MASTER EN SCIENCES INFORMATIQUES

Tolérance aux pannes dans les systèmes informatiques de gestion

Chanteux, Christian; Thiry, Jacques

Award date:
1987

Awarding institution:
Universite de Namur

FACULTÉS
UNIVERSITAIRES
N.D. DE LA PAIX
NAMUR

INSTITUT D'INFORMATIQUE

Année Académique 1986 - 1987

Tolérance aux Pannes dans les Systèmes Informatiques de Gestion

Christian Chanteux
Jacques Thiry

Promoteur : J. Ramaekers

Mémoire présenté en vue de l'obtention du grade de Licencié et Maître en Informatique.

Adresse : rue Grandgagnage, 21, B-5000 NAMUR (Belgium) - ☎ 081 - 22 90 65 - Télex 59.222 FAC NAM-B

Nous remercions cordialement notre promoteur Jean Ramaekers ainsi que son assitante Cécile Mahiat pour les conseils qu'ils nous ont prodigués tout au long de notre travail.

Que soit également remercié Monsieur De Rivet (Bull Transac) pour la documentation fournie lors d'un stage à Paris.

Nous tenons aussi à remercier Monsieur Deneyer (Tandem) pour la documentation fournie ainsi que pour les explications qu'il a bien voulu nous donner.

Nous adressons également nos remerciements à Monsieur Adams (Olivetti) pour la documentation fournie sur le système Stratus.

Enfin, nous tenons à remercier nos parents respectifs pour la patience dont ils ont fait preuve durant nos études et leurs continuels encouragements.

Namur, le 31 août 1987

Christian Chanteux
Jacques Thiry

Ce mémoire est un travail de synthèse des différents principes existants pour la construction de systèmes informatiques orientés Base de Données à haute fiabilité.

Les systèmes de Tandem et de Stratus sont présentés.

Les sujets abordés comprennent la Continuité de Service et l'Intégrité des données.

This paper provides a study of principles involved in achieving high reliability for data bases oriented systems.

It includes a description of Tandem and Stratus systems.

Topics covered include Continuous Processing and Data Integrity.

Table des Matières

Introduction

A la recherche d'ordinateurs fiables, l'homme, ne pouvant maîtriser la complexité ni l'environnement des systèmes qu'il construit, introduit dans l'architecture de ces derniers des moyens pour "cacher" les pannes éventuelles. Les systèmes qui en découlent sont dits "tolérants aux pannes".

Il y a quelques années encore, l'utilisation et le développement des systèmes informatiques fiables, ou "tolérants aux pannes", étaient limités aux applications industrielles, aérospatiales et militaires; c'est-à-dire là où une panne d'ordinateur avait un impact économique significatif ou mettait en jeu des vies humaines. Actuellement, les besoins dans ces systèmes fiables s'étendent pour maintes raisons, dont les principales sont:

1. L' informatisation croissante des processus industriels impose à l' ordinateur de quitter sa chambre stérile pour un environnement hostile, avec des variations fréquentes de température et d' humidité, des interférences électromagnétiques, et des fluctuations de courant.

2. Le coût du matériel diminue alors que le coût de la main d'oeuvre augmente. Il devient donc économiquement préférable de miser davantage sur les coûts d' acquisition afin de minimiser le nombre d' appel service.

3. La dépendance vis-à-vis de l' ordinateur s'accroît. Dans les organisations, l' ordinateur tient un rôle de plus en plus primordial, à un point tel qu'on ne peut plus s'en passer dans certains cas.

A l'heure actuelle, l'utilisation des systèmes fiables s'est étendue jusqu'aux applications bancaires, où une panne risque de provoquer une perte d'information qui peut coûter cher [RUSH85]. Ce type d'application, dit de gestion, est ainsi supportée par des systèmes fournissant des garanties sur l'intégrité de la base de données et sur la continuité du service fourni.

Le terme intégrité de données recouvre la capacité qu'a un système de prévoir les erreurs dans la Base de Données, de les détecter aussitôt que possible, de les corriger ou de limiter leurs effets. Une erreur peut être une donnée erronée ou tout simplement un manque de donnée. Aucune erreur dans la Base de Données ne doit jamais être laissée non détectée. Aucune transaction entrée ne peut jamais être perdue.

