et un message $m\_pr\_sync_{GTF}^i$ est automatiquement émis vers le GTF pour transmettre la mise à jour de $pr\_sync_{GTF}[i]$. Un processus ne peut pas changer sa variable pr_sync_i pendant qu'il est en cours de constitution d'un point de reprise.

Un processus P_i constitue un point de reprise soit parce qu'il n'en a pas constitué depuis τ_i^0 secondes, soit parce qu'il se synchronise avec d'autres processus de l'URR. Suite à l'expiration de τ_i^0, P_i transmet l'ordre de constitution $m\_pr\_ordre_*^i$ au GTF ou à lui-même, selon que pr_sync_i est évalué à "vrai" ou "faux". Puisque tous les processus peuvent lancer des constitutions synchronisées, P_i n'est pas forcément le premier. L'ordre de constitution vient alors du GTF. Aussi, la réception du message $m\_pr\_ordre_i^{GTF}$ provoque l'action `enlever`(τ_i^0) puis la saisie de l'état de P_i. En outre, le GTF insère son vecteur $pr\_en\_cours_{GTF}$. P_i met à jour son vecteur $pr\_en\_cours_i$.

À la réception d'un message $m\_pr\_ordre_{GTF}^i$, le GTF vérifie que toutes les constitutions synchronisées de points de reprise sont terminées. Si c'est le cas, le GTF ajoute pr_sync_{GTF} à $m\_pr\_ordre_{GTF}^i$ et le diffuse à tous les processus concernés (à l'instant T). Dans le cas contraire, le GTF répond au processus par un message $m\_pr\_refus_i^{GTF}$.

La réception d'un message de refus par P_i provoque la ré-initialisation de la temporisation τ_i^0.

Tous les messages émis pendant les constitutions de points de reprise sont marqués par les émetteurs [Lai87]. Sur réception d'un message marqué, si un processus participe

aux constitutions synchronisées, il débute la constitution d'un point de reprise s'il ne l'a pas déjà fait. Le prédicat `marqué`(m) est évalué à "vrai" ssi m est marqué. L'action de marquage d'un message m avant son émission est notée `marquer`(m).

Enfin, pour que la diffusion de $m\_pr\_ordre_i^{GTF}$ soit fiable, tous les processus l'acquittent positivement. Le GTF ré-émet sélectivement ce message aux processus n'ayant pas répondu au temps $T + 2\delta$, avec δ la durée moyenne de transmission d'un message sur le réseau local[8]. Pour ne pas surcharger les algorithmes, nous ne montrons pas les acquittements des messages de diffusion.

La première phase débute donc à l'instant T_0 et est terminée lorsque l'état du processus est complètement enregistré en mémoire stable (à l'instant T_1). La figure 3.1 visualise le mécanisme de constitution d'un point de reprise pour une constitution synchronisée. Seuls les messages concernant le dialogue entre le processus et le GTF sont dessinés.

Si tous les processus se synchronisent, Lai et Yang montrent que le marquage des messages émis après les saisies des états des processus assure la cohérence de la coupure [Lai87].

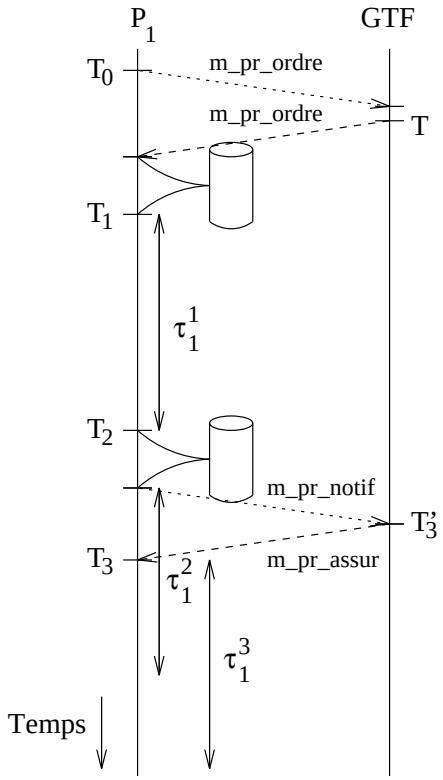


FIG. 3.1 – Les trois phases des constitutions de points de reprise.

2e phase : L'enregistrement des messages

La deuxième phase débute par l'initialisation d'une temporisation τ_i^1 avant l'enregistrement des messages en mémoire stable (*cf.* figure 3.1). Cette temporisation est supposée

8. δ comprend les temps moyens de traitements estimés pour l'émission et la réception du message ainsi que le temps de propagation du message sur le support physique.

assez grande pour que tous les processus en cours de synchronisation aient reçu l'ordre de constitution. Dans la réalité, la diffusion fiable est de l'ordre de la seconde, τ_i^1 de l'ordre de la minute. De même, il est supposé que τ_i^1 est supérieure à la durée de sauvegarde de l'état du processus en mémoire stable.

Si tous les processus participent aux constitutions synchronisées, les messages manquants traversant la coupure cohérente sont les messages non marqués et reçus par un processus en constitution [Lai87]. Seuls ces messages sont enregistrés en mémoire stable.

Dans le cas de constitutions de points de reprise non coordonnés, l'intervalle de temps $]T_0, T_1 + \tau_i^1[$ sert à connaître les constitutions concomitantes. Étant donnés P_i et P_j deux processus voisins, la *constitution* du point de reprise de P_j est *concomitante* pour P_i ssi P_i en a connaissance avant le début de l'enregistrement en mémoire stable des messages. La connaissance s'effectue grâce au marquage et au message $m\_pr\_ordre_i^{GTF}$:

$$\texttt{pr\_concomitant}(P_i, P_j) \stackrel{\text{def}}{=} \begin{cases} & \texttt{pr\_en\_cours}_{GTF}[j] \\ \vee & \exists m_i^j : (\texttt{recevoir}(m_i^j) \prec_{\mathbf{A}} \texttt{fin}(\tau_i^1, nra)) \wedge \texttt{marqué}(m_i^j) \end{cases}$$

avec le prédicat $\texttt{pr\_concomitant}(P_i, P_j)$ évalué à "vrai" ssi la constitution de P_j est concomitante pour P_i.

Par défaut, tous les messages émis depuis la dernière constitution des points de reprise sont enregistrés en mémoire stable. Les messages demandés sont inclus dans cet ensemble de messages. Les processus utilisent la connaissance des constitutions concomitantes pour optimiser le nombre de messages à enregistrer en mémoire stable. En effet, les messages traversant la coupure concomitante sont émis non marqués et reçus par un processus en constitution. D'après la définition des messages demandés, il est simple de montrer que les messages émis non marqués sont demandés. De même, il est clair que les messages marqués, ou non marqués et reçus par un processus non en constitution ne sont pas demandés. Par conséquent, seuls les messages émis non marqués et reçus marqués sont enregistrés en mémoire stable.

Pour chaque processus, la deuxième phase est terminée lorsque les messages demandés sont enregistrés en mémoire stable (à l'instant T_2) (*cf.* figure 3.1).

3e phase : L'assurance ou terminaison de la constitution des points de reprise

Si le prédicat $\texttt{pr\_concomitant}(P_i, P_j)$ est évalué à "vrai", le processus P_i minimise le nombre de messages qu'il enregistre en mémoire stable. Pour que le point de reprise R_i^x soit permanent - la constitution du point de reprise terminée -, il faut que P_i sache que P_j a terminé la constitution du point de reprise R_j^y. C'est le GTF qui calcule la terminaison des constitutions. Nous appelons ce calcul "l'assurance".

La troisième phase débute avec l'émission par le processus d'un message $m\_pr\_notif_{GTF}^i$ de notification à destination du GTF (*cf.* figure 3.1). Ce message notifie la fin de l'enregistrement en mémoire stable des informations de recouvrement. Il contient le vecteur $pr\_en\_cours_{GTF}$ et le vecteur vdd_i de l'instant T_0.

À la suite de $m\_pr\_notif_{GTF}^i$, le processus initialise une temporisation τ_i^2. Le processus attend au maximum jusqu'à l'instant $T_2 + \tau_i^2$ la confirmation par le GTF de l'assurance de

son point de reprise. Le GTF exécute un algorithme d'assurance (*cf.* figure 3.2) à la suite duquel il confirme la constitution de points de reprise par le message $m\_pr\_assur_i^{GTF}$. Dans la figure 3.1, $m\_pr\_assur_i^{GTF}$ est reçu avant la fin de la temporisation.

Un point de reprise R_i^x est assuré ssi tous les processus appartenant à la même coupure concomitante ont terminé de constituer leur point de reprise :

$$\texttt{assuré}(R_i^x) \stackrel{\text{def}}{=} \begin{cases} & \forall P_j \in URR, \texttt{pr\_concomitant}(P_i, P_j) \Rightarrow \texttt{terminé}(P_j) \\ \wedge & \forall (P_j, P_k) \in URR \times URR, \\ & (\texttt{pr\_concomitant}(P_i, P_j) \wedge \texttt{pr\_concomitant}(P_j, P_k)) \\ & \Rightarrow \texttt{terminé}(P_k) \end{cases}$$

avec le prédicat $\texttt{terminé}(P_j)$ évalué à "vrai" ssi le message $m\_pr\_notif_{GTF}^j$ est reçu.

La figure 3.2 présente l'algorithme d'assurance des points de reprise exécuté par le GTF à chaque réception d'un message $m\_pr\_notif_{GTF}^i$. Les variables temporaires de l'algorithme sont les suivantes :

- **N** est l'ensemble des processus ayant leur point de reprise nouvellement assuré par l'exécution courante de la procédure `Assurance`,
- **X** est l'ensemble des processus ayant leur point de reprise ne pouvant pas encore être assuré,
- P_i, P_j et P_k sont des processus voisins.

La fonction `Assure` calcule la coupure concomitante en cours d'assurance chez le GTF. Elle vérifie que le GTF a bien reçu tous les messages $m\_pr\_notif_{GTF}^i$ des processus appartenant à la coupure concomitante. La fonction est bien entendu récursive (*cf.* ligne 50). Les deux conditions d'arrêt de la récursion sont la non-réception d'une notification (*cf.* lignes 43 et 58) et la réception des notifications de tous les processus de la coupure (*cf.* lignes 45 et 47).

Au début de l'exécution, **AS** est égal à l'ensemble vide. La procédure `Assurance` retire de **AS** définitivement les processus dont le point de reprise vient d'être assuré (*cf.* ligne 28) et provisoirement ceux qui ne peuvent pas encore l'être (*cf.* lignes 25 puis 27). En outre, la procédure initialise **N** à l'ensemble vide (*cf.* ligne 05) et y ajoute les processus de la dernière coupure concomitante construite : **Coupure** (*cf.* ligne 20). Le GTF enregistre en mémoire stable les constitutions de points de reprise assurées par l'intermédiaire de la procédure `Nouveaux_points_de_reprise`. Ensuite, il émet un message d'assurance $m\_pr\_assur_i^{GTF}$ vers tous les processus $P_i \in URR$ attendant l'assurance de leur point de reprise (*cf.* lignes 30..36). Les processus constituant des points de reprise non coordonnés et tels que $\mathbf{C}_{P_k} = \emptyset$ n'ont pas besoin d'assurer leur point de reprise auprès du GTF (*cf.* ligne 31). Dans ce cas, la notification $m\_pr\_notif_{GTF}^i$ a pour but de prévenir le GTF et ainsi rendre le point de reprise permanent [Koo87].

À l'instant $T_2 + \tau_i^2$, si le GTF n'a pas encore assuré le point de reprise du processus, le processus recommence l'enregistrement des messages. Mais, cette fois-ci, le processus ignore les marqueurs. Tous les messages émis entre la constitution du précédent point de reprise et l'instant T_0 sont considérés comme des messages demandés. Ils doivent être enregistrés en mémoire stable. Enfin, le processus émet une nouvelle notification à destination du GTF. C'est la signification du test de la ligne 06 dans la procédure `Assurance`.

La constitution d'un point de reprise se termine à l'instant T_3 soit par l'émission de la notification (cas $\mathcal{C}_{P_i} = \emptyset$), soit par la réception de l'assurance (cas $\mathcal{C}_{P_i} \neq \emptyset$). Dans la figure 3.1, la constitution du point de reprise de P_1 se termine par la réception du message $m\_pr\_assur_i^{GTF}$.

Enfin, pour les constitutions synchronisées, un processus P_i peut recevoir des messages marqués après l'instant T_3. Afin de ne pas provoquer une nouvelle constitution de points de reprise, les marqueurs sont coloriés [Lai87]. Les messages marqués de l'ancienne constitution de points de reprise sont alors ignorés. En outre, le GTF n'accepte aucune nouvelle constitution de points de reprise cohérente avant l'instant $T_3' + \tau_i^3$, tel que T_3' est l'instant d'émission des messages d'assurance et τ_i^3 est strictement supérieur à 2δ. Par souci de simplicité, la gestion des temporisations τ_i^3 n'est pas incluse dans la description des algorithmes.

```
00 Procedure Assurance (processus P_i)
01 {
02   ens_processus N, X, Coupure;
03   processus P_j ;
04   /*aucun nouveau point de reprise assuré */
05   N := ∅;
06   Si C_{P_i} == ∅ ∧ P_i ∈ AS Alors
07     /* deuxième notification */
08     N := N ∪ {P_i};
09   Sinon
10     /* première notification */
11     AS := AS ∪ {P_i};
12   Finsi;
13   /* on suppose que tous les points
14    de reprise sont assurés */
15   X := ∅;
16   Tantque AS ≠ ∅ Faire
17     P_j := Premier_élément_de (AS);
18     Coupure := {P_j};
19     Si Assure (P_j) Alors
20       N := N ∪ Coupure;
21     Sinon
22       X := X ∪ Coupure;
23       AS := AS − Coupure;
24     Finsi;
25   Fintantque;
26   /* préparation du prochain appel */
27   AS := AS ∪ X;
28   AS := AS − N;
29   Nouveaux_points_de_reprise (N);
30   Pour P_j ∈ N Faire
31     Si C_{P_j} ≠ ∅ Alors
32       /* P_j attend l'assurance */
33       émettre(P_j, m_pr_assur);
34       C_{P_j} := ∅ ;
35     Finsi;
36   Finpour;
37 }

39 Fonction Assure (processus P_j) : Booleen
40 {
41   processus P_k;
42   Pour P_k ∈ C_{P_j} Faire
43     Si P_k ∈ AS Alors
44       /* notification recue */
45       Si P_k ∈ Coupure Alors
46         /* C_{P_k} déjà parcouru */
47         retourner vrai;
48       Sinon
49         Coupure := Coupure ∪ {P_k};
50         Si ¬ Assure (P_k) Alors
51           retourner faux;
52         Finsi
53       Finsi
54     Sinon
55       /* notification non recue
56        donc, points de reprise
57        de Coupure non assurés */
58       retourner faux;
59     Finsi
60   Finpour
61 }
```

FIG. 3.2 – L'algorithme d'assurance des points de reprise.

La tolérance aux fautes du mécanisme

Dans notre mécanisme de recouvrement arrière, toute faute annule tous les points de reprise en cours de constitution, et ceci quelle que soit la phase d'avancement des constitutions.

Il reste alors à vérifier que la mémoire stable contient tous les éléments pour récupérer les informations de recouvrement. Lorsque le GTF assure un point de reprise, la procédure `Nouveaux_points_de_reprise` enregistre en mémoire stable l'action de création du point de reprise (*cf.* ligne 29 de la figure 3.2). Cette procédure enregistre les noms des fichiers

contenant les informations de recouvrement, plus d'autres informations telles que la liste des processus de la coupure concomitante. Ces informations n'étant pas volumineuses, nous considérons que leur enregistrement en mémoire stable est atomique. Par conséquent, l'opération de constitution concomitante de points de reprise est une opération atomique. En cas de faute globale du réseau local, le nouveau GTF récupère toutes les actions de création des points de reprise de la coupure concomitante.

3.2.3.4 Discussion

Chandy et Lamport démontrent que seul un protocole à deux phases est à même de constituer des coupures cohérentes. Les auteurs s'intéressent plus particulièrement à la première phase et introduisent les marqueurs [Chan85]. Lai et Yang incluent les marqueurs dans les messages interprocessus [Lai87]. Koo et Toueg montrent l'efficacité en terme de degré de tolérance aux fautes du schéma proposé par Chandy et Lamport. Ils développent la deuxième phase du protocole et introduisent deux catégories de points de reprise: provisoires et permanents [Koo87].

La décomposition en trois phases de notre mécanisme s'inspire de ces protocoles à deux phases. Elle permet de construire des constitutions synchronisées de points de reprise ou non. Pour les coupures cohérentes, la première phase diffuse de manière fiable un marqueur et ajoute des marqueurs aux messages interprocessus. La troisième et dernière phase assure les points de reprise et ainsi les rend permanents. Pour les constitutions de points de reprise non coordonnés, la deuxième phase enregistre en mémoire stable les messages demandés [Stro88].

L'hypothèse de non-optimisation de la durée des constitutions de points de reprise autorise l'insertion d'une phase entre les phases de lancement et d'assurance. Ce point constitue l'originalité la plus importante de notre mécanisme de constitution de points de reprise. Pendant la deuxième phase, les processus repèrent sans échange explicite de messages les processus de la coupure concomitante. En outre, grâce à la phase d'assurance, les processus se synchronisent de manière automatique[9], contrairement au mécanisme décrit dans [Tong89]. Si les horloges locales évoluent sensiblement à la même vitesse, il est possible de constituer des points de reprise virtuellement synchronisés [Tong89, Cris91a].

Dans [Kaas92], les auteurs expérimentent un surcoût de stockage important lorsque beaucoup de processus appartiennent à la coupure cohérente. Pour échelonner les accès à la mémoire stable, Plank retarde au maximum la transmission des marqueurs [Plan93]. Dans le mécanisme que nous décrivons, il est aisé d'ordonner les processus de l'URR et ainsi d'échelonner dans le temps les débuts des constitutions de points de reprise. En conséquence, les processus commenceraient l'enregistrement de leur état soit à la suite de la réception d'un marqueur, soit à la fin de la temporisation.

Le mécanisme permet aussi de disperser dans le temps l'enregistrement en mémoire stable des messages demandés. Ceci est obtenu en agissant sur les temporisations τ_i^1 de la deuxième phase.

3.2.4 Le mécanisme de journalisation

Le mécanisme de journalisation autorise aussi bien une journalisation optimiste que pessimiste.

9. L'algorithme d'assurance des points de reprise émet les messages d'assurance pour tous les processus de la coupure dans la même exécution de la procédure `Assurance`.

Le premier paragraphe motive la prise en compte des deux types de journalisation ainsi que l'aspect centralisé de notre mécanisme de journalisation. Puis, le paragraphe 3.2.4.2 introduit les structures de données des algorithmes. Ensuite, nous développons les trois niveaux de journalisation : en mémoire volatile de l'émetteur, en mémoire volatile du GTF et en mémoire stable (*cf.* § 3.2.4.3). Ces choix sont discutés dans un dernier paragraphe (*cf.* § 3.2.4.4).

3.2.4.1 Un mécanisme de journalisation centralisé optimiste ou pessimiste

L'utilisateur désirant que toute faute ne provoque pas la ré-exécution de processus orphelins choisit la journalisation pessimiste pour tous les processus.

Par contre, la journalisation pessimiste entraîne des surcoûts importants de stockage, de charge du réseau et d'exécution. Aussi, pour des raisons d'efficience pendant l'exécution, l'utilisateur choisit souvent une journalisation optimiste.

D'après le paragraphe 3.2.1.2, les messages émis à destination du monde extérieur à l'URR sont gérés par le GTF. Or, tous ces messages doivent être stables. Pour des raisons de simplicité, nous choisissons d'utiliser un algorithme de validation centralisé (*cf.* § 2.4.2.1). Le GTF est tout indiqué pour valider ces messages. Par ailleurs, la validation d'un message consiste à parcourir l'histoire répartie journalisée. Donc, les processus enregistrent leur histoire séquentielle en mémoire volatile du nœud sur lequel s'exécute le GTF. Le GTF valide les messages lorsque cela est nécessaire. Il enregistre périodiquement l'histoire répartie en mémoire stable, notamment lors de la validation d'un message.

Avant de présenter en détail le mécanisme de journalisation de l'exécution, nous en donnons une description informelle.

1. Les processus de l'unité de reprise répartie :
   (a) Lorsqu'un processus P_i émet un message m à destination d'un processus voisin P_j, P_i estampille m en y ajoutant son numéro d'ordre d'émission en entête. P_i enregistre la future dépendance de P_j par rapport à m dans un vecteur d'émission. Une copie de m est gardée en mémoire volatile du nœud de P_i.
   (b) Lorsque le processus P_j reçoit le message m, P_j estampille m avec son numéro d'ordre de réception. L'estampille du message m est gardée en mémoire volatile du nœud de P_j. L'ensemble des estampilles du processus P_j constitue l'histoire séquentielle de P_j. P_j enregistre la dépendance par rapport à P_i dans un vecteur des dépendances.
   (c) Quant aux messages à destination de processus distants, ils sont estampillés par les émetteurs et dirigés vers le GTF de départ (*cf.* 2c). P_i attend que le GTF confirme la transmission du message au GTF distant.
   (d) Les messages en provenance de processus distants sont enregistrés en mémoire stable par le GTF d'arrivée. Ensuite, ils sont fournis aux processus récepteurs.
2. Le gestionnaire de tolérance aux fautes de l'unité de reprise répartie :
   (a) Régulièrement, P_i transmet son histoire séquentielle et son vecteur des dépendances au GTF.
   (b) Le GTF garde l'histoire séquentielle de P_i en mémoire volatile du nœud sur lequel il s'exécute. À chaque estampille de message correspond un vecteur des

dépendances directes. C'est celui de l'instant d'émission du message de journalisation de P_i.

(c) Lorsque le GTF reçoit un message à destination du monde extérieur en provenance de P_i, le GTF exécute l'algorithme de validation d'un message (*cf.* figure 3.4). Puis, le GTF transmet le message au GTF distant, et ensuite, répond à P_i.

(d) Plus rarement, le GTF diffuse de manière non fiable un message contenant un vecteur donnant l'état d'avancement de l'enregistrement en mémoire stable de l'histoire répartie.

3. La mémoire stable :

(a) Régulièrement, le GTF enregistre en mémoire stable l'histoire répartie. Les vecteurs des dépendances directes enregistrés sont ceux des derniers messages de journalisation reçus, un vecteur par processus de l'URR.

3.2.4.2 Les structures de données, les messages et les temporisations

Les structures de données maintenues par chaque processus P_i

- noc_i : numéro de communication. noc_i est initialisé à 0.
- noe_i : numéro d'émission. noe_i est initialisé à 0.
- nor_i : numéro de réception. nor_i est initialisé à 0.
- $\mathbf{M}_i$: ensemble des contenus des messages reçus et gardés en mémoire volatile. $\mathbf{M}_i$ est initialisé à l'ensemble vide.
- $\mathbf{E}_i$: ensemble des estampilles des messages reçus et gardées en mémoire volatile. $\mathbf{E}_i$ est initialisé à l'ensemble vide.
- ve_i : vecteur des numéros d'émission tel que $ve_i[j]$ est égal à noe_i lors de la dernière émission de message m_j^i. Toutes les composantes de ve_i sont initialisées à 0.
- vdd_i : vecteur des dépendances directes tel que $vdd_i[j]$ est égal à noe_j lors de la dernière réception de message m_i^j, et tel que $vdd_i[i]$ est égal à nor_i lors de la dernière réception de message m_i^j. Toutes les composantes de vdd_i sont initialisées à 0.
- je_vdd_i : vecteur des dépendances directes au moment de la dernière journalisation par un message $m\_je\_proc_{GTF}^i$. Toutes les composantes de je_vdd_i sont initialisées à 0.
- ν_i^4 : nombre maximum de messages émis entre deux constitutions de points de reprise. ν_i^4 est un paramètre donné par l'utilisateur.
- nra_i : numéro de recouvrement arrière ou numéro de l'exécution du mécanisme de recouvrement arrière (*cf.* sous-section 3.2.5). nra_{GTF} est initialisé à 0.

Les structures de données maintenues par le GTF

- noc_{GTF} : numéro de communication. noc_{GTF} est initialisé à 0.
- noe_{GTF} : numéro d'émission. noe_{GTF} est initialisé à 0.

- nor_{GTF} : numéro de réception. nor_{GTF} est initialisé à 0.

- $\mathbf{E}_{GTF}$: ensemble des estampilles des messages reçus et gardées en mémoire volatile. $\mathbf{E}_{GTF}$ est initialisé à l'ensemble vide.

- max_stable : vecteur de l'état global recouvrable maximal de l'URR. Toutes les composantes de max_stable sont initialisées à 0.

- $\mathbf{PRNV}_{GTF}$: ensemble des couples (point de reprise non validé, vecteur des dépendances directes). $\mathbf{PRNV}_{GTF}$ est initialisé à l'ensemble vide.

- $dprv_{GTF}$: vecteur des derniers points de reprise validés tel que $dprv_{GTF}[i]$ est égal à $vdd_i[i]$ du dernier point de reprise validé R_i^x de P_i. Toutes les composantes de $dprv_{GTF}$ sont initialisées à 0.

- $\mathbf{PRVD}_{GTF}$: ensemble des couples (point de reprise validé demandé, vecteur des dépendances directes). $\mathbf{PRVD}_{GTF}$ est initialisé à l'ensemble vide.

- ν^5_{GTF} : nombre maximum de messages $m\_je\_proc^i_{GTF}$ reçus entre deux sauvegardes des estampilles en mémoire stable.

- ν^6_{GTF} : nombre maximum de messages $m\_je\_proc^i_{GTF}$ reçus entre deux diffusions non fiables du message $m\_je\_gtf_i^{GTF}$.

- nra_i : numéro de recouvrement arrière ou numéro de l'exécution du mécanisme de recouvrement arrière (*cf.* sous-section 3.2.5). nra_{GTF} est initialisé à 0.

Les messages

- $m\_je\_proc^i_{GTF}$: transmission des estampilles des messages reçus tels que $\forall(j \neq i), vdd_i[j] > je\_vdd_i[j]$.

- $m\_je\_gtf_i^{GTF}$: transmission du vecteur max_stable et du vecteur $dprv_{GTF}$.

- $m\_je\_valid^i_{GTF}$: demande de transmission d'un message à destination du monde extérieur à l'URR, donc demande de validation.

- $m\_je\_je_i^{GTF}$: demande de journalisation adressée lors de la validation d'un message à destination du monde extérieur.

Les temporisations

- τ_i^4 : durée maximale entre deux journalisations par le message $m\_je\_proc^i_{GTF}$.

- τ^5_{GTF} : durée maximale entre deux diffusions du message $m\_je\_gtf_i^{GTF}$.

- τ^6_{GTF} : durée maximale entre deux journalisations de l'histoire répartie par le GTF.

- τ^7_{GTF} : durée maximale entre deux calculs du vecteur $dprv_{GTF}$.

- τ^{valid}_{GTF} : durée maximale avant la diffusion de l'ordre de journalisation $m\_je\_je_i^{GTF}$.

3.2.4.3 Les trois niveaux de journalisation

La journalisation de l'exécution répartie est construite sur trois niveaux : chez les processus, puis, des processus au GTF, et ensuite, du GTF à la mémoire stable. Par la suite, nous présentons l'algorithme de validation d'un message par le GTF. Enfin, nous étudions l'effacement des informations de recouvrement et la tolérance aux fautes du mécanisme.

1^er^ niveau : Les processus de l'unité de reprise répartie

Lorsqu'un processus P_i émet un message m à destination d'un processus voisin P_j, P_i estampille m en y ajoutant son numéro d'ordre d'émission noe_i en entête. Le numéro d'ordre d'émission noe_i est égal au numéro d'ordre de communication noc_i du processus P_i de l'instant d'émission. Les numéros d'ordre de communication sont incrémentés après chaque émission et chaque réception de messages. En outre, le contenu de m est gardé en mémoire volatile du nœud sur lequel s'exécute P_i [John87]. L'ensemble des contenus des messages émis par P_i et gardés en mémoire volatile est appelé $\mathbf{M}_i$. Enfin, P_i enregistre la future dépendance de P_j par rapport à m dans un vecteur d'émission noté ve_i : $ve_i[j] := m_{noe_i}$.

Lorsque le processus P_j reçoit le message m, P_j estampille m avec son numéro d'ordre de réception nor_j. Le numéro d'ordre de réception nor_j est égal au numéro d'ordre de communication noc_j du processus P_j de l'instant de réception. L'estampille du message m complète[10] est gardée en mémoire volatile du nœud de P_j. L'ensemble des estampilles du processus P_j constitue l'histoire séquentielle de P_j. L'ensemble des estampilles des messages reçus par P_j et gardées en mémoire volatile est appelé $\mathbf{E}_j$. Enfin, le processus récepteur P_j mémorise la dépendance vis-à-vis du processus P_i dans un vecteur des dépendances dites directes vdd_j : $vdd_j[i] := m_{noe_i}$. En outre, nor_j est aussi placé dans $vdd_j[j]$.

Quant aux messages à destination de processus distants, ils sont estampillés par les émetteurs et dirigés vers le GTF de départ. Le prédicat `distant`(j) est évalué à "vrai" ssi $P_j \notin URR$. Le GTF valide ces messages et les enregistre lui-même en mémoire stable (*cf.* le 2^e^ niveau). Ils ne sont donc pas gardés en mémoire volatile des nœuds sur lesquels s'exécutent les processus émetteurs.

Enfin, les messages en provenance de processus distants sont enregistrés en mémoire stable par le GTF d'arrivée. Ensuite, ils sont fournis aux processus récepteurs.

2^e^ niveau : Le gestionnaire de tolérance aux fautes de l'unité de reprise répartie

Régulièrement, c.à-d. tous les ν_i^4 messages reçus ou toutes les τ_i^4 secondes, les processus transmettent leur histoire séquentielle au GTF. La figure 3.3 visualise le mécanisme de journalisation pour une URR composée de deux processus. Le paramètre pris en compte dans la figure est τ_i^4, d'où $\nu_i^4 > 4$ ici. Le message $m\_je\_proc^i_{GTF}$ est composé du vecteur des dépendances directes de l'instant de journalisation, plus de l'histoire séquentielle depuis la dernière journalisation.

Le GTF garde l'histoire répartie en mémoire volatile du nœud sur lequel il s'exécute. À chaque estampille de message correspond un vecteur des dépendances directes. C'est celui de l'instant d'émission du message $m\_je\_proc^i_{GTF}$. Pour le cas particulier des messages à destination de processus distants, les vecteurs de dépendances sont égaux aux vecteurs

10. Code d'identification de l'émetteur + m_{noe_i} + Code d'identification du récepteur + m_{nor_j}.

des dépendances transitives calculés lors de leur validation(*cf.* ci-dessous la validation d'un message par le GTF).

En outre, plus rarement, c.à-d. tous les ν^6_{GTF} messages reçus ou toutes les τ^6_{GTF} secondes, le GTF diffuse de manière non fiable un message $m\_je\_gtf_i^{GTF}$ contenant les vecteurs *max_stable* (*cf.* le 3^e niveau) et $dprv_{GTF}$ (*cf.* l'effacement des informations de recouvrement). $m\_je\_gtf_i^{GTF}$ donne l'état d'avancement de l'enregistrement en mémoire stable de l'histoire répartie. Le paramètre pris en compte dans la figure 3.3 est τ^6_{GTF}, d'où $\nu^6_{GTF} > 3$ ici.

3^e niveau : La mémoire stable

Régulièrement, c.à-d. tous les ν^5_{GTF} messages de journalisation reçus ou toutes les τ^5_{GTF} secondes ou encore lors de la validation d'un message, le GTF enregistre en mémoire stable l'histoire répartie. Le paramètre pris en compte dans la figure 3.3 est τ^5_{GTF}, d'où $\nu^5_{GTF} > 2$ ici. Les vecteurs des dépendances directes enregistrés sont ceux des derniers messages de journalisation reçus, un vecteur par processus de l'URR.

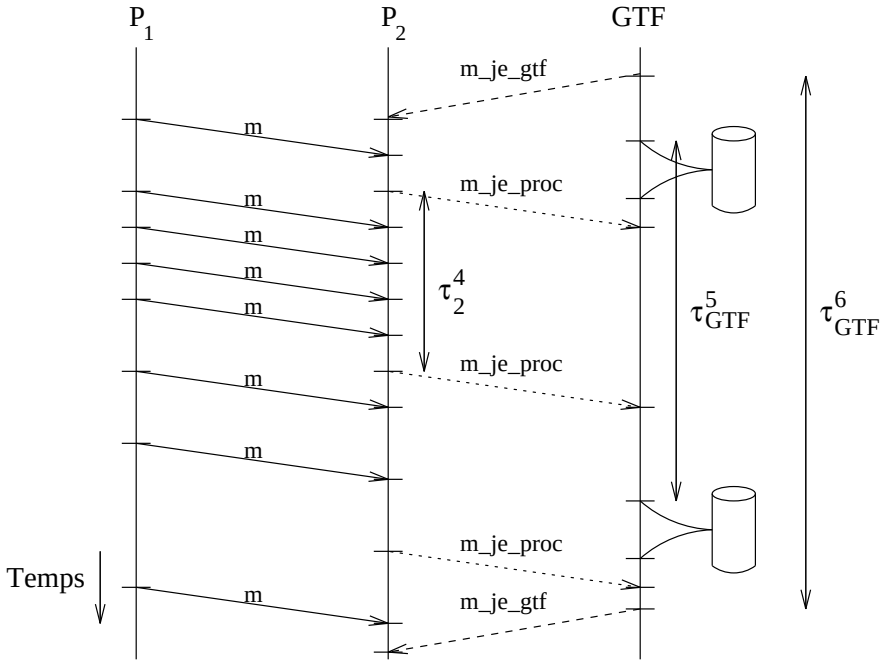


FIG. 3.3 – Les trois niveaux de journalisation de l'exécution répartie.

La validation d'un message par le gestionnaire de tolérance aux fautes

Lorsqu'un processus P_i désire émettre un message m à destination du monde extérieur, il l'estampille de l'histoire séquentielle non encore transmise. Le message transmis au GTF est noté $m\_je\_valid^i_{GTF}$. Le GTF répond par un message $m\_je\_gtf_i^{GTF}$ contenant le vecteur *max_stable* tel que $max\_stable[i] = noe_i$.

La figure 3.4 présente l'algorithme de validation d'un message par le GTF. Il est exécuté

par le GTF à chaque fois qu'un message est émis à destination du monde extérieur. Les notations de l'algorithme sont les suivantes :

- P_i est le processus émetteur du message m à valider,
- *noe* est le numéro d'ordre d'émission de m,
- *vd_dim* est le nombre de processus de l'URR,
- *vap* est le vecteur de valeurs booléennes indiquant les composantes du vecteur des dépendances restant à parcourir,
- *vdd* est le vecteur des dépendances correspondant à m,
- $transitif$ est la variable booléenne indiquant si les dépendances transitives de *vdd* ont pu être calculées,
- *continuer* est la variable booléenne indiquant si le calcul du vecteur des dépendances transitives est terminé,
- v est le vecteur des dépendances directes correspondant au message repéré par une composante du vecteur *vdd*,
- I et J sont des processus voisins.

La procédure `Validation` vérifie que m n'est pas orphelin (*cf.* ligne 11). Elle attend l'histoire répartie jusqu'à ce que l'émission de m soit enregistrée en mémoire stable. C'est la signification de la procédure `Attendre_journalisation` appelée à la ligne 12. `Attendre_journalisation` attend que n messages de journalisation $m\_je\_proc^i_{GTF}$ soient reçus par le GTF, avec n le nombre de composantes de *vap* égale à vrai. Enfin, puisque la journalisation est initiée par les processus, le GTF peut explicitement demander aux processus d'avancer les émissions des messages de journalisation; ceci est effectué par la procédure `Diffuser_journalisation` exécutée si le message n'est pas stable en τ^{valid}_{GTF} secondes (*cf.* lignes 13..19). Lors du premier appel, la procédure `Diffuser_journalisation` diffuse de manière non fiable un ordre de journalisation $m\_je\_je^{GTF}_i$ à tous les processus de l'URR. Les appels suivants ré-émettent sélectivement l'ordre aux processus n'ayant pas encore répondu.

La fonction `Orphelin` lance le calcul du vecteur des dépendances transitives correspondant au vecteur des dépendances directes de l'instant d'émission du message de journalisation. Le vecteur des dépendances transitives est calculé tel qu'il l'aurait été si, au lieu d'ajouter le numéro d'ordre d'émission, chaque processus ajoutait son vecteur des dépendances [Stro85, Sist89]. Si le vecteur des dépendances est effectivement calculé, le dernier message reçu avant l'émission de m est stable. En effet, tous les vecteurs des dépendances directes ont été trouvés, et par déduction, tous les messages journalisés [John93]. La condition antérieure au calcul du vecteur des dépendances transitives est donc la présence des vecteurs des dépendances directes en mémoire volatile du nœud sur lequel s'exécute le GTF. C'est le rôle de la fonction `Chercher_vd` appelée aux lignes 26 et 70. Elle parcourt les histoires séquentielles en recherchant le vecteur des dépendances directes du dernier message reçu par P_i avant l'émission du message de numéro d'ordre d'émission *noe*. Notons qu'afin d'optimiser cette recherche, le GTF mémorise le vecteur des dépendances transitives du dernier message stable dans un vecteur noté *max_stable*. Observons alors que les composantes de *max_stable* évoluent de manière monotone et croissante.