La continuité de service est une propriété permettant aux systèmes qui en disposent de continuer à travailler malgré l'apparition d'une panne.

Notre travail décrit les concepts et les techniques permettant l'élaboration de ces systèmes tolérants aux pannes. La tolérance aux pannes est donc abordée dans les deux aspects

- de continuité de service
- d'intégrité de données

Le mémoire se compose de deux parties principales.

La première, composée des quatre premiers chapitres, propose une synthèse théorique des différents concepts et des techniques associées pour assurer cette tolérance aux pannes dans un système d'information de gestion, c'est-à-dire principalement orienté base de données.

La seconde, composée des chapitres cinq à sept, concerne l'étude de systèmes particuliers (Tandem et Stratus) et de leurs possibilités respectives.

Dans le chapitre 1, intitulé "Définitions et Concepts", nous donnons les définitions de base nécessaires à la bonne compréhension de l'ensemble du mémoire. Un certain nombre de concepts primordiaux sont déjà abordés succinctement.

Dans le chapitre 2, intitulé "Techniques de support à la Continuité de Service", nous décrivons des techniques employées pour assurer une continuité de service.

Le chapitre 3 s'intitule "Modèle d'Implémentation". Il décrit une méthodologie de construction de logiciels tolérants aux pannes basée sur la structuration hiérarchique et la découpe modulaire.

Enfin, dans le chapitre 4 "Techniques de Reprise BD", nous présentons l'ensemble des techniques communément admises et décrites dans la littérature qui permettent de sauvegarder l'intégrité des données en cas de panne du système.

Dans la seconde partie consacrée à l'étude de systèmes tolérants aux pannes particuliers, nous présentons dans le chapitre 5 le système "Tandem" et dans le chapitre 6, le système "Stratus".

Chacun des deux systèmes est décrit en fonction des termes définis dans la partie théorique. Nous avons donc conservé une cohérence dans la dénomination de certaines techniques, ce qui permet, dans le chapitre 7, de faire une évaluation des possibilités de chacun des systèmes en fonction de critères bien définis.

Dans l'ensemble du mémoire, nous renvoyons à de nombreux articles dont nous donnons les références et des résumés succincts.

PARTIE 1

Théorie

des

Systèmes Informatiques

de Gestion

Tolérants aux Pannes

Chapitre 1

Concepts de base

pour la

Tolérance aux pannes

1 Concepts de base pour la tolérance aux pannes

Dans l'introduction, nous avons situé le problème de la tolérance aux pannes, tant dans ses aspects de "continuité de service" que "d'intégrité de données". Avant de présenter différentes techniques permettant de résoudre ces problèmes, il convient de présenter une série de concepts de base qui serviront de point d'appui, de référence aux différentes solutions proposées.

Après avoir donné les notions fondamentales pour la tolérance aux pannes (1.1) et après avoir discuté de la disponibilité des systèmes (1.2), nous introduisons dans un premier temps les notions de reprise et de point de cohérence (1.3). Ensuite, nous décrivons les concepts d'atomicité et de transaction (1.4). Nous analysons ensuite leur influence sur la reprise (1.5). Enfin, nous détaillons les problèmes de la concurrence et proposons certaines solutions communément admises (1.6). En effet, par la suite, nous prendrons toujours comme hypothèse que les problèmes de concurrence sont résolus.

1.1 Notions fondamentales de tolérance aux pannes

1.1.1 Introduction

Dans la suite de ce travail, nous utilisons un certain nombre de concepts qu'il est nécessaire de comprendre de manière non-équivoque. Ces concepts font l'objet du point suivant et sont repris sous le nom: "définitions". Nous identifions ensuite, dans le paragraphe intitulé 'à quelles fautes résister ?', les classes de fautes auxquelles les systèmes sont confrontés. Et, pour terminer, nous situons la tolérance aux pannes par rapport à ces classes de fautes dans 'quid de la tolérance ?'.

1.1.2 Définitions

Les définitions qui suivent sont celles que B. Randell et A. Lee en [RAND78] attribuent à ces concepts.

"Un système est constitué d'un ensemble de composants interagissant sous le contrôle d'une architecture, ces composants pouvant eux-mêmes être vus comme des systèmes". Cette définition s'adresse aussi bien aux systèmes matériels qu'aux systèmes logiciels. C'est ainsi qu'une plaquette de circuits imprimés est un système matériel constitué de composants électroniques soudés et possède une architecture implémentée par des fils interconnectant les composants. De même, un système informatique simplifié a comme composants un processeur central, une mémoire centrale, une mémoire auxiliaire et des périphériques, et met en oeuvre son architecture par des bus de données les interconnectant. En ce qui concerne les systèmes logiciels, systèmes et composants sont synonymes de programmes et sous-programmes. Cette définition de système logiciel s'applique aussi bien aux Systèmes d'Exploitation qu'à des programmes d'application.

L' environnement d'un composant d'un système est l'ensemble des composants avec lesquels il communique directement.

L' interface est le lieu d'interaction entre deux systèmes (matériels ou logiciels). Le lieu d'interaction particulier entre le matériel et le logiciel est appelé l'interface d'interprétation. C'est une interface maintenue par le matériel sur laquelle le logiciel (sous forme de langage machine) est exécuté, interprété. La partie du matériel qui interprète le logiciel est appelée l'interpréteur.

La fiabilité d'un système est la mesure dans laquelle le comportement du système est conforme à sa spécification. Quand le système dévie de celle-ci, il s'en suit une défaillance (ou panne).

L'état externe d'un système peut-être assimilé à "l'output" de celui-ci. L'état interne, quant à lui, est défini comme l'ensemble ordonné des états externes de ses composants.

Un état erroné est un état interne pouvant amener le système à une défaillance qui ne pourrait être imputée à une faute ultérieure.

L' erreur désigne la partie de l'état erroné qui est incorrecte.

Comprenons bien que l'état d'un système est décrit par des bits, de l'information. C'est à ce titre que nous pouvons le tester.

La faute est la cause mécanique (matérielle) ou algorithmique (logicielle) d'une erreur.

En résumé, une erreur est donc la manifestation d'une faute matérielle ou logicielle dans une structure de données et une défaillance est la manifestation d'une erreur sur le service rendu par le système (matériel ou logiciel).

Une faute potentielle est une construction mécanique ou algorithmique du système qui, dans certaines circonstances conformes aux spécifications du système, pourrait amener le système dans un état erroné.

Les fautes peuvent être classifiées selon leur durée, leur étendue et leur valeur.

La durée d'une faute définit si celle-ci est transitoire, intermittente ou permanente.

Une faute transitoire est présente dans le système pour une période limitée dans le temps après laquelle elle disparaît spontanément. Elle provient d'une combinaison de conditions exceptionnelles.

Une faute intermittente est une faute transitoire qui réapparaît par à-coups, souvent en fonction de la charge du système. Toute faute présente dans le système pour une période dépassant un seuil fixé est dite permanente.

L'étendue d'une faute exprime la mesure dans laquelle la faute est localisée ou distribuée.

La valeur d'une faute indique si celle-ci crée des valeurs erronées fixes ou variables.

Etant donné que ces trois concepts: de faute, d'erreur et de défaillance, représentent des événements se succédant avec une relation de cause à effet, nous pouvons les situer les uns par rapport aux autres sur la droite du temps. Nous voyons sur la figure 1.1 T_f, T_e et T_d représentant respectivement les temps d'apparition d'une faute, de l'erreur et de la défaillance qui s'ensuivent.

Figure 1.1: Enchaînement 'faute -> erreur -> défaillance'

Nous parlons de faute simple lorsqu'il est question de l'occurrence d'une seule faute, et de faute double lorsqu'il est question de l'occurrence de deux fautes successives, dont la deuxième apparaît avant que la première ne soit

complètement traitée.

1.1.3 A quelles fautes résister ?

Dans ce paragraphe nous identifions quatre grandes classes de fautes à partir desquelles nous fixons les limites de notre étude.

Commençons par analyser les quatre grandes raisons pour lesquelles un système peut être mis dans un état erroné.