La fonction `Vd_transitif` cherche les vecteurs des dépendances directes correspondant aux messages repérés par les composantes du vecteur des dépendances *vdd*. Au début du calcul, toutes les composantes de *vdd* sont à parcourir. D'où, toutes les composantes du vecteur *vap* sont initialisées à la valeur *vrai* (*cf.* lignes 06..09). Si un vecteur des dépendances directes n'est pas trouvé alors le vecteur des dépendances transitives *vdd* ne peut pas être complètement calculé (*cf.* lignes 60 puis 76..81). Par contre, lorsqu'un nouveau vecteur des dépendances est trouvé, le maximum entre *vdd* et ce nouveau vecteur est calculé [Stro85]. La composante de *vdd* correspondant au nouveau vecteur n'est plus à parcourir (*cf.* lignes 66). Toute composante de *vdd* ayant pris la valeur de la composante du nouveau vecteur devra être à nouveau parcourue (*cf.* ligne 71). Le calcul se termine donc sur une impossibilité (*cf.* ligne 81) ou lorsque toutes les composantes de *vdd* ne sont plus à parcourir (*cf.* lignes 57, 68, 72 et 85).

```
00 Procedure Validation (processus Pi, entier noe)
01 {
02   vecteur vap[vd_dim];
03   processus I;
04
05   seconde t;
06   Pour I de 1 a vd_dim Faire
07     vap[I]:= vrai;
08   Finpour;
09   vap[Pi]:= faux;
10   premier_appel:= vrai;
11   Tantque Orphelin (Pi, noe, &vap) Faire
12     t:= Attendre_journalisation (τ^valid_GTF);
13     Si t == 0 Alors
14       /* journalisation non arrivée
15        d'où, diffusion non fiable d'un
16        ordre de journalisation */
17       Diffuser_journalisation (premier_appel);
18       premier_appel:= faux;
19     Finsi;
20   Fintantque;
21 }

23 Fonction Orphelin (processus Pi, entier noe) : Booleen
24 {
25   vecteur vdd[vd_dim];
26   Si Chercher_vd (Pi, noe, &vdd) == vrai Alors
27     /* le vecteur des dépendances existe
28      d'où, le message noe de Pi est journalisé
29      donc, calcul du vecteur des dépendances
30      transitives correspondant */
31     Si Vd_transitif (&vdd, &vap) Alors
32       /* journal complet
33        d'où, noe non orphelin */
34       retourner faux;
35     Sinon
36       /* le vecteur des dépendances
37        transitives n'est pas construit
38        d'où, noe orphelin */
39       retourner vrai;
40     Finsi
41   Sinon
42     /* le vecteur des dépendances n'existe pas
43      d'où, le message noe de Pi n'est pas
44      encore journalisé
45      donc, noe orphelin */
46     retourner vrai;
47   Finsi
48 }

49 Fonction Vd_transitif (vecteur &vdd,
50                        verb+vecteur+ &vap): Booleen
51 {
52   booleen continuer, transitif;
53   vecteur v[vd_dim];
54   processus I, J;
55   Repeter
56     transitif:= vrai;
57     continuer:= faux;
58     Pour I de 1 a vd_dim Faire
59       Si vap[I] == vrai Alors
60         Si Chercher (I, vdd[I], &v)
61                  == vrai Alors
62           /* le vecteur des dépendances
63            v existe
64            d'où, vdd[I] est journalisé
65            donc, on met à jour vdd */
66           vap[I]:= faux;
67           Pour J de 1 a vd_dim Faire
68             Si v[J] > vdd[J] Alors
69               vdd[J]:= v[J];
70               /* vdd[J] à analyser */
71               vap[J]:= vrai;
72               continuer:= vrai;
73             Finsi
74           Finpour
75         Sinon
76           /* le vecteur des dépendances
77            v n'existe pas
78            d'où, vdd[I] non journalisé
79            donc, vdd transitif non
80            calculable */
81           transitif:= faux;
82         Finsi
83       Finsi
84     Finpour
85   Jusqu'a continuer == faux
86           ∨ transitif == faux;
87   retourner transitif;
88 }
```

FIG. 3.4 – L'algorithme de validation d'un message.

L'effacement des informations de recouvrement

À chaque point de reprise R_i^x est associé un vecteur des dépendances directes $vdd_{R_i^x}$ (*cf.* sous-section 3.2.3). Soit $\mathbf{PRNV}_{GTF}$ l'ensemble des couples (point de reprise non validé, vecteur des dépendances directes). En outre, soit $dprv_{GTF}$ le vecteur des derniers points de reprise validés tel que $dprv_{GTF}[i]$ est égal à $vdd_i[i]$ du dernier point de reprise validé R_i^x de P_i. Enfin, soit $\mathbf{PRVD}_{GTF}$ l'ensemble des couples (point de reprise validé demandé, vecteur des dépendances directes). Un point de reprise est automatiquement inséré dans $\mathbf{PRNV}_{GTF}$. Dès que la relation $(vdd_{R_i^x}[i] \leq max\_stable[i])$ est vérifiée, le point de reprise R_i^x est retiré de $\mathbf{PRNV}_{GTF}$ et inséré dans $\mathbf{PRVD}_{GTF}$. Par ailleurs, les parties d'histoires séquentielles antérieures au vecteur $dprv_{GTF}$ sont effacées de la mémoire stable. Cette opération est effectuée toutes les τ^7_{GTF} secondes. $\texttt{fin}(\tau^7_{GTF}, nra)$ commence par le calcul de l'état global recouvrable maximal (*cf.* le 2^{e} niveau).

Les contenus des messages correspondants qui étaient gardés en mémoire volatile sont effacés. Pour cette dernière opération, le GTF insère le vecteur $dprv_{GTF}$ dans les messages $m\_je\_gtf_i^{GTF}$.

La tolérance aux fautes du mécanisme

D'après [Stro88], le fait d'enregistrer en mémoire stable l'histoire répartie rend l'application tolérante aux fautes globales. Dans notre cas, le GTF, passage obligé de la journalisation, ne possède aucune information sensible concernant le mécanisme de journalisation. En effet, si le GTF est fautif alors les processus de l'URR demandent la création d'un nouveau GTF. Le processus créant le premier le fichier dans lequel figurera l'adresse du prochain GTF est responsable de la mise à niveau du nouveau GTF. L'histoire est répartie entre la mémoire stable et la mémoire volatile des nœuds du réseau local. Comme pour l'enregistrement en mémoire stable de l'action de création d'un point de reprise, nous supposons que l'enregistrement des parties de l'histoire répartie est atomique. D'où, le nouveau GTF récupère les parties de l'histoire répartie disponibles dans la mémoire stable et invoque le mécanisme de recouvrement arrière. Le mécanisme de recouvrement arrière commence par récupérer l'histoire répartie qui n'est pas encore enregistrée en mémoire stable. Ensuite, il calcule l'état global recouvrable maximal (*cf.* § 3.2.5.3).

3.2.4.4 Discussion

Elnozahy et Zwaenepoel ont expérimenté plusieurs mécanismes de journalisation et conclu ceci [Elno94] :

- l'enregistrement en mémoire stable de l'histoire répartie est plus efficiente que l'enregistrement en mémoire stable des messages,
- l'enregistrement du contenu des messages en mémoire volatile est plus efficiente que l'enregistrement en mémoire stable,
- l'enregistrement de l'histoire répartie par les processus récepteurs est plus efficiente que l'enregistrement par les processus émetteurs.

Toutes ces conclusions tiennent aussi pour notre mécanisme de journalisation.

Sistla et Welch ont montré que les estampilles des messages comportent au minimum les codes d'identification des processus émetteurs et récepteurs, plus les numéros d'ordre

d'émission et de réception. Ils ont aussi démontré que la complexité de l'algorithme réparti de validation est au mieux en $\mathcal{O}(\mid URR \mid)$ et au pire en $\mathcal{O}(\mid URR \mid^3)$ messages, avec n le nombre de processus de l'URR [Sist89]. Le meilleur des cas est obtenu lorsque le processus émetteur attend que les autres processus l'avertissent de l'état d'avancement de leur journalisation. Le pire des cas est obtenu lorsque le processus demande explicitement aux autres processus de l'avertir de l'état d'avancement de leur journalisation[11]. Remarquons que le mécanisme de Strom et Yemini est inclus dans l'étude de Sistla et Welch : les estampilles sont les vecteurs des dépendances transitives et les processus attendent les vecteurs de journalisation régulièrement diffusés par les autres processus [Stro85]. La journalisation étant centralisée, la taille des estampilles est optimale et la complexité de notre algorithme de validation est en $\mathcal{O}(\mid URR \mid)$ messages.

Le mécanisme de Johnson et Zwaenepoel centralise la journalisation sur un serveur s'exécutant sur le nœud du serveur de gestion répartie des fichiers [John89, John90b]. Ce nœud du réseau local est supposé sûr de fonctionnement par les auteurs. L'algorithme de validation centralisé calcule l'état global recouvrable maximal au fur et à mesure de la journalisation de l'histoire répartie et parcourt toute l'histoire répartie. C'est le mécanisme de journalisation le plus proche du nôtre. Grâce aux vecteurs des dépendances directes journalisés, la validation d'un message ne nécessite pas le parcours de toute l'histoire répartie. Par ailleurs, le GTF n'a pas besoin d'être sûr de fonctionnement. En conséquence, l'application tolère les fautes globales et le GTF peut s'exécuter sur n'importe quel nœud du réseau local. Johnson fait évoluer le mécanisme pour que les processus gardent les vecteurs des dépendances directes [John93]. Lors de la validation, il construit, un vecteur des dépendances transitives à l'aide d'un algorithme réparti [John93]. Certes, le principe de notre mécanisme est le même, mais l'option centralisée est plus efficiente et plus simple.

Avec notre mécanisme, la journalisation pessimiste est obtenue en initialisant les paramètres ν_i^4 et ν_{GTF}^5 à la valeur 1. Remarquons que ce n'est pas une vraie journalisation pessimiste car la réception d'un message n'est pas synchronisée avec sa journalisation. Ici, le terme "pessimiste" est interprété comme "sauvegarde en mémoire stable de l'histoire dès que possible en passant par le GTF". La quantité d'information enregistrée en mémoire stable est petite. Deux messages sont émis avant l'enregistrement en mémoire stable de l'histoire : le message interprocessus pouvant être long, plus un message court. Avec les mécanismes de journalisation pessimistes classiques [Powe83, Borg89, Sens94, Sens95], la quantité d'information enregistrée en mémoire stable est grande et au minimum un message interprocessus est émis. Nous espérons que le surcoût d'exécution engendré par un message court est souvent compensé par le gain à ne pas enregistrer le contenu du message interprocessus. En revanche, notre mécanisme est moins efficient que les mécanismes de journalisation répartie [Alvi93, Elno93, Alvi96]. En effet, nous avons montré dans le chapitre 2 que ces mécanismes combinent les avantages des journalisations pessimiste et optimiste. Néanmoins, ces derniers sont beaucoup plus complexes.

3.2.5 Le mécanisme de recouvrement arrière

Le mécanisme de recouvrement arrière bloque tous les processus de l'URR et exécute les quatre étapes du recouvrement arrière.

Le premier paragraphe motive l'aspect bloquant du mécanisme (*cf.* § 3.2.5.1). Puis, le paragraphe 3.2.5.2 introduit les structures de données des algorithmes. Ensuite, nous développons les quatre étapes des recouvrements arrière : la détection, la reconfiguration, la

11. Les deux cas n'incluent aucunement l'enregistrement en mémoire stable de la fin de l'histoire répartie.

récupération et la reprise (*cf.* § 3.2.5.3). Ces choix sont discutés dans un dernier paragraphe (*cf.* § 3.2.5.4).

3.2.5.1 Un mécanisme de recouvrement arrière bloquant

Le premier objectif du mécanisme de recouvrement arrière est le maintien de la cohérence de l'état global. Puisque la journalisation peut être optimiste, nous devons exécuter un algorithme de calcul de l'état global recouvrable maximal. Dans le chapitre 2, nous avons montré que, pour limiter la propagation des fautes par message orphelin, le calcul doit précéder le début de la ré-exécution. Par conséquent, nous choisissons d'arrêter l'application - en fait, blocage des émissions de message - pour :

1. vider les canaux de communication,
2. calculer l'état global recouvrable maximal,
3. reconfigurer l'URR,
4. récupérer les informations de recouvrement pour la ré-exécution,
5. lancer la ré-exécution.

Le deuxième objectif du mécanisme de recouvrement arrière est d'éviter le bouclage infini des recouvrements arrière. La solution est d'estampiller les messages d'un numéro dit d'incarnation (*cf.* § 2.5.4.2). Nous construisons une variante du numéro d'incarnation. Le but implicite de ce "numéro de recouvrement arrière" est d'autoriser les fautes multiples.

Avant de présenter en détail le mécanisme de recouvrement arrière, nous en donnons une description informelle.

1. La détection :
   (a) Les fautes sont détectées par rupture de canaux de communication. Lorsqu'un processus P_i détecte la faute d'un processus voisin, il avertit le GTF.
   (b) Sur réception d'un avertissement de fautes, le GTF incrémente le numéro de recouvrement arrière et diffuse de manière fiable un ordre d'arrêt des émissions de messages. Toute prochaine détection de faute provoque la ré-exécution de l'algorithme de recouvrement arrière à partir de cette étape.
   (c) Les processus continuent la journalisation de leur histoire séquentielle. Le GTF détecte la fin de la journalisation. Par ailleurs, pour ne pas attendre indéfiniment la journalisation d'un processus devenant fautif, le GTF émet un message au processus n'ayant pas encore journalisé au bout d'un certain temps.
   (d) Si l'utilisateur de l'application ou l'administrateur du système le désirent, ils peuvent exécuter leur propre mécanisme de détection des fautes. Pour cela, avant le début de l'exécution répartie, ils indiquent au GTF l'identité du processus devant être averti des fautes. L'entité externe peut décider de supprimer des processus, et dans ce cas, en avertit le GTF.
   (e) Sur réception de la réponse de l'entité externe de détection des fautes, le GTF exécute l'algorithme de calcul de l'état global recouvrable maximal (*cf.* figure 3.6) et calcule la ligne de reprise.

2. La reconfiguration :

   (a) Dans la plupart des cas, c'est le GTF qui place les nouvelles images des processus défaillants ou orphelins. Si l'utilisateur de l'application ou l'administrateur du système le désirent, ils peuvent exécuter leur propre mécanisme de reconfiguration. Pour cela, avant le début de l'exécution répartie, ils indiquent au GTF l'identité du processus devant être contacté pour la reconfiguration.

   (b) Sur réception de la réponse de l'entité externe de reconfiguration, le GTF diffuse de manière fiable un ordre de reconnexion aux nouvelles images des processus défaillants ou orphelins pour la reconstruction des canaux de communications.

   (c) Les processus concernés acquittent positivement l'ordre de reconnexion.

3. La récupération :

   (a) Conjointement aux informations de reconnexion, le GTF insère les informations de ré-exécution. Ce message contient ainsi l'histoire séquentielle et le nom du fichier contenant les messages demandés.

4. La reprise :

   (a) Le GTF diffuse de manière fiable l'ordre de reprise.

3.2.5.2 Les structures de données, les messages et les temporisations

Les structures de données maintenues par chaque processus P_i

- $\mathbf{D}_i$: ensemble des processus défaillants. $\mathbf{D}_i$ est initialisé à l'ensemble vide. Lorsque $\mathbf{D}_i$ n'est pas égal à l'ensemble vide, cela signifie que l'URR est en cours de recouvrement arrière.

- $\mathbf{ND}_i$: ensemble des triplets (processus non défaillant,*ve*,*vdd*). $\mathbf{ND}_i$ est initialisé à l'ensemble vide.

- $\mathbf{ORPH}_i$: ensemble des triplets (processus orphelin,*ve*,*vdd*). $\mathbf{ORPH}_i$ est initialisé à l'ensemble vide.

- nra_i : numéro de recouvrement arrière ou numéro de l'exécution du mécanisme de recouvrement arrière. nra_i est initialisé à 0. nra_i prend la valeur de nra_{GTF} transmise par les messages $m\_ra\_arret_i^{GTF}$. P_i insère nra_i dans les messages qu'il émet (y compris les messages inter-processus) et initialise les temporisations avec nra_i. P_i exécute une action de réception d'un message uniquement si le numéro fourni par le message est supérieur ou égal à nra_i. De même, P_i exécute une action de fin de temporisation uniquement si le numéro associé est supérieur ou égal à nra_i. C'est ainsi une variante du numéro d'incarnation.

Les structures de données maintenues par le GTF

- $\mathbf{D}_{GTF}$: ensemble des processus défaillants. $\mathbf{D}_{GTF}$ est initialisé à l'ensemble vide. Lorsque $\mathbf{D}_{GTF}$ n'est pas égal à l'ensemble vide, cela signifie que l'URR est en cours de recouvrement arrière.

- $\mathbf{ND}_{GTF}$: ensemble des triplets (processus non défaillant,*ve*,*vdd*). $\mathbf{ND}_{GTF}$ est initialisé à l'ensemble vide.

- $\mathbf{ORPH}_{GTF}$: ensemble des triplets (processus orphelin,*ve*,*vdd*). $\mathbf{ORPH}_{GTF}$ est initialisé à l'ensemble vide.
- $\mathbf{NVEAU}_{GTF}$: ensemble des processus défaillants ou orphelins pour lesquels la nouvelle image est recréée. $\mathbf{NVEAU}_{GTF}$ est initialisé à l'ensemble vide.
- $\mathbf{PRET}_{GTF}$: ensemble des processus prêts pour la reprise. $\mathbf{PRET}_{GTF}$ est initialisé à l'ensemble vide.
- $\mathbf{R}_{GTF}$: ensemble des processus ayant commencé la ré-exécution. $\mathbf{R}_{GTF}$ est initialisé à l'ensemble vide.
- $velr$: vecteur d'émission de la ligne de reprise. Toutes les composantes de $velr$ sont initialisées à 0.
- nra_{GTF} : nombre d'exécutions des actions RECEVOIR($m\_ra\_avert^*_{GTF}$). nra_{GTF} est initialisé à 0. Comme les processus de l'URR, le GTF insère nra_{GTF} dans les messages qu'il émet et les temporisations qu'il initialise. De même, il exécute les actions de réception de message et de fin de temporisation uniquement si le numéro fourni est supérieur ou égal à nra_{GTF}.
- $déjà\_fait1$ et $déjà\_fait2$: variables booléennes utilisées lorsqu'une entité externe intervient pour, respectivement, la détection et la reconfiguration.

Les messages

- $m\_ra\_avert^*_{GTF}$: avertissement de fautes émis à destination du GTF.
- $m\_ra\_arret_i^{GTF}$: ordre d'arrêt des émissions de messages inter-processus, ceci suite à l'avertissement de fautes $m_ra_faute^i_{GTF}$d.
- $m\_ra\_acq^i_{GTF}$: acquittement de l'ordre d'arrêt des émissions $m\_ra\_arret_i^{GTF}$.
- $m\_ra\_je_i^{GTF}$: demande de journalisation aux processus n'ayant pas terminé leur journalisation.
- $m\_ra\_detect\_ext_{GTF}$: fin de détection des fautes par une entité externe.
- $m\_ra\_pr_i^{GTF}$: ordre de constitution de points de reprise non coordonnés.
- $m\_ra\_notif^i_{GTF}$: notification de constitution d'un point de reprise non coordonné.
- $m\_ra\_reconf\_ext_{GTF}$: fin de reconfiguration par une entité externe.
- $m\_ra\_recup_i^{GTF}$: transmission des informations pour la récupération des messages demandés.
- $m\_ra\_reprise_i^{GTF}$: ordre de reprise.

Les temporisations

- τ^9_{GTF}: durée maximale entre deux émissions de messages $m\_ra\_arret_i^{GTF}$.
- τ^{10}_{GTF}: durée maximale entre deux demandes de journalisation $m\_ra\_je_i^{GTF}$.
- τ^{11}_{GTF}: durée maximale d'attente du message $m\_ra\_detect_{GTF}$.

- τ_{GTF}^{12}: durée maximale entre deux détections par un `ping`(...) de la défaillance des processus en constitution de points de reprise.

- τ_{GTF}^{13}: durée maximale d'attente du message $m\_ra\_reconf\_ext_{GTF}$.

- τ_{GTF}^{15}: durée maximale entre deux détections par un `ping`(...) de la défaillance des processus en récupération.

- τ_{GTF}^{16}: durée maximale entre deux demandes de reprise $m\_ra\_reprise_i^{GTF}$.

- τ_i^{17}: durée maximale entre deux détections par un `ping`(...) de la défaillance du GTF.

3.2.5.3 Les quatre étapes des recouvrements arrière

Le recouvrement arrière de l'URR à la suite d'occurrences de fautes est effectué en quatre étapes. Ensuite, nous étudions la constitution des points de reprise et la journalisation pendant la ré-exécution. Nous terminons le paragraphe avec la tolérance aux fautes du mécanisme de recouvrement arrière.

1re étape : La détection

Les fautes sont détectées par rupture de canaux de communication. Lorsqu'un processus détecte la faute d'un processus voisin, il avertit le GTF avec le message $m\_ra\_avert_{GTF}^{i}$. Si c'est le GTF qui détecte une rupture de connexion, il émet le message $m\_ra\_avert_{GTF}^{GTF}$. Suite à cette action, le GTF incrémente nra_{GTF}, ajoute le nouveau processus défaillant à l'ensemble $\mathbf{D}_{GTF}$ et diffuse de manière non fiable un ordre d'arrêt $m\_ra\_arret_i^{GTF}$ des émissions de messages. Le GTF insère nra_{GTF} et l'identification du processus défaillant dans le message $m\_ra\_arret_i^{GTF}$. Sur réception de cet ordre, les processus non défaillants acquittent positivement par un message $m\_ra\_acq_{GTF}^{i}$ et continuent leur exécution jusqu'à leur prochaine émission de message. Le message $m\_ra\_acq_{GTF}^{i}$ contient les vecteurs d'émission ve_i et des dépendances directes vdd_i. Le GTF ré-émet sélectivement l'ordre d'arrêt aux processus n'ayant pas acquitté l'ordre d'arrêt au bout de τ_{GTF}^{9} secondes. Toute prochaine détection de faute provoque la ré-exécution de l'algorithme de recouvrement arrière à partir du début - à partir de l'action `RECEVOIR`($m\_ra\_avert_{GTF}^{*}$). Par conséquent, à la fin de cette première phase, le GTF connaît l'ensemble $\mathbf{D}_{GTF}$ des processus défaillants et l'ensemble $\mathbf{ND}_{GTF}$ des processus non défaillants respectivement à la dernière faute détectée.

La figure 3.5 visualise le mécanisme de recouvrement arrière pour une URR composée de deux processus. Le processus P_2 connaît la faute du processus P_1 à l'instant T_8. Le GTF diffuse le message $m\_ra\_arret_i^{GTF}$ (à l'instant T_9).

Lorsque tous les processus ont acquitté l'ordre d'arrêt (à l'instant T_{10}), l'exécution répartie est considérée comme arrêtée. Avant de commencer le calcul de l'état global recouvrable maximal, le GTF attend que les canaux de communication se vident. Pour ce faire, les processus continuent la journalisation de leur histoire séquentielle. Le GTF détecte la fin de la journalisation par la vérification de la condition $(\forall i \in \mathbf{ND}_{GTF}, (\forall (j \neq i) \in \mathbf{ND}_{GTF}, vdd_i[j] \geq ve_j[i])$. Par ailleurs, pour ne pas attendre indéfiniment la journalisation d'un processus devenant fautif, le GTF émet un message $m\_ra\_je_i^{GTF}$ à tous les processus

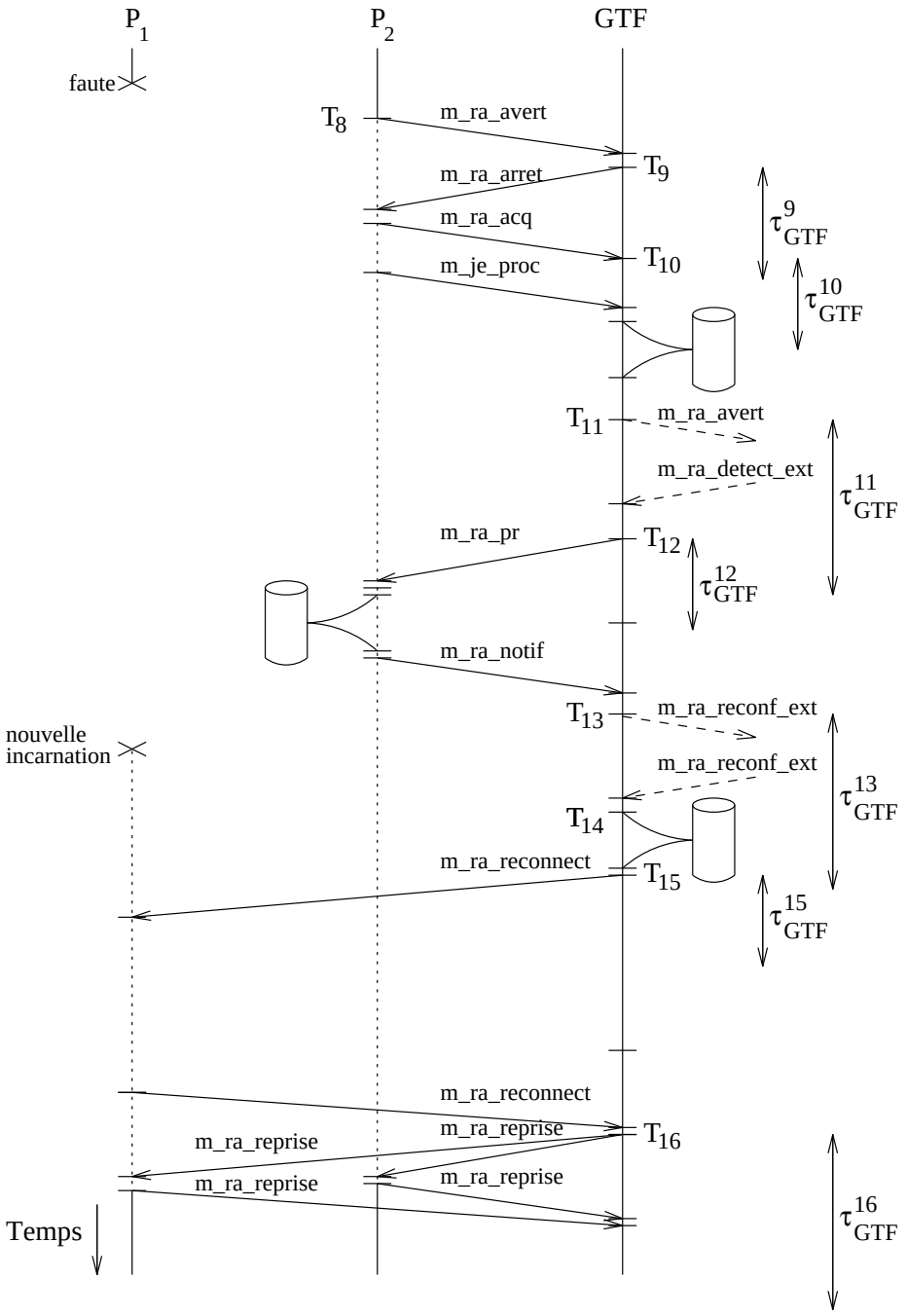


FIG. 3.5 – Les quatre étapes du recouvrement arrière des processus de l'unité de reprise répartie.

n'ayant pas terminé leur journalisation au bout de τ^{10}_{GTF} secondes. Enfin, le GTF enregistre en mémoire stable l'histoire répartie.

Si l'utilisateur de l'application ou l'administrateur du système le désirent, ils peuvent exécuter leur propre mécanisme de détection des fautes. Pour cela, avant le début de l'exécution répartie, ils indiquent au GTF l'identité du processus devant être averti des fautes. Le message d'avertissement (à l'instant T_{11}) précise les nœuds du réseau local et les processus défaillants ainsi que les nœuds et les processus non défaillants. Le GTF

attend un message de fin de détection $m\_ra\_detect\_ext_{GTF}$ en provenance du processus "détecteur de fautes" pendant τ^{11}_{GTF} secondes. L'entité externe peut décider de supprimer des processus, et dans ce cas, en avertit le GTF. La variable booléenne $déjà\_fait1$ permet d'exécuter soit l'action `RECEVOIR`($m\_ra\_detect\_ext_{GTF}$) soit l'action `fin`(τ^{11}_{GTF}, nra).

Puisque la journalisation de l'exécution répartie peut être optimiste, le GTF calcule l'état global recouvrable maximal. La figure 3.6 présente l'algorithme de calcul de l'état global recouvrable maximal. Les notations sont les suivantes :

- *vd_dim* est le nombre de processus composant l'URR,
- *vap* et *vdd* sont les vecteurs utilisés pour le calcul du dernier message reçu stable,
- *continuer* est la variable booléenne indiquant si le calcul est terminé,
- *nor* et *noe* sont les numéros d'ordre de réception et d'émission du prochain message reçu à analyser,
- I et J sont des processus de l'URR.

L'algorithme de calcul de l'état global recouvrable maximal consiste en deux parcours de l'histoire répartie.

Premièrement, l'histoire répartie est parcourue dans le sens décroissant du temps, par sauts successifs en partant du dernier message reçu. Ce premier parcours recherche les vecteurs des dépendances directes et calcule les vecteurs des dépendances transitives correspondants. Les vecteurs des dépendances directes des derniers messages reçus sont obtenus par les appels à la procédure `Dernier_vdd`. Les vecteurs des dépendances transitives sont calculés par la fonction `Vd_transitif` appelée à la ligne 20 (*cf.* l'algorithme de validation pour le détail de cette fonction). Si un vecteur des dépendances transitives n'est pas calculable, cela signifie que le message associé au vecteur des dépendances directes est orphelin. Le parcours de l'histoire répartie continue avec le vecteur des dépendances directes précédant celui qui n'a pas pu être calculé (*cf.* ligne 25). Le parcours continue jusqu'à trouver un vecteur des dépendances directes correspondant à un message non orphelin. Puisque les vecteurs des dépendances directes sont ceux des instants de journalisation, tous les messages reçus ne sont pas examinés. D'où, le vecteur des dépendances transitives calculé est inférieur ou égal au vecteur repérant l'état global recouvrable maximal.

Deuxièmement, l'histoire répartie est parcourue dans le sens croissant du temps, pas à pas en partant de max_stable. Ce deuxième parcours recherche les prochains messages reçus par les processus de l'URR et regarde si ces messages sont orphelins. L'estampille complète du prochain message reçu est donnée par la fonction `Suivant_reçu` (*cf.* ligne 44). S'il n'existe aucun message reçu après l'état repéré par le vecteur max_stable, alors la fonction `Suivant_reçu` renvoie $nor = max\_stable[I] + 1$, $J = I$ et $noe = max\_stable[I] + 1$. D'où, l'état n'évolue pas (*cf.* lignes 50 et 51). Le parcours de l'histoire séquentielle s'interrompt au premier message non encore calculé stable (*cf.* ligne 56). Le parcours de l'histoire répartie s'arrête lorsqu'aucun nouveau message n'est déclaré non orphelin par la dernière itération (*cf.* ligne 37, 54 et 61).

À la suite du calcul de l'état global recouvrable maximal, le GTF exécute l'action `fin`(τ^{7}_{GTF}, nra) pour calculer la ligne de reprise repérée par $dprv_{GTF}$. En outre, il calcule l'ensemble $\mathbf{ORPH}_{GTF}$ des processus orphelins. Un processus orphelin P_i vérifie la condition $(vdd_i[i] > max\_stable[i])$. Ensuite, le GTF diffuse de manière fiable un ordre de constitution de points de reprise non coordonnés $m\_ra\_pr_i^{GTF}$ (à l'instant T_{12}). Le GTF

```
00 Procedure État_maximal ()
01 {
02   vecteur vdd[vd_dim][vd_dim];
03   vecteur vap[vd_dim][vd_dim];
04   booleen continuer;
05   entier nor, noe;
06   processus I, J;
07   /* parcours dans le sens décroissant
08    du temps de l'histoire répartie */
09   Pour I de 1 a vd_dim Faire
10     Pour J de 1 a vd_dim Faire
11       vap[I][J] := vrai;
12     Finpour;
13     vap[I][I] := faux;
14   Finpour;
15   /* en partant du dernier message recu */
16   Pour I de 1 a vd_dim Faire
17     Dernier_vdd (I, &vdd[I]);
18   Finpour;
19   Pour I de 1 a vd_dim Faire
20     Tantque ¬ Vd_transitif (&vdd[I], &vap[I])
21      Faire
22       /* prendre le précédent vdd
23        différent du vdd courant,
24        d'où, saut en arrière dans l'histoire répartie */
25       Précédent_vdd (I);
26     Fintantque;
27     /* message vdd[I][I] non orphelin,
28      d'où, mise à jour de max_stable */
29     Pour J de 1 a vd_dim Faire
30       Si max_stable[J] < vdd[I][J] Alors
31         max_stable[J] := vdd[I][J];
32       Finsi;
33     Finpour;
34   Finpour;
35   /* parcours dans le sens croissant
36    du temps de l'histoire répartie */
37   continuer := faux;
38   Repeter
39     Pour I de 1 a vd_dim Faire
40       Repeter
41         /* en partant de max_stable,
42          aller jusqu'au prochain
43          message recu m par l */
44         nor := Suivant_recu (I,
45          max_stable[I],
46          &J, &noe);
47         /* avancer max_stable jusqu'au
48          numéro de communication
49          précédent la réception de m */
50         max_stable[I] := nor - 1;
51         Si noe ≤ max_stable[J] Alors
52           /* m est stable */
53           max_stable[I] ++;
54           continuer := vrai;
55         Finsi;
56       Jusqu'a noe > max_stable[J];
57       /* le prochain message m recu par l
58        sera à valider dans une
59        prochaine itération */
60     Finpour;
61   Jusqu'a continuer == faux;
62 }
```

FIG. 3.6 – L'algorithme de calcul de l'état global recouvrable maximal.

surveille par des **ping**(...) toutes les (τ^{12}_{GTF}) secondes les processus en constitution, et ce, jusqu'à la fin des constitutions. Les processus acquittent les points de reprise par le message $m\_ra\_pr^i_{GTF}$. Ces processus sont donc prêts pour la reprise; ils sont ajoutés à l'ensemble $\mathbf{PRET}_{GTF}$.

2e étape : La reconfiguration

Dans la plupart des cas, c'est le GTF qui place les nouvelles images des processus défaillants ou orphelins. Si l'utilisateur de l'application ou l'administrateur du système le désirent, ils peuvent exécuter leur propre mécanisme de reconfiguration. Pour cela, avant le début de l'exécution répartie, ils indiquent au GTF l'identité du processus devant être contacté pour la reconfiguration. Le message de reconfiguration émis par le GTF (à l'instant T_{13}) précise les processus devant être recréés. Le GTF attend un message de fin de reconfiguration $m\_ra\_reconf\_ext_{GTF}$ du processus "placeur" pendant τ^{13}_{GTF} secondes. $m\_ra\_reconf\_ext_{GTF}$ indique la localisation des nouvelles images. Le GTF enregistre en mémoire stable la nouvelle configuration de l'URR. Le GTF redémarre les processus non prêts (à l'instant T_{14}) - les processus non placés par l'entité externe et les processus orphelins. Dans la figure 3.5, c'est le processus "placeur" extérieur à l'application qui installe la nouvelle incarnation du processus P_1. La variable booléenne $déjà\_fait2$ permet d'exécuter soit l'action RECEVOIR($m\_ra\_reconf\_ext_{GTF}$) soit l'action **fin**(τ^{13}_{GTF}, nra). Les processus ayant effectué la reconfiguration sont ajoutés à l'ensemble $\mathbf{NVEAU}_{GTF}$.

Le GTF diffuse (à l'instant T_{15}) un ordre de reconnexion $m\_ra\_reconnect^{GTF}_i$ aux nou-

velles images des processus défaillants ou orphelins pour la reconstruction des canaux de communications. Les processus concernés acquittent positivement l'ordre de reconnexion. Le GTF surveille par des `ping`(...) toutes les (τ^{15}_{GTF}) secondes les processus en reconnexion.

3^{e} étape : La récupération

Conjointement aux informations de reconnexion, le GTF insère les informations de ré-exécution dans le message $m\_ra\_reconnect_i^{GTF}$. $m\_ra\_reconnect_i^{GTF}$ contient ainsi l'histoire séquentielle et le nom du fichier contenant les messages demandés. Les processus ayant effectué la récupération sont ajoutés à l'ensemble $\mathbf{PRET}_{GTF}$. La reprise commence lorsque $\mathbf{PRET}_{GTF}$ contient tous les processus de l'URR.

4^{e} étape : La reprise

Le GTF diffuse de manière non fiable l'ordre de reprise $m\_ra\_reprise_i^{GTF}$ (à l'instant T_{16}). Ce message de reprise contient le vecteur $dprv_{GTF}$, plus le vecteur d'émission de la ligne de reprise $velr_{GTF}$. $velr_{GTF}$ est défini par la relation $velr_{GTF}[i] = ve\_dprv_i[i]$, avec ve_dprv_i le vecteur d'émission du processus P_i correspondant à la ligne de reprise. Les processus acquittent positivement l'ordre de reprise. Le GTF reprend individuellement les processus ne répondant pas au bout de τ^{16}_{GTF} secondes. Les ensembles **D**, **ND** et **ORPH** sont ré-initialisés à l'ensemble vide.