- La première raison met en cause la conception (le "design") d'un composant. En effet, que celui-ci soit matériel ou logiciel, son architecture doit être conforme à sa spécification, et c'est loin d'être toujours le cas. C'est pourquoi des techniques de construction et de vérification de système, appelées encore techniques de prévention ont été mises au point pour éliminer les fautes de conception [DENN78]. Malheureusement cet objectif n'est pas atteint: après la phase de développement d'un système, il subsiste des fautes de conception, et cela, principalement dans les composants logiciels. Ces fautes, que nous appelons "fautes résiduelles de conception" , constituent la première cause de panne à combattre.

- La seconde raison concerne exclusivement les composants physiques: il s'agit du "vieillissement" du matériel. En effet, les composants physiques s'usent avec le temps. Nous pouvons même mesurer leur temps de vie moyen ("Mean Time Between Failure" (MTBF)), ce qui nous permet d'évaluer la probabilité qu'un système a de s'écrouler à cause de ce type de faute.

- La troisième raison a pour origine l'environnement du système. Cela concerne les coupures de courant, les problèmes de conditionnement d'air, les explosions et inondations, les champs magnétiques, les particules alfa ...enfin toute une série d'événements se produisant en dehors du "monde informatique", mais qui agissent plus ou moins fort sur le fonctionnement du système.

- La quatrième et dernière cause provient de l'opérateur qui s'occupe de la gestion du système. Celui-ci peut, par distraction ou méconnaissance du système le mettre dans un état erroné. C'est un point plus important qu'il n'y paraît, d'autant plus que les systèmes se compliquent

sans cesse et que leur gestion en devient d'autant plus délicate.

Avant de continuer la discussion, il est bon de détailler la première cause de panne: les fautes de conception.

Les techniques de prévention étant bien au point pour la conception des systèmes matériels, ceux-ci sont généralement libres de fautes d'architecture. Malheureusement, ce n'est pas le cas pour les logiciels. Leur complexité est telle que les techniques de prévention s'y appliquent moins bien. En effet, si celles-ci sont efficaces contre les fautes de logiciels permanentes, induisant des fautes qui peuvent être reproduites systématiquement (tel que l'oubli d'une ligne de code dans un programme), elles le sont nettement moins contre les fautes de logiciel transitoires (intermittentes) apparaissant suite à une combinaison de conditions difficiles à reproduire.

Ainsi que le montre une enquête réalisée par TANDEM, ces fautes de logiciel sont fréquentes.

Cette enquête, menée sur 200 sites TANDEM pendant une période de 7 mois, a dénombré près de 166 pannes dont les causes ont été attribuées principalement, comme le montre le tableau de la figure 1.2, à des erreurs d'opérateur et de conception de logiciel.

Opérateur	42 %	Maintenance	25 %
		Opérations	9 %
		Configuration	8 %
Logiciel	25 %	OS	21 %
		Applications	4 %
Matériel	18 %	Processeur	1 %
		Disques	7 %
		Bandes Magnétiques	2 %
		Contrôle de Comm.	6 %
		Source d' Energie	2 %
Environnement	14 %	Source d' Energie	9 %
		Communication	3 %
Inconnue	1 %		

Figure 1.2: Détails de l'enquête menée par TANDEM

Notons:

- que les causes attribuées au matériel concernent aussi bien le vieillissement que l'architecture des composants.

- que les opérations d'administration du système sont liées à des problèmes "d'opérabilité" donc supportées par le logiciel.

En résumé, le tableau de la figure 1.3 reprend les différentes classes de fautes identifiées ci-dessus en les regroupant selon le type de composants (matériel/logiciel) qu'elles concernent. Par abus de language, nous parlons de "fautes du matériel/logiciel" au lieu de fautes de composants mat/log.

PS: les numéros dans l'extrémité droite du tableau renvoient à un exemple de chaque cas particulier.

fautes du matériel	conception		permanente intermittente	(1) (2)
	vieillesse		permanente	(3)
	environnement		permanente transitoire	(4) (5)
fautes du logiciel	conception	logiciel d' exploitation	permanente intermittente	(6) (7)
		logiciel d' application	permanente intermittente	(8) (9)
	opérateur		permanente transitoire	(10) (11)

Figure 1.3: Tableau récapitulatif des types de fautes

Exemples:

(1) court-circuit dans un circuit intégré sortant de la chaîne de production

(2) contrôleur d'Entrées/Sorties saturé par une forte charge. Il ne répond plus dans la fraction de seconde conforme à sa spécification

(3) erreur double en mémoire, défaillance d'un microprocesseur

(4) coupure d'électricité, altération du matériel par un agent externe (feu, eau ...)

(5) perturbation électro-magnétique, rayonnement de particules alfa ...

(6) et (8) faute de programmation aisément détectable par des tests

(7) et (9) faute de programmation apparaissant dans des conditions exceptionnelles
exemple: problème de pile, de sémaphore ...

(10) et (11) fautes dans le dialogue avec l'opérateur, directive erronée dans le manuel de maintenance ou d'opération.

La classification précédente, basée sur les causes des pannes, nous est très utile car elle met en évidence des classes de problèmes, donc des classes de techniques de support à la tolérance aux pannes.

Il y a peu de temps encore, toute l'attention des concepteurs de systèmes fiables s'orientait vers les fautes du matériel. Maintenant, avec l'accroissement du temps de vie des composants matériels et le déplacement de la complexité du matériel vers le logiciel, l'attention s'oriente plutôt vers les fautes de logiciel. En pratique, il est certain que les classes de fautes qui peuvent être traitées par un système déterminé dépendent de la nature des applications auxquelles celui-ci est dédié.

Pour notre part, l'objectif que nous nous fixons est de rechercher des moyens pour résister à toutes ces fautes à l'exception des fautes de l'opérateur dont l'étude sort des limites de ce travail.

1.1.4 Quid de la tolérance ?

Comme nous l'avons déjà dit, la majorité des fautes de conception, que ce soit du matériel ou du logiciel, peut être éliminée grâce aux méthodes de prévention. Celles-ci

concernent l'élimination systématique des fautes lors de l'étape de conception. Les méthodes de prévention concernent aussi les techniques de protection des composants matériels contre les perturbations externes. Cette approche s'appelle méthode de prévention des fautes puisqu'elle "prévient" l'occurrence de fautes dans le système opérationnel. Malheureusement, une élimination totale des fautes n'est pas possible pour des raisons de coût, de vieillissement des composants matériels et parce que le système n'est pas à l'abri d'incidents inopinés. Ces fautes, qui ne peuvent être prévenues, devront être combattues lors de leur apparition dans le système en masquant leurs conséquences aux yeux des utilisateurs. C'est cette méthode, appellée méthode de tolérance aux fautes que nous détaillons dans ce travail.

PS: jusqu'à présent nous avons parlé de tolérance "aux pannes" (et non "aux fautes") parce que c'est un concept facile à comprendre intuitivement. En réalité, il est préférable de parler de tolérance aux fautes puisque, ce que l'on désire, c'est justement éviter qu'une "faute" ne devienne une "panne" visible pour l'utilisateur (rappelons qu'une faute en elle-même n'est pas visible).

Les fautes tolérées peuvent encore être réparties en deux catégories à partir de leur "degré d'anticipation". En effet, puisqu'on connaît le temps de vie moyen des composants matériels, on peut connaître la probabilité d'apparition d'erreur de vieillesse. Ces fautes étant bien connues, on peut facilement identifier leurs conséquences et incorporer des mesures spécifiques afin de pouvoir les cacher lorsqu'elles apparaissent. Elles peuvent donc être anticipées.
A côté de celles-ci, il y a d'autres fautes dont on ne peut caractériser l'occurrence et les effets précis. Il s'agit principalement de fautes résiduelles de logiciel. Le traitement de telles fautes exige des méthodes de récupération systématique.

La figure 1.4 qui suit intègre les différentes distinctions faites aussi bien sur les fautes que sur les manières de les combattre. Ainsi, les fautes permanentes de conception sont éliminées au moyen de techniques de prévention avant que le système soit opérationnel. De même, les techniques de tolérance aux fautes, permettent de résister aux fautes provenant d'une des trois premières causes de défaillance évoquées en 1.1.3

c'est-à-dire :

- la conception du système
- le vieillissement des composants matériels
- l'environnement du système.