La tolérance aux fautes du mécanisme

Toute détection de faute durant l'exécution du mécanisme de recouvrement arrière provoque l'émission d'un message $m\_ra\_avert^{*}_{GTF}$ et par conséquent l'exécution de l'action `RECEVOIR`($m\_ra\_avert_i^{GTF}$). Le mécanisme se ré-exécute entièrement. Afin de détecter la faute du GTF pendant les recouvrements arrière, les processus de l'URR vérifient toutes les τ_i^{17} secondes que le GTF n'est pas défaillant. Si le GTF devient fautif alors un processus de l'URR crée un nouveau GTF. L'état global recouvrable maximal est celui calculé à partir de l'histoire répartie enregistrée en mémoire stable par l'ancien GTF. Dans la figure 3.5, pour ne pas surcharger trop le schéma, les temporisations τ_i^{17} ne sont pas tracées.

La ré-exécution

Pendant la ré-exécution, les processus constituent des points de reprise et journalisent leur exécution comme pendant l'exécution. Par contre, les processus ne ré-émettent pas les messages à destination d'un processus P_i tels que $noe \leq dprv_{GTF}[i]$. Il en est de même pour les messages à destination du monde extérieur. Les processus prennent en mémoire stable les messages en provenance du processus P_i tels que $nor \leq velr[i]$.

3.2.5.4 Discussion

Grâce à la présence du GTF, la complexité en nombre de messages de notre algorithme est de 10 $\mid URR \mid$ dans le pire des cas. Notre algorithme nécessite l'étape de détection. Il

en est de même pour les politiques à journalisation répartie [Alvi93, Elno93]. En revanche, les politiques à journalisation pessimiste n'en ont pas besoin [Powe83, Borg89, Sens95].

Les politiques à journalisation pessimiste n'ont pas besoin de l'étape de détection [Powe83, Borg89, Sens95]. Parce qu'il ne distingue pas ce cas particulier de journalisation, notre algorithme nécessite quand même l'étape de détection. Il en est de même pour les politiques à journalisation répartie [Alvi93, Elno93].

Sistla et Welch ont montré que les algorithmes de calcul de l'état global recouvrable maximal possèdent une complexité au pire en $\mathcal{O}(\mid URR \mid^3)$ et au mieux en $\mathcal{O}(\mid URR \mid^2$), selon que les estampilles des messages émis contiennent seulement le numéro d'ordre d'émission ou le vecteur des dépendances transitives [Sist89]. Pour pallier l'inconvénient de la complexité, l'algorithme de Johnson et Zwaenepoel est centralisé, mais ne tolère pas les fautes du serveur de reprise central [John89]. Notre algorithme est lui aussi centralisé mais, d'une part, nous tolérons les fautes du GTF, d'autre part, nous utilisons les vecteurs des dépendances directes pour accélérer le calcul de l'état global recouvrable maximal.

Enfin, l'aspect bloquant du mécanisme de recouvrement arrière permet la prise en compte de mécanismes de détection et de reconfiguration spécifiques à l'utilisateur ou à l'administrateur du système.

La reconfiguration est classique, si l'on excepte la possibilité de déléguer le placement des processus à une entité extérieure à l'application.

La récupération des informations pour la ré-exécution est elle aussi classique. Nous profitons de la présence du GTF pour regrouper les messages demandés par processus. Rappelons que les politiques à journalisation répartie obligent l'utilisation d'un algorithme de récupération réparti [Alvi93, Elno93].

3.3 Les extensions au mécanisme global

Par défaut, le mécanisme global décrit dans la section précédente permet une ré-exécution équivalente et supporte des applications p-déterministes. Il prend en compte la plupart des optimisations possibles améliorant l'efficacité. Dans cette section, deux extensions sont étudiées qui améliorent encore l'efficience et agrandissent le champ d'action du recouvrement arrière.

Le traitement de la sémantique des transmissions de messages consiste à classer les messages en quatre catégories : exactement une fois, au moins une fois, au plus une fois et sans contrainte (*cf.* sous-section 2.2.2). Jusqu'à présent, la sémantique des transmissions des messages était exactement une fois. Aussi, les trois mécanismes de base (constitution des points de reprise, journalisation de l'exécution et recouvrement arrière) sont étendus pour supporter les trois autres sémantiques (*cf.* sous-section 3.3.1).

L'étude des ré-exécutions non équivalentes est prolongée dans une deuxième extension (*cf.* sous-section 3.3.2). Cette fois, les processus de l'application sont qualifiés par l'utilisateur de p-déterministes ou indéterministes. Les mécanismes de base sont étendus pour accepter la non-journalisation par les processus indéterministes. Toutefois, le mécanisme global est assez souple pour que les modifications soient restreintes. D'une manière générale, l'indéterminisme d'exécution coûte cher lors des validations de messages vers le monde extérieur et lors des recouvrements arrière. Cependant, en utilisant conjointement le traitement des sémantiques des transmissions de messages, nous montrons que la cohabitation de processus p-déterministes avec des processus indéterministes est réalisable à un

coût raisonnable. Ce coût est supporté par l'utilisateur qui doit programmer la tolérance aux fautes de son application. Nous montrons comment rendre cette programmation aisée.

3.3.1 La sémantique des transmissions de messages

La sémantique (des transmissions de messages) est spécifiée par un champ ajouté à l'estampille des messages. Tout d'abord, la sémantique est définie en relation avec les définitions des messages manquant et orphelin. Elle influence donc la cohérence de la ligne de reprise (*cf.* § 3.3.1.1). Par déduction, tous les mécanismes de base du recouvrement arrière doivent être étendus pour tenir compte de ce nouveau paramètre (*cf.* § 3.3.1.2, 3.3.1.3 et 3.3.1.4). Les choix de réalisation de cette extension sont discutés dans un dernier paragraphe (*cf.* § 3.3.1.5).

3.3.1.1 Les définitions préalables

Un message est *manquant* si l'action *émettre* du message existe dans l'histoire répartie, mais pas l'action *recevoir* qui lui correspond (*cf.* § 2.2.2.4).

Un message est *orphelin* si l'action *recevoir* du message existe dans l'histoire répartie, mais pas l'action *émettre* qui lui correspond (*cf.* § 2.2.2.4).

Le tableau 3.1 donne la signification des différentes sémantiques par rapport à la définition de la cohérence de la ligne de reprise [Brze95].

Sémantique	$\texttt{cohérence}(\widehat{R})$
Exactement une fois	Vrai s'il n'existe aucun message orphelin ou manquant
Au moins une fois	Vrai s'il n'existe aucun message manquant
Au plus une fois	Vrai s'il n'existe aucun message orphelin
Sans contrainte	Toujours vrai

TAB. 3.1 – La cohérence de la ligne de reprise en fonction de la sémantique des transmissions de messages.

3.3.1.2 Le mécanisme de constitution des points de reprise

La prise en compte de la sémantique modifie la définition des messages demandés, et par suite, la définition de la cohérence de la constitution d'un point de reprise.

Les messages demandés sont les messages devenant manquants à la suite de la faute des processus récepteurs. Les processus les gardent en mémoire volatile jusqu'à ce qu'ils les enregistrent en mémoire stable lors des constitutions de points de reprise. Puisqu'ils ne rendent pas la ligne de reprise incohérente, les messages à sémantique au plus une fois ou sans contrainte ne sont jamais demandés. Nous en déduisons le tableau 3.2.

À partir du tableau 3.2 et de la première définition du paragraphe 2.4.1.1, vient immédiatement la définition de la cohérence de la constitution d'un point de reprise en fonction de la sémantique des transmissions de messages (*cf.* tableau 3.3).

3.3.1.3 Le mécanisme de journalisation

Jusqu'à présent, l'histoire répartie est journalisée afin d'obtenir une ré-exécution équivalente et afin de détecter les messages orphelins. Pour ce faire, le mécanisme de jour-

Sémantique	demandé(m)
Exactement une fois	Identiques à celles de [Stro88]
Au moins une fois	Identiques à celles de [Stro88]
Au plus une fois	Toujours faux
Sans contrainte	Toujours faux

TAB. 3.2 – La définition d'un message demandé en fonction de la sémantique des transmissions de messages.

Sémantique	Conditions
Exactement une fois	Identiques à celles du paragraphe 2.2.2.3
Au moins une fois	Identiques à celles du paragraphe 2.2.2.3
Au plus une fois	Toujours vrai
Sans contrainte	Toujours vrai

TAB. 3.3 – La cohérence de la constitution d'un point de reprise en fonction de la sémantique des transmissions de messages.

nalisation utilise des numéros d'ordre d'émission et de réception. Il enregistre aussi le contenu des messages en mémoire volatile des nœuds sur lesquels s'exécutent les processus émetteurs. Enfin, il valide les messages à destination du monde extérieur.

Le numéro d'ordre de communication *noc* est incrémenté lors de l'émission d'un message exactement une fois. Le but est de replacer l'émission dans la ré-exécution. En outre, *noc* est incrémenté lors de la réception d'un message exactement une fois. Le but est de détecter *a posteriori* les messages orphelins. D'après le tableau 3.1, les messages à sémantique sans contrainte ne peuvent être ni manquants ni orphelins. D'où, ils n'ont pas besoin d'être repérés dans l'histoire répartie. Donc, les numéros d'ordre de communication des processus ne sont incrémentés ni lors de l'émission ni lors de la réception de messages sans contrainte. Par ailleurs, ces messages n'ont pas besoin de posséder d'estampilles. Par conséquent, nous pouvons construire le tableau 3.4.

Sémantique	Opérations sur les émissions et les réceptions
Exactement une fois	$\text{noc}_{\text{émetteur}}++$; noe := $\text{noc}_{\text{émetteur}}$; $\text{noc}_{\text{récepteur}}++$; nor := $\text{noc}_{\text{récepteur}}$;
Au moins une fois	$\text{noc}_{\text{émetteur}}++$; noe := $\text{noc}_{\text{émetteur}}$; $\text{noc}_{\text{récepteur}}++$; nor := $\text{noc}_{\text{récepteur}}$;
Au plus une fois	$\text{noc}_{\text{émetteur}}++$; noe := $\text{noc}_{\text{émetteur}}$; $\text{noc}_{\text{récepteur}}++$; nor := $\text{noc}_{\text{récepteur}}$;
Sans contrainte	

TAB. 3.4 – Le calcul des numéros d'ordre d'émission et de réception en fonction de la sémantique des transmissions de messages.

Selon le tableau 3.2, les messages au plus une fois ou sans contrainte ne sont jamais manquants. D'où il s'ensuit qu'ils ne sont pas gardés en mémoire volatile par le processus émetteur (*cf.* tableau 3.5).

La fonction `Vd_transitif` décrite dans la figure 3.3 construit le vecteur des dépendances transitives à partir du vecteur des dépendances directes. Or, d'après le tableau 3.1, les messages au moins une fois ou sans contrainte ne sont jamais orphelins. Déjà, le tableau 3.4 indique que l'histoire répartie ne contient pas les opérations d'émission et de

Sémantique	Contenu en mémoire volatile
Exactement une fois	Oui
Au moins une fois	Oui
Au plus une fois	Non
Sans contrainte	Non

TAB. 3.5 – L'enregistrement en mémoire volatile du contenu des messages en fonction de la sémantique des transmissions de messages.

réception des messages sans contrainte. Le vecteur des dépendances transitives calculé par la fonction `Vd_transitif` contient encore les dépendances vis-à-vis des messages au moins une fois. Par conséquent, le prédicat `Orphelin` est trop contraignant.

Pour ne pas tenir compte des messages au moins une fois, le calcul du vecteur des dépendances directes est modifié pour ne pas inclure les dépendances vis-à-vis des messages à sémantique au moins une fois (*cf.* tableau 3.6).

Sémantique	Opérations sur réception
Exactement une fois	vdd[i] := $\text{nor}_{m_j^i}$;
Au moins une fois	
Au plus une fois	vdd[i] := $\text{nor}_{m_j^i}$;
Sans contrainte	

TAB. 3.6 – Le calcul du vecteur des dépendances directes en fonction de la sémantique des transmissions de messages.

Les messages à destination ou en provenance du monde extérieur possèdent eux aussi une sémantique de transmission. D'après le tableau 3.5, les premiers ne sont pas gardés en mémoire stable si la sémantique est au plus une fois ou sans contrainte. D'après le tableau 3.6, les derniers ne sont pas stables si la sémantique est au moins une fois ou sans contrainte. Nous en déduisons le tableau 3.7.

Sémantique	Validation sur émission	Contenu en mémoire stable sur réception
Exactement une fois	Oui	Oui
Au moins une fois	Non	Oui
Au plus une fois	Oui	Non
Sans contrainte	Non	Non

TAB. 3.7 – Les émissions à destination et les réceptions en provenance du monde extérieur en fonction de la sémantique des transmissions de messages.

3.3.1.4 Le mécanisme de recouvrement arrière

En ce qui concerne le mécanisme de recouvrement arrière, la prise en compte de la sémantique des transmissions de messages modifie l'étape de détection.

Le GTF calcule l'état global recouvrable maximal. Ce calcul doit respecter les conditions de cohérence de la ligne de reprise (*cf.* tableau 3.1). Il est clair que la définition de la cohérence de l'état global recouvrable maximal suit les mêmes conditions (*cf.* tableau 3.8).

Le GTF débute le recouvrement arrière en attendant que tous les processus de l'URR n'évoluent plus. Le GTF s'assure que l'histoire des messages devant participer au calcul de

Sémantique	Conditions
Exactement une fois	Vrai s'il n'existe aucun message orphelin ou manquant
Au moins une fois	Vrai s'il n'existe aucun message manquant
Au plus une fois	Vrai s'il n'existe aucun message orphelin
Sans contrainte	Toujours vrai

TAB. 3.8 – La cohérence de l'état global recouvrable maximal en fonction de la sémantique des transmissions de messages.

l'état global recouvrable maximal est enregistrée en mémoire stable. D'après le tableau 3.8, cela concerne les messages devant ne pas être orphelins. Or, l'attente est contrôlée par les vecteurs d'émission. Par conséquent, les vecteurs d'émission ne sont pas mis à jour pour les émissions de messages au moins une fois ou sans contrainte (*cf.* tableau 3.9).

Sémantique	Opérations sur émission
Exactement une fois	ve[j] := $\text{noe}_{m_j^i}$;
Au moins une fois	
Au plus une fois	vdd[j] := $\text{nor}_{m_j^i}$;
Sans contrainte	

TAB. 3.9 – Le calcul du vecteur d'émission en fonction de la sémantique des transmissions de messages.

La première partie de la procédure `État_maximal` de la figure 3.6 utilise les vecteurs des dépendances directes. Puisque les vecteurs des dépendances directes prennent en compte les messages exactement une fois ou au plus une fois, l'état *max_stable* est une approximation correcte de l'état global recouvrable maximal.

En revanche, la deuxième partie de la procédure `État_maximal` est un parcours de l'histoire répartie dans le sens croissant du temps, pas à pas en partant de l'état *max_stable*. Ce parcours s'arrête pour un processus P_i au premier message reçu non encore calculé stable. D'où, le parcours s'arrête aussi sur un message reçu à sémantique au moins une fois. Par conséquent, le test de la ligne 51 de la figure 3.6 est trop contraignant. Il est modifié pour devenir :

$$(\text{noe} \leq \mathit{max\_stable}[\text{J}]) \vee (\text{sémantique}(\text{nor}, \text{I}) = \text{au\_moins\_1\_fois}).$$

D'après le tableau 3.2, les opérations d'émission et de réception des messages au plus une fois ou sans contrainte ne doivent pas être ré-exécutées. Par ailleurs, les messages au moins une fois peuvent être reçus plusieurs fois. Les opérations d'émission et de réception des messages pendant la ré-exécution sont résumées dans le tableau 3.10.

Sémantique	Émission pendant la ré-exécution	Réception pendant la ré-exécution
Exactement une fois	Oui	Oui
Au moins une fois	Oui	Oui
Au plus une fois	Non	Non
Sans contrainte	Non	Non

TAB. 3.10 – La ré-exécution des opérations d'émission et de réception en fonction de la sémantique des transmissions de messages.

3.3.1.5 Discussion

Dans [Manc89], Mancini et Shrivastava continuent leur recherche sur la dualité du modèle des conversations avec le modèle des objets et actions. Les auteurs sont convaincus que, et montrent comment, les techniques et les mécanismes d'un modèle sont transférables et applicables à l'autre domaine. Les travaux qui suivent, y compris le nôtre, poursuivent la même idée.

Lin et Ahamad étudient le recouvrement arrière pour les systèmes répartis basés sur les objets [Lin90]. Les ressources sont placées dans des objets communiquant par invocation. Les invocations sont classées en deux catégories selon qu'elles modifient ou consultent seulement les données de l'objet. Chaque objet O maintient deux listes. L'une repère les objets devant constituer un point de reprise si O constitue un point de reprise. L'autre repère les objets devant se ré-exécuter si O se ré-exécute. Un objet peut décider de provoquer la constitution synchronisée d'un point de reprise si une opération importante doit être validée. L'invocation d'un objet est un appel de procédure à distance et se concrétise par deux messages : l'appel et le retour. En outre, le processus appelant est bloqué dans l'attente de la réponse.

La sémantique étudiée par les auteurs est la sémantique des traitements de messages. Il est cependant possible d'établir une correspondance entre les deux types de sémantiques : transmission et traitement. Supposons comme dans [Lin90] :

1. que les constitutions de points de reprise sont synchronisées,
2. que tous les processus dépendant d'un processus défaillant se ré-exécutent aussi,
3. que les processus qui se ré-exécutent effacent leur histoire séquentielle depuis le dernier point de reprise avant la ré-exécution.

Les messages d'appel et de retour d'une invocation de modification possèdent une sémantique exactement une fois. Le message d'appel d'une invocation de consultation n'a pas besoin d'apparaître dans l'histoire répartie. Il possède donc une sémantique sans contrainte. Enfin, le message de retour d'une invocation de consultation doit ou non apparaître dans l'histoire répartie selon que l'état de l'objet appelé a été modifié ou non depuis le dernier point de reprise. Il possède donc respectivement une sémantique exactement une fois et au moins une fois. Le mécanisme de recouvrement arrière est alors similaire à celui de Koo et Toueg [Koo87], appliqué à la gestion d'objets.

Leong et Agrawal élargissent le travail de Lin et Ahamad au système réparti à base de communication par messages[Leon94]. Les changements d'états des processus sont divisés en deux types. Les changements d'états dits explicites sont les changements de contenu du processus. Les changements d'état dits implicites sont l'obligation d'émettre un message en réponse au message traité. Les messages sont ensuite classés en deux catégories selon qu'ils sont significatifs ou non dans l'exécution répartie. Grossièrement, les messages non significatifs sont des messages :

1. qui ne provoquent pas de changements d'état explicites ou implicites, ou,
2. qui provoquent des changements d'état explicites sans changement d'état implicite, ou encore,
3. qui provoquent deux changements d'état explicites mais non implicites dont le deu xième annule le premier.

Le principe est alors de repérer les messages non significatifs pour, d'une part, les enlever de la ré-exécution, et d'autre part, ne pas les considérer comme des messages orphelins. L'histoire répartie est compressée pour la ré-exécution.

Notre travail sur la sémantique ne s'inscrit pas au même niveau. En effet, dans [Leon94], l'utilisateur ne programme pas la tolérance aux fautes de son application. La sémantique des traitements de messages est en partie calculée par une analyse *post mortem*. Par ailleurs, il est *a priori* possible de combiner les sémantiques de transmission et de traitement des messages. Reste aussi à évaluer lequel, de l'utilisateur ou du système, est le plus apte à établir la sémantique des traitements ou des transmissions de messages. Pour conclure, observons que la sémantique des transmissions de messages permet d'optimiser non seulement le recouvrement arrière mais aussi la constitution des points de reprise et la journalisation.

Wang et Fuchs exploitent la propriété selon laquelle l'ordre de réception des messages peut souvent être changé durant l'exécution sans en affecter le comportement attendu [Wang92b]. Le principe est ici de ne pas journaliser l'exécution répartie. En outre, la réception d'un message est retardée dans le but de faire diminuer la probabilité de dépendances vis-à-vis des processus défaillants. Un algorithme d'ordonnancement des réceptions de messages est construit. Il tient compte d'un champ appelé "index de hasard" ajouté aux estampilles des messages par les processus émetteurs. L'index de hasard prôné par les auteurs est le temps avant la prochaine constitution d'un point de reprise de l'émetteur. Ainsi, la réception d'un message est retardée jusqu'à ce que le processus émetteur ait probablement constitué un point de reprise.

La fréquence des constitutions de points de reprise est supposée élevée. Sinon la probabilité qu'un message soit très prochainement validé par la constitution d'un point de reprise est faible. Le mécanisme est simulé avec un intervalle de temps entre deux constitutions consécutives de points de reprise égal à 30 secondes. Par conséquent, cette caractéristique est incompatible avec notre hypothèse sur l'intervalle de temps entre deux constitutions de points de reprise. Enfin, remarquons que la sémantique des transmissions de messages est un autre exemple d'index de hasard. En effet, les messages au moins une fois ou sans contrainte ne sont jamais orphelins et seraient donc prioritaires par rapport aux messages exactement une fois ou au plus une fois.

3.3.2 L'indéterminisme d'exécution

Dans cette sous-section, nous prolongeons notre recherche sur les ré-exécutions non équivalentes. Cette fois, les processus de l'application sont qualifiés par l'utilisateur de p-déterministes ou indéterministes (*cf.* § 3.3.2.1). Jusqu'à présent, l'exécution des processus était supposée p-déterministe. Aussi, les trois mécanismes de base (constitution de points de reprise, journalisation de l'exécution et recouvrement arrière) sont étendus pour supporter l'indéterminisme d'exécution (*cf.* § 3.3.2.2, 3.3.2.3 et 3.3.2.4). Les choix de réalisation de cette extension sont discutés dans un dernier paragraphe (*cf.* § 3.3.2.6).

3.3.2.1 Le repérage de l'indéterminisme d'exécution

Le repérage des actions indéterministes est en partie un problème d'implantation. En effet, il est *a priori* possible de définir un ensemble d'appels système connus pour s'exécuter de manière indéterministe. Toutefois, notre conviction est qu'il est intéressant de prendre en compte l'indéterminisme d'exécution dès la conception de l'application par l'utilisateur. Cette sous-section a pour but de montrer l'intérêt qu'a l'utilisateur à regrouper les actions

indéterministes dans certains processus. Nous optons donc pour une programmation par l'utilisateur du repérage de l'indéterminisme.

Chaque processus P_i maintient une variable booléenne $indéterministe_i$ évaluée à "vrai" ssi P_i contient des actions indéterministes. $indéterministe_i$ est initialisée à la valeur "faux". Les processus indéterministes exécutent l'action `EXEC_INDÉTER()`. Cette action comprend l'émission du message $m\_indéterministe^i_{GTF}$ contenant la variable $indéterministe_i$.

Le GTF mémorise les décisions dans le vecteur de booléens $indéterministe_{GTF}$ mis à jour à la réception des messages $m\_indéterministe^i_{GTF}$:

$$indéterministe_{GTF}[i] = indéterministe_i.$$

Le changement de $indéterministe_{GTF}[i]$ de la valeur "faux" à la valeur "vrai" provoque la constitution d'un point de reprise de P_i.

3.3.2.2 La constitution de points de reprise

L'indéterminisme d'exécution nécessite l'utilisation d'un mécanisme de constitution de points de reprise du type "barrière de synchronisation" ou "adaptatif" (*cf.* sous-section 2.4.2). Toutefois, nous avons aussi ajouté qu'un mécanisme de coupure cohérente est efficace pourvu que les actions entre l'état de reprise et les réceptions des messages manquants journalisés soient p-déterministes. Il est à noter que tous les mécanismes de coupures cohérentes que nous connaissons font la même hypothèse. En conséquence, le mécanisme de constitution de points de reprise peut aisément être étendu pour la prise en compte de l'indéterminisme.

Un processus indéterministe doit obligatoirement participer aux constitutions synchronisées (qui construisent des coupures cohérentes) :

$$indéterministe_i \Rightarrow pr\_sync_i.$$

Un processus dépendant causalement d'un processus indéterministe doit aussi participer aux constitutions synchronisées. Deux alternatives sont possibles. La première alternative est le calcul de l'état global recouvrable correspondant à l'état local du processus demandant la constitution, ceci avant la diffusion de l'ordre $m\_pr\_ordre^{GTF}_i$. Dans ce cas, l'histoire des processus indéterministes est journalisée. L'autre alternative est la participation de tous les processus à toutes les constitutions synchronisées. Nous optons pour cette dernière solution car elle n'impose pas la journalisation de l'histoire des processus indéterministes :

$$\forall P_i \in URR, pr\_sync_i.$$

Une dernière solution serait le calcul des messages dépendant causalement des processus indéterministes. L'algorithme réparti correspondant à cette dernière solution est similaire à celui de Koo et Toueg [Koo87]. Il est plus complexe et génère un nombre important de messages. Nous estimons qu'il n'est pas efficient. En effet, puisque l'intervalle de temps entre deux constitutions de points de reprise est important, la probabilité d'apparition de ces dépendances causales est proche de 1. En revanche, cette dernière solution est utilisée lors de la validation des messages à destination du monde extérieur (*cf.* § 3.3.2.3). La différence est que le vecteur *max_stable* calculé par l'algorithme de validation permet de connaître l'ensemble des processus dépendant indirectement des processus indéterministes.

Si l'indéterminisme d'exécution interdit l'utilisation d'un mécanisme de coupures cohérentes, le mécanisme de constitution de points de reprise arrête les communications

inter-processus avant de diffuser l'ordre $m\_pr\_ordre_i^{GTF}$. Ceci est par exemple déjà le cas avant le début du calcul de l'état global recouvrable maximal. Pour ce faire, le GTF diffuse de manière fiable un message $m\_pr\_arret_i^{GTF}$ demandant l'arrêt des émissions de message et la transmission des vecteurs ve et vdd.

3.3.2.3 La journalisation de l'exécution

Un processus indéterministe ne se ré-exécutera pas de manière équivalente. Il ne journalise donc pas son histoire séquentielle :

$$indéterministe_i \Rightarrow (\nu_i^4 = \infty).$$

Lors de la validation d'un message m à destination du monde extérieur, les processus indéterministes doivent constituer un point de reprise. Rappelons que la constitution doit construire une coupure cohérente.

L'algorithme de validation calcule le vecteur des dépendances transitives de l'état précédant l'émission de m. Les processus indéterministes ne journalisent pas leur histoire séquentielle mais transmettent leur vecteur des dépendances directes.

Suite au calcul du vecteur des dépendances transitives, le GTF calcule l'ensemble des points de reprise validés - met à jour le vecteur $dprv$. Puisque les constitutions de points de reprise sont synchronisées et le vecteur max_stable mis à jour, $dprv$ repère les dernières constitutions de points de reprise.

Ensuite, le GTF regarde si l'état précédant l'émission de m dépend de l'état d'un processus indéterministe P_i postérieur à l'état du dernier point de reprise de P_i. Le GTF maintient un vecteur de booléens $pr\_obligatoire_{GTF}$ tel que $pr\_obligatoire_{GTF}[i]$ est évalué à "vrai" ssi P_i est un processus indéterministe dont l'état du dernier point de reprise précède causalement l'émission de m :

$$pr\_obligatoire_{GTF}[i] \stackrel{\text{def}}{=} (max\_stable[i] > dprv[i]) \wedge indéterministe_{GTF}[i].$$

Si au moins une des composantes de $indéterministe_{GTF}[i]$ est évalué à "vrai", le GTF diffuse un ordre de constitution de points de reprise $m\_pr\_ordre_i^{GTF}$ en y insérant $dprv_{GTF}$ et $indéterministe_{GTF}$. Seuls les processus P_i dépendant de processus indéterministes participent à la constitution synchronisée. La dépendance est soit directe soit indirecte. La relation de dépendance directe s'exprime naturellement par la relation :

$$\exists (j \neq i) : (vdd_i[j] > dprv[j]) \wedge indéterministe_{GTF}[j].$$

La relation de dépendance indirecte s'établit simplement comme suit :

$$\exists (j \neq i) : (vdd_i[j] > max\_stable_{GTF}[j]) \wedge indéterministe_{GTF}[j].$$

Par conséquent, un processus P_i constitue un point de reprise si la relation suivante est évaluée à "vrai" :

$$\begin{array}{ll} & indéterministe_{GTF}[i] \\ \vee & \exists (j \neq i) : ((vdd_i[j] > max\_stable_{GTF}[j]) \vee (vdd_i[j] > dprv[j])) \ . \\ & \wedge indéterministe_{GTF}[j] \end{array}$$

La relation précédente est une relation nécessaire mais non suffisante, car max_stable n'est qu'une approximation de l'état global précédant l'émission de m.

3.3.2.4 Le recouvrement arrière

Contrairement aux processus p-déterministes, les processus indéterministes ne journalisent pas leur histoire séquentielle. Il s'ensuit que leurs ré-exécutions ne sont pas équivalentes.

Le recouvrement arrière débute avec le calcul de l'état global recouvrable maximal. Comme pour la validation d'un message, l'algorithme ne possède que deux vecteurs de dépendances directes au maximum pour les processus indéterministes. Il s'agit de celui du dernier point de reprise et de celui de l'état courant (cas $P_i \in \mathbf{ND}_{GTF}$). Mises à part ces différences, l'algorithme de calcul de l'état global recouvrable maximal est identique.

Les autres phases du recouvrement arrière sont elles aussi identiques. Les processus dont l'état dépend de l'état d'un processus indéterministe postérieur à l'état du dernier point de reprise sont orphelins.

3.3.2.5 Le couplage avec la sémantique des transmissions de messages

En guise de conclusion sur la prise en compte de l'indéterminisme d'exécution, le mécanisme de base étendu au seul indéterminisme supporte très mal les applications communiquant régulièrement avec le monde extérieur. Cependant, le couplage des deux extensions (sémantique et indéterminisme d'exécution) permet d'espérer une très bonne efficience. En effet, l'indéterminisme mesure, dans un sens, l'incapacité d'un processus à ré-exécuter de manière équivalente des actions internes. Le traitement de la sémantique des transmissions de messages consiste à enlever de la ré-exécution certaines émissions et certaines réceptions de messages. Par conséquent, la sémantique des transmissions de messages peut être utilisée pour limiter et contrôler la propagation par message de l'indéterminisme de processus en processus.

Prenons un exemple fictif, mais qui se veut être quelque peu réaliste : une application de suivi de cours boursiers.

La plus grosse partie des processus de l'application coopèrent pour faire des calculs sur les cours actuels et passés afin de prévoir les cours futurs. Ces processus sont appelés les processus "calculateurs".

Un processus de l'application est dédié à la récupération des cours boursiers en provenance des places boursières étrangères. Ce processus est appelé le processus "enquêteur". Son exécution est indéterministe car il initialise des temporisations pour périodiquement demander les cours étrangers. Il est capable, en relation avec le monde extérieur, de récupérer les cours des n dernières minutes. Les messages qu'il reçoit du monde extérieur peuvent donc être reçus à sémantique au plus une fois. Les contenus de ces messages ne sont pas sauvegardés en mémoire stable. Le processus enquêteur émet ces demandes avec des messages à sémantique "sans contrainte". Ils ne sont pas validés par le GTF. Les processus calculateurs demandent les valeurs des cours au processus enquêteur. Si les processus calculateurs sont capables de gérer des données qui arrivent en plusieurs exemplaires, les messages échangés lors des demandes sont à sémantique "au moins une fois". Les échanges des processus calculateurs avec le monde extérieur ne sont pas précédés de la constitution d'un point de reprise du processus enquêteur. En conclusion, le processus enquêteur ne peut pas provoquer le recouvrement arrière des processus calculateurs.

Un processus de l'application affiche sur un écran l'évolution des cours. Ce processus est appelé le processus "afficheur". Comme dans [Gold90], les utilisateurs estiment que les affichages peuvent expérimenter un phénomène de *bégaiement* (en anglais, *stuttering*)

lors de la ré-exécution. En d'autres termes, les messages d'affichage peuvent apparaître plusieurs fois. Toutefois, les utilisateurs préfèrent visualiser une valeur fausse parce qu'ancienne (clignotant) plutôt qu'une valeur par défaut. Les ordres d'affichage transmis par les processus calculateurs sont donc à sémantique "au moins une fois". Le processus afficheur ne peut donc pas devenir orphelin. En outre, les valeurs anciennes n'intéressent pas les utilisateurs. Les messages émis par le processus afficheur à destination du gestionnaire de l'écran - à destination du monde extérieur - possèdent donc une sémantique "sans contrainte". Ces messages ne sont pas validés.

En conclusion, le couplage des deux extensions permet de construire une application non fermée sur elle-même. Le principe est de regrouper les actions indéterministes dans des processus - ici, l'enquêteur et l'afficheur - et de les isoler en paramétrant la sémantique des transmissions de messages.

3.3.2.6 Discussion

Les trois catégories de politiques prenant en compte l'indéterminisme d'exécution sont les barrières de synchronisation, les coupures cohérentes et les constitutions adaptatives (*cf.* sous-section 2.4.2). Rappelons que les coupures cohérentes supposent que les processus possèdent un comportement p-déterministe entre l'état de reprise et les réceptions des messages manquants de la coupure.

Notre extension bénéficie des travaux des deux premières catégories. Les mécanismes de constitution de points de reprise adaptatifs n'ont été trouvés que récemment ([Xu93] et surtout [Bald95a]). Des améliorations peuvent donc être apportées à nos travaux.

En ce qui concerne la cohabitation de processus p-déterministes avec des processus déterministes, peu de travaux ont été publiés. Citons les recherches de Johnson dont nous nous sommes inspirés [John93]. Les processus indéterministes ne journalisent pas leur exécution. Par déduction, l'état global recouvrable maximal n'évolue pas. La conséquence est l'obligation de constituer des points de reprise pour le faire avancer. Les algorithmes de validation des messages et de calcul des états sont répartis. Chaque processus peut choisir entre la journalisation de son histoire séquentielle et la constitution d'un point de reprise.

Notre mécanisme global est étendu pour la conception d'applications "mixtes". Le mécanisme de constitution de points de reprise construit des coupures cohérentes. Peu de modifications sont à apporter pour obtenir des barrières de synchronisation. Les processus indéterministes ne journalisent pas leur histoire séquentielle. Ils ne transmettent que leur vecteur des dépendances lors de la validation d'un message. La validation d'un message m peut provoquer la construction d'une coupure cohérente si l'état précédant l'émission de m dépend d'une action indéterministe. Quant au mécanisme de recouvrement arrière, les mêmes modifications que pour la validation d'un message sont apportées à l'algorithme de calcul de l'état global recouvrable maximal. Les processus dépendant de processus indéterministes défaillants ou orphelins sont orphelins.

Ensuite, nous montrons comment les deux extensions peuvent être conjointement utilisées pour supporter ces applications "mixtes" à moindre coût. Le résultat est que les interactions avec le monde extérieur ne sont plus incompatibles avec l'indéterminisme d'exécution. La contre-partie est l'intervention de l'utilisateur pour la programmation de la tolérance aux fautes. Néanmoins, nous démontrons que cette programmation est aisée si les actions indéterministes sont regroupées dans un nombre restreint de processus à isoler.

3.4 Les politiques possibles

En conclusion, l'architecture du mécanisme global structure l'application en regroupant les processus s'exécutant sur les nœuds d'un même réseau local dans une URR. La gestion d'une URR est uniquement locale, elle est confiée à un processus dédié appelé le GTF. Par conséquent, l'utilisateur ou l'administrateur du système fixent les paramètres des mécanismes de base pour chaque URR. Chaque URR possède donc sa propre politique de recouvrement arrière. L'utilisateur peut par exemple grouper les processus de son application selon ce critère.

Le mécanisme de constitution des points de reprise constitue des coupures cohérentes aussi bien que des points de reprise non coordonnés. Certes, les constitutions de points de reprise cohérentes sont globales, mais le mécanisme peut assez facilement être modifié pour autoriser des mécanismes équivalents à ceux décrits dans [Spez86, Koo87, Lai87].

Le mécanisme global de recouvrement arrière suppose toujours la journalisation pessimiste ou optimiste de l'exécution répartie.

Le mécanisme de recouvrement arrière calcule l'état global recouvrable maximal à partir de l'histoire répartie. Par défaut, l'histoire répartie comprend les opérations d'émission et de réception de tous les messages. Puisque le GTF délivre leur histoire séquentielle aux processus fautifs avant le début de la reprise, la ré-exécution est équivalente.

La première des deux extensions au mécanisme global modifie la journalisation de l'histoire répartie en ignorant les messages à sémantique sans contrainte. Par ailleurs, la ré-exécution des opérations d'émission et de réception est elle aussi modifiée pour ignorer les messages à sémantique au plus une fois. Par conséquent, le mécanisme global plus le traitement de la sémantique des transmissions de messages supporte une ré-exécution non équivalente.

La deuxième extension permet le support d'applications indéterministes. Les mécanismes de base sont revus pour tenir compte de la non-journalisation par les processus indéterministes. Cette deuxième extension autorise des applications "mixtes" composées de processus p-déterministes et de processus indéterministes.