Parmi ces causes, nous distinguons celles qui induisent des fautes anticipées ,telles que l'environnement et le vieillissement du matériel, de celles qui provoquent des

fautes non-anticipées telles que les fautes résiduelles.

Nous remarquons :

- que les fautes transitoires de conception (fautes résiduelles) sont des fautes non-anticipées pour les techniques de tolérance
- que les fautes d'environnement et de vieillissement des composants constituent des fautes anticipées pour les techniques de tolérance
- que les fautes d'opérations ne font pas l'objet de techniques de tolérance (et ne sont donc pas "tolérées").

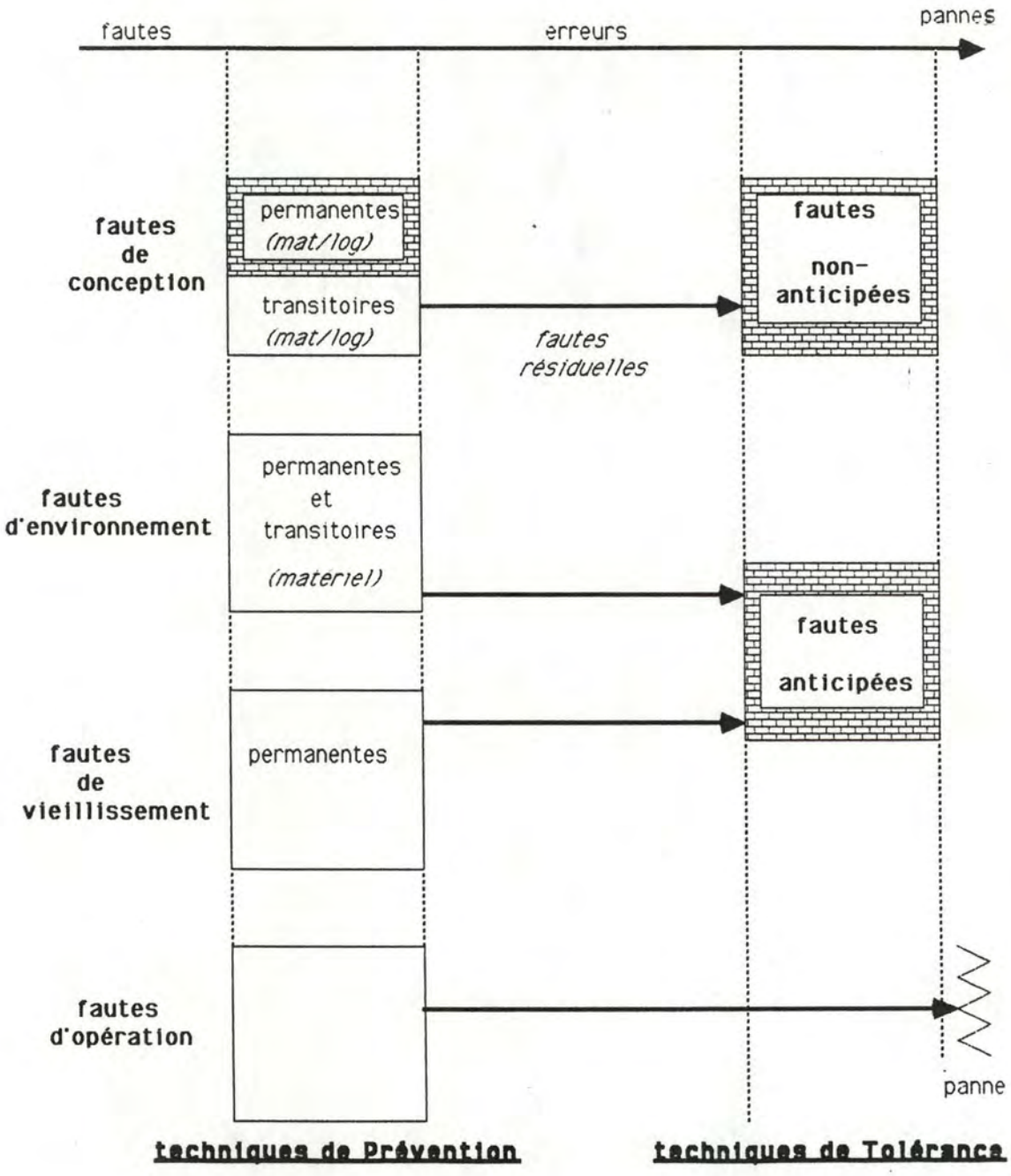

Figure 1.4: Types de fautes et méthodes de combat

Au travers des mesures de tolérance prises envers les fautes d'environnement, nous remarquons un effort pour diminuer la dépendance du système vis-à-vis de l'extérieur. Il s'agit, par exemple, d'incorporer dans le système des batteries de secours pour suppléer le secteur en cas de coupure de courant.

Les fautes d'opérations ne peuvent être "tolérées", mais cela ne signifie pas qu'elles ne peuvent être combattues. En effet, la tendance actuelle est de rechercher à réduire au maximum les interventions humaines et, quand elles sont indispensables, de les guider grâce à un Système Expert. Ce sujet sort du problème de la tolérance aux fautes et ne sera donc plus abordé dans la suite du travail.

1.2 Disponibilité

La disponibilité d'un système [GIBB76] est souvent exprimée sous forme d'un pourcentage, ou ratio, qui détermine la proportion du temps de service effectif sur le temps total de service d'un système. La littérature, pour ce fait, emploie les notions de "Mean Time Before Failure" (MTBF), qui est le temps moyen qui s'écoule avant que le système ne s'effondre (MTBF signifie parfois "Mean Time Between Failure"); et de "Mean Time To Repair" (MTTR), qui est le temps moyen nécessaire à la remise en route du système.