Enfin, ces deux extensions fournissent un outil de programmation de la tolérance aux fautes supportant efficacement les applications "mixtes" communiquant fréquemment avec le monde extérieur. Nous montrons que l'intervention de l'utilisateur peut être restreinte à l'étude de quelques processus. Le principe est de regrouper les actions indéterministes dans ces quelques processus.

Chapitre 4

Les deux réalisations : au dessus de SunOS et dans Chorus/MiX

Le mécanisme global développé dans le chapitre précédent a donné lieu à deux prototypes. Le premier prototype est réalisé au dessus du noyau SunOS. Sa particularité est qu'il est entièrement portable. Cette implantation a pour but l'évaluation des limites du concept de la portabilité. Le deuxième prototype est réalisé au dessus du micro-noyau Chorus, dans le sous-système Chorus/MiX. Cette deuxième implantation a pour but l'évaluation des apports et des limites de la technologie micro-noyau pour la construction de systèmes intégrant le recouvrement arrière.

Les sections 4.2 et 4.5 présentent respectivement les implantations au-dessus de SunOS et dans Chorus/MiX. Les sections 4.3 et 4.6 évaluent respectivement la conception des deux prototypes. Enfin, les sections 4.4 et 4.7 évaluent respectivement les performances intrinsèques des deux prototypes.

4.1 Un double objectif

L'objectif global des deux implantations reste l'évaluation des systèmes d'exploitation répartis pour l'intégration du recouvrement arrière. Rappelons que la sous-section 2.5.1 a précisé un certain nombre de pré-requis à la conception d'un mécanisme de recouvrement arrière. En outre, au paragraphe 3.2.1.1, nous avons supposé l'existence d'abstractions système telles que la mémoire stable et la diffusion non fiable.

Les deux systèmes étudiés sont volontairement pris très différents : SunOS est un système monolithique traditionnel et Chorus/MiX un système à micro-noyau.

Les systèmes monolithiques traditionnels ont comme principal inconvénient, au niveau conception, leur faible configurabilité. Cet inconvénient résulte en partie d'un avantage apprécié par tous les utilisateurs : l'existence de la norme UNIX. La conséquence est l'apparition de plates-formes logicielles pour offrir de nouveaux services aux utilisateurs : migration, partage de charge, programmation du parallélisme, mémoire répartie partagée, tolérance aux fautes, etc. Le leitmotiv de toutes ces plates-formes est la portabilité. Le concept regroupe aussi bien la capacité à s'exécuter sur des matériels différents[1] que la possibilité de s'exécuter au dessus du noyau. Les limites de la portabilité sont de deux ordres. D'une part, un processus de l'utilisateur ne peut pas se substituer au noyau pour certaines tâches. D'autre part, un processus utilisateur n'a accès qu'à un nombre restreint

1. Moyennant une nouvelle compilation et une nouvelle édition de liens.

de fonctionnalités à travers un ensemble fini d'appels système. Enfin, observons qu'à l'origine les systèmes monolithiques traditionnels ont été conçus pour les systèmes centralisés. La section 4.3 tire les leçons de l'implantation au dessus du noyau monolithique SunOS 4.1 et évalue les limites de la portabilité pour l'intégration du recouvrement arrière.

Les systèmes à micro-noyau ont été construits avec comme principal but de faciliter la construction de systèmes d'exploitation répartis. Le système d'exploitation est décomposé en un micro-noyau réalisant les services de base au dessus duquel sont ajoutés à la demande des émulations de systèmes traditionnels. Les émulations sont aussi appelées indifféremment des personnalités ou des sous-systèmes. En ce qui nous concerne, la deuxième implantation est réalisée au dessus du micro-noyau Chorus, dans le sous-système Chorus/MiX. Autrement dit, la tolérance aux fautes est intégrée à une émulation du système UNIX. Chorus/MiX a été choisi pour pouvoir comparer les deux implantations. La sous-section 4.6 évalue les apports et les limites de la technologie micro-noyau pour l'implantation du recouvrement arrière.

4.2 L'implantation au dessus de SunOS

La sous-section 4.2.1 présente l'environnement matériel et logiciel. La sous-section 4.2.2 montre l'architecture générale de la plate-forme logicielle. La sous-section 4.2.3 décrit les actions effectuées lors du démarrage d'un processus désirant être tolérant aux fautes. Les sous-sections suivantes: 4.2.4, 4.2.5 et 4.2.6 détaillent la réalisation des trois mécanismes de base. Les choix effectués sont discutés dans une dernière sous-section (*cf.* sous-section 4.2.7).

NB: Cette implantation correspond à une version antérieure du mécanisme de base. Certains algorithmes sont donc différents de ceux présentés dans le chapitre 3, nous les indiquons explicitement. En outre, le but étant l'évaluation du système d'exploitation, certaines fonctionnalités présentées dans le chapitre 3 n'ont pas été intégrées. Il s'agit des communications entre réseaux locaux, du traitement de la sémantique des transmissions de messages et de la prise en compte de l'indéterminisme.

4.2.1 L'environnement matériel et logiciel

La maquette est implantée en langage C sur un ensemble de stations de travail sans disque Sun4. Les nœuds sont équipés d'un processeur dont la fréquence d'horloge est de 25 MHz et d'une mémoire centrale volatile de 16 Mo. Les nœuds accèdent tous à une mémoire stable réalisée par des disques durs de 1.3 Gb SCSI possédant chacun une vitesse de transfert de 5 Mo/s. Le nœud gérant les disques est un serveur SPARC670MP (40 MHz, 64 Mo). L'accès aux disques est contrôlé par le système de gestion répartie de fichiers NFS [Sand85]. Pour des raisons historiques, cette version du mécanisme de base ne tolère pas les fautes du GTF. Le système de gestion répartie de fichiers NFS ne tolère pas les défaillances du serveur. Le GTF s'exécute donc sur le nœud du serveur NFS.

Le système d'exploitation est le système UNIX SunOS 4.1. Pour une présentation détaillée du système d'exploitation UNIX, le lecteur est prié de se reporter aux livres suivants: [Bach86], [Riff89] et [Riff90]. Le protocole de communication fiable est le protocole TCP (en anglais, Transport Control Protocol) [Come91]. Le mécanisme de diffusion non fiable est fourni par une option de l'appel système sendto(2) du protocole de communication non fiable UDP [Come91]. Aucun mécanisme de détection des fautes logicielles n'est utilisé. Les défaillances de nœuds sont détectées par rupture de connexion TCP.

Le point de départ de la plate-forme logicielle est le logiciel `Epsilon` [Bern89]. Quant au mécanisme de constitution de points de reprise, nous reprenons le mécanisme de saisie de l'état d'un processus conçu pour la migration par Alard [Alar91, Alar92].

4.2.2 L'architecture générale

En sous-section 3.2.2, nous avons indiqué qu'un contrôleur est associé à chaque processus. Nous en donnons maintenant la raison. L'un des objectifs du recouvrement arrière (*cf.* sous-section 2.3.3) est la minimisation de l'inhibition. Pour ce faire, toutes les actions des mécanismes de base pouvant s'exécuter en parallèle avec les actions des processus de l'application sont regroupées dans un autre processus : le contrôleur. Les actions des mécanismes de base sont donc divisées en trois catégories :

- les actions dépendant uniquement des processus de l'application : la saisie et la récupération de l'état du processus de l'application,
- les actions dépendant uniquement des contrôleurs : la journalisation périodique des histoires séquentielles, etc,
- les actions dépendant des processus de l'application et des contrôleurs : les émissions et les réceptions de messages inter-processus.

Par conséquent, sur chaque nœud est installé un processus exécutant les actions des mécanismes de base pour les processus de l'application du nœud. Ce processus est appelé le démon `epsid`.

En outre, les temporisations τ_i^n et les variables ν_i^n sont des paramètres que l'utilisateur peut désirer contrôler. Un programme de contrôle appelé `epsic` est ajouté pour leur paramétrage.

La figure 4.1 représente l'architecture générale du mécanisme global au dessus de `SunOS`. Les sous-sections suivantes présentent avec plus de détails chaque élément de l'architecture.

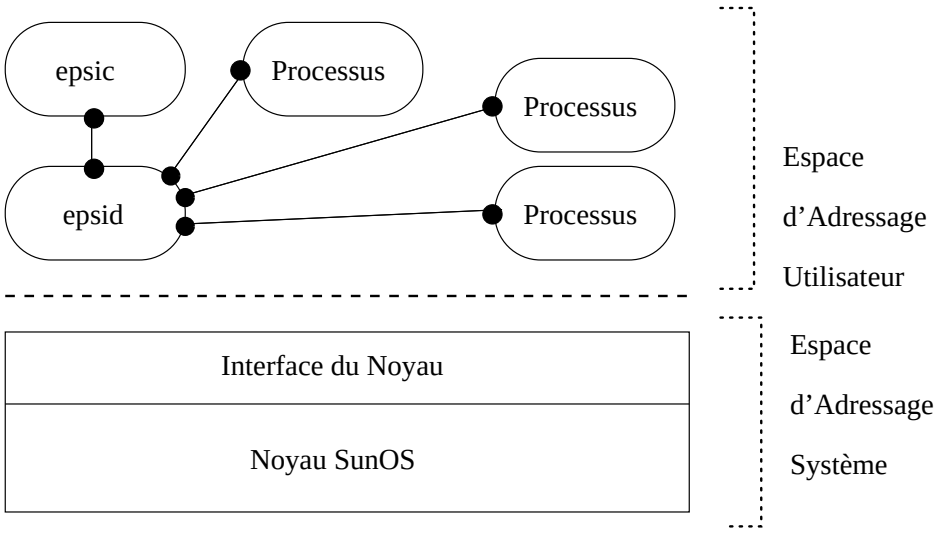


FIG. 4.1 – L'architecture générale - réalisation au dessus de `SunOS`.

4.2.2.1 Le programme utilisateur `epsic`

Il reçoit les ordres de l'utilisateur à partir du clavier. Les ordres sont de trois types :

- la configuration du réseau local : l'ensemble des nœuds sur lesquels l'application peut se diffuser,
- le démarrage des démons `epsid` locaux ou distants,
- le paramétrage de la tolérance aux fautes.

La transmission des ordres s'effectue par échange de messages `UDP` non fiables. `epsid` acquitte explicitement les ordres reçus et traités.

`epsic` est une boucle infinie lisant les ordres au clavier et les transmettant au démon `epsid` concerné.

4.2.2.2 Le processus démon `epsid`

Il exécute la plus grande partie des actions des mécanismes de base et toutes les actions traitant les requêtes en provenance des processus `epsic`. `epsid` initialise un `socket UDP` de service pour recevoir :

- les ordres des processus `epsic`,
- les messages de diffusion émis par le démon `epsid` GTF,
- les demandes d'ouverture de canaux de communication avec les autres démons `epsid`.

`epsid` ouvre un canal de communication avec chacun des processus de l'application du nœud.

`epsid` consiste en une boucle infinie recevant les messages sur les `sockets` ouverts et exécutant les actions désirées. Il initialise aussi les temporisations et exécute les actions `fin`(τ_i^n, nra) des mécanismes de base.

Cette structure du programme `epsid` possède un inconvénient majeur. L'appel système d'attente de réception des messages sur les connexions ouvertes est `select(2)`. Or, cette appel système fait partie des appels système[2] non interruptibles[3]. Les signaux sont donc masqués pendant l'exécution de l'appel système `select` et ne sont reçus qu'une fois l'exécution de l'action `RECEVOIR` terminée. Donc, lorsqu'aucun message n'est reçu, `epsid` ne peut exécuter aucune action `fin`(τ_i^n, nra). La conséquence est que, pendant les recouvrements arrière, les démons `epsid` autres que le GTF ne peuvent pas exécuter l'action `fin`(τ_i^{17}, nra) vérifiant la non-défaillance du GTF. Notons cependant que cette implantation n'a initialement pas été prévue pour tolérer la faute du GTF.

Une meilleure solution consisterait à créer un autre processus local serveur de temporisations. Ce processus réaliserait un mécanisme de temporisations à base de messages. Cette solution simple nous est apparue naturellement lors de la réalisation de l'implantation dans `Chorus/MiX`. En effet, dans `Chorus/MiX`, un signal `UNIX` est un message.

2. avec `recv(2)`, `recvfrom(2)`, `send(2)` et `sendto(2)`, etc.

3. La réception d'un signal pendant l'appel système `select` provoque un retour avec une erreur `EINTR`. Il serait donc possible de modifier le talon de l'appel système pour le ré-exécuter. Cette modification relativement compliquée n'a pas été effectuée.

4.2.2.3 Les processus de l'application

Les processus de l'application s'exécutent dans des espaces d'adressage séparés et non accessibles par les démons `epsid`. La seule manière de saisir l'état d'un processus à la place du processus lui-même serait de le faire à partir du noyau. Or, nous ne désirons pas modifier le noyau du système d'exploitation. En conséquence, les processus de l'application saisissent et récupèrent eux-mêmes leur état.

Les actions ÉMETTRE(m_i^j) et RECEVOIR(m_i^j) engendrent un dialogue entre les processus de l'application et les démons `epsid`. Afin de diminuer le nombre de connexions, chaque processus n'ouvre qu'un `socket UDP` avec le démon `epsid` local[4]. Les démons `epsid` gèrent des numéros de connexion et ouvrent des `sockets TCP` entre eux pour les transmissions de messages.

4.2.3 Le démarrage et la terminaison d'un processus tolérant aux fautes

Les processus de l'application sont démarrés par l'utilisateur à travers le processus `epsic`. La figure 4.2 trace les 5 étapes du démarrage d'un processus P_i.

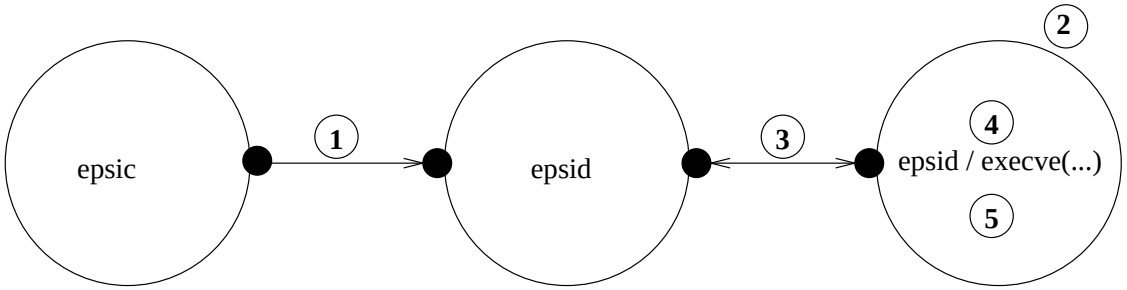


FIG. 4.2 – Le démarrage d'un processus tolérant aux fautes - réalisation au dessus de `SunOS`.

1. `epsic` transmet la requête de démarrage au démon `epsid` concerné.
2. Le démon `epsid` se dédouble en exécutant l'appel système `fork(2V)`.
3. Avant d'exécuter le code de l'application, le futur P_i et `epsid` créent le canal $c_{P_i,\texttt{epsid}}$.
4. Le nouveau processus se transforme en P_i par l'appel système `execve(2V)`.
5. Au début de son exécution, en guise de prologue, P_i affecte le gestionnaire du signal `SIGUSR1` à la fonction qui effectuera les saisies de l'état de P_i.

En conséquence, un processus désirant être tolérant aux fautes inclut un prologue dans le code du programme. Ce programme doit alors être à nouveau compilé et l'édition de liens refaite. Il est possible d'inclure un prologue et un épilogue sans modifier le source du programme, mais en modifiant le fichier `crt0.o` [Bric92]. Toutefois, une nouvelle édition de liens est quand même nécessaire.

En terminant sont exécution, P_i exécute un épilogue prévenant le démon `epsid` de sa terminaison normale.

4. Dans `Epsilon`, les connexions entre les processus et les démons utilisent le protocole `UDP` plutôt que le protocole `TCP`. La raison est que la plate-forme était installée aussi bien sur des stations de travail (possédant une capacité mémoire importante) que sur des ordinateurs personnels (possédant une capacité mémoire limitée).

4.2.4 La constitution de points de reprise

Les étapes de la constitution d'un point de reprise de P_i qui nous intéressent le plus sont celles concernant P_i lui-même. La figure 4.3 en trace les 4 premières.

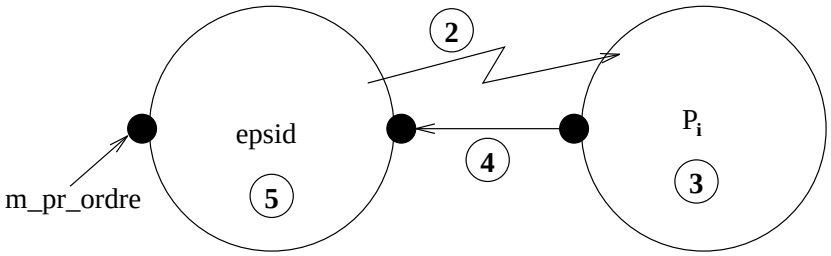


FIG. 4.3 – La saisie de l'état d'un processus de l'application - réalisation au dessus de `SunOS`.

1. La constitution débute avec la réception par le démon `epsid` d'un message *m_pr_ordre* ou avec la fin de la temporisation τ_i^0.
2. `epsid` émet un signal `SIGUSR1` à P_i. Notons que `epsid` connaît le numéro d'identification de P_i car P_i est l'un de ses fils. `epsid` continue la transmission des messages à destination de P_i.
3. Dès que P_i n'est pas bloqué par un appel système bloquant (tel que la réception d'un message), il saisit son état et l'enregistre en mémoire stable.
4. P_i prévient le démon `epsid` de la fin de la saisie de son état.
5. `epsid` continue l'exécution du mécanisme de constitution de points de reprise.

Rappelons que le mécanisme de saisie de l'état d'un processus est celui de Alard [Alar91]. L'état d'un processus est constitué de sa mémoire virtuelle et de ses registres. L'auteur a recours à une astuce de programmation pour la saisie des registres matériels. L'unique optimisation effectuée est le non-enregistrement du code du processus. L'exécution du processus est interrompue tout le temps de l'enregistrement de l'espace d'adressage en mémoire stable. L'inhibition est donc très importante. Une seconde optimisation consiste à dédoubler par un appel à `vfork(2V)` le processus et à effectuer l'enregistrement à partir du fils [Sens94]. C'est la copie en mémoire centrale. Plus précisément, dès que P_i accède à une page de la zone des données, une copie complète de son espace d'adressage est effectuée par le noyau pour le processus fils. Ce n'est donc pas une vraie copie sur écriture telle qu'elle est décrite au paragraphe 2.5.2.2.

À part ces deux optimisations et celles développées au paragraphe 2.5.2.2, aucune autre ne peut être réalisée de manière efficiente.

Tous les appels système non interruptibles sont remplacés par un talon masquant les signaux. Cela est nécessaire car la réception d'un signal pendant l'un d'eux met en erreur le processus. Le mécanisme de constitution de points de reprise du chapitre 3 spécifie qu'un point de reprise doit être constitué avant la réception d'un message marqué. Or, un processus ne peut pas interrompre l'exécution d'un appel système pour constituer son point de reprise. Par conséquent, les constitutions de coupures cohérentes ne sont pas possibles. Une solution serait d'émettre des messages de type marqueur et de transformer le talon de l'appel système `recv(2)` pour recevoir le marqueur puis saisir l'état du processus

et ensuite ré-exécuter la réception du message. Cette solution est correcte uniquement si les communications sont FIFO. Mais, rappelons que l'implantation au dessus de `SunOS` suppose que les processus de l'application sont tous p-déterministes. Nous nous intéressons donc aux coupures concomitantes.

Quant aux connexions et aux fichiers ouverts, ils sont ignorés et seront ré-ouverts lors de la reconfiguration et de la récupération.

4.2.5 La journalisation de l'histoire répartie

Le chemin d'un message m_j^i passe par les démons `epsid` locaux à P_i et P_j. Les primitives de connexion entre P_i et P_j sont remplacées et créent des numéros de connexion dans chaque démon `epsid`. Si aucune connexion n'existait entre les deux démons `epsid`, une connexion `TCP` est créée. Les démons `epsid` préparent l'ouverture du nouveau canal en échangeant des messages entre leur `socket UDP` de service. Par conséquent, un canal $c_{i,j}$ est identifié dans chaque démon `epsid` par deux `sockets` : un vers le processus de l'application et un vers le démon distant. La figure 4.4 trace les 4 étapes de l'émission d'un message m_j^i.

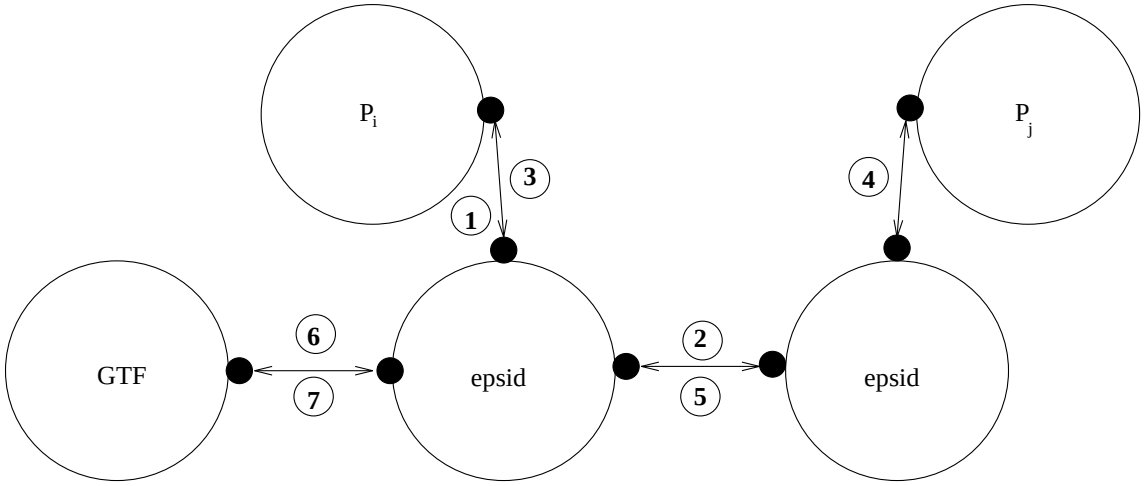


FIG. 4.4 – L'émission d'un message inter-processus - réalisation au dessus de `SunOS`.

1. P_i émet le message vers le démon `epsid` local en indiquant son numéro de connexion local.

2. Le démon `epsid` de P_i établit la correspondance entre le numéro de connexion local et le numéro de connexion distant. Puis, il ajoute le numéro de connexion distant à m_j^i. Ensuite, il émet m_j^i vers le démon `epsid` distant de P_j.

3. Le démon `epsid` de P_j acquitte la transmission de m_j^i auprès du démon de P_i. L'acquittement contient le numéro d'ordre de réception de P_j.

4. Si nécessaire, le démon `epsid` de P_i émet un message de journalisation vers le GTF. Ces messages de journalisation sont émis par le protocole `UDP` : de manière optimiste, ils seront reçus par le GTF.

5. Le GTF acquitte le message de journalisation en indiquant l'état d'avancement de la journalisation de P_i au démon `epsid` de P_i. Les acquittements sont eux-aussi émis par le protocole `UDP`.

Rappelons que le point de départ de cette implantation des canaux de communication est le projet `Epsilon` [Bern89].

La commande démarrant les démons `epsid` à partir des processus `epsic` possède en argument le nom du site du démon `epsid` GTF. Lorsqu'un canal entre processus de l'application est ouvert, le démon `epsid` initiateur de la demande enregistre les informations dans un fichier lui appartenant. Tous ces fichiers sont lus par le démon `epsid` GTF avant la reconfiguration.

L'implantation au dessus de `SunOS` ne possède pas d'algorithme de validation d'un message à destination du monde extérieur. L'objectif est ici surtout d'évaluer le coût des communications à travers les démons `epsid` et la charge du nœud du démon `epsid` GTF.

4.2.6 Le recouvrement arrière

Les défaillances de nœuds sont détectées par les démons `epsid`. Les actions du recouvrement arrière s'enchaînent comme indiqué dans le chapitre 3. Nous détaillons seulement les étapes de reconfiguration et de récupération, et en particulier, le redémarrage des processus de l'application et la reconnexion.

Les processus de l'application sont redémarrés avec en argument l'option "`-RESTART nomfichier`". L'option est interprétée par le prologue qui récupère l'état du processus contenu dans le fichier `nomfichier`. À la suite de la récupération de son état, le processus commence sa ré-exécution. Il est bloqué lors de sa première émission ou réception de message. Observons que le `socket` avec le démon local est recréé. Il en est de même pour les fichiers. Les anciennes entrées des tables des `sockets` et des fichiers ne sont pas récupérées. Puisque la taille de ces tables est fixe, le nombre de redémarrages est donc limité.

4.2.7 Discussion

Peu de mécanismes de recouvrement arrière sont portables.

Dans [Alvi93], les auteurs étudient plus spécialement la journalisation répartie. La journalisation répartie est plus efficiente que notre journalisation optimiste par les émetteurs.

Dans Star [Sens94], le mécanisme de constitution de points de reprise met en œuvre les optimisations de la copie en mémoire centrale (en créant un processus fils) et des points de reprise incrémentaux. Pour les points de reprise incrémentaux, le processus fils du dernier point de reprise continue d'exister jusqu'au point de reprise suivant. L'espace d'adressage du premier point de reprise est transmis au nouveau par un tube nommé. Remarquons que l'espace d'adressage du premier fils a de fortes chances d'être placé sur le disque de "swap" entre les deux points de reprise. La méthode est efficiente pour Star car la plate-forme réalise la mémoire stable en répliquant les informations sur les disques de plusieurs nœuds. Dans notre cas, les nœuds sont des stations de travail sans disque. Nous ne pensons donc pas observer les mêmes effets bénéfiques.

Peu de mécanismes de migration sont portables.

Dans Condor [Bric92], le fichier `crt0.o` est modifié et ainsi seule une nouvelle édition de liens est nécessaire. Condor utilise aussi les signaux pour déclencher la saisie de l'état. Celle-ci est réalisée en deux temps. Premièrement, le processus exécute l'appel système `setjmp(3V)` pour récupérer les registres matériels dans l'espace d'adressage et émet un

autre signal pour lui-même. Deuxièmement, le gestionnaire de ce second signal sauvegarde l'espace d'adressage dans un fichier. Enfin, les connexions avec d'autres processus ne sont pas supportées.

Dans PVM [Geis94], une plate-forme logicielle pour la programmation du parallélisme, les communications fiables sont construites à partir du protocole de communication non fiable `UDP`. Les auteurs avancent deux raisons pour ne pas utiliser `TCP`. Premièrement, qui dit mode connecté dit connexion. Or, le nombre de connexions par processus et par site est limité. En conséquence, `TCP` ne permet pas de construire une plate-forme extensible. Deuxièmement, le mécanisme de détection des fautes du protocole `TCP` met en œuvre des temporisations ralentissant beaucoup (toujours selon les auteurs) les communications. Un nouveau mécanisme avec des contextes de communication et de nouvelles temporisations (adaptées) est conçu. Le mécanisme résultant est complexe. En ce qui concerne notre mécanisme, il faut noter que les communications entre réseaux locaux sont regroupées chez les GTFs.

4.3 Les limites de la portabilité

Les limites sont de deux ordres. D'une part, un processus utilisateur ne peut pas se substituer au noyau pour certaines tâches. Par exemple, le démon `epsid` ne peut pas saisir l'état d'un processus de l'application. D'autre part, un processus utilisateur n'a accès qu'à un nombre restreint de fonctionnalités à travers un ensemble fini d'appels système. Par exemple, un processus ne peut pas modifier les bits d'état des pages de sa mémoire virtuelle.

Dans cette section, nous faisons la synthèse des limites rencontrées lors de l'implantation au dessus du noyau `SunOS 4.1`.

1. La saisie de l'état d'un processus est déclenchée par la réception du signal `SIGUSR1`. Or, certains appels système ne sont pas interruptibles. Il est donc nécessaire d'écrire des talons pour ces appels système. En conséquence, la tolérance aux fautes d'un processus oblige au minimum une nouvelle édition de liens.

2. Les démons `epsid` émettent le signal `SIGUSR1` pour la saisie de l'état d'un processus. Le nombre de signaux est limité. Nous en utilisons un pour la tolérance aux fautes. Il n'en reste donc qu'un pour les autres fonctionnalités système.

3. La fonction de saisie de l'état d'un processus est attaché au signal lors de l'exécution d'un prologue. Le prologue nécessite au minimum une nouvelle édition de liens.

4. Un processus utilisateur n'a pas accès aux informations du noyau concernant la gestion en mémoire centrale de sa mémoire virtuelle. Il n'a par exemple pas accès au bit de modification des pages en mémoire centrale [Appe91]. L'implantation efficiente des points de reprise incrémentaux n'est alors pas possible.

5. Un processus utilisateur ne peut pas accéder à l'espace d'adressage d'un autre processus (sans accord explicite de ce dernier). Le processus de l'application est donc obligé de sauvegarder lui-même son espace d'adressage en mémoire stable. Les méthodes de sauvegarde de la mémoire virtuelle les plus efficientes gèrent de manière globale tous les processus de l'application d'un nœud. Elles allouent par exemple des zones de mémoire dédiées aux opérations de sauvegarde en mémoire stable [Li90b]. Ces méthodes ne sont pas réalisables au dessus du noyau.

6. Les processus de l'application masquent la réception des signaux pendant les appels système non interruptibles tels que les appels système bloquants. Un processus utilisateur ne peut donc pas interrompre l'exécution d'un appel système pour saisir son état. Par conséquent, le mécanisme de constitution de coupures cohérentes ne peut pas être implanté.

7. Les systèmes monolithiques n'ont pas été conçus pour la répartition. Les noms sont souvent locaux et aucun mécanisme de localisation automatique n'existe. C'est le cas avec les descripteurs de `socket`. Les connexions ne sont donc pas fiables[5] et doivent être ré-ouvertes.

8. L'état du système d'exploitation concernant un processus est dispersé dans le noyau. En outre, aucune primitive ne permet d'y accéder et de le modifier. Il est alors nécessaire de créer une copie de certaines variables dans l'espace d'adressage du processus. C'est par exemple le cas avec la position courante de lecture ou d'écriture dans un fichier, et du répertoire courant de travail.

9. L'introduction d'un contrôleur entre le processus de l'application et le noyau pour la gestion des communications multiplie les connexions à ouvrir. C'est pour cette raison qu'il n'existe qu'un seul démon `epsid` par nœud et qu'une connexion entre les processus de l'application et les démons `epsid`. En conséquence, tous les talons des appels systèmes des `socket`s sont modifiés.

10. Le système d'exploitation `SunOS 4.1` ne réalise pas le concept des acteurs multi-activités. Le démon `epsid` sérialise donc toutes les actions du recouvrement pour tous les processus de l'application du nœud. Il est à noter que la plupart des actions concernant un processus de l'application peuvent être exécutées en parallèle avec celles des autres processus de l'application.

11. La structure du démon `epsid` est un boucle d'attente sur l'appel système `select(2)`. Or, cet appel système n'est pas interruptible facilement. Cela empêche la détection de la faute du GTF pendant l'algorithme de recouvrement arrière. La solution est d'ajouter un processus serveur de temporisations par nœud.

La section 4.6 reprend ces différents points et illustre les apports et les limites de la technologie micro-noyau `Chorus` pour l'intégration de la tolérance aux fautes.

4.4 Les mesures de performances de l'implantation au dessus de `SunOS`

Les mesures que nous présentons évaluent les performances intrinsèques du prototype. Autrement dit, les applications réparties utilisées pour les mesures sont construites de façon à mesurer les performances des mécanismes de base dans les cas les plus défavorables. Ces applications ne sont donc pas réalistes.

L'objectif est l'évaluation des surcoûts d'exécution et de l'inhibition pendant l'exécution : engendrés par les mécanismes de constitution de points de reprise (*cf.* sous-section 4.4.1) et de journalisation (*cf.* sous-section 4.4.2).

Puisque les stations de travail ne possèdent pas de disque local, les processus des applications réparties sont construits de manière à ce qu'aucun défaut de page ne survienne

5. Au sens où leur identificateur ne survit pas aux défaillances.

pendant les exécutions. Durant les tests, les stations de travail du réseau local n'exécutent aucune application autre que celles de gestion du réseau.

Le code source du prototype et des programmes de tests est modifié pour récupérer les informations d'utilisation des ressources locales. Ces informations sont obtenues à l'aide de l'appel système `SunOS getrusage(2)`. La précision de l'horloge physique est de 10ms. Dans certains cas, le code source du prototype est compilé avec l'option de profilage. Les résultats sont alors analysés avec l'utilitaire `SunOS gprof(1)`.

4.4.1 La constitution de points de reprise

L'enregistrement en mémoire stable de l'état des processus de l'application répartie est effectué par les processus eux-mêmes. L'exécution des processus est donc inhibée pendant tout le temps de la saisie de l'état. Quant aux messages demandés, ils sont enregistrés par les démons `epsid`. Raisonnablement, il est possible de paramétrer les démons `epsid` pour constituer des coupures concomitantes. Nous estimons donc que le surcoût d'exécution se limite à l'enregistrement en mémoire stable des états des processus. La concomitance est donc nulle.

Pour mesurer l'inhibition, nous construisons un programme de test qui exécute itérativement (100 fois) la fonction de saisie de l'état du processus. Le tableau 4.1 montre le nombre d'opérations d'entrée/sortie, le temps CPU système et le temps effectif pour des processus dont la taille de l'état (données plus pile) varie entre 20Ko et 5Mo. Remarquons que jusqu'à 1Mo la durée effective de la saisie de l'état est proportionnelle à la taille de l'état : environ 800ms par 100Ko. Les performances se dégradent lorsque les processus occupent une place importante en mémoire volatile : plus de 1Mo.

Taille données + pile (Ko)	**Nb. op. E/S**	**Tps CPU syst.** (s)	**Tps effectif** (s)
20	3	0,01 ± 0,00	0,17 ± 0,02
100	13	0,04 ± 0,00	0,81 ± 0,18
500	62	0,19 ± 0,01	3,94 ± 0,28
1 000	123	0,38 ± 0,02	8,70 ± 0,53
5 000	610	1,89 ± 0,03	44,83 ± 2,67

TAB. 4.1 – Le surcoût engendré par l'enregistrement de l'état d'un processus - réalisation au dessus de `SunOS`.

4.4.2 La journalisation de l'histoire répartie

La journalisation de l'histoire répartie est effectuée par les démons `epsid`. En conséquence, tous les messages passent par les démons `epsid` des processus émetteurs et récepteurs. En moyenne[6], hors journalisation des messages inter-processus, pour un message inter-processus de taille inférieur à 1444 octets[7], 6 messages (4 `UDP` intra-nœud et 2 `TCP` inter-nœud) sont transmis. En outre, en moyenne[8], ce message inter-processus nécessite

6. c.-à-d., en considérant qu'un message `TCP` constitue un segment `IP`, et, en ignorant les acquittements et les ré-émissions du protocole `TCP`.

7. 1444 = 1500 (taille d'un segment `IP`) - 40 (taille de l'entête `IP` plus taille de l'entête `TCP`) - 16 (taille de l'entête `epsid`).

8. c.-à-d., en considérant qu'il n'y a pas de congestion de chaque côté, et, que les demandes de réceptions de messages inter-processus en provenance des processus récepteurs précèdent les réceptions de messages inter-processsus par les démons `epsid` du côté récepteur.

l'exécution de 6 appels système `recv(2)` (ou `recvfrom(2)`) et de 7 appels système `send(2)` (ou `sendto(2)`).

À titre de comparaison, le prototype initial (réalisant la journalisation par l'émetteur) est modifié pour réaliser la journalisation par le récepteur. Dans ce cas, nous faisons l'économie d'un message `TCP` inter-nœud, et donc, d'un appel système `send(2)` et d'un appel système `recv(2)`. Par la suite, le prototype initial est noté `TFE` et le dernier `TFR`.

Pour chacun d'eux, les parties de code réalisant la constitution des points de reprise sont enlevées, exceptées celles permettant le vidage des journaux enregistrés en mémoire volatile. La périodicité des constitutions de points de reprise est tous les 500 messages. En outre, puisque le nœud sur lequel s'exécute le GTF est considéré comme sûr de fonctionnement, l'histoire répartie n'est pas enregistrée en mémoire stable. Enfin, les résultats sont des valeurs moyennes sur 3 exécutions comportant 100 000 transmissions de messages inter-processus. L'écart type observé est toujours inférieur à 5% de la valeur moyenne. Nous ne l'indiquons donc pas.

Pour évaluer les surcoûts d'exécution et l'inhibition, un programme de test est construit qui crée deux processus sur deux nœuds du réseau local. L'un des processus exécute itérativement (100 000 fois) une émission de messages et l'autre autant de réceptions de messages. Nous appelons ce programme "flux unidirectionnel". La taille des messages est successivement fixée à 1 octet [9], puis 1000 octets [10], et enfin 2000 octets [11].

Afin d'évaluer la dégradation des performances causée par le recouvrement arrière, une version du programme test est construite qui réalise la même fonction mais sans tolérance aux fautes. Dans la suite, il est noté `TF0`. La transmission d'un message inter-processus engendre un message `TCP` inter-nœuds, et, nécessite deux exécutions de l'appel système `recv(2)` et une exécution de l'appel système `send(2)`. Nous supposons ainsi que l'utilisateur introduit une entête à ces messages.

En plus d'évaluer les surcoûts d'exécution et l'inhibition (*cf.* § 4.4.2.1), nous observons la charge du nœud sur lequel s'exécute le démon `epsid` GTF (*cf.* § 4.4.2.2).

4.4.2.1 Le surcoût d'exécution et l'inhibition

À partir de chaque prototype (`TFE` et `TFR`), un nouveau prototype (respectivement appelé `TFE'` et `TFR'`) est construit. Ils n'enregistrent pas en mémoire volatile le contenu des messages inter-processus émis. Ces deux prototypes permettent l'évaluation du surcoût d'exécution et de l'inhibition dus au premier niveau de journalisation. Ensuite, avec les prototypes `TFE` et `TFR`, nous faisons varier la périodicité de la journalisation de "tous les 500 messages" (pour comparaison avec `TFE'` et `TFR'`) à "tous les messages" (journalisation pessimiste), en passant par "tous les 10 messages" (valeur communément prise pour la journalisation optimiste).

Les tableaux 4.2, 4.3 et 4.4 montrent les résultats des exécutions pour les trois tailles de messages : 1, 1000 et 2000 octets. Chaque tableau présente le pourcentage de la charge des nœuds émetteur et récepteur ainsi que la charge du nœud GTF, et, le débit ou nombre de messages transmis par seconde. Nous pouvons faire les remarques suivantes :

- Lorsque la journalisation est effectuée par l'émetteur, excepté dans le cas pessimiste, la charge est équilibrée entre les nœuds émetteur et récepteur. En effet, le nœud

9. c.-à-d., quantité d'information minimale nécessitant 7 exécutions des appels système `recv(2)` (ou `recvfrom(2)`).

10. c.-à-d., presque la taille d'un segment `IP`.

11. c.-à-d., plus que la taille d'un segment `IP`.

émetteur exécute un appel système `send(2)` et un appel système `recv(2)` sur la connexion `TCP` et le nœud récepteur un appel système `send(2)` et deux appels système `recv(2)`. Quant à la journalisation par le récepteur, le nœud récepteur est beaucoup plus chargé. Le nœud émetteur exécute un appel système `send(2)` sur la connexion `TCP` et le nœud récepteur deux appels système `recv(2)`. Notons que, dans le cas pessimiste, la valeur 100% est atteinte pour les tailles 1000 et 2000 octets.

- Dans le cas optimiste, la journalisation par le récepteur est environ deux fois plus efficiente que la journalisation par l'émetteur. Dans le cas pessimiste, elle l'est une fois et demie fois plus.

- Le premier niveau de la journalisation intervient pour une part insignifiante dans la dégration des performances.

- Les performances obtenues avec la journalisation optimiste sont comparables à celles obtenues sans journalisation.

- Dans le cas optimiste, le facteur de dégradation des performances de l'ensemble du système varie de 11 à 3 pour la journalisation par l'émetteur, et, de 5 à 1,5 pour la journalisation par le récepteur.

- Dans le cas pessimiste, le facteur de dégradation varie de 14 à 3 pour la journalisation par l'émetteur, et, de 10 à 2 pour la journalisation par le récepteur.

L'inhibition lors de l'émission et de la réception d'un message est évaluée avec l'utilitaire `SunOS gprof(1)`. L'opération de profilage étant effectuée par le noyau, nous supposons que le surcoût introduit est négligeable. Les figures 4.5 et 4.6 représentent le diagramme temporel de la transmission d'un message inter-processus pour les journalisations par l'émetteur et le récepteur, respectivement.

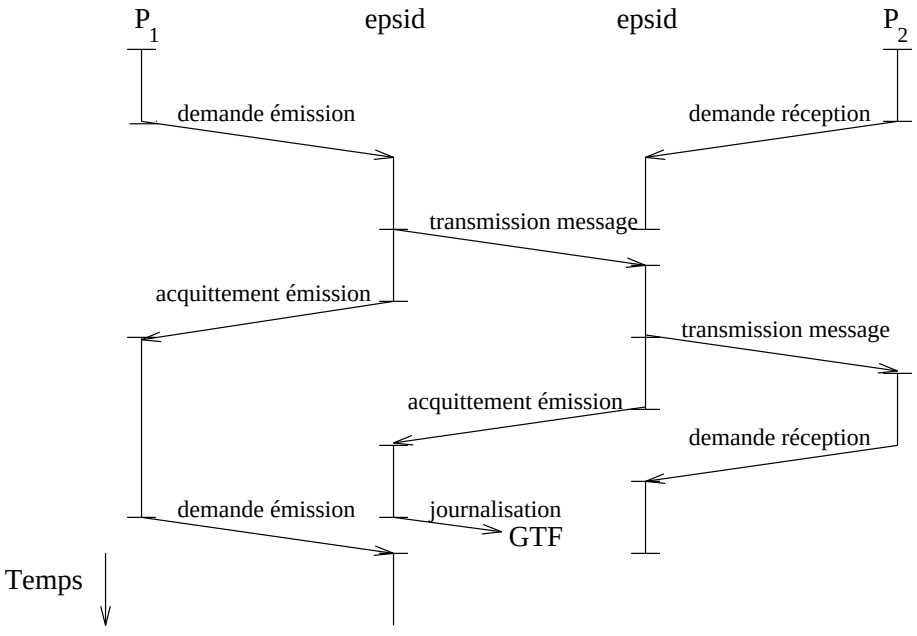


FIG. 4.5 – Le diagramme temporel de la transmission d'un message - journalisation par l'émetteur au dessus de `SunOS`.

Prototype	Période journalisation (Nb msg)	Émetteur proc + epsid (%CPU)	Récepteur proc + epsid (%CPU)	GTF (%CPU)	Débit (msg/s)
TF0	-	96	90	-	2 885
TFE'	-	10 + 51	10 + 53	-	252
TFE	500	10 + 51	10 + 53	0	251
	10	10 + 52	9 + 49	6	246
	1	8 + 79	9 + 38	47	202
TFR'	-	24 + 51	24 + 76	-	575
TFR	500	25 + 51	24 + 75	0	568
	10	21 + 45	21 + 78	12	502
	1	19 + 27	10 + 84	70	299

TAB. 4.2 – Le surcoût d'exécution engendré par la journalisation des messages (messages de 1 octet) - réalisation au dessus de SunOS.

Prototype	Période journalisation (Nb msg)	Émetteur proc + epsid (%CPU)	Récepteur proc + epsid (%CPU)	GTF (%CPU)	Débit (msg/s)
TF0	-	41	72	-	830
TFE'	-	11 + 37	9 + 38	-	171
TFE	500	10 + 39	10 + 38	0	170
	10	10 + 44	10 + 37	5	166
	1	10 + 65	9 + 31	27	142
TFR'	-	21 + 43	22 + 78	-	346
TFR	500	21 + 45	22 + 78	0	345
	10	21 + 44	19 + 81	7	341
	1	14 + 28	16 + 84	51	226

TAB. 4.3 – Le surcoût d'exécution engendré par la journalisation des messages (messages de 1 000 octets) - réalisation au dessus de SunOS.

Prototype	Période journalisation (Nb msg)	Émetteur proc + epsid (%CPU)	Récepteur proc + epsid (%CPU)	GTF (%CPU)	Débit (msg/s)
TF0	-	33	35	-	404
TFE'	-	11 + 33	12 + 32	-	131
TFE	500	11 + 38	13 + 32	0	130
	10	11 + 40	17 + 35	4	129
	1	17 + 32	17 + 32	27	120
TFR'	-	23 + 35	22 + 75	-	251
TFR	500	23 + 37	22 + 77	0	251
	10	22 + 37	17 + 82	6	251
	1	17 + 28	13 + 87	44	189

TAB. 4.4 – Le surcoût d'exécution engendré par la journalisation des messages (messages de 2 000 octets) - réalisation au dessus de SunOS.

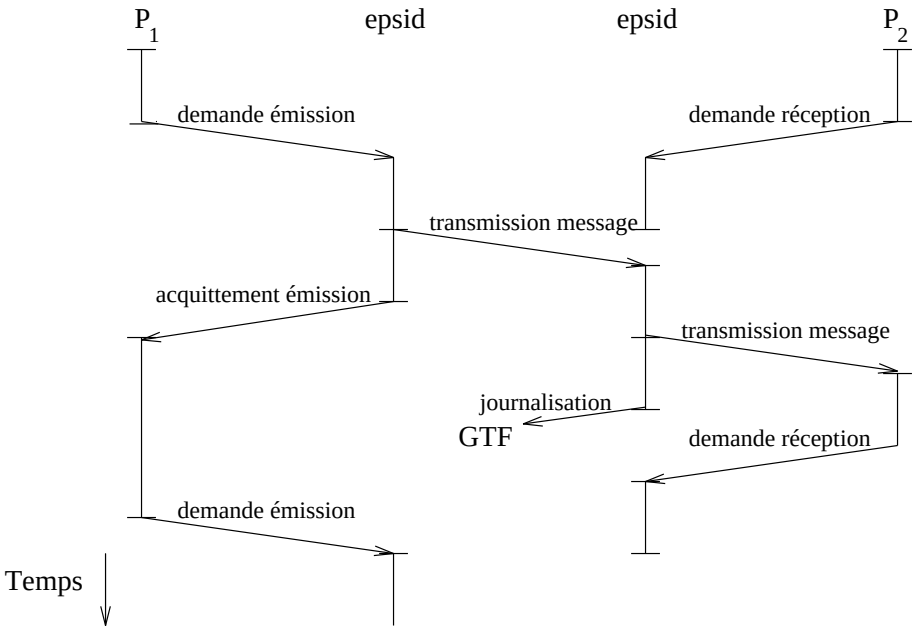


FIG. 4.6 – Le diagramme temporel de la transmission d'un message - journalisation par le récepteur au dessus de `SunOS`.

Le tableau 4.5 donne les durées minimales des opérations d'émission et de réception. Les résultats confirment ce qui est pressenti dans les diagrammes. En effet, pour la journalisation par l'émetteur, l'émission des messages de journalisation ne fait pas partie du chemin critique qui va du processus émetteur au processus récepteur. En revanche, pour la journalisation par le récepteur, le message de journalisation est émis à la suite de la réception du message inter-processus. Cela explique la dégradation proportionnellement plus importante des performances lorsque l'on passe du cas optimiste au cas pessimiste pour la journalisation par le récepteur. Pour indication, nous calculons la concomitance du mécanisme de journalisation en prenant la moyenne des valeurs obtenues pour les démons `epsid` à partir du tableau 4.5. Pour la journalisation par l'émetteur (respectivement, par le récepteur), la concomitance est inférieure à 40% du côté de l'émetteur (respectivement, du récepteur).

4.4.2.2 La charge du GTF

Les tableaux 4.2, 4.3 et 4.4 rassemblent les résultats des exécutions du programme "flux unidirectionnel". Nous extrapolons la valeur de la fréquence de transmission de messages inter-processus à partir de laquelle le GTF devient un goulet d'étranglement. Nous appelons cette valeur critique la fréquence d'étranglement du GTF. Par définition, elle correspond à une charge du GTF égale à 100%. Une valeur est calculée pour chaque cas de journalisation (optimiste et pessimiste) en prenant la moyenne des valeurs obtenues avec les tableaux 4.2, 4.3 et 4.4. Les fréquences d'étranglement du GTF pour les cas optimistes et pessimistes sont respectivement 4 100 messages par seconde et 450 messages par seconde. Si nous multiplions les valeurs calculées respectivement par les tailles maximales et minimales des messages inter-processus transmis, nous obtenons les intervalles suivants, en Mb/s : [67 ; 1,9] et [7 ; 0,2], respectivement pour les cas optimistes et pessimistes. Nous en concluons que le GTF ne sera pas un goulet d'étranglement pour la journalisation optimiste, mais qu'il peut le devenir pour la journalisation pessimiste.

Prototype	Taille des **messages** (octet)	**Période journalisation** (Nb msg)	**Émetteur** (ms/msg)	**Récepteur** (ms/msg)	**Tps transmission 1 message** (ms/msg)
TFE	1	10	0,76	1,00	4,07
		1	1,2	0,9	4,94
	1 000	10	0,97	1,13	6,01
		1	1,61	1,38	7,03
	2 000	10	1,48	1,80	7,76
		1	1,94	1,79	8,31
TFR	1	10	0,89	0,73	1,99
		1	0,91	1,26	3,34
	1 000	10	1,2	0,88	2,93
		1	1,21	1,50	4,42
	2 000	10	1,48	1,07	3,99
		1	1,48	2,17	5,28

TAB. 4.5 – L'inhibition engendrée par la journalisation des messages - réalisation au dessus de SunOS.

Par ailleurs, les messages de journalisation sont émis par le protocole UDP. Leur transmission n'est donc pas fiable. Par exemple, avec un programme de test réalisant la circulation d'un jeton sur un anneau de processus, nous avons observé jusqu'à 20% de perte de messages de journalisation, ceci dans le cas pessimiste avec 8 processus. Il serait intéressant de remplacer les messages UDP de journalisation par des messages TCP. Qu'en serait-il alors du surcoût d'exécution et de l'inhibition pour la journalisation par le récepteur ? Quelle serait la valeur de la fréquence d'étranglement du GTF ?

4.5 L'implantation dans Chorus/MiX

Une deuxième implantation du mécanisme global est réalisée dans Chorus/MiX. Cette section en développe les points importants.

La sous-section 4.5.1 présente l'environnement matériel et logiciel. La sous-section 4.5.2 montre l'architecture générale. La sous-section 4.5.3 décrit les actions effectuées lors du démarrage d'un processus désirant être tolérant aux fautes. Les sous-sections 4.5.4, 4.5.5 et 4.5.6 détaillent chacune un des mécanismes de base. Les choix effectués sont discutés dans la dernière section (*cf.* sous-section 4.5.7).

4.5.1 L'environnement matériel et logiciel

La maquette est implantée en partie en langage C et en partie en langage C++ sur un ensemble de micro-ordinateurs 486DX tous équipés d'un disque local. Les nœuds sont équipés d'un processeur dont la fréquence d'horloge est de 33 MHz et d'une mémoire centrale volatile de 16 Mo. Un des disques durs des postes de travail réalise la mémoire stable du réseau local. Les disques durs possèdent une capacité de 510 Mo et un temps d'accès de 12 ms. L'accès au disque dur réalisant la mémoire stable est contrôlé par le gestionnaire d'objets du sous-système Chorus/MiX du nœud.

Le système d'exploitation est le système Chorus/MiX. C'est un sous-système construit au dessus du micro-noyau Chorus. Avant de développer la conception du prototype, les sous-sections suivantes présentent le micro-noyau Chorus puis le sous-système Chorus/MiX et ensuite le sous-système des c_acteurs. Cette brève introduction définit les principaux

concepts de la technologie micro-noyau `Chorus` qui nous ont été utiles pour l'intégration du recouvrement arrière dans `Chorus/MiX`. D'autres concepts moins généraux seront définis au fur et à mesure de la présentation de l'implantation.

4.5.1.1 Le micro-noyau `Chorus`

La figure 4.7 présente l'architecture du micro-noyau `Chorus v3r5` [Rozi88, Abro89, Rozi92, CS95a, CS95b, CS95c].

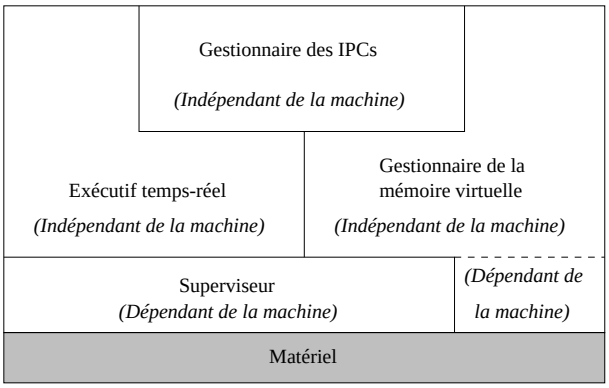


FIG. 4.7 – L'architecture du micro-noyau `Chorus`.

Il est composé de quatre entités :

- Le *superviseur* gère les interruptions, les trappes et les exceptions. Le mécanisme de trappe est le mécanisme d'interception des appels systèmes et de redirection vers le sous-système concerné. Ainsi, il y aura une table de trappes pour le sous-système `Chorus/MiX`.
- L'*exécutif temps-réel* gère l'ordonnancement et la synchronisation des entités d'exécution.
- Le *gestionnaire de mémoire virtuelle* est responsable de la gestion de la mémoire virtuelle des processeurs de la famille `Intel 486`.
- Le *gestionnaire de communication inter-processus* fournit un mécanisme d'échange de messages asynchrones et d'appel de procédures à distance. Le terme générique des primitives de communication est IPC (en anglais, Inter Process-Communication). Toutes les tâches de plus haut niveau que celles réalisées par le micro-noyau sont implantées par des entités communiquant par échange de messages. C'est notamment le cas pour la gestion de la mémoire virtuelle au niveau utilisateur.

Ces différents services sont accessibles aux utilisateurs par un ensemble d'abstractions de base. Celles-ci sont illustrées dans la figure 4.8.

- Un *acteur* est une unité d'allocation de ressources. Il encapsule les ressources qui suivent. Un acteur possède un nom global. Sa connaissance donne accès à toutes ses ressources.
- Une *région mémoire* ou zone de mémoire virtuelle est une unité de manipulation par l'utilisateur de la mémoire virtuelle de l'acteur. Une région possède des droits d'accès. Le gestionnaire de mémoire virtuelle implante la politique de copie sur écriture : une

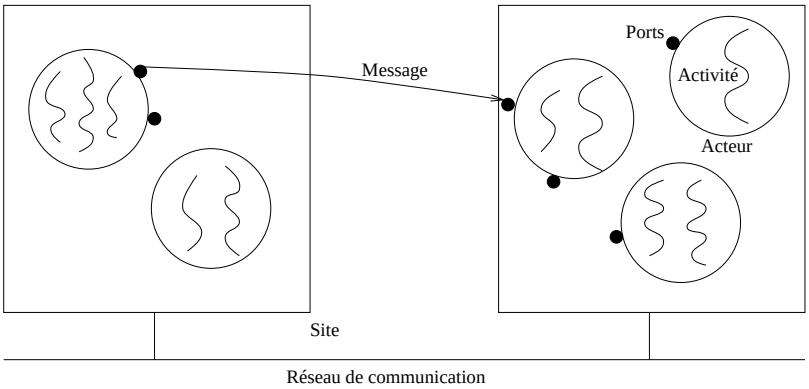


FIG. 4.8 – Les abstractions de base du micro-noyau `Chorus`.

page "mappée" par plusieurs régions n'est présente qu'en un exemplaire en mémoire centrale jusqu'à ce qu'un des propriétaires en modifie le contenu.

- Un *port* est un point d'entrée d'une connexion. Les acteurs communiquent par échange de messages d'un port d'émission vers un port de réception. La connaissance du nom global transparent d'un port est un droit d'émission de messages vers l'acteur qui l'héberge. Un port est une entité mobile. L'interface du micro-noyau `Chorus` fournit un appel système pour construire le nom d'un port. Il est donc possible de reconstruire un port disparu à la suite de défaillances. Par conséquent, nous dirons qu'un port est fiable. En outre, plusieurs ports peuvent être regroupés dans une entité de nommage supérieure appelée un *groupe de ports*. Ce niveau d'indirection dans le nommage facilite considérablement la configuration et la reconfiguration. Les noms de ports dans les appels système des IPCs peuvent être remplacés par des noms de groupes de ports. L'adressage est alors dit fonctionnel : un message est émis vers un port du groupe sur tel site ou bien vers un port quelconque du groupe ou bien vers un port du groupe n'étant pas sur tel site ou encore vers tous les ports du groupe [12].

- Une *activité* est une unité d'exécution. Elle s'exécute dans un acteur et partage toutes les ressources (régions mémoire et ports) avec les autres activités de l'acteur.

Les sous-systèmes implantés au dessus du micro-noyau `Chorus` utilisent les concepts d'acteur, de région mémoire, de port et d'activité pour mettre en œuvre des concepts plus classiques tels que les processus pour `Chorus/MiX`.

Du point de vue de l'accès aux services de l'interface du noyau, les acteurs possèdent deux propriétés :

- Un acteur est *sûr* (en anglais, trusted) s'il a le droit d'exécuter des appels systèmes sensibles.

- Un acteur est *privilégié* (en anglais, privileged) s'il a le droit d'exécuter des instructions privilégiées : généralement concernant le matériel.

12. Le micro-noyau `Chorus` ne maintient pas d'informations sur les membres du groupe. La diffusion n'est donc pas fiable.

Les acteurs sont divisés en trois catégories :

1. Les *acteurs utilisateur* ne sont ni sûrs ni privilégiés. Ils n'ont accès à aucun des appels système du noyau. Ils accèdent uniquement aux services fournis par un sous-système. C'est par exemple le cas des processus dans `Chorus/MiX`.

2. Les *acteurs système* sont sûrs mais non privilégiés. Ils accèdent à la plus grande partie des appels système du micro-noyau, excepté par exemple ceux gérant les temporisations. Leur espace d'adressage est dans l'espace utilisateur.

3. Les *acteurs superviseur* sont sûrs et privilégiés. Au moins un acteur par sous-système est un acteur superviseur. Il gère au minimum la table des trappes du sous-système. Dans `Chorus/Mix`, pour des raisons de performances, tous les serveurs sont des acteurs superviseurs. Ces acteurs partagent l'espace d'adressage système avec le micro-noyau.

4.5.1.2 Le sous-système `Chorus/MiX`

Le sous-système `Chorus/MiX v4r2` est le sous-système réalisant la personnalité `UNIX` au dessus du micro-noyau `Chorus` [Rozi88, Rozi92, CS93c]. La figure 4.9 représente les 5 serveurs qui le compose : le gestionnaire de processus (`PM`), le gestionnaire d'objets [13] (`OM`), le gestionnaire de flots d'entrée/sortie (`StM`) et deux autres serveurs (`IPCM` et `NDM`) spécifiques au système d'exploitation `V-system`.

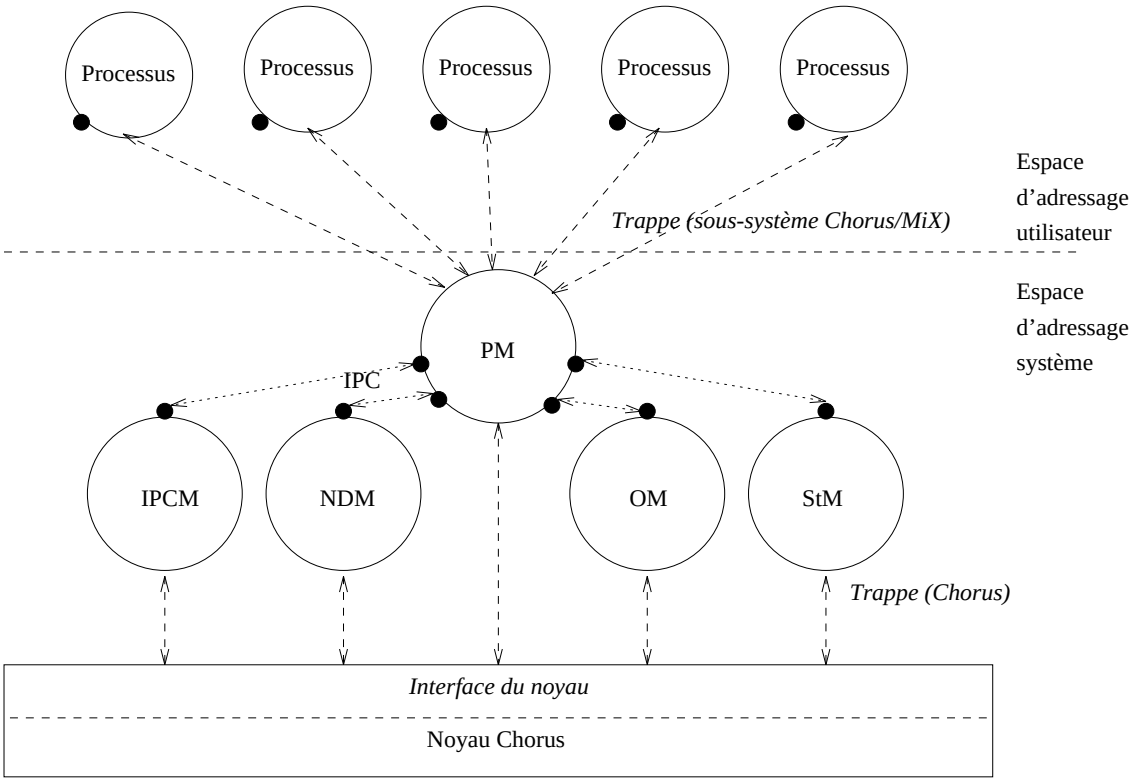


FIG. 4.9 – L'architecture du sous-système `Chorus/Mix`.

Le `PM` implante le concept de processus `UNIX`. Il intercepte (par la trappe) tous les appels système. De nouveaux appels système sont ajoutés à la norme `UNIX`. Il s'agit des

13. mémoire virtuelle et fichiers.

appels système concernant les IPCs. Le PM est écrit en C++. Tout l'état d'un processus est contenu dans une instance d'une classe (et de ses sous-classes).

L'OM agit comme "mappeur" externe de la mémoire virtuelle et comme gestionnaire des fichiers. Il implante par exemple la structure inode des fichiers UNIX.

Le StM est responsable de toutes les interactions d'un processus avec le matériel. C'est lui qui met en œuvre le protocole TCP. D'ailleurs, par un mécanisme d'appel de procédure remontant (en anglais, upcall) le micro-noyau fait appel au StM pour fiabiliser les IPCs.

Dans ce contexte, un processus est un acteur utilisateur ayant une activité[14] et un port de contrôle.

Le port de contrôle est un point d'entrée spécial par lequel une entité extérieure peut modifier le fonctionnement du processus. C'est par exemple par le port de contrôle que sont reçus sous forme de messages les signaux de la sémantique UNIX. Contrairement aux signaux dans UNIX, un processus MiX ne peut pas éviter le traitement d'un message arrivant sur son port de contrôle. Le nom global du port de contrôle peut être construit en connaissant l'identificateur UNIX (pid) du processus. Cette opération nécessite l'utilisation d'un appel système du micro-noyau, et donc, interdit aux acteurs utilisateurs. Les actions exécutées lors de la réception d'un message sur le port de contrôle ne sont pas exécutées par l'activité du processus. La réception d'un message sur le port de contrôle génère la création d'une autre activité appelée *gestionnaire de message* (en anglais, message handler). Les exécutions des deux activités sont parallèles. Mais, le gestionnaire de message (appartenant au PM) a accès à toutes fonctionnalités du micro-noyau. Cette propriété est utilisée pour la saisie et la récupération de l'état des processus de l'application.

Lorsqu'un processus exécute un appel système, l'exécution de son activité s'exécute dans le PM après le passage par le mécanisme de gestion des trappes du superviseur. Le contexte d'exécution de l'activité principale du processus est sauvegardé et une nouvelle activité (un nouveau contexte d'exécution) exécute les actions de l'appel système.

La figure 4.10 visualise la structure interne d'un processus.

4.5.1.3 Le sous-système des c_acteurs

C'est le sous-système associé à Chorus/MiX pour le développement de nouvelles fonctionnalités dans Chorus/MiX [CS93a, CS93b]. En effet, les processus MiX n'ont pas accès aux appels système du micro-noyau. Les c_acteurs ont été développés pour pallier cet inconvénient.

Un c_acteur est un acteur système ou un acteur superviseur ayant en plus l'accès à certaines fonctionnalités du sous-système Chorus/MiX comme les primitives sur les fichiers. Un c_acteur est contrôlé à partir d'un processus MiX par des commandes particulières : arun pour le démarrage, akill pour la terminaison et aps pour la liste des c_acteurs en cours d'exécution. En revanche, un c_acteur ne peut pas créer de processus MiX.

4.5.2 L'architecture générale

Afin d'utiliser les abstractions du micro-noyau, le serveur de la tolérance aux fautes par recouvrement arrière (FTM) est un c_acteur. Le FTM joue le rôle équivalent au démon epsid de l'implantation SunOS. La prudence veut que le développement d'un nouveau serveur se fasse dans l'espace utilisateur. En conséquence, le FTM est un acteur système.

14. Chorus/MiX supporte les processus multi-activités. Mais, notre mécanisme global du recouvrement arrière ne les autorise pas.

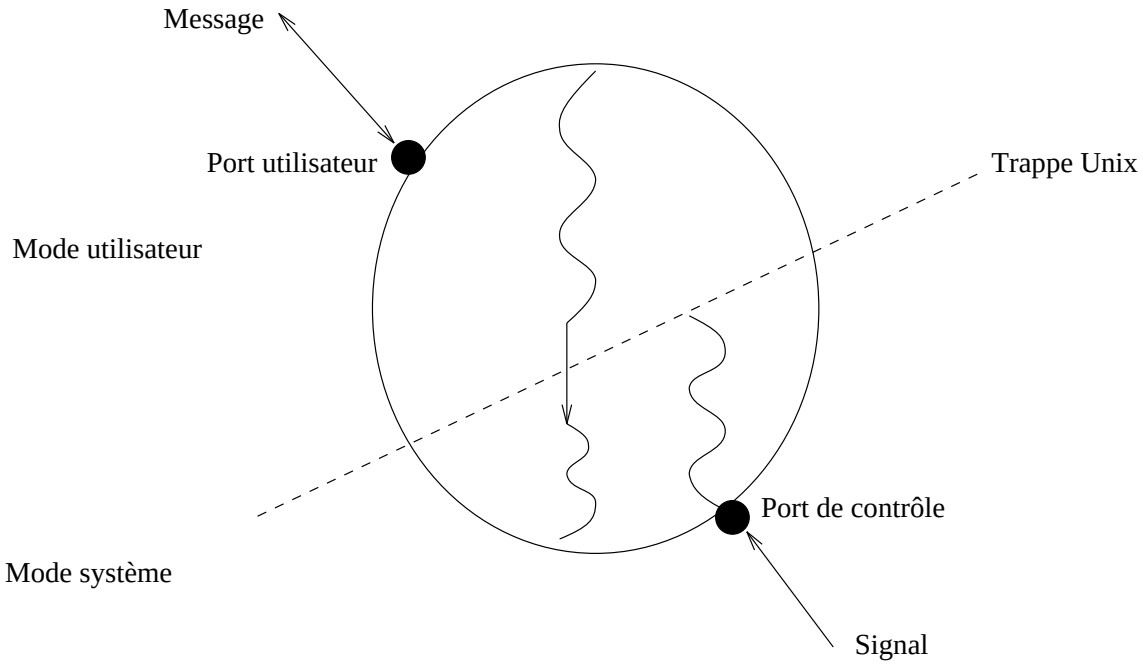


FIG. 4.10 – La structure interne d'un processus dans `Chorus/Mix`.

Cela oblige à introduire un `c_acteur` superviseur pour la gestion des temporisations, il est appelé le `TFTM`.

L'équivalent du processus utilisateur `epsic` de l'implantation `SunOS` est un processus `MiX` appelé `UFTM`. Puisque le `FTM` ne peut pas créer de processus `MiX`, le processus `UFTM` agit comme un relais du `FTM` pour démarrer les processus tolérants aux fautes. En conséquence, contrairement à l'implantation au dessus de `SunOS`, le processus `UFTM` est nécessaire pendant toute la durée de l'exécution de l'application.

Enfin, nous modifions le `PM` pour y intégrer la saisie et la récupération des processus `MiX`.

La figure 4.11 présente l'architecture générale du mécanisme global dans `Chorus/MiX`. Les sous-sections suivantes présentent avec plus de détails chaque élément de l'architecture.

4.5.2.1 Le processus utilisateur `UFTM`

En plus des fonctionnalités du processus `epsic` de l'implantation `SunOS`, il interrompt régulièrement son exécution pour recevoir les messages en provenance des `FTM`s. Ce sont les opérations de lecture du clavier qui sont interrompues. Contrairement aux émissions et aux réceptions de messages, celles-là sont interruptibles. L'opération de réception de message du `FTM` est paramétrée pour être non bloquante.

C'est toujours le processus `UFTM` qui démarre le `c_acteur FTM`. Pour communiquer avec les `FTM`s, le processus `UFTM` crée un port qu'il insère dans un groupe de ports statique utilisateur : un groupe de ports dont le nom peut être construit par tous les acteurs à partir d'une clé. Remarquons qu'un processus `UFTM` peut alors recevoir des messages en provenance de tous les `FTM`s et de tous les autres `UFTM`s.

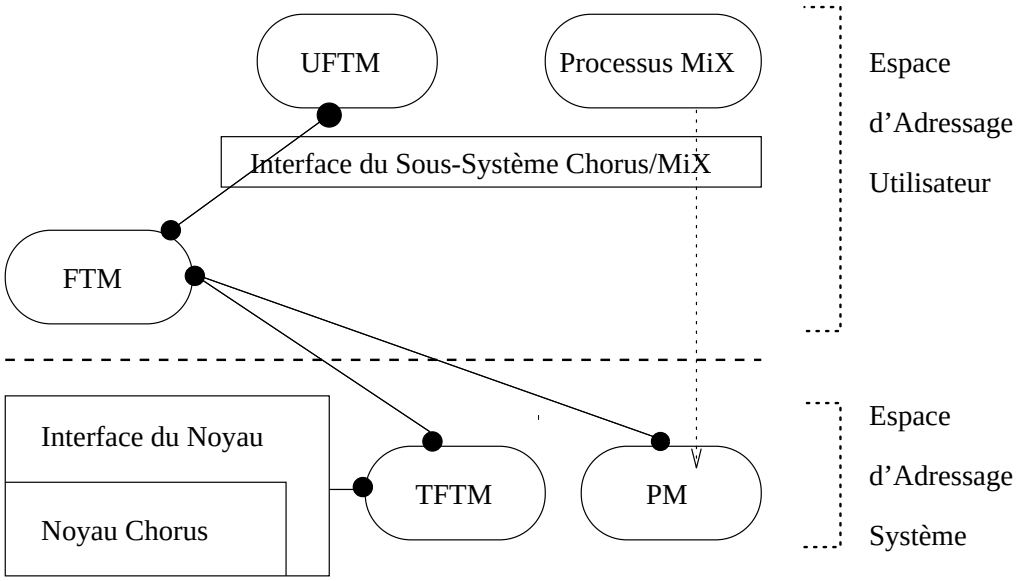


FIG. 4.11 – L'architecture générale - réalisation dans `Chorus/Mix`.

4.5.2.2 Le c_acteur utilisateur FTM

Il implante toutes les fonctionnalités du démon `epsid` de l'implantation `SunOS`. Mais, il n'est pas visible par les processus de l'application. En fait, le `FTM` dialogue avec le `PM` pour toutes les actions dépendant des processus de l'application et des contrôleurs (*cf.* sous-section 4.2.2).

Pour communiquer avec les processus `UFTM`s, le `FTM` crée un port qu'il insère dans un groupe de ports statique utilisateur. En outre, il crée un deuxième port qu'il insère dans un groupe de ports statique système. À la différence d'un nom de groupe utilisateur, un nom de groupe système ne peut être construit que par un acteur système ou un acteur superviseur. Le deuxième port est dédié aux communications avec le `PM`, les `TFTM`s et avec les autres `FTM`s. La sécurité des serveurs du sous-système `Chorus/MiX` est alors préservée à partir du moment où les interactions avec les processus `UFTM`s sont limitées et "sûres".

Le `FTM` bénéficie du concept des acteurs multi-activités. Toutes les activités du `FTM` exécutent le même code. Les actions pour un même processus de l'URR (y compris le GTF) sont sérialisées.

Le `FTM` n'étant qu'un acteur système, il ne peut pas initialiser de temporisations. Les temporisations sont gérées par le `TFTM`. Les actions `init`(τ_i^n, nra) et `enlever`(τ_i^n, nra) sont réalisées par l'émission d'un message à destination du `TFTM`. L'action `fin`(τ_i^n, nra) est exécutée à la réception d'un message en provenance du `TFTM`.

4.5.2.3 Le c_acteur superviseur TFTM

C'est le serveur de temporisations des mécanismes de base. Sa structure interne est présentée dans la figure 4.12.

Les messages d'initialisation de temporisation sont reçus par l'activité principale du `TFTM`. Cette activité enregistre la temporisation dans un tableau, puis, exécute l'appel système `svTimeOut(K)` pour l'initialiser. `svTimeOut` prend en argument le nom de la fonction à exécuter (`tftmTimeOutHandler`) à la fin de la temporisation et un numéro: ici c'est l'indice de la nouvelle temporisation dans la table des temporisations. Enfin, l'activité acquitte la requête d'initialisation.

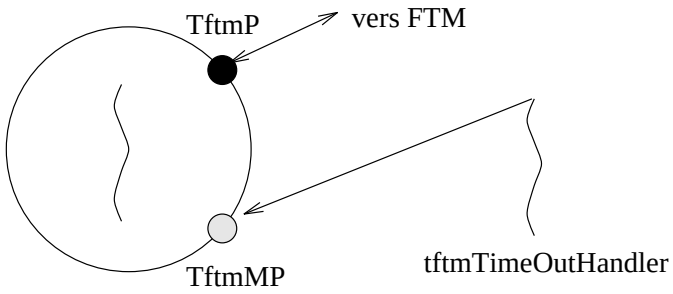


FIG. 4.12 – Le fonctionnement du `TFMT`- réalisation dans `Chorus/Mix`.