On obtient ainsi la mesure mathématique de la disponibilité:

$$\frac{MTBF}{MTBF+MTTR}$$

Mais cette mesure de la disponibilité doit être prise avec prudence. En effet, elle peut représenter de fréquentes interruptions de courtes durées, ou des défaillances moins fréquentes mais d'un temps de remise en route plus long. Ainsi, un arrêt de trois minutes en moyenne toutes les heures donne une disponibilité de 95%. Mais la même disponibilité est donnée pour un arrêt total d' une journée par mois (en comptant 20 jours de travail).

1.3 Reprise et nécessité d'un point de cohérence

Comme le dit si bien la loi de Murphy, "Tout appareil qui peut tomber en panne, tombe en panne", nous ne pouvons assurer à cent pour cent qu'une défaillance d'un système informatique ne se produise. Nous devons donc mettre en oeuvre des techniques de protection contre les incidents.

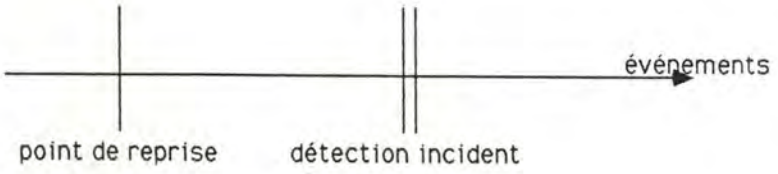

Figure 1.5: Notion de point de reprise

Nous pressentons tous le besoin de revenir "un peu en arrière" lorsqu'une défaillance ou un incident est détecté. Ce retour en arrière doit permettre de remettre le système dans un état "cohérent" , c'est-à-dire dans un état connu et bien déterminé à un instant donné (figure 1.5). Ainsi, après correction de la (ou les) cause(s) de défaillance, nous effectuons une "reprise" , un recommencement des opérations (les mêmes ou des opérations similaires) qui avaient été exécutées juste avant la défaillance. Cette reprise s'effectue à partir d'un état cohérent qui avait été sauvegardé, mémorisé. Ce point de cohérence dans le temps s'appelle point de reprise ("checkpoint").

Les procédures de reprise sont élaborées lors de la conception du système et exigent des informations sur l'état courant des programmes. Ces informations sont acquises lors d'interruptions volontairement créées selon une certaine périodicité. Ces interruptions correspondent aux points de cohérence, et donc aux points de reprise décrits