À la fin de la temporisation, le micro-noyau alloue une activité qui exécute la fonction fournie lors de l'appel à `svTimeOut`. Cette fonction prend en argument un indice de la table des temporisations. L'activité créée est appelée un gestionnaire d'interruption dans la terminologie `Chorus`. Les actions concernant les temporisations étant considérées comme des actions privilégiées, seuls quelques appels système sont exécutables par les gestionnaires d'interruption. Le gestionnaire d'activité émet un message vers un *mini-port* [15] qui a été précédemment créé par l'activité principale du `TFTM`.

Enfin, l'activité principale du `TFTM` reçoit le message arrivé sur le mini-port et émet un message à destination du `FTM` pour l'avertir de la fin de la temporisation.

4.5.2.4 Les processus de l'application

Contrairement à l'implantation au dessus de `SunOS`, la tolérance aux fautes est transparente à l'utilisateur. Aucune compilation et aucune édition de liens particulières n'est nécessaire.

4.5.3 Le démarrage et la terminaison d'un processus tolérant aux fautes

Les processus de l'application sont démarrés par l'utilisateur à travers le processus `UFTM`. L'objectif est de changer l'état du processus de l'application (objet `ProcFlags`) pour indiquer au `PM` que le processus est tolérant aux fautes. La figure 4.13 trace les 5 étapes du démarrage d'un processus P_i.

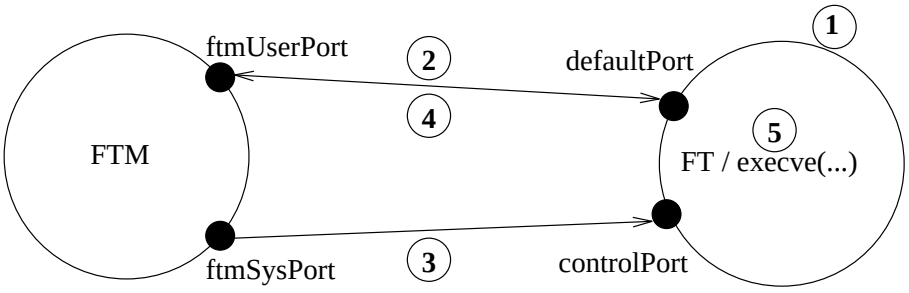


FIG. 4.13 – Le démarrage d'un processus tolérant aux fautes - réalisation dans `Chorus/Mix`.

1. Le processus `UFTM` démarre l'utilitaire `FT`. `FT` est un processus `MiX` qui deviendra le processus P_i à la fin de la phase de démarrage.

15. Du point de vue de la sémantique, un mini-port est équivalent à un port, excepté que seuls les acteurs superviseurs peuvent les manipuler. Dans la réalité, les messages reçus sont de taille fixe et les opérations de réception et d'émission sont optimisées : ce sont de simples appels de procédure dans l'espace système.

2. Le processus `FT` crée un port et émet un message vers le `FTM` s'exécutant sur le même site. Ce message contient le `pid` de `FT` et demande au `FTM` l'enregistrement de la création du nouveau processus.

3. Le `FTM` reçoit le message en provenance du processus `FT` et calcule le nom global du port de contrôle de `FT`. Puis, il émet un message vers le port de contrôle de `FT`.

4. La réception du message sur le port de contrôle provoque le passage à "vrai" du drapeau "tolérant aux fautes" du processus : objet `ProcFlags` de la structure de gestion du processus dans le `PM`. Dorénavant, le `PM` coopérera avec le `FTM` pour les actions des mécanismes de base du recouvrement arrière.

5. Dès l'émission du message vers le port de contrôle, le `FTM` acquitte la demande d'enregistrement.

6. À la réception du message d'acquittement, le processus `FT` se transforme en P_i pour exécuter le code l'application. Le drapeau "tolérant aux fautes" de l'objet `ProcFlags` est hérité par les processus fils et non modifié par l'appel système `execve(2V)`.

Notons qu'il n'est pas obligatoire de passer par le processus `UFTM` pour rendre tolérant aux fautes un processus. Il suffit d'utiliser l'utilitaire `FT`.

4.5.4 La constitution de points de reprise

Comme pour l'implantation au dessus de `SunOS`, les étapes de la constitution d'un point de reprise de P_i qui nous intéressent le plus sont celles concernant P_i lui-même. La figure 4.14 en trace les 7 premières.

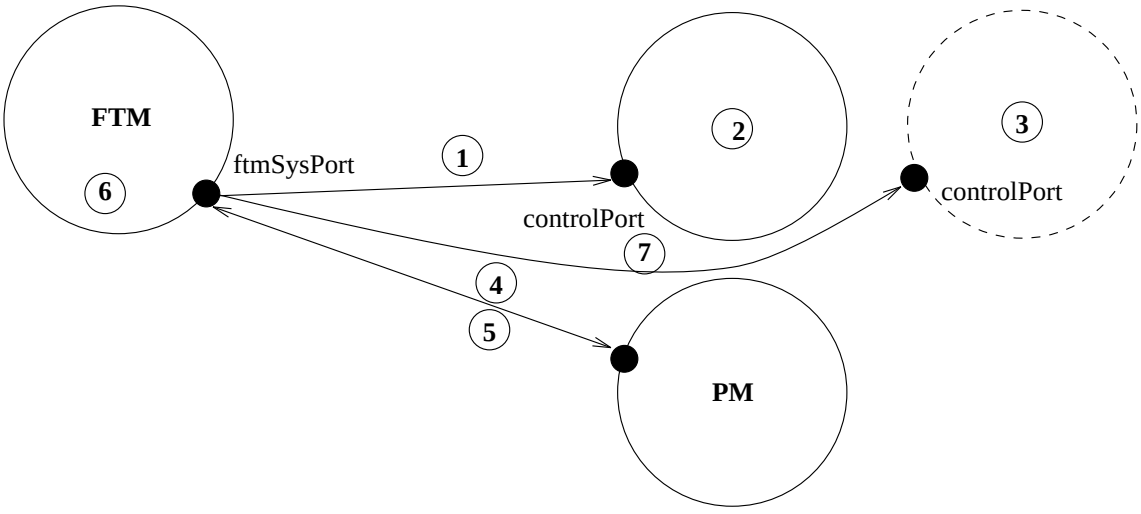


FIG. 4.14 – La saisie de l'état d'un processus de l'application - réalisation dans `Chorus/Mix`.

1. Le `FTM` émet un message à destination du port de contrôle de P_i. Ce message provoque le passage à "vrai" du drapeau "en constitution" de l'objet `ProcFlags`. Si P_i n'est pas en cours d'exécution d'un appel système, l'étape suivante n'existe pas.

2. Le `PM` attend la fin de l'exécution de l'appel système.

3. Le PM crée un *pseudo*-processus qui possède le même état et les mêmes régions mémoire que P_i. C'est un *pseudo*-processus dans le sens où seules les parties de l'état à récupérer sont copiées. Il possède une activité (pour la dernière étape) mais son exécution reste suspendue.

4. Le PM transmet l'état de P_i au FTM.

5. Le FTM acquitte la saisie de l'état au PM. L'exécution de l'activité principale de P_i reprend.

6. L'état de P_i comprend le nom du *pseudo*-processus. Le FTM peut alors accéder à l'espace d'adressage du *pseudo*-processus pour l'enregistrer en mémoire stable.

7. Lorsque l'espace d'adressage du *pseudo*-processus est sauvegardé en mémoire stable, le FTM émet un message vers le port de contrôle du *pseudo*-processus. Le message provoque la disparition du *pseudo*-processus.

Rappelons que le micro-noyau réalise la copie sur écriture entre l'espace d'adressage de P_i et celui du *pseudo*-processus. En outre, les régions mémoire du code du processus de l'application ne sont pas sauvegardées en mémoire stable ni même répliquées dans le contexte du *pseudo*-processus.

La seule optimisation (simple[16]) existant dans la littérature et restant à réaliser est la constitution de points de reprise incrémentaux. À ce sujet, nous n'avons pas trouvé où et comment sont gérés les bits de modifications des pages de la mémoire centrale [Appe91].

Pour la sauvegarde de l'espace d'adressage en mémoire stable, le FTM utilise les appels système du micro-noyau. Ainsi, les régions mémoire du *pseudo*-processus sont "mappées" dans son espace d'adressage. Ensuite, le FTM utilise les appels système de l'interface MiX concernant les fichiers pour écrire les pages sur disque. Pour minimiser le nombre de copies mémoire des pages, la fonction d'écriture sur fichier de la libraire libMiX.a a été ré-écrite pour éliminer la première copie dans le "buffeur" d'écriture du FTM.

Comme pour l'implantation au dessus de SunOS, la saisie de l'état commence à la fin des appels système non interruptibles tels que les réceptions de messages. La raison évidente est que la saisie de l'état du sous-système Chorus/MiX et du micro-noyau Chorus pendant un appel système MiX est une tâche très compliquée. D'autres avant nous ont essayé pour le sous-système des c_acteurs, mais ne sont pas arrivés à la fin de l'implantation [O'Co93, O'Co94]. Par conséquent, les constitutions de coupures cohérentes ne sont pas réalisées. La solution proposée pour l'implantation SunOS est plus difficile à réaliser dans Chorus/MiX pour être transparente au processus de l'application. En effet, c'est *a priori* le FTM qui détecte la réception du marqueur et indique au PM que l'état est à saisir. Le problème est que le processus MiX est encore en cours d'exécution de l'appel système de réception de message. La réalisation est donc plus compliquée car elle implique la saisie de l'état du sous-système Chorus/MiX: notamment pour l'enregistrement des paramètres de l'appel système.

4.5.5 La journalisation de l'histoire répartie

Le chemin d'un message inter-processus n'est pas modifié pour la journalisation. Seulement, avant l'émission et avant la réception, le PM émet une copie du message vers le FTM pour l'estampillage et la sauvegarde du contenu en mémoire volatile. Les messages Chorus

16. *cf.* section 3.1.

sont constitués de deux zones de données distinctes et gérées différemment : le corps du message et son annexe. Les estampilles trouvent naturellement leur place dans les annexes des messages.

Pendant les ré-exécutions, un autre drapeau de l'objet `ProcFlags` indique au `PM` que les messages à recevoir sont donnés par le `FTM` et que les messages à émettre ne sont pas forcément ré-émis.

Le concept de groupe de ports est encore utilisé pour adresser les `FTM`s de tous les sites sur lesquels l'application se diffuse. Les informations sur les membres du groupe sont gérées par le `FTM` GTF.

Enfin, les noms des ports des processus en ré-exécution sont reconstruits à la suite d'occurrences de fautes. Le mécanisme de localisation du micro-noyau les retrouve automatiquement. Toutefois, la localisation des ports ayant changé de site générera un nombre important de diffusions dans le réseau local.

4.5.6 Le recouvrement arrière

Les défaillances de nœuds sont détectées par les `PM`s via la rupture des connexions `TCP` inter-sites[17]. Le `PM` détectant la défaillance d'un nœud lors de l'émission d'un message inter-processus en avertit le `FTM`. Nous considérons que la détection des défaillances de sites est un problème orthogonal au recouvrement arrière. Par conséquent, les détections à d'autres instants et par d'autres serveurs ne sont pas traitées [Sens94].

Les actions du recouvrement arrière s'enchaînent comme indiqué dans le chapitre 3. Ci-après, nous détaillons uniquement la récupération de l'état d'un processus `MiX`. La figure 4.15 en trace les 10 étapes pour le processus P_i.

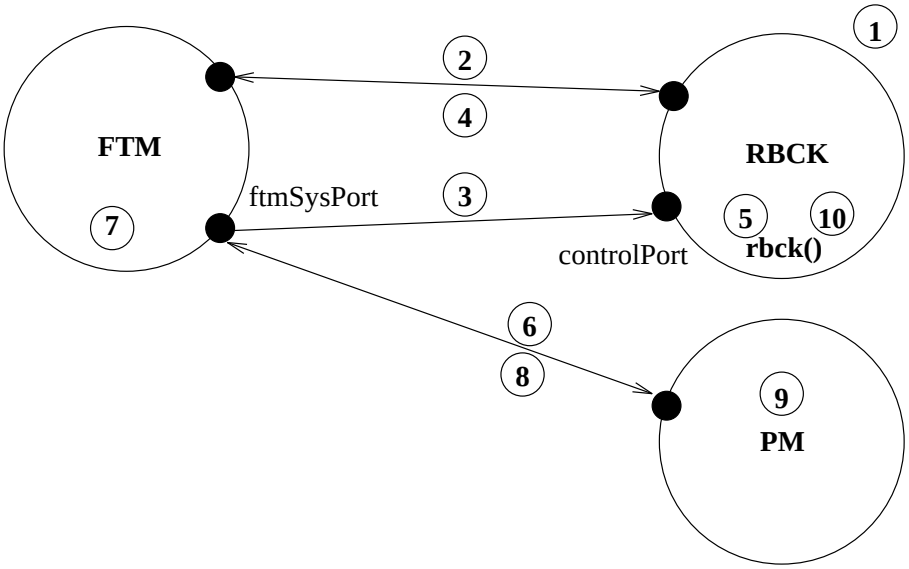


FIG. 4.15 – La récupération d'un processsus de l'application - réalisation dans `Chorus/Mix`.

Pour des raisons de programmation du `PM`, il n'est pas possible d'envisager la récupération de l'état d'un processus `MiX` par la programmation du gestionnaire de message attaché au port de contrôle. Il a donc fallu ajouter l'appel système `rbck` à l'interface du sous-système `Chorus/MiX`. Le processus en ré-exécution est démarré par le processus `UFTM`

17. Dans `Chorus v3r5`, les micro-noyaux des sites ouvrent une connexion avec chacun des micro-noyaux des autres sites du réseau local.

à la demande du FTM. Les 4 premières étapes sont celles du démarrage de l'utilitaire RBCK (*cf.* sous-section 4.5.3).

5. Le processus RBCK exécute l'appel système rbck. Son exécution se poursuit donc dans le PM. L'appel système rbck commence par exécuter l'équivalent d'un appel système execve(2v) pour récupérer les régions mémoire du code de P_i.

6. Le PM émet un message de demande de l'état de reprise de P_i vers le FTM.

7. Le FTM récupère en mémoire stable l'état et la mémoire virtuelle de P_i. Ainsi, l'espace d'adressage de P_i est mis à jour. Pour minimiser le nombre de copies mémoire des pages, la fonction de lecture sur fichier de la libraire libMiX.a a été ré-écrite pour éliminer la première copie dans le 'buffeur d'écriture du FTM.

8. Le FTM transmet l'état de reprise de P_i au PM.

9. Le PM met à jour l'état du processus P_i.

10. La ré-exécution peut commencer.

4.5.7 Discussion

Nous ne connaissons pas d'autre implantation du recouvrement arrière autour du micro-noyau Chorus. Il existe toutefois quelques réalisations de mécanismes de migration.

Dans [O'Co93, O'Co94], ce sont les acteurs au dessus du micro-noyau Chorus v3r2 qui sont migrés. Le mécanisme de migration est divisé en trois modules: le module de saisie de l'état d'un acteur, le module du choix de l'instant de migration et le module de transfert de l'état. Celui qui nous intéresse est le module de saisie de l'état d'un acteur. Il rassemble les actions non fournies par le micro-noyau pour la saisie de l'état. Il s'agit des appels système encapsulant l'état des objets acteur, activité et port. Le module de saisie de l'état de l'acteur attend que toutes les activités ne soient plus en train de s'exécuter dans le micro-noyau pour commencer la migration. Pour le recouvrement arrière de processus MiX, nous n'avons pas eu besoin de saisir les objets du micro-noyau. La raison principale est que les processus MiX sont mono-activité et les opérations de gestion de l'activité sont toutes définies dans le PM. Il a donc suffi de ré-utiliser les fonctions programmées dans le PM.

Dans [Phil95], ce sont les processus MiX possédant plusieurs activités qui sont migrés. L'article donne peu de détails d'implantation. Néanmoins, il semble que l'implantation de la saisie de l'état soit similaire à la nôtre. Avant la saisie de l'état, le PM attend que les activités ne soient pas en exécution d'un appel système et arrête leur exécution. Mais, l'article passe sous silence le cas où les activités sont en inter-blocage sur des opérations bloquantes.

4.6 Les apports et les limites de la technologie micro-noyau Chorus

Dans cette section, nous reprenons les limites citées dans la section 4.3 et évaluons l'apport de la technologie micro-noyau Chorus. Parmi les limites citées, nous estimons que

l'une d'entre elles n'est pas encore résolue de manière satisfaisante : l'interruption d'un appel système bloquant.

1. La saisie de l'état d'un processus est déclenchée par la réception d'un message sur le port de contrôle. L'état du processus est modifié pour provoquer sa saisie au retour de l'appel système. En conséquence, la tolérance aux fautes est transparente.

2. Le port de contrôle permet d'intervenir par message sur l'état du processus. Le nombre de fonctionnalités qu'il est possible d'ajouter simultanément n'est pas limité par le nombre de signaux `UNIX`.

3. La saisie de l'état est programmée dans le `PM`. Elle est donc transparente au processus.

4. L'état d'un processus est entièrement défini dans le sous-système `Chorus/MiX` - ne comprend aucun objet du micro-noyau. Cependant, nous n'avons pas trouvé comment est géré le bit de modification des pages de la mémoire centrale. Nous espérons qu'il est possible de les gérer à partir du sous-système `Chorus/MiX`.

5. Un acteur possède un nom global autorisant la manipulation des ressources qu'il héberge. Il s'ensuit que l'espace d'adressage du processus peut être enregistré en mémoire stable par le `FTM`. Toutes les optimisations[18] peuvent être implantées. Remarquons que le micro-noyau implante déjà la "copie sur écriture". Enfin, le concept de région mémoire réalisé par le micro-noyau rend aisées les opérations sur la mémoire virtuelle.

6. Le problème de l'interruption des appels bloquants n'est toujours pas résolu de manière satisfaisante. Une analyse approfondie est nécessaire pour savoir si la saisie de l'état du sous-système `Chorus/MiX` est réalisable sans modification du micro-noyau.

7. Le nommage et la localisation des ports sont transparents. C'est une avancée importante par rapport aux systèmes d'exploitation traditionnels tel que `SunOS` et un pas significatif vers la conception de système à image unique.

8. Le concept d'acteur multi-activités permet de paralléliser les actions du mécanisme global. Les actions concernant un même processus ou un même mécanisme de base sont sérialisées.

9. Toute interaction au-dessus du micro-noyau peut être exprimée par des messages. C'est même le cas pour les interruptions et les exceptions. En conséquence, il est aisé de rendre les actions du `FTM` atomiques.

4.7 Les mesures de performances de l'implantation dans `Chorus/MiX`

Nous avons tenté d'effectuer les mêmes types de mesures que pour l'implantation au dessus de `SunOS`. Les programmes de tests sont les mêmes. Cependant, les appels système d'émission et de réception sur les `sockets` sont remplacés par les appels correpondant pour les `IPCs`. Malheureusement, l'utilitaire `getrusage(2)` dont nous disposons sur `SunOS` n'est pas disponible dans `Chorus/MiX`. Le micro-noyau `Chorus` fournit un acteur `PROF` pour profiler les acteurs, donc les processus `MiX`. Pour des raisons encore inexpliquées,

18. excepté peut-être les points de reprise incrémentaux pour lesquels nous n'avons pas résolu la gestion des bits de modification des pages de la mémoire centrale.

nous n'avons pas réussi a l'utiliser. Par conséquent, les seules mesures que nous sommes capables de fournir à ce jour sont des durées. Pour les obtenir, nous utilisons l'appel système `gettimeofday(2)` qui possède une précsion de 10ms, et, le profileur du dévermineur du micro-noyau `KDB` qui possède une précision de 1μs.

4.7.1 La constitution de points de reprise

L'enregistrement en mémoire stable de l'état des processus de l'application répartie est effectué par le `FTM`. L'exécution des processus est inhibée uniquement le temps du *pseudo*-`fork` et du dialogue avec le `FTM`. L'inhibition est évaluée à 18,133ms $\pm$ 1,722ms, ceci quelque soit la taille de l'espace d'adressage alloué, dans la limite de 5Mo. À titre de comparaison, l'appel système `fork(2)` s'exécute en moyenne en 7ms.

L'unité de transfert de l'espace d'adressage vers la mémoire stable est la page mémoire (soit 4Ko). Dans cette configuration, sauvegarder en mémoire stable 100Ko pendant que le processus exécute une boucle infinie, prend en moyenne 7s. Nous attendons de pouvoir nous servir de l'acteur `PROF` pour évaluer précisément le surcoût d'exécution et l'inhibition dus à cette sauvegarde en mémoire stable. Il sera aussi intéressant de faire varier la taille du tampon d'enregistrement.

4.7.2 La journalisation de l'exécution

La journalisation de l'histoire répartie est effectuée par le `FTM` du nœud récepteur. La journalisation d'un message interprocessus nécessite 3 messages courts intra-nœuds (2 lors de l'émission, 1 à la réception). Contrairement à l'implantation au dessus de `SunOS`, l'histoire répartie est sauvegardée en mémoire stable.

Les tailles des messages sont les mêmes que pour les mesures dans l'implantation `SunOS`. Rappelons en effet que les `IPC Chorus` passent par des canaux de communication `TCP`. Les programmes de test sont presque les mêmes. La différence est que les pages mémoire du processus sont modifiées entre deux émissions, ceci pour éviter l'effet bénéfique des copies-sur-écriture du micro-noyau. L'écart type observé est toujours inférieur à 10% de la valeur moyenne.

Le tableau 4.6 montrent les débits obtenus pour 4 configurations différentes : sans tolérance aux fautes, journalisations uniquement du premier niveau, optimiste, et, pessimiste. Nous pouvons faire les remarques suivantes :

En considérant que l'écart type peut atteindre 10% de la valeur moyenne, nous déduisons que les performances du système avec la tolérance aux fautes sont équivalentes quelle que soit la méthode de journalisation. Le fait que la journalisation pessimiste n'écroule pas le débit signifie que la concomitance du mécanisme de journalisation est bonne.

La dégradation maximale des performances observée est de 35%. C'est un peu moins que la meilleure valeur rencontrée dans l'implantation `SunOS`.

Enfin, dans le tableau 4.7, l'inhibition est évaluée en mesurant :

- du côté de l'émetteur, la durée entre le début de l'appel système `u_ipcSend(2MIX)` et l'appel à `ipcSend(K)`,
- du côté du récepteur, la durée entre le retour de l'appel à `ipcReceive(K)` et la fin de l'appel système `u_ipcReceive(2MIX)`.

Il nous faudrait le surcoût global du mécanisme de journalisation (à obtenir avec `PROF`) pour calculer la concomitance.

Prototype	**Taille des messages** (octet)	**Débit** (msg/s)
Sans	1	349
tolérance aux fautes	1 000	186
	2 000	120
Tolérance aux fautes	1	238
avec 1er niveau	1 000	158
de journalisation	2 000	111
Tolérance aux fautes	1	232
avec journalisation	1 000	150
optimiste	2 000	109
Tolérance aux fautes	1	228
avec journalisation	1 000	146
pessimiste	2 000	108

TAB. 4.6 – Le surcoût engendré par la journalisation des messages - réalisation dans `Chorus/MiX`.

Prototype	**Taille des messages** (octet)	**Durée de l'appel u_ipcSend(2MIX)** (ms)	**Durée de l'appel u_ipcReceive(2MIX)** (ms)
Sans	1	0,83	0,06
tolérance aux fautes	1 000	0,83	0,06
	2 000	0,83	0,06
Avec	1	1,33	0,30
tolérance aux fautes	1 000	1,54	0,30
	2 000	1,91	0,30

TAB. 4.7 – L'inhibition engendrée par la journalisation des messages - réalisation dans `Chorus/MiX`.

Conclusion

La tolérance aux fautes est un domaine complexe nécessitant une définition précise des fautes, des systèmes et des applications supportés. Les fautes que nous avons considérées sont accidentelles, permanentes et d'origine matérielle ou bien humaine (procédurale ou de maintenance), ou encore environnementale. Les erreurs détectées sont des arrêts francs (de processeurs) de stations de travail. Le mode de défaillance est le mode "silence sur défaillance". De plus, les défaillances sont perçues de manière cohérente et les conséquences sont bénignes. Nous nous sommes intéressés aux applications commerciales à usage général - tolérant les interruptions momentanées du service, pourvu que les rétablissements soient automatiques - s'exécutant sur les nœuds de réseaux de stations de travail faiblement couplés. Ces applications ne sont pas critiques. La méthode employée doit donc être "légère".

Le deuxième axe d'étude est l'intégration des mécanismes de la tolérance aux fautes dans les systèmes d'exploitation. Aussi, notre travail ne porte pas spécialement sur les techniques de génie logiciel ni sur les paradigmes de programmation de la tolérance aux fautes, mais sur les abstractions du système d'exploitation. La méthode de tolérance aux fautes utilisée est la reprise sur erreur par recouvrement arrière. Elle est fondée sur le concept des processus fiables, qui lui-même dépend de la gestion de groupe, de la diffusion, de l'horloge globale commune et de la mémoire stable.

Les résultats que nous avons obtenus se résument ainsi.

Premièrement, la littérature sur le sujet est abondante. Mais, à notre connaissance, aucun article de synthèse n'analysait et ne comparait les différentes politiques possibles. Notre travail débute par une recherche (la plus exhaustive possible) des modèles, des politiques et des mécanismes du recouvrement arrière. À partir de cette étude bibliographique, nous dégageons les problèmes (le déterminisme d'exécution et la cohérence d'état global) et les objectifs (le degré de tolérance aux fautes, les surcoûts, l'inhibition et la quantité de travail à défaire ou à refaire). Puis, en déclinant les propriétés du déterminisme et de la cohérence, nous classons les politiques en trois catégories suivant les types d'applications réparties : les applications indéterministes, les applications déterministes à communications inter-processus intenses et les applications déterministes à grand nombre de processus. Ensuite, en tenant compte des objectifs, nous déterminons les politiques possibles (efficaces) et souhaitables (efficientes). Ces résultats sont résumés dans le tableau 2.1 à la page 46. Enfin, les principes des algorithmes des mécanismes de base des différentes classes sont présentés.

Parallèlement à l'étude algorithmique, nous présentons les problèmes d'implantation des mécanismes de base. Tout d'abord, le concept des processus fiables s'appuie sur quatre autres abstractions du système d'exploitation : la mémoire stable, les communications fiables, la gestion de groupe, la diffusion et l'horloge globale commune. Il faut aussi y ajouter d'autres pré-requis communs aux mécanismes de migration regroupés sous la propriété

de “transparence”. L'implantation des mécanismes de base est d'autant plus compliquée que le système d'exploitation ne réalise pas la transparence du nommage et la transparence de la localisation. Par ailleurs, le mécanisme de constitution des points de reprise, de loin le plus difficile à réaliser, pose le problème de la définition de l'état de reprise d'un processus. L'état volatile est automatiquement et facilement inclus dans l'état de reprise. Par contre, l'état persistant ainsi que l'état du système d'exploitation sont beaucoup plus difficiles à déterminer et à insérer dans l'état de reprise.

Deuxièmement, nous construisons un mécanisme global efficace et efficient supportant les applications *presque*-déterministes, et, transparent pour l'utilisateur. Ce mécanisme global simple comprend les mécanismes de base composés de leurs optimisations les plus efficientes dans le modèle de système réparti. En outre, il est assez souple pour permettre l'ajout d'extensions. Ces extensions autorisent le support d'applications composées de processus *presque*-déterministes et de processus indéterministes, ceci en contre-partie de la programmation par l'utilisateur de la tolérance aux fautes.

Le mécanisme global est étudié pour s'adapter aux systèmes répartis construits comme une agglomération de réseaux locaux. Les processus s'exécutant sur les nœuds d'un même réseau local sont groupés pour former une unité de reprise répartie (URR). Ainsi, les mécanismes de base bénéficient des abstractions disponibles dans un réseau local : système de gestion répartie de fichiers réalisant la mémoire stable et diffusion non fiable. En outre, un processus appelé le gestionnaire de la tolérance aux fautes (GTF) est ajouté à l'URR. Il transforme le groupe de processus en une unité de confinement, de détection, de reconfiguration, de récupération et de reprise.

Le mécanisme de constitution de points de reprise construit aussi bien des coupures cohérentes que des points de reprise non coordonnés. Ce choix résulte de la décomposition en trois phases d'une constitution : le lancement synchronisé ou non avec les autres processus, l'attente de l'enregistrement pour connaître les “constitutions concomitantes”, et l'assurance pour rendre le point de reprise persistant. La dernière phase permet aussi de re-synchroniser les processus pour la constitution du point de reprise suivant. La non-optimisation de la durée des constitutions de points de reprise permet de disperser dans le temps les sauvegardes en mémoire stable des informations de reprise.

Le mécanisme de journalisation autorise aussi bien une journalisation optimiste que pessimiste. Pour des raisons de simplicité, c'est le GTF qui valide les messages à destination du monde extérieur. Pour ce faire, les processus enregistrent leur histoire séquentielle en mémoire volatile du nœud sur lequel s'exécute le GTF. La journalisation étant centralisée, la taille des estampilles est optimale et la complexité de notre algorithme de validation est en $\mathcal{O}(\mid URR \mid)$ messages. Grâce aux vecteurs des dépendances directes journalisés, la validation d'un message ne nécessite pas le parcours de toute l'histoire répartie. Par ailleurs, le GTF n'a pas besoin d'être sûr de fonctionnement. En conséquence, l'application tolère les fautes globales et le GTF peut s'exécuter sur n'importe quel nœud du réseau local. Notre mécanisme est moins efficient que les mécanismes de journalisation répartie, qui sont en revanche plus compliqués.

L'objectif principal du mécanisme de recouvrement arrière est le maintien de la cohérence de l'état global. Puisque la journalisation peut être optimiste, nous devons exécuter un algorithme de calcul de l'état global recouvrable maximal avant le début de la ré-exécution. Par conséquent, l'application est arrêtée - en fait, blocage des émissions de message - pour vider les canaux de communication, calculer l'état global recouvrable maximal, reconfigurer l'URR, récupérer les informations de recouvrement et lancer la ré-exécution. L'aspect bloquant du mécanisme permet la prise en compte de mécanismes de

détection et de reconfiguration externes spécifiques à l'utilisateur ou à l'administrateur du système.

Troisièmement, le mécanisme global permet une ré-exécution équivalente et supporte des applications *presque*-déterministes, par défaut. Deux extensions sont étudiées qui améliorent encore l'efficience et agrandissent le champ d'action du recouvrement arrière. Tout d'abord, le traitement de la sémantique des transmissions de messages consiste à classer les messages en quatre catégories : exactement une fois, au moins une fois, au plus une fois et sans contrainte. Jusqu'à présent, la sémantique des transmissions des messages était exactement une fois. C'est le concepteur de l'application qui programme la sémantique. Notre travail a consisté à revoir les trois mécanismes de base pour supporter ces différentes sémantiques.

Ensuite, le mécanisme global est étendu pour la conception d'applications "mixtes" : composées de processus déterministes et de processus indéterministes. Le concepteur de l'application déclare lui-même l'ensemble des processus indéterministes. Les constitutions de points de reprise construisent des coupures cohérentes et les processus indéterministes ne journalisent pas leur histoire séquentielle. Ils ne transmettent que leur vecteur des dépendances directes lors de la validation d'un message. La validation d'un message m peut provoquer la construction d'une coupure cohérente si l'état précédant l'émission de m dépend d'une action indéterministe. Quant au mécanisme de recouvrement arrière, les processus dépendant de processus indéterministes défaillants ou orphelins sont orphelins et doivent donc se ré-exécuter.

Enfin, nous montrons comment les deux extensions peuvent être conjointement utilisées pour supporter ces applications "mixtes" à moindre coût. L'indéterminisme mesure, dans un sens, l'incapacité d'un processus à ré-exécuter de manière équivalente des actions internes. Le traitement de la sémantique des transmissions de messages consiste à enlever de la ré-exécution certaines émissions et certaines réceptions de messages. Par conséquent, la sémantique des transmissions de messages peut être utilisée pour limiter et contrôler la propagation de l'indéterminisme de processus en processus par message. Le concepteur de l'application regroupe les actions indéterministes dans certains processus qu'il déclare indéterministes et isole ceux-ci en programmant la sémantique des transmissions des messages échangés avec eux.

Quatrièmement, le mécanisme global développé a donné lieu à deux prototypes.

Le premier prototype est réalisé au dessus du noyau `SunOS`. Sa particularité est qu'il est entièrement portable. Cette implantation nous a permis d'évaluer les limites du concept de la portabilité. Ces limites sont de deux ordres. D'une part, un processus utilisateur ne peut pas se substituer au noyau pour certaines tâches. D'autre part, un processus utilisateur n'a accès qu'à un nombre restreint de fonctionnalités à travers un ensemble fini d'appels système. Les principales limites de la portabilité sont les suivantes : la non-transparence du recouvrement arrière - l'utilisateur doit au minimum faire une nouvelle édition de liens de ces programmes -, l'impossibilité d'optimiser les constitutions de points de reprise - un processus saisit lui-même son état et ne peut pas optimiser l'enregistrement de son espace d'adressage -, l'impossibilité d'accéder à l'état du noyau, et, le non-parallélisme - les processus `UNIX` standards sont des processus mono-activité. Les mesures sont effectuées avec deux prototypes : un deuxième prototype a été ébauché afin de comparer les journalisations par l'émetteur (cas du premier prototype) et par le récepteur. L'objectif des mesures est d'évaluer le surcoût d'exécution et l'inhibition maximum. Les tests du mécanisme de journalisation montrent que le facteur de dégradation des performances est compris entre 14 et 3 pour la journalisation par l'émetteur, et, entre 10 et 1,5 pour la journalisation

par le récepteur. Si l'utilisateur choisit une journalisation optimiste, la journalisation par le récepteur s'avère deux fois plus efficiente que la journalisation par l'émetteur. Pour les deux prototypes, dans ces conditions extrèmes, la concomitance est évaluée à seulement 30%. Cela explique le coût principal de la portabilité. L'évaluation montre aussi que le GTF n'est pas un goulet d'étranglement si la journalisation est optimiste, et qu'il peut le devenir si elle est pessimiste. Quand à la constitution des points de reprise, la durée d'inhibition des procesus est proportionnelle à la vitesse d'enregistrement de la taille des zones de données et de pile des processus par le système de gestion répartie des fichiers.

Une deuxième implantation du mécanisme global est réalisée dans `Chorus/MiX`. Aucune modification du micro-noyau n'a été nécessaire. Cette deuxième implantation a pour but l'évaluation des apports et des limites de la technologie micro-noyau pour la construction de système intégrant le recouvrement arrière. Les apports se résument comme suit : le contrôle aisé d'un processus - l'état d'un processus peut être modifié par une entité extérieur grâce au gestionnaire de message attaché au port de contrôle (inconnu de l'activité principale) -, l'accès aux ressources d'un acteur par son nom - la sauvegarde de l'espace d'adressage est effectuée par un acteur et peut donc être optimisée -, la manipulation aisée de la mémoire virtuelle - avec la notion de région mémoire -, la transparence du nommage et de la localisation - les ports sont fiables -, le parallélisme - les acteurs possèdent plusieurs activités. Pour le mécanisme de journalisation, les mesures effectuées montrent que, toujours dans les cas les plus défavorables, la dégradation des performances est toujours inférieures à 30%, que la journalisation soit optimiste ou pessimiste. Cela laisse présager une bonne concomitance. Quant à la constitution des points de reprise, les processus sont interrompus pendant moins de 20ms - l'équivalent de moins de 3 appels à `fork()/exit()`.

Enfin, dans les deux implantations, nous n'avons pas été capables de construire de coupures cohérentes. En effet, dans le cas de communications non FIFO, la construction de coupures cohérentes suppose que l'on puisse saisir l'état d'un processus avant de recevoir un message marqué. Dans `SunOS`, un processus ne peut pas interrompre l'appel système de réception de message pour saisir son état. Dans `Chorus/MiX`, une analyse approfondie est nécessaire pour savoir si la saisie de l'état du sous-système `Chorus/MiX` est réalisable sans modification du micro-noyau.

Pour conclure, nous voudrions mentionner quelques perspectives et suites possibles aux recherches entreprises.

Au niveau algorithmique, on pourrait :

- intégrer les mécanismes de recouvrement arrière conçus pour les systèmes fortement couplés,
- intégrer les mécanismes de recouvrement arrière conçus pour la mémoire partagée répartie,
- ré-évaluer l'efficience du mécanisme global pour les réseaux à haut débit,
- adapter le mécanisme global pour les systèmes mobiles,
- approfondir les analogies avec le modèle des variables partagées et le modèle orienté-objet pour découvrir de nouvelles voies de recherche (telles que le traitement de la sémantique des transmissions de messages),
- prendre en compte de nouveaux modèles de fautes, etc.

En ce qui concerne la conception de système d'exploitation réparti, on pourrait :

- prendre en compte l'état persistant de l'application,
- prendre en compte l'état du système d'exploitation,
- permettre l'interruption d'un appel système bloquant,
- supporter les échanges par mémoire partagée et ensuite accepter les acteurs multi-activités,
- repérer de manière automatique les actions indéterministes des processus, etc.

Annexe A

Le détail des actions des mécanismes de base

A.1 Les structures de données, les messages et les temporisations

A.1.1 Le mécanisme de constitution de points de reprise

Les structures de données maintenues par chaque processus P_i

- pr_sync_i : booléen évalué à "faux" ssi P_i participe aux constitutions de points de reprise synchronisées. pr_sync_i est initialisé à "faux".
- $pr\_en\_cours_i$: vecteur de booléens tel que $pr\_en\_cours_i[j]$ est évalué à "vrai" ssi, à la connaissance de P_i, P_j est en cours de constitution d'un point de reprise. Toutes les composantes de ce vecteur sont initialisées à "faux".
- vdd_i : vecteur des dépendances directes construit par P_i. Toutes les composantes de ce vecteur sont initialisées à 0.
- nra_i : numéro de l'exécution du mécanisme de recouvrement arrière. nra_i est initialisé à 0.

Les structures de données maintenues par le GTF

- pr_sync_{GTF} : vecteur de booléens tel que $pr\_sync_{GTF}[i]$ est évalué à "vrai" ssi P_i participe aux constitutions de points de reprise synchronisées. Toutes les composantes de ce vecteur sont initialisées à "faux".
- $pr\_en\_cours_{GTF}$: vecteur de booléens tel que $pr\_en\_cours_{GTF}[i]$ est évalué à "vrai" ssi P_i est en cours de constitution d'un point de reprise. Toutes les composantes de ce vecteur sont initialisées à "faux".
- **AS** : ensemble des processus ayant leur point de reprise en cours d'assurance - pour lesquels le message $m\_pr\_notif^i_{GTF}$ a été reçu et le message $m\_pr\_assur^{GTF}_i$ non émis. Cet ensemble est utilisé dans l'algorithme d'assurance (*cf.* figure 3.2). **AS** est initialisé à l'ensemble vide.

- $\mathbf{C}_{P_i}$: ensemble des processus qui, à la connaissance de P_i, appartiennent à la même coupure concomitante. Il existe un ensemble par processus. Ces ensembles sont initialisés à l'ensemble vide.

- **Coupure** : ensemble des processus participant à la même coupure concomitante. Cet ensemble est utilisé dans l'algorithme d'assurance. **Coupure** est initialisé à l'ensemble vide.

- nra_{GTF} : numéro de l'exécution du mécanisme de recouvrement arrière. nra_{GTF} est initialisé à 0.

Les messages

- $m\_pr\_sync^i_{GTF}$: transmission du résultat de la décision de participer ou non aux constitutions synchronisées.

- $m\_pr\_ordre^*_*$: ordre de constitution de points de reprise.

- $m\_pr\_refus^{GTF}_i$: refus par le GTF d'une constitution de points de reprise synchronisée.

- $m\_pr\_notif^i_{GTF}$: notification par un processus de la fin de la constitution d'un point de reprise.

- $m\_pr\_assur^{GTF}_i$: assurance par le GTF de la constitution d'un point de reprise.

Les temporisations

- τ^0_i : durée maximale entre deux constitutions de points de reprise.

- τ^1_i : durée maximale entre la saisie de l'état et l'enregistrement en mémoire stable des messages.

- τ^2_i : durée maximale d'attente du message d'assurance $m\_pr\_assur^{GTF}_i$.

- τ^3_i : durée minimale entre deux constitutions de points de reprise.

A.1.2 Le mécanisme de journalisation de l'exécution

Les structures de données maintenues par chaque processus P_i

- noc_i : numéro de communication. noc_i est initialisé à 0.

- noe_i : numéro d'émission. noe_i est initialisé à 0.

- nor_i : numéro de réception. nor_i est initialisé à 0.

- $\mathbf{M}_i$: ensemble des contenus des messages reçus et gardés en mémoire volatile. $\mathbf{M}_i$ est initialisé à l'ensemble vide.

- $\mathbf{E}_i$: ensemble des estampilles des messages reçus et gardées en mémoire volatile. $\mathbf{E}_i$ est initialisé à l'ensemble vide.

- ve_i : vecteur des numéros d'émission tel que $ve_i[j]$ est égal à noe_i lors de la dernière émission de message m_j^i. Toutes les composantes de ve_i sont initialisées à 0.

- vdd_i : vecteur des dépendances directes tel que $vdd_i[j]$ est égal à noe_j lors de la dernière réception de message m_i^j, et tel que $vdd_i[i]$ est égal à nor_i lors de la dernière réception de message m_*^j. Toutes les composantes de vdd_i sont initialisées à 0.

- je_vdd_i : vecteur des dépendances directes au moment de la dernière journalisation par un message $m\_je\_proc_{GTF}^i$. Toutes les composantes de je_vdd_i sont initialisées à 0.

- n_mis_i : nombre de messages émis depuis la dernière constitution de points de reprise. n_mis_i est initialisé à 0.

- ν_i^4 : nombre maximum de messages émis entre deux constitutions de points de reprise. ν_i^4 est un paramètre donné par l'utilisateur.

- nra_i : numéro de l'exécution du mécanisme de recouvrement arrière. nra_{GTF} est initialisé à 0.

Les structures de données maintenues par le GTF

- noc_{GTF} : numéro de communication. noc_{GTF} est initialisé à 0.

- noe_{GTF} : numéro d'émission. noe_{GTF} est initialisé à 0.

- nor_{GTF} : numéro de réception. nor_{GTF} est initialisé à 0.

- $\mathbf{E}_{GTF}$: ensemble des estampilles des messages reçus et gardées en mémoire volatile. $\mathbf{E}_{GTF}$ est initialisé à l'ensemble vide.

- max_stable : vecteur de l'état global recouvrable maximal de l'URR. Toutes les composantes de max_stable sont initialisées à 0.

- $\mathbf{PRNV}_{GTF}$: ensemble des couples (point de reprise non validé,vecteur des dépendances directes). $\mathbf{PRNV}_{GTF}$ est initialisé à l'ensemble vide.

- $dprv_{GTF}$: vecteur des derniers points de reprise validés tel que $dprv_{GTF}[i]$ est égal à $vdd_i[i]$ du dernier point de reprise validé R_i^x de P_i. Toutes les composantes de $dprv_{GTF}$ sont initialisées à 0.

- $\mathbf{PRVD}_{GTF}$: ensemble des couples (point de reprise validé demandé,vecteur des dépendances directes). $\mathbf{PRVD}_{GTF}$ est initialisé à l'ensemble vide.

- n_recu_{GTF} : nombre de messages $m\_je\_proc_{GTF}^i$ reçus depuis la dernière sauvegarde en mémoire stable. n_recu_{GTF} est initialisé à 0.

- ν_{GTF}^5 : nombre maximum de messages $m\_je\_proc_{GTF}^i$ reçus entre deux sauvegardes des estampilles en mémoire stable. ν_{GTF}^5 est un paramètre donné par l'utilisateur.

- $n\_diff\_stable_{GTF}$: nombre de messages $m\_je\_proc_{GTF}^i$ reçus depuis la dernière diffusion non fiable du message $m\_je\_gtf_i^{GTF}$. $n\_diff\_stable_{GTF}$ est initialisé à 0.

- ν_{GTF}^6 : nombre maximum de messages $m\_je\_proc_{GTF}^i$ reçus entre deux diffusions non fiables du message $m\_je\_gtf_i^{GTF}$. ν_{GTF}^6 est un paramètre donné par l'utilisateur.

– nra_i : numéro de l'exécution du mécanisme de recouvrement arrière. nra_{GTF} est initialisé à 0.

Les messages

– $m\_je\_proc^i_{GTF}$: transmission des estampilles des messages reçus tels que $\forall(j \neq i), vdd_i[j] > je\_vdd_i[j]$.

– $m\_je\_gtf_i^{GTF}$: transmission du vecteur max_stable et du vecteur $dprv_{GTF}$.

– $m\_je\_valid^i_{GTF}$: demande de transmission d'un message à destination du monde extérieur à l'URR, donc demande de validation.

– $m\_je\_je_i^{GTF}$: demande de journalisation adressée lors de la validation d'un message à destination du monde extérieur.

Les temporisations

– τ_i^4 : durée maximale entre deux journalisations par le message $m\_je\_proc^i_{GTF}$.

– τ^5_{GTF} : durée maximale entre deux diffusions du message $m\_je\_gtf_i^{GTF}$.

– τ^6_{GTF} : durée maximale entre deux journalisations de l'histoire répartie par le GTF.

– τ^7_{GTF} : durée maximale entre deux calculs du vecteur $dprv_{GTF}$.

A.1.3 Le mécanisme de recouvrement arrière

Les structures de données maintenues par chaque processus P_i

– $\mathbf{D}_i$: ensemble des processus défaillants. $\mathbf{D}_i$ est initialisé à l'ensemble vide. Lorsque $\mathbf{D}_i$ n'est pas égal à l'ensemble vide, cela signifie que l'URR est en cours de recouvrement arrière.

– $\mathbf{ND}_i$: ensemble des triplets (processus non défaillant,ve,vdd). $\mathbf{ND}_i$ est initialisé à l'ensemble vide.

– $\mathbf{ORPH}_i$: ensemble des triplets (processus orphelin,ve,vdd). $\mathbf{ORPH}_i$ est initialisé à l'ensemble vide.

– nra_i : numéro de l'exécution du mécanisme de recouvrement arrière. nra_i est initialisé à 0. nra_i prend la valeur de nra_{GTF} transmise par les messages $m\_ra\_arret_i^{GTF}$. P_i insère nra_i dans les messages qu'il émet (y compris les messages inter-processus) et initialise les temporisations avec nra_i. P_i exécute une action de réception de message uniquement si le numéro fourni par le message est supérieur ou égal à nra_i. De même, P_i exécute une action de fin de temporisation uniquement si le numéro associé est supérieur ou égal à nra_i. C'est ainsi une variante du numéro d'incarnation.

Les structures de données maintenues par le GTF

– $\mathbf{D}_{GTF}$: ensemble des processus défaillants. $\mathbf{D}_{GTF}$ est initialisé à l'ensemble vide. Lorsque $\mathbf{D}_{GTF}$ n'est pas égal à l'ensemble vide, cela signifie que l'URR est en cours de recouvrement arrière.

- $\mathbf{ND}_{GTF}$: ensemble des triplets (processus non défaillant,*ve*,*vdd*). $\mathbf{ND}_{GTF}$ est initialisé à l'ensemble vide.

- $\mathbf{ORPH}_{GTF}$: ensemble des triplets (processus orphelin,*ve*,*vdd*). $\mathbf{ORPH}_{GTF}$ est initialisé à l'ensemble vide.

- $\mathbf{NVEAU}_{GTF}$: ensemble des processus défaillants ou orphelins pour lesquels la nouvelle image est recréée. $\mathbf{NVEAU}_{GTF}$ est initialisé à l'ensemble vide.

- $\mathbf{PRET}_{GTF}$: ensemble des processus prêts pour la reprise. $\mathbf{PRET}_{GTF}$ est initialisé à l'ensemble vide.

- $\mathbf{R}_{GTF}$: ensemble des processus ayant commencé la reprise. $\mathbf{R}_{GTF}$ est initialisé à l'ensemble vide.

- $velr$: vecteur d'émission de la ligne de reprise. Toutes les composantes de $velr$ sont initialisées à 0.

- nra_{GTF} : nombre d'exécutions des actions `RECEVOIR`($m\_ra\_avert^{*}_{GTF}$). nra_{GTF} est initialisé à 0. Comme les processus de l'URR, le GTF insère nra_{GTF} dans les messages qu'il émet et dans les temporisations qu'il initialise. De même, il exécute les actions de réception de message et de fin de temporisation uniquement si le numéro fourni est supérieur ou égal à nra_{GTF}.

- *déjà_fait*1 et *déjà_fait*2: variables booléennes utilisées lorsqu'une entité externe intervient pour, respectivement, la détection et la reconfiguration.

Les messages

- $m\_ra\_avert^{*}_{GTF}$: avertissement de fautes émis à destination du GTF.

- $m\_ra\_arret_i^{GTF}$: ordre d'arrêt des émissions de messages inter-processus, ceci suite l'avertissement de fautes $m\_ra\_faute^{i}_{GTF}$.

- $m\_ra\_acq^{i}_{GTF}$: acquittement de l'ordre d'arrêt des émissions $m\_ra\_arret_i^{GTF}$.

- $m\_ra\_je_i^{GTF}$: demande de journalisation aux processus n'ayant pas terminé leur journalisation.

- $m\_ra\_detect\_ext_{GTF}$: fin de détection des fautes par une entité externe.

- $m\_ra\_pr_i^{GTF}$: ordre de constitution de points de reprise non coordonnés.

- $m\_ra\_notif^{i}_{GTF}$: notification de constitution d'un point de reprise non coordonné.

- $m\_ra\_reconf\_ext_{GTF}$: fin de reconfiguration par une entité externe.

- $m\_ra\_recup_i^{GTF}$: transmission des informations pour la récupération des messages demandés.

- $m\_ra\_reprise_i^{GTF}$: ordre de reprise.

Les temporisations

- τ^{9}_{GTF}: durée maximale entre deux émissions de messages $m\_ra\_arret_i^{GTF}$.

- τ_{GTF}^{10}: durée maximale entre deux demandes de journalisation $m\_ra\_je_i^{GTF}$.
- τ_{GTF}^{11}: durée maximale d'attente du message $m\_ra\_detect_{GTF}$.
- τ_{GTF}^{12}: durée maximale entre deux détections par un `ping`(...) de la défaillance des processus en constitution de points de reprise.
- τ_{GTF}^{13}: durée maximale d'attente du message $m\_ra\_reconf\_ext_{GTF}$.
- τ_{GTF}^{15}: durée maximale entre deux détections par un `ping`(...) de la défaillance des processus en récupération.
- τ_{GTF}^{16}: durée maximale entre deux demandes de reprise $m\_ra\_reprise_i^{GTF}$.
- τ_i^{17}: durée maximale entre deux détections par un `ping`(...) de la défaillance du GTF.

A.2 Le mécanisme de constitution de points de reprise

```
RECEVOIR (m_pr_sync^i_GTF) ≝
01 {
02   booléen pr_syn ;
03   entier nra ;
04   récupérer_nra(nra, m_pr_sync) ;
05   Si (nra ≥ nra_GTF) Alors
06     récupérer_pr_sync(pr_syn, m_pr_sync) ;
07     pr_sync_GTF[i] := pr_sync_i ;
08   Finsi
09 }
```

```
fin (τ_i^0, nra) ≝
01 {
02   Si (nra ≥ nra_i) Alors
03     Si (pr_sync_i) Alors
04       émettre(GTF, m_pr_ordre) ;
05     Sinon
06       émettre(P_i, m_pr_ordre) ;
07     Finsi;
08   Finsi
09 }
```

```
RECEVOIR (m_pr_ordre_i^GTF) ≝
01 {
02   entier nra ;
03   récupérer_nra(nra, m_pr_ordre) ;
04   Si (nra ≥ nra_i) Alors
05     Si (¬pr_en_cours_i[i] ∧ pr_sync_i) Alors
06       enlever(τ_i^0) ;
07       C_{R_i^x} ;
08       Pour j ∈ URR − {GTF} Faire
09         pr_en_cours_i[j] := pr_en_cours_GTF[j] ∨ pr_en_cours_i[j] ;
```

```
10        Finpour;
11        init(τ_i^1, nra_i) ;
12      Finsi;
13    Finsi
14  }
```

RECEVOIR $(m\_pr\_ordre^i_{GTF}) \stackrel{\text{def}}{=}$

```
01  {
02    entier nra ;
03    récupérer_nra(nra, m_pr_ordre) ;
04    Si (nra ≥ nra_GTF) Alors
05      Pour j ∈ URR − {GTF} Faire
06        Si (pr_en_cours_GTF[j]) Alors
07          émettre(P_i, m_pr_refus) ;
08          retourner;
09        Finsi;
10      Finpour;
11      Pour j ∈ URR − {GTF} Faire
12        Si (pr_sync_GTF[j]) Alors
13          pr_en_cours_GTF[j] := vrai;
14        Finsi;
15      Finpour;
16      ajouter_nra(m_pr_ordre)
17      diffuser(m_pr_ordre) ;
18    Finsi
19  }
```

RECEVOIR $(m\_pr\_refus_i^{GTF}) \stackrel{\text{def}}{=}$

```
01  {
02    entier nra ;
03    récupérer_nra(nra, m_pr_refus) ;
04    Si (nra ≥ nra_GTF) Alors
05        init(τ_i^0, nra_i) ;
06    Finsi
07  }
```

fin $(\tau_i^1, nra) \stackrel{\text{def}}{=}$

```
01  {
02    booleen optimisation := faux;
03    Si (nra ≥ nra_i) Alors
04      Pour j ∈ URR − {GTF} Faire
05        Si (pr_en_cours_i[j]) Alors
06          enreg_msg_optimisé(j) ;
07          optimisation := vrai;
08        Sinon
09          enreg_msg__non_optimisé(j) ;
10        Finsi;
11      Finpour;
12      Si (optimisation) Alors
```

```
13        init(τ_i^2, nra_i) ;
14      Sinon
15        init(τ_i^0, nra_i) ;
16      Finsi;
17      ajouter_pr_en_cours(m_pr_notif) ;
18      ajouter_nra(m_pr_notif) ;
19      émettre(GTF, m_pr_notif) ;
20    Finsi
21  }
```

RECEVOIR $(m\_pr\_notif^i_{GTF}) \stackrel{\text{def}}{=}$

```
01  {
02    vecteur pr_en_cours ;
03    entier nra ;
04    récupérer_nra(nra, m_pr_notif) ;
05    Si (nra ≥ nra_GTF) Alors
06      récupérer_pr_en_cours(pr_en_cours, m_pr_notif) ;
07      Pour (j ≠ i) ∈ URR − {GTF} Faire
08        Si (pr_en_cours_i[j]) Alors
09          C_Pi := C_Pi ∪ {P_j} ;
10        Finsi;
11      Finpour;
12      Assurance(P_i) ;
13    Finsi
14  }
```

fin $(\tau_i^2, nra) \stackrel{\text{def}}{=}$

```
01  {
02    Si (nra ≥ nra_i) Alors
03      Pour j ∈ URR − {GTF} Faire
04        pr_en_cours_i[j] := faux;
05        enreg_msg_non_optimisé(j) ;
06      Finpour;
07      init(τ_i^0, nra_i) ;
08      init(τ_i^3, nra_i) ;
09    Finsi
10  }
```

RECEVOIR $(m\_pr\_assur_i^{GTF}) \stackrel{\text{def}}{=}$

```
01  {
02    entier nra ;
03    récupérer_nra(nra, m_pr_assur) ;
04    Si (nra ≥ nra_i) Alors
05      Si (pr_en_cours_i[i]) Alors
06        enlever(τ_i^2) ;
07        Pour j ∈ URR − {GTF} Faire
08          pr_en_cours_i[j] := faux;
09        Finpour;
10        init(τ_i^3, nra_i) ;
11      Finsi
```

```
12    Finsi
13  }
```

```
fin (τ_i^3, nra) ≝
01  {
02    Si (nra ≥ nra_i) Alors
03      init(τ_i^0 − τ_i^3, nra_i) ;
04    Finsi
05  }
```

A.3 Le mécanisme de journalisation de l'exécution

```
ÉMETTRE (m_j^i) ≝
01  {
02    message m ;
03    ensemble Nouveau = ∅ ;
04    Si (distant(j)) Alors
05      enlever(τ_i^4) ;
06      noc_i ++ ;
07      noe_i := noc_i ;
08      ve_i[GTF] := noe_i ;
09      ajouter_noe(m_j^i) ;
10      M_i := M_i ∪ {m_j^i} ;
11      Pour m ∈ E_i Faire
12        Si (m_{noe_j} > je_vdd_i[j]) Alors
13          Nouveau := Nouveau ∪ {m} ;
14          je_vdd_i[i] := vdd_i[j]; ;
15        Finsi
16      Finpour
17      ajouter_Nouveau_vdd(m_je_valid) ;
18      ajouter_nra(m_je_valid) ;
19      émettre(GTF, m_je_valid) ;
20      attendre_recevoir(m_je_gtf) ;
21      n_recu_i := 0 ;
22      init(τ_i^4), nra_i ;
23    Sinon
24        Si (pr_en_cours_i[i]) Alors
25          marquer(m_j^i) ;
26        Finsi
27      noc_i ++ ;
28      noe_i := noc_i ;
29      ve_i[j] := noe_i ;
30      ajouter_noe(m_j^i) ;
31      M_i := M_i ∪ {m_j^i} ;
32      ajouter_nra(m_j^i) ;
33      émettre(P_j, m_j^i) ;
34    Finsi
35  }
```

```
RECEVOIR (m_i^j) ≝
01 {
02   message m ;
03   ensemble Nouveau = ∅ ;
04   entier nra ;
05   récupérer_nra(nra, m) ;
06   Si (nra ≥ nra_i) Alors
07     Si (distant(j)) Alors
08       noc_i ++ ;
09       nor_i := noc_i ;
10       vdd_i[i] := nor_i ;
11       vdd_i[GTF] := noe_j ;
12       ajouter_nor(m_i^j) ;
13       E_i := E_i ∪ {m_i^j} ;
14     Sinon
15       Si (marqué(m_i^j)) Alors
16         pr_en_cours_i[j] := vrai;
17         Si (¬pr_en_cours_i[j]) Alors
18           enlever(τ_i^0) ;
19           C_{R_i^x} ;
20           pr_en_cours_i[i] := vrai;
21           init(τ_i^1, nra_i) ;
22         Finsi
23       Finsi
24       noc_i ++ ;
25       nor_i := noc_i ;
26       vdd_i[i] := nor_i ;
27       vdd_i[j] := noe_j ;
28       ajouter_nor(m_i^j) ;
29       E_i := E_i ∪ {m_i^j} ;
30     Finsi
31     n_recu_i ++ ;
32     Si (n_recu_i ≥ ν_i^4) Alors
33       enlever(τ_i^4) ;
34       Pour m ∈ E_i Faire
35         Si (m_{noe_j} > je_vdd_i[j]) Alors
36           Nouveau := Nouveau ∪ {m} ;
37           je_vdd_i[i] := vdd_i[j]; ;
38         Finsi
39       Finpour
40       ajouter_Nouveau_vdd(m_je_proc) ;
41       ajouter_nra(m_je_proc) ;
42       émettre(GTF, m_je_proc) ;
43       n_recu_i := 0 ;
44       init(τ_i^4, nra_i) ;
45     Finsi
46     recevoir(P_j, m_i^j) ;
47   Finsi
48 }
```

```
fin (τ_i^4, nra) ≝
01 {
02   message m ;
03   ensemble Nouveau = ∅ ;
04   Si (nra ≥ nra_i) Alors
05     Si (n_recu_i ≥ 0) Alors
06       Pour m ∈ E_i Faire
07         Si (m_{noe_j} > je_vdd_i[j]) Alors
08           Nouveau := Nouveau ∪ {m} ;
09           je_vdd_i[i] := vdd_i[j]; ;
10         Finsi
11       Finpour
12       ajouter_Nouveau(m_je_proc) ;
13       ajouter_nra(m_je_proc) ;
14       émettre(GTF, m_je_proc) ;
15       n_recu_i := 0 ;
16     Finsi
17     init(τ_i^4, nra_i) ;
18   Finsi
19 }

fin (τ_GTF^5, nra) ≝
01 {
02   Si (nra ≥ nra_GTF) Alors
03     Si (n_recu_GTF > 0) Alors
04       enreg_E_GTF() ;
05       n_recu_GTF := 0 ;
06     Finsi
07     init(τ_GTF^6, nra_GTF) ;
08   Finsi
09 }

fin (τ_GTF^6, nra) ≝
01 {
02   Si (nra ≥ nra_GTF) Alors
03     Si (n_diff_stable_GTF > 0) Alors
04       ajouter_nra(m_je_gtf)
05       diffuser(m_je_gtf) ;
06       n_diff_stable_GTF := 0 ;
07     Finsi
08     init(τ_GTF^6, nra_GTF) ;
09   Finsi
10 }

RECEVOIR (m_je_proc^i_GTF) ≝
01 {
02   ensemble Nouveau = ∅ ;
03   entier nra ;
04   récupérer_nra(nra, m_je_proc) ;
05   Si (nra ≥ nra_GTF) Alors
```

```
06      récupérer_Nouveau(Nouveau, m_je_proc) ;
07      E_GTF ::= E_GTF ∪ Nouveau ;
08      Si (n_recu_GTF ≥ ν^5_GTF) Alors
09        enlever(τ^5_GTF) ;
10        enreg_E_GTF() ;
11        n_recu_GTF := 0 ;
12        init(τ^5_GTF, nra_GTF) ;
13      Finsi
14      Si (n_diff_stable_GTF ≥ ν^6_GTF) Alors
15        ajouter_nra(m_je_gtf)
16        diffuser(m_je_gtf) ;
17        n_diff_stable_GTF := 0 ;
18        init(τ^6_GTF, nra_GTF) ;
19      Finsi
20    Finsi
21  }
```

RECEVOIR $(m\_je\_je_i^{GTF}) \stackrel{\text{def}}{=}$

```
01  {
02    message m ;
03    ensemble Nouveau = ∅ ;
04    entier nra ;
05    récupérer_nra(nra, m_je_je) ;
06    Si (nra ≥ nra_i) Alors
07      enlever(τ^4_i) ;
08      Pour m ∈ E_i Faire
09        Si (m_noe_j > je_vdd_i[j]) Alors
10          Nouveau := Nouveau ∪ {m} ;
11          je_vdd_i[i] := vdd_i[j]; ;
12        Finsi
13      Finpour
14      ajouter_Nouveau(m_je_proc) ;
15      ajouter_nra(m_je_proc) ;
16      émettre(GTF, m_je_proc) ;
17      n_recu_i := 0 ;
18      init(τ^4_i, nra_i) ;
19    Finsi
20  }
```

RECEVOIR $(m\_je\_valid^i_{GTF}) \stackrel{\text{def}}{=}$

```
01  {
02    entier noe ;
03    entier nra ;
04    récupérer_nra(nra, m_je_valid) ;
05    Si (nra ≥ nra_GTF) Alors
06      enlever(τ^5_GTF) ;
07      enlever(τ^6_GTF) ;
08      récupérer_noe(noe, m_je_valid) ;
09      validation(P_i, noe_i) ;
10      enreg_E(E_GTF) ;
```

```
11      ajouter_nra(m_je_gtf) ;
12      émettre(P_i, m_je_gtf) ;
13      init(τ^5_GTF, nra_GTF) ;
14      init(τ^6_GTF, nra_GTF) ;
15    Finsi
16  }
```

fin $(\tau^7_{GTF}, nra) \stackrel{\text{def}}{=}$

```
01  {
02    vecteur vdd_i ;
03    Si (nra ≥ nra_GTF) Alors
04      Pour vdd_i ∈ PRNV_GTF Faire
05        Si (vdd_i[i] ≤ max_stable[i]) Alors
06          PRNV_GTF := PRNV_GTF − {vdd_i} ;
07          PRVD_GTF := PRVD_GTF ∪ {vdd_i} ;
08          Si (vdd_i[i] > dprv_GTF[i]) Alors
09            dprv_GTF[i] := vdd_i[i] ;
10          Finsi
11        Finsi
12      Finpour
13      Pour vdd_i ∈ PRVD_GTF Faire
14        Si (vdd_i[i] ≤ dprv_GTF[i]) Alors+
15          PRVD_GTF := PRVD_GTF − {vdd_i} ;
16          effacer_histoire(vdd_i, i) ;
17          effacer_pr(vdd_i, i) ;
18        Finsi
19      Finpour
20      init(τ^7_GTF, nra_GTF) ;
21    Finsi
22  }
```

RECEVOIR $(m\_je\_gtf_i^{GTF}) \stackrel{\text{def}}{=}$

```
01  {
02    vecteur max_stable , dprv ;
03    message m ;
04    entier nra ;
05    récupérer_nra(nra, m_je_gtf) ;
06    Si (nra ≥ nra_i) Alors
07      récupérer_max_stable_dprv(max_stable, dprv, m_je_proc) ;
08      Pour m ∈ E_i Faire
09        Si (m_nor_i ≤ max_stable[i]) Alors
10          E_i := E_i − {m} ;
11        Finsi
12      Finpour
13      Pour m ∈ M_i Faire
14        Si (m_nor_i ≤ dprv_GTF[i]) Alors
15          M_i := M_i − {m} ;
16        Finsi
17      Finpour
18    Finsi
```

```
19 }
```

A.4 Le mécanisme de recouvrement arrière

RECEVOIR $(m\_ra\_avert^{*}_{GTF}) \stackrel{\text{def}}{=}$

```
01 {
02   processus P_j ;
03   entier nra ;
04   récupérer_nra(nra, m_ra_avert) ;
05   Si (nra ≥ nra_GTF) Alors
06     nra_GTF ++ ;
07     enlever(τ^5_GTF) ;
08     enlever(τ^6_GTF) ;
09     enlever(τ^7_GTF) ;
10     enlever(τ^9_GTF) ;
11     enlever(τ^10_GTF) ;
12     enlever(τ^11_GTF) ;
13     enlever(τ^12_GTF) ;
14     enlever(τ^13_GTF) ;
15     enlever(τ^15_GTF) ;
16     enlever(τ^16_GTF) ;
17     Pour i ∈ URR − {GTF} Faire
18       pr_en_cours_GTF[i] := faux;
19     Finpour
20     ORPH_GTF := ∅ ;
21     NVEAU_GTF := ∅ ;
22     PRET_GTF := ∅ ;
23     R_GTF := ∅ ;
24     récupérer_id_défaillant(P_j, m_ra_avert) ;
25     Si (jnotin D_GTF) Alors
26       D_GTF := D_GTF ∪ {P_j} ;
27     Finsi
28     init(τ^9_GTF, nra_GTF) ;
29     ajouter_nra(m_ra_arret)
30     diffuser(m_ra_arret) ;
29   Finsi
30 }
```

RECEVOIR $(m\_ra\_arret_i^{GTF}) \stackrel{\text{def}}{=}$

```
01 {
02   processus P_j ;
03   entier nra ;
04   récupérer_nra(nra, m_ra_arret) ;
05   Si (nra ≥ nra_i) Alors
06     nra_i := nra ;
07     enlever(τ^0_i) ;
08     enlever(τ^1_i) ;
09     enlever(τ^2_i) ;
10     enlever(τ^3_i) ;
```

```
11      Pour j ∈ URR − {GTF} Faire
12        pr_en_cours_i[j] := faux;
13      Finpour
14      ND_i := ∅ ;
15      ORPH_i := ∅ ;
16      récupérer_id_défaillant(P_j, m_ra_arret) ;
17      D_i := D_i ∪ {P_j} ;
18      ajouter_id_défaillant(m_ra_acq) ;
19      ajouter_nra(m_ra_acq) ;
20      émettre(GTF, m_ra_acq) ;
21      init(τ_i^17, nra_i) ;
22    Finsi
23  }
```

RECEVOIR $(m\_ra\_acq^{i}_{GTF}) \stackrel{\text{def}}{=}$

```
01  {
02    vecteur ve_i , vdd_i ;
03    entier nra ;
04    récupérer_nra(nra, m_ra_acq) ;
05    Si (nra ≥ nra_GTF) Alors
06      Si (i ∉ ND_GTF) Alors
07        récupérer_ve_vdd(ve_i, vdd_i, m_ra_acq) ;
08        ND_GTF := ND_GTF ∪ {(P_i, ve_i, vdd_i)} ;
09      Finsi
10    Finsi
11  }
```

fin $(\tau^{9}_{GTF}, nra) \stackrel{\text{def}}{=}$

```
01  {
02    Si (nra ≥ nra_GTF) Alors
03      Pour i ∉ (ND_GTF ∪ D_GTF) Faire
04        ajouter_nra(m_ra_arret) ;
05        émettre(P_i, m_ra_arret) ;
06      Finpour
07      Si ((ND_GTF ∪ D_GTF) ≠ (URR − {GTF})) Alors
08        init(τ_GTF^9, nra_GTF) ;
09      Sinon
10        init(τ_GTF^10, nra_GTF) ;
11      Finsi
12    Finsi
13  }
```

fin $(\tau^{10}_{GTF}, nra) \stackrel{\text{def}}{=}$

```
01  {
02    boolen fin_je := vrai;
03    Si (nra ≥ nra_GTF) Alors
04      Pour i ∈ ND_GTF Faire
05        Pour (j ≠ i) ∈ ND_GTF Faire
06          Si (vdd_i[j] < ve_j[i]) Alors
07            fin_je := faux;
```

```
08          ajouter_nra(m_ra_je) ;
09          émettre(P_i, m_ra_je) ;
10        Finsi
11      Finpour
12    Finpour
13    Si (¬fin_je) Alors
14      init(τ^10_GTF, nra_GTF) ;
15    Sinon
16      Si (détection_externe()) Alors
17        enreg_E_GTF() ;
18        ajouter_nra(m_ra_detect_ext) ;
19        déjà_fait1 := faux;
20        émettre(externe, m_ra_detect_ext) ;
21      Finsi
22      init(τ^11_GTF, nra_GTF) ;
23    Finsi
24  Finsi
25 }
```

RECEVOIR $(m\_ra\_je_i^{GTF}) \stackrel{\text{def}}{=}$ RECEVOIR $(m\_je\_je_i^{GTF})$

RECEVOIR $(m\_ra\_detect\_ext_{GTF}) \stackrel{\text{def}}{=}$

```
01 {
02   ensemble D ;
03   entier nra ;
04   récupérer_nra(nra, m_ra_detect_ext) ;
05   Si (nra ≥ nra_GTF) Alors
06     Si (¬déjà_fait1) Alors
07       déjà_fait1 := vrai;
08       enlever(τ^11_GTF) ;
09       retirer_défaillants(D, m_je_ext) ;
10       ND_GTF := ND_GTF − D ;
11       État_maximal() ;
12       fin(τ^7_GTF, nra_GTF) ;
13       Pour i ∈ ND_GTF Faire
14         Si (vdd_i[i] > max_stable[i]) Alors
15           ND_GTF := ND_GTF − {P_i} ;
16           ORPH_GTF := ORPH_GTF ∪ {P_i} ;
17         Finsi
18       Finpour
19       ajouter_D_ND_ORPH(m_ra_pr) ;
20       Pour i ∈ ND_GTF Faire
21         ajouter_nra(m_ra_pr) ;
22         émettre(P_i, m_ra_pr) ;
23       Finpour
24       init(τ^12_GTF, nra_GTF) ;
25     Finsi
26   Finsi
27 }
```

```
fin (τ^11_GTF, nra) ≝
01 {
02   processus i ;
03   Si (nra ≥ nra_GTF) Alors
04     Si (¬déjà_fait1) Alors
05       déjà_fait1 := vrai;
06       État_maximal() ;
07       fin(τ^7_GTF, nra_GTF) ;
08       Pour i ∈ ND_GTF Faire
09         Si (vdd_i[i] > max_stable[i]) Alors
10           ND_GTF := ND_GTF − {P_i} ;
11           ORPH_GTF := ORPH_GTF ∪ {P_i} ;
12         Finsi
13       Finpour
14       ajouter_D_ND_ORPH(m_ra_pr) ;
15       Pour i ∈ ND_GTF Faire
16         ajouter_nra(m_ra_pr) ;
17         émettre(P_i, m_ra_pr) ;
18       Finpour
19       init(τ^12_GTF, nra_GTF) ;
20     Finsi
21   Finsi
22 }
```

```
RECEVOIR (m_ra_pr^GTF_i) ≝
01 {
02   pr_en_cours_i[i] := vrai;
03   entier nra ;
04   récupérer_nra(nra, m_ra_pr) ;
05   Si (nra ≥ nra_i) Alors
06     C_{R^x_i} ;
07     attente_fin_enreg_état() ;
08     Pour j ∈ URR − {GTF} Faire
09       enreg_msg_non_optimisé(j) ;
10     Finpour
11     ajouter_nra(m_ra_notif) ;
12     émettre(GTF, m_ra_notif) ;
13     pr_en_cours_i[i] := faux;
14   Finsi
15 }
```

```
RECEVOIR (m_ra_notif^i_GTF) ≝
01 {
02   entier nra ;
03   récupérer_nra(nra, m_ra_notif) ;
04   Si (nra ≥ nra_GTF) Alors
05     PRET_GTF := PRET_GTF ∪ {P_i} ;
06     Si (ND_GTF = PRET_GTF) Alors
```

```
07        Si (reconfiguration_externe()) Alors
08          ajouter_nra(m_ra_reconf_ext) ;
09          émettre(externe, m_ra_reconf_ext) ;
10        Finsi
11        déjà_fait2 := vrai;
12        init(τ^13_GTF, nra_GTF) ;
13      Finsi
14    Finsi
15  }
```

fin $(\tau^{12}_{GTF}, nra) \stackrel{\text{def}}{=}$

```
01  {
02    Si (nra ≥ nra_GTF) Alors
03      Pour j ∈ ND_GTF Faire
04        ping(...) ;
05      Finpour
06    Finsi
07  }
```

RECEVOIR $(m\_ra\_reconf\_ext_{GTF}) \stackrel{\text{def}}{=}$

```
01  {
02    entier nra ;
03    récupérer_nra(nra, m_ra_reconf) ;
04    Si (nra ≥ nra_GTF) Alors
05      Si (¬déjà_fait2) Alors
06        déjà_fait2 := vrai;
07        récupérer_nveau(NVEAU, m_ra_reconf) ;
08        enreg_configuration() ;
09        Pour i ∈ URR − ({GTF} ∪ PRET ∪ NVEAU)_GTF Faire
10          placer(P_i) ;
11          NVEAU_GTF := NVEAU_GTF ∪ {P_i} ;
12        Finpour
13        enreg_configuration() ;
14        efface_msg_orphelin() ;
15        Pour i ∈ NVEAU_GTF Faire
16          ajouter_info_ra(, m_ra_recup)
17          ajouter_nra(m_ra_recup) ;
18          émettre(P_i, m_ra_recup)
19        Finpour
20        init(τ^15_GTF, nra_GTF) ;
21      Finsi
22    Finsi
23  }
```

fin $(\tau^{13}_{GTF}, nra) \stackrel{\text{def}}{=}$

```
01  {
02    Si (nra ≥ nra_GTF) Alors
```

```
03      Si (¬déjà_fait2) Alors
04        déjà_fait2 := vrai;
05        Pour i ∈ URR − ({GTF} ∪ PRET ∪ NVEAU)_GTF Faire
06          placer(P_i) ;
07          NVEAU_GTF := NVEAU_GTF ∪ {P_i} ;
08        Finpour
09        enreg_configuration() ;
10        efface_msg_orphelin() ;
11        Pour i ∉ PRET_GTF Faire
12          ajouter_info_ra(, m_ra_recup)
13          ajouter_nra(m_ra_recup) ;
14          émettre(P_i, m_ra_recup)
15        Finpour
16        init(τ^15_GTF, nra_GTF) ;
17      Finsi
18    Finsi
19  }
```

RECEVOIR $(m\_ra\_recup_i^{GTF}) \stackrel{\text{def}}{=}$

```
01  {
02    entier nra ;
03    récupérer_nra(nra, m_ra_recup) ;
04    Si (nra ≥ nra_GTF) Alors
05      récupération() ;
06      ajouter_nra(m_ra_recup) ;
07      émettre(GTF, m_ra_recup)
08    Finsi
09  }
```

RECEVOIR $(m\_ra\_recup_{GTF}^{i}) \stackrel{\text{def}}{=}$

```
01  {
02    entier nra ;
03    récupérer_nra(nra, m_ra_recup) ;
04    Si (nra ≥ nra_GTF) Alors
05      NVEAU_GTF := NVEAU_GTF − {P_i} ;
06      PRET_GTF := PRET_GTF ∪ {P_i} ;
07      Si (PRET_GTF = (URR − {GTF})) Alors
08        ajouter_nra(m_ra_reprise)
09        diffuser(m_ra_reprise) ;
10        init(τ^16_GTF, nra_GTF) ;
11      Finsi
12    Finsi
13  }
```

fin $(\tau_{GTF}^{15}, nra) \stackrel{\text{def}}{=}$

```
01  {
02    Si (nra ≥ nra_GTF) Alors
```

```
03      Pour j ∈ ND_GTF Faire
04        ping(...) ;
05      Finpour
06    Finsi
07  }
```

RECEVOIR $(m\_ra\_reprise_i^{GTF}) \stackrel{\text{def}}{=}$

```
01  {
02    entier nra ;
03    récupérer_nra(nra, m_ra_reprise) ;
04    Si (nra ≥ nra_i) Alors
05      enlever(τ_i^17) ;
06      init(τ_i^0, nra_i) ;
07      init(τ_i^1, nra_i) ;
08      init(τ_i^2, nra_i) ;
09      init(τ_i^3, nra_i) ;
10      D_i := ∅ ;
11      ND_i := ∅ ;
12      ORPH_i := ∅ ;
13    Finsi
14  }
```

RECEVOIR $(m\_ra\_reprise_i^{GTF}) \stackrel{\text{def}}{=}$

```
01  {
02    entier nra ;
03    récupérer_nra(nra, m_ra_reprise) ;
04    Si (nra ≥ nra_i) Alors
05      R_GTF := R_GTF ∪ {P_i} ;
06      Si (R_GTF = (URR − {GTF}) Alors
07        enlever(τ_GTF^16) ;
08        init(τ_GTF^5, nra_GTF) ;
09        init(τ_GTF^6, nra_GTF) ;
10        init(τ_GTF^7, nra_GTF) ;
11        D_GTF := ∅ ;
12        ND_GTF := ∅ ;
13        ORPH_GTF := ∅ ;
14      Finsi
15    Finsi
16  }
```

fin $(\tau_{GTF}^{16}, nra) \stackrel{\text{def}}{=}$

```
01  {
02    Si (nra ≥ nra_GTF) Alors
03      Pour i ∈ (URR − ({GTF} ∪ R_GTF)) Faire
04        ajouter_nra(m_ra_reprise) ;
05        émettre(P_i, m_ra_reprise) ;
06      Finpour
```

```
07    Finsi
08  }
```

Bibliographie

[Abro89] V. Abrossimov and M. Rozier. Generic Virtual Memory Management for Operating System Kernels. In *Proc. 12th ACM Symposium on Operating Systems Principles*, Litchfield Park(USA), December 1989.

[Ahme90] 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.

[Ahuj93] 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.

[Alar91] E. Alard. Un mécanisme de migration dans un réseau homogène de stations de travail. Rapport de stage du DEA systèmes informatiques, Université Pierre et Marie Curie, PARIS VI (France), 1991.

[Alar92] E. Alard and G. Bernard. Preemptive Process Migration in Networks of UNIX Workstations. In *Proc. 7th International Symposium on Computer and Information Sciences*, Antalya (Turkey), November 1992.

[Alvi93] 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.

[Alvi94] 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.

[Alvi96] L. Alvisi and K. Marzullo. Trade-Offs in Implementing Optimal Message Logging Protocols. In *Proc. 15th ACM Symposium on Principles of Distributing Computing*, August 1996.

[Appe91] A.W. Appel and K. Li. Virtual Memory Primitives for User Programs. In *Proc. 4th International Conference on Architectural Support for Programming Languages and Operating Systems*, Santa Clara (USA), April 1991.

[Aviz85] A. Avizienis. The N-Version Approach to Fault-Tolerant Software. *IEEE Transactions on Software Engineering*, SE-11(12), December 1985.

[Bach86] M.J. Bach. *The Design of The UNIX Operating System*. Prentice-Hall Internationnal Editions, 1986.

[Baco91] D. Bacon. Transparent Recovery in Distributed Systems - *Position Paper*. *ACM Operating Systems Review*, 25(2), April 1991.

[Bald95a] 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.

[Bald95b] 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.

[Bana91a] J.P. Banatre and M. Banâtre. Fast stable storage as a basis for fault tolerant architectures. In A. Karshmer and J. Nehmer, editors, *Proc. International Workshop on Operating Systems of the 90s and Beyond*, Dagstuhl Castle (Germany), July 1991. Springer-Verlag, Lecture Notes in Computer Science, Volume 563.

[Bana91b] M. Banatre, G. Muller, B. Rochat, and P. Sanchez. Design Decisions for the FTM: A General Purpose Fault Tolerant Machine. In *Proc. 21st IEEE Symposium on Fault Tolerant Computing*, Montréal (Canada), June 1991.

[Bana93] 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.

[Bana94] M. Banatre, A. Gefflaut, and C. Morin. Tolerating Node Failures in Cache Only Memory Architectures. In *Proc. of Supercomputing '94*, Washington (USA), November 1994.

[Bari83] G. Barigazzi and L. Strigini. Application-Transparent Setting of Recovery Points. In *Proc. 13th IEEE Symposium on Fault Tolerant Computing*, Milano (Italy), June 1983.

[Barr90] P.A. Barret, A.M. Hilborne, P. Verissimo, L. Rodrigues, P.G. Bond, D.T. Seaton, and N.A. Speirs. The Delta-4 Extra Performance Architecture (XPA). In *Proc. 20th IEEE Symposium on Fault Tolerant Computing*, Newcastle-upon-Tyne (UK), June 1990.

[Bart81] J.F. Bartlett. A 'NonStop' Kernel. In *Proc. 8th ACM Symposium on Operating Systems Principles*, Pacific Grove (USA), October 1981.

[Bern88] P.A. Bernstein. Sequoia: A Fault-Tolerant Tightly Coupled Multiprocessor for Transaction Processing. *IEEE Computer*, February 1988.

[Bern89] G. Bernard, A. Duda, Y. Haddad, and G. Harrus. Primitives for Distributed Computing in a Heterogeneous Local Area Network Environment. *IEEE Transactions on Software Engineering*, SE-15(12), December 1989.

[Bhar88] 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.

[Bhid90] A. Bhide, E.N. Elnozahy, and S.P. Morgan. Implicit Replication in a Network File Server. In *IEEE Workshop on the Management of Replicated Data*, Houston (USA), November 1990.

[Birm85] K.P. Birman. Replication and Fault-Tolerance in the ISIS System. In *Proc. 10th ACM Symposium on Operating Systems Principles*, Orcas Island (USA), December 1985.

[Birm87] K.P. Birman and T.A. Joseph. Reliable Communication in the Presence of Failures. *ACM Transactions on Computer Systems*, 5(1), February 1987.

[Blac92] D.L. Black, D.B. Golub, D.P. Julin, R.F. Rashid, R.P. Draves, R.W. Dean, A. Forin, J. Barrera, H. Tokuda, G. Malan, and D. Bohman. Microkernel Operating System Architecture and Mach. In *Proc. USENIX Workshop on Micro-Kernels and Other Kernel Architectures*, Seattle (USA), April 1992.

[Borg83] 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.

[Borg89] A. Borg, W. Blau, W. Graetsch, F. Herrmann, and W. Oberle. Fault Tolerance Under UNIX. *ACM Transactions on Computer Systems*, 7(1), February 1989.

[Bric92] A. Bricker, M. Litzkow, and M. Livny. Condor Technical Summary. Technical Report TR 1069, University of Wisconsin, Madison (USA), January 1992.

[Brze95] J. Brzeziński, J.-M. Helary, and M. Raynal. Semantics of recovery lines for backward recovery in distributed systems. Publication Interne PI-899, Institut de Recherche en Informatique et Systèmes Aléatoires, Rennes (France), Février 1995.

[Burr92] M. Burrows, C. Jerian, B. Lampson, and T. Mann. On-line Data Compression in a Log-structured File System. In *Proc. 5th IEEE International Conference on Architectural Support for Programming Languages and Operating Systems*, October 1992.

[Chan84] J.-M. Chang and N.F. Maxemchuk. Reliable Broadcast Protocols. *ACM Transactions on Computer Systems*, 2(3), August 1984.

[Chan85] K.M. Chandy and L. Lamport. Distributed Snapshots: Determining Global States of Distributed Systems. *ACM Transactions on Computer Systems*, 3(1), February 1985.

[Cher92] M. Cherèque, D. Powell, P. Reynier, J.L. Richier, and J. Voiron. Active Replication in Delta-4. In *Proc. 22th IEEE Symposium on Fault Tolerant Computing*, Boston (USA), July 1992.

[Come91] D. Comer. *INTERNETORKING WITH TCP/IP Volume 1: Principles protocols and architecture.* Prentice-Hall International Editions, 1991.

[Coop85] E.C. Cooper. Replicated Distributed Programs. In *Proc. 10th ACM Symposium on Operating Systems Principles*, Orcas Island (USA), December 1985.

[Cris89] 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.

[Cris90] F. Cristian, B. Dancey, and J. Dehn. Fault-Tolerance in the Advanced Automation System. In *Proc. 20th IEEE Symposium on Fault Tolerant Computing*, Newcastle-upon-Tyne (UK), June 1990.

[Cris91a] F. Cristian. Basic concepts and issues in fault-tolerant distributed systems. In A. Karshmer and J. Nehmer, editors, *Proc. International Workshop on Operating Systems of the 90s and Beyond*, Dagstuhl Castle (Germany), July 1991. Springer-Verlag, Lecture Notes in Computer Science, Volume 563.

[Cris91b] 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.

[Cris96] F. Cristian. Synchronous and Asynchronous Group Communication. *Communications of the ACM*, 39(4), April 1996.

[CS93a] CHORUS/MiX V.4r2 C_actor Programmer's Guide. Technical Report CS/TR-92-108, Chorus Systèmes, Saint-Quentin-en-Yvelines (France), February 1993.

[CS93b] CHORUS/MiX V.4r2 C_actor User's Guide. Technical Report CS/TR-92-109, Chorus Systèmes, Saint-Quentin-en-Yvelines (France), February 1993.

[CS93c] CHORUS/MiX V.4r2 Protocols between Unix Sub System Servers. Technical Report CS/TR-91-43.1, Chorus Systèmes, Saint-Quentin-en-Yvelines (France), February 1993.

[CS95a] Chorus Kernel v3r5 : Implementation Guide. Technical Report CS/TR-94-73.2, Chorus Systèmes, Saint-Quentin-en-Yvelines (France), March 1995.

[CS95b] Chorus Kernel v3r5 : Programmer's Reference Manual. Technical Report CS/TR-92-26.6, Chorus Systèmes, Saint-Quentin-en-Yvelines (France), March 1995.

[CS95c] Chorus Kernel v3r5 : Specification and Interface. Technical Report CS/TR-91-69.4, Chorus Systèmes, Saint-Quentin-en-Yvelines (France), March 1995.

[Deco93] 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.

[Dixo87] 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.

[Doug91] F. Douglis and J. Ousterhout. Transparent Process Migration: Design Alternatives and the Sprite Implementation. *Software - Practice and Experience*, 21(8), August 1991.

[Doug93] F. Douglis. The Compression Cache: Using On-Line Compression to Extend Physical Memory. In *Proc. Winter 1993 USENIX Technical Conference*, San Diego, January 1993.

[Elno92a] E.N. Elnozahy, D.B. Johnson, and W. Zwaenepoel. The Performance of Consistent Checkpointing. In *Proc. 11th IEEE Symposium on Reliable Distributed Systems*, 1992.

[Elno92b] E.N. Elnozahy and W. Zwaenepoel. Replicated Distributed Processes in Manetho. In *Proc. 22th IEEE Symposium on Fault Tolerant Computing*, Boston (USA), July 1992.

[Elno92c] 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.

[Elno93] E.N. Elnozahy. *Manetho: Fault Tolerance in Distributed Systems Using Rollback-Recovery and Process Replication.* PhD thesis, Rice University (USA), October 1993.

[Elno94] 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.

[Elno95] E.N. Elnozahy. On the Relevance of Communication Costs of Rollback-Recovery Protocols. In *Proc. 14th ACM Symposium on Principles of Distributing Computing*, Ottawa (Canada), August 1995.

[Fidg91] C. Fidge. Logical Time in Distributed Computing Systems. *IEEE Computer*, August 91.

[From94] E. Fromentin and M. Raynal. Local states in distributed computations: a few relations and formulas. *ACM Operating Systems Review*, 28(2), April 1994.

[Geis94] A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek, and V. Sunderam. *PVM: Parallel Virtual Machine - A User's Guide and Tutorial for Networked Parallel Computing.* The MIT Press, Cambridge, Massachusetts (USA), 1994.

[Gold90] A.P. Goldberg, A. Gopal, K. Li, R. Strom, and D.F. Bacon. Transparent Recovery of Mach Applications. In *Proc. 1st USENIX Mach Symposium*, 1990.

[Gray81] J. Gray. The Transaction Concepts: Virtues and Limitations. In *Proc. 7th International Conference on Very Large Databases*, September 1981.

[Gray86] J. Gray. Why do computers stop and what can be done about it? In *Proc. 5th Symposium on Reliability in Distributed Software and Database Systems*, Los Angeles (USA), January 1986.

[Gray88] J. Gray. The cost of messages. In *Proc. 7th ACM Symposium on Principles of Distributing Computing*, 1988.

[Gray90] J. Gray. A Census of Tandem System Availability Between 1985 and 1990. *IEEE Transactions on Reliability*, 39(4), October 1990.

[Guer95] R. Guerraoui and A. Schiper. Transaction model vs Virtual Synchrony model: bridging the gap. In *Proc. International Workshop on Theory and Practice in Distributed Systems*. Springer-Verlag, Lecture Notes in Computer Science, Volume 938, 1995.

[Haer83] T. Haerder and A. Reuter. Principles of Transaction-Oriented Database Recovery. *ACM Computing Surveys*, 15(4), December 1983.

[Hask88] R. Haskin, Y. Malachi, W. Sawdon, and G. Chan. Recovery Management in QuickSilver. *ACM Transactions on Computer Systems*, 6(1), February 1988.

[Hela93] 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.

[Hisg93] A. Hisgen, A. Birrell, C. Jerian, T. Mann, and G. Swart. New-Value Logging in the Echo Replicated File System. Technical Report SRC-104, Digital Equipment Corporation Systems Research Center, Palo Alto (USA), June 1993.

[Horn91] C. Horn, B. Coghlan, N. Harris, and J. Jones. Stable memory - another look. In A. Karshmer and J. Nehmer, editors, *Proc. International Workshop on Operating Systems of the 90s and Beyond*, Dagstuhl Castle (Germany), July 1991. Springer-Verlag, Lecture Notes in Computer Science, Volume 563.

[Huan95] 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.

[Huda93] J. Hudak, B.-H. Suh, D. Siewiorek, and Z. Segall. Evaluation & Comparison of Fault-Tolerant Software Techniques. *IEEE Transactions on Reliability*, 42(2), June 1993.

[Hwan93a] K. Hwang. Parallel Computer Models. In *Advanced Computer Architecture*, chapter 1. McGraw-Hill, 1993.

[Hwan93b] K. Hwang. Parallel Models, Languages, and Compilers. In *Advanced Computer Architecture*, chapter 10. McGraw-Hill, 1993.

[Jalo86] P. Jalote. Using Broadcasting for Multiprocess Recovery. In *Proc. 6th IEEE International Conference on Distributed Computing Systems*, Cambridge (USA), May 1986.

[Jans93] 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.

[Jans94] 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.

[Jard94] C. Jard and G.-V. Jourdan. Dependency tracking and filtering in distributed computations. Publication Interne PI-581, Institut de Recherche en Informatique et Systèmes Aléatoires, Rennes (France), Août 1994.

[Jewe91] D. Jewett. Integrity S2: A Fault-Tolerant Unix Platform. In *Proc. 21st IEEE Symposium on Fault Tolerant Computing*, Montréal (Canada), June 1991.

[John87] D.B. Johnson and W. Zwaenepoel. Sender-based Message Logging. In *Proc. 17th IEEE Symposium on Fault Tolerant Computing*, June 1987.

[John89] D.B. Johnson. *Distributed System Fault Tolerance Using Message Logging and Checkpointing.* PhD thesis, Rice University, December 1989.

[John90a] 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.

[John90b] D.B. Johnson and W. Zwaenepoel. Recovery in Distributed Systems Using Optimistic Message Logging and Checkpointing. *Journal of Algorithms*, 11, September 1990.

[John91] D.B. Johnson and W. Zwaenepoel. Transparent Optimistic Rollback Recovery. *ACM Operating Systems Review*, 25(2), April 1991.

[John93] D.B. Johnson. Efficient Transparent Optimistic Rollback Recovery for Distributed Applications Programs. In *Proc. 12th IEEE Symposium on Reliable Distributed Systems*, October 1993.

[Juan91] 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.

[Kaas92] 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.

[Kaas94] M.F. Kaashoek and A.S. Tanenbaum. Efficient Reliable Group Communication for Distributed Systems. Technical Report IR-295, Vrije Universiteit, Amsterdam (Netherlands), 1994.

[Kim82] K.H. Kim. Approaches to Mechanization of the Conversation Scheme Based on Monitors. *IEEE Transactions on Software Engineering*, 8(3), May 1982.

[Kim86] K.H. Kim and A. Abouelnaga. A Scheme for Coordinated Execution of Independently Designed Recoverable Distributed Processes. In *Proc. 16th IEEE Symposium on Fault Tolerant Computing*, Vienna (Austria), July 1986.

[Kim90] K.H. Kim and A. Abouelnaga. A Highly Decentralized Implementation Model for the Programmer-Transparent Coordination (PTC) Scheme for Cooperative Recovery. In *Proc. 20th IEEE Symposium on Fault Tolerant Computing*, Newcastle-upon-Tyne (UK), 1990.

[Koo87] R. Koo and S. Toueg. Checkpointing and Rollback Recovery for Distributed Systems. *IEEE Transactions on Software Engineering*, SE-13(1), January 1987.

[Kron86] N. Kronenberg, H. Levy, and W. Strecker. VAXclusters: A Closely-Coupled Distributed System. *ACM Transactions on Computer Systems*, 2(4), 1986.

[Lai87] T.H. Lai and T.H. Yang. On Distributed Snapshots. *Information Processing Letters*, 25, May 1987.

[Lamp78] L. Lamport. Time, clocks and the ordering of events in a distributed system. *Communications of the ACM*, 21(7), July 1978.

[Lamp81] B.W. Lampson. Atomic transactions. In *Distributed Systems - Architecture and Implementation*. Springer-Verlag, Lecture Notes in Computer Science Volume 105, 1981.

[Lapr85] J.-C. Laprie. Sûreté de fonctionnement des systèmes informatiques et tolérance aux fautes: concepts de base. *Technique et Science Informatiques*, 4(5), Septembre-Octobre 1985.

[Lapr90] J.-C. Laprie, J. Arlat, C. Béounes, and K. Kanoun. Definition and Analysis of Hardware- and Software-Fault-Tolerant Architectures. *IEEE Computer*, July 1990.

[Lapr94] J.-C. Laprie. Concepts de base de la tolérance aux fautes. In Observatoire Français des Techniques Avancées, editor, *Informatique Tolérante aux Fautes*. O.F.T.A. - Masson, Paris (France), 1994.

[Lee95] I. Lee and R. Iyer. Software Dependability in the Tandem GUARDIAN System. *IEEE Transactions on Software Engineering*, 21(5), May 1995.

[Leon93] 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.

[Leon94] 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.

[Leu88] P.-J. Leu and B. Bhargava. Concurrent Robust Checkpointing and Recovery in Distributed Systems. In *Proc. 4th IEEE International Conference on Data Engineering*, 1988.

[Li87] 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.

[Li90a] C.-C.J. Li and W.K. Fuchs. CATCH - Compiler-Assisted Techniques for Checkpointing. In *Proc. 20th IEEE Symposium on Fault Tolerant Computing*, June 1990.

[Li90b] K. Li, J.F. Naughton, and J.S. Plank. Real-Time, Concurrent Checkpoint for Parallel Programs. In *Proc. 2nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 1990.

[Lin90] 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.

[Lisk83] B. Liskov and R. Scheifler. Guardians and Actions: Linguistic Support for Robust, Distributed Programs. *ACM Transactions on Programming Languages and Systems*, 5(3), July 1983.

[Lisk87] B. Liskov, D. Curtis, P. Johnson, and R. Scheifler. Implementation of Argus. In *Proc. 11th ACM Symposium on Operating Systems Principles*, Austin, Texas (USA), November 1987.

[Lisk91] B. Liskov, S. Ghemavat, R. Gruber, P. Johnson, L. Shrira, and M. Williams. Replication in the Harp File System. In *Proc. 13th ACM Symposium on Operating Systems Principles*, October 1991.

[LL94] G. Le Lann, P. Minet, and D. Powell. Tolérance aux fautes et systèmes répartis : Concepts et mécanismes. In Observatoire Français des Techniques Avancées, editor, *Informatique Tolérante aux Fautes*. O.F.T.A. - Masson, Paris (France), 1994.

[Long92] J. Long, W.K. Fuchs, and J.A. Abraham. Compiler-Assisted Static Checkpoint Insertion. In *Proc. 22th IEEE Symposium on Fault Tolerant Computing*, Boston (USA), July 1992.

[Lowr91] 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.

[Mahm88] A. Mahmood and E.J. McCluskey. Concurrent Error Detection Using Watchdog Processors - A Survey. *IEEE Transactions on Computers*, 37(2), February 1988.

[Manc89] 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.

[Mish91] S. Mishra, L.L. Peterson, and R.D. Schlichting. Consul: A Communication Substrate for Fault-Tolerant Distributed Programs. Technical Report TR 91-32, Department of Computer Science, The University of Arizona at Tucson (USA), November 1991.

[Mish92] S. Mishra and R.D. Schlichting. Abstractions for Constructing Dependable Distributed Systems. Technical Report TR 92-19, Department of Computer Science, The University of Arizona at Tucson (USA), August 1992.

[Moss82] J.E.B. Moss. Nested Transactions and Reliable Distributed Computing. In *Proc. 2nd IEEE Symposium on Reliability in Distributed Software and Database Systems*, July 1982.

[MS89] P.M. Melliar-Smith and L.E. Moser. Fault-Tolerant Distributed Systems Based on Broadcast Communication. In *Proc. 9th IEEE International Conference on Distributed Computing Systems*, Newport Beach, California (USA), 1989.

[Nels90] V.P. Nelson. Fault-Tolerant Computing: Fundamental Concepts. *IEEE Computer*, July 1990.

[Netz94] R.H.B. Netzer and B.P. Miller. Optimal Tracing and Replay for Debugging Message-Passing Parallel Programs. *The Journal of Supercomputing*, 1994.

[Netz95] R.H.B. Netzer and J. Xu. Necessary and Sufficient Conditions for Consistent Global Snapshots. *IEEE Transactions on Parallel and Distributed Systems*, 6(2), February 1995.

[O'Co93] M. O'Connor. Process migration on chorus. Master's thesis, Trinity College, Dublin (Eire), 1993.

[O'Co94] M. O'Connor, B. Tangney, V. Cahill, and N. Harris. Micro-kernel Support for Migration. Technical Report TCD-CS-94-43, Trinity College, Dublin (Ireland), 1994.

[OFTA94] Informatique Tolérante aux Fautes. Observatoire Francais des Techniques Avancées - Masson, 1994.

[Phil95] L. Philippe and G.-R. Perrin. Migration de processus dans Chorus/MiX. *Revue Électronique sur les Réseaux et l'Informatique Répartie*, (1), Avril 1995.

[Plan93] J.S. Plank. *Efficient Checkpointing on MIMD Architectures.* PhD thesis, Princeton University, June 1993.

[Plan95] J.S. Plank, M. Beck, G. Kingsley, and K. Li. Libckpt: Transparent Checkpointing under Unix. In *Proc. Winter 1995 USENIX Technical Conference*, New Orleans (USA), January 1995.

[Pope90] G.J. Popek, R.G. Guy, T.W. Page, and J.S. Heidemann. Replication in Ficus Distributed File Systems. In *IEEE Workshop on the Management of Replicated Data*, Houston (USA), November 1990.

[Powe83] 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.

[Powe88] 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.

[Powe91] D. Powell, M. Chérèque, and D. Drackley. Fault-Tolerance in DELTA-4. *ACM Operating Systems Review*, 25(2), April 1991.

[Powe92] D. Powell. Failure Mode Assumptions and Assumption Coverage. In *Proc. 22th IEEE Symposium on Fault Tolerant Computing*, Boston (USA), June 1992.

[Prad94a] D.K. Pradhan and N.H. Vaidya. Roll-Forward and Rollback Recovery: Performance-Reliability Trade-Off. In *Proc. 24th IEEE Symposium on Fault Tolerant Computing*, Austin (USA), June 1994.

[Prad94b] D.K. Pradhan and N.H. Vaidya. Roll-Forward Checkpointing Scheme: A Novel Fault-Tolerant Architecture. *IEEE Transactions on Computers*, 43(10), October 1994.

[Rama93] P. Ramanathan and K.G. Shin. Use of Common Time Base for Checkpointing and Rollback Recovery in a Distributed System. *IEEE Transactions on Software Engineering*, SE-19(6), June 1993.

[Rand75] B. Randell. System Structure for Software Fault Tolerance. *IEEE Transactions on Software Engineering*, SE-1(2), June 1975.

[Rash87] R. Rashid, A. Tevanian, M. Young, D. Golub, R. Baron, D. Black, W. Bolosky, and J. Chew. Machine-Independent Virtual Memory Management for Paged Uniprocessor and Multiprocessor Architectures. In *Proc. 2nd ACM Conference on Architectural Support for Programming Languages and Operating Systems*, October 1987.

[Rayn94] M. Raynal. Ré-exécution et analyse de calculs répartis. Publication Interne PI-814, Institut de Recherche en Informatique et Systèmes Aléatoires, Rennes (France), Avril 1994.

[Rayn95] M. Raynal and M. Singhal. Logical Time: A Way to Capture Causality in Distributed Systems. Publication Interne PI-900, Institut de Recherche en Informatique et Systèmes Aléatoires, Rennes (France), Janvier 1995.

[Riff89] J.-M. Rifflet. *La programmation sous UNIX.* Mc Graw-Hill, 1989.

[Riff90] J.-M. Rifflet. *La communication sous UNIX.* Mc Graw-Hill, 1990.

[Rose91] M. Rosenblum and J.K. Ousterhout. The Design and Implementation of a Log-Structured File System. In *Proc. 13th ACM Symposium on Operating Systems Principles*, Pacific Grove (USA), October 1991.

[Rozi88] M. Rozier, V. Abrossimov, F. Armand, I. Boule, M. Gien, M. Guillemont, F. Herrmann, C. Kaiser, S. Langlois, P. Léonard, and W. Neuhauser. Chorus Distributed Operating Systems. *Computing Systems Journal, The USENIX Association*, 1(4), December 1988.

[Rozi92] M. Rozier, V. Abrossimov, F. Armand, I. Boule, M. Gien, M. Guillemont, F. Herrmann, C. Kaiser, S. Langlois, P. Léonard, and W. Neuhauser. Overview of the Chorus Distributed Operating System. In *Proc. USENIX Workshop on Micro-Kernels and Other Kernel Architectures*, Seattle (USA), April 1992.

[Russ80] D.L. Russell. State Restoration in Systems of Communicating Processes. *IEEE Transactions on Software Engineering*, SE-6(2), March 1980.

[Russ93] M. Russinovich, Z. Segall, and D. Siewiorek. Application Transparent Fault Management in Tolerant Mach. In *Proc. 23th IEEE Symposium on Fault Tolerant Computing*, Toulouse (France), June 1993.

[Sand85] R. Sandberg, D. Goldberg, S. Kleiman, D. Walsh, and B. Lyon. Design and Implementation of the Sun Network File System. In *Proc. Summer 1985 USENIX Technical Conference*, Portland (USA), June 1985.

[Saty90a] M. Satyanarayanan. Scalable, Secure, and Highly Available Distributed File Access. *IEEE Computer*, May 1990.

[Saty90b] M. Satyanarayanan, J. Kistler, P. Kumar, M.E. Okasaki, E.H. Siegel, and D.C. Steere. Coda: A Highly Available File System for a Distributed Workstation Environment. *IEEE Transactions on Computers*, 39(4), April 1990.

[Saty94] 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.

[Schl83] 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.

[Schl91] R.D. Schlichting. Operating Systems and Fault-Tolerance. In A. Karshmer and J. Nehmer, editors, *Proc. International Workshop on Operating Systems of the 90s and Beyond*, Dagstuhl Castle (Germany), July 1991. Springer-Verlag, Lecture Notes in Computer Science, Volume 563.

[Schn90] F.B. Schneider. Implementing Fault-Tolerant Services Using the State Machine Approach: A Tutorial. *ACM Computing Surveys*, 22(4), December 1990.

[Schw92] T. Schwarz and F. Mattern. Detecting Causal Relationships in Distributed Computations: In Search of the Holy Grail. Technical Report SFB 124-15/92, Department of Computer Science, University of Kaiserslautern (Germany), December 1992. revised version of report 215/91.

[Sens94] P. Sens. *Conception et mise en œuvre d'une plate-forme logicielle de tolérance aux fautes pour le support d'applications réparties.* PhD thesis, Université Paris VI (France), Décembre 1994.

[Sens95] P. Sens. The Performance of Independent Checkpointing in Distributed Systems. In *3rd ACM Hawaii International Conference on System Sciences*, Hawaii (USA), 1995.

[Shri85] 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.

[Shri87] S.K. Shrivastava, L.V. Mancini, and B. Randell. On the Duality of Fault Tolerant System Structures. In J. Nehmer, editor, *Experiences with Distributed Systems.* Springer-Verlag, 1987.

[Shri90] S.K. Shrivastava and S.M. Wheater. Implementing Fault-Tolerant Distributed Applications Using Objects and Multi-Coloured Actions. In *Proc. 10th IEEE International Conference on Distributed Computing Systems*, Paris (France), May 1990.

[Shri91] S.K. Shrivastava, G.N. Dixon, and G.D. Parrington. An Overview of the Arjuna Distributed Programming System. *IEEE Software*, 1(8), January 1991.

[Shri92] S.K. Shrivastava, P.D. Ezhilchelvan, N.A. Speirs, S. Tao, and A. Tully. Principal Features of the VOLTAN Family of Reliable Node Architectures for Distributed Systems. *IEEE Transactions on Computers*, 41(5), May 1992.

[Siew90] D.P. Siewiorek. Fault Tolerance in Commercial Computers. *IEEE Computer*, July 1990.

[Siew91] D.P. Siewiorek. Architecture of Fault-Tolerant Computers: An Historical Perspective. *Proc. of the IEEE*, 79(12), December 1991.

[Silv92] L.M. Silva and J.G. Silva. Global Checkpointing for Distributed Programs. In *Proc. 11th IEEE Symposium on Reliable Distributed Systems*, 1992.

[Sist89] A.P. Sistla and J.L. Welch. Efficient distributed recovery using message logging. In *Proc. 8th ACM Symposium on Principles of Distributed Computing*, 1989.

[Spei89] N.A. Speirs and P.A. Barret. Using Passive Replicates in Delta-4 tp provide Dependable Distributed Computing. In *Proc. 19th IEEE Symposium on Fault Tolerant Computing*, Chicago (USA), 1989.

[Spez86] M. Spezialetti and P. Kearns. Efficient Distributed Snapshots. In *Proc. 6th IEEE International Conference on Distributed Computing Systems*, 1986.

[Stri91] L. Strigini and Di Giandomenico. Flexible Schemes for Application-Level Fault Tolerance. In *Proc. 10th IEEE Symposium on Reliable Distributed Systems*, 1991.

[Stro85] R.E. Strom and S.A. Yemini. Optimistic Recovery in Distributed Systems. *ACM Transactions on Computer Systems*, 3(3), August 1985.

[Stro88] 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.

[Surf76] Mémoire de définition du projet-pilote sûreté de fonctionnement des systèmes informatiques. Document conjoint UPS-LSI, Onera-Cert, CNRS-Laas, Toulouse (France), 1976.

[Tayl90] D.J. Taylor, D.E. Morgan, and J.P. Black. Redundancy in data structure: improving software fault tolerance. *IEEE Transactions on Software Engineering*, SE-16(11), November 1990.

[Thei85] M.M. Theimer, K.A. Lantz, and D.R. Cheriton. Preemptable Remote Execution Facilities for the V-System. In *Proc. 10th ACM Symposium on Operating Systems Principles*, Orcas Island, Washington (USA), December 1985.

[Thie91] G. Thiel. LOCUS operating System, a transparent system. *Computer Communications*, 14(6), July/August 1991.

[Tong89] 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.

[Tyrr86] A.M. Tyrrel and D.J. Holding. Design of Reliable Software in Distributed Systems Using the Conversation Scheme. *IEEE Transactions on Software Engineering*, SE-12(9), September 1986.

[Vaid93a] N. Vaidya. *Low-Cost Schemes for Fault Tolerance.* PhD thesis, University of Massachusetts, Amherst (USA), February 1993.

[Vaid93b] N. Vaidya. Distributed recovery units: An approach for hybrid and adaptative distributed recovery. Technical Report 93-052, Texas A&M University, Texas (USA), November 1993.

[Vaid94] N. Vaidya. Consistent logical checkpointing. Technical Report 94-051, Texas A&M University, Texas (USA), July 1994.

[Venk87] 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.

[Venk89] S. Venkatesan. Message-Optimal Incremental Snapshots. In *Proc. 9th IEEE International Conference on Distributed Computing Systems*, 1989.

[Walk83] B. Walker, G. Popek, R. English, C. kline, and G. Thiel. The LOCUS Distributed Operating System. In *Proc. 9th ACM Symposium on Operating Systems Principles*, Bretton Woods (USA), October 1983.

[Wang92a] 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.

[Wang92b] Y.-M. Wang and W.K. Fuchs. Scheduling Message Processing for Reducing Rollback Propagation. In *Proc. 22th IEEE Symposium on Fault Tolerant Computing*, Boston (USA), June 1992.

[Wang95] 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.

[Webb91] S. Webber and J. Beirne. The Stratus Architecture. In *Proc. 21st IEEE Symposium on Fault Tolerant Computing*, Montréal (Canada), June 1991.

[Wens78] J.F. Wensley, L. Lamport, J. Goldberg, M.W. Green, K.N. Levitt, P.M. Melliar-Smith, R.E. Shostak, and C.B. Weinstock. SIFT: Design and analysis of a fault-tolerant computer for aircraft control. *Proceedings of the IEEE*, 66(10), October 1978.

[Wu90a] K.-L. Wu and W.K. Fuchs. Recoverable Distributed Shared Virtual Memory. *IEEE Transactions on Computers*, 39(4), April 1990.

[Wu90b] 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.

[Xu93] 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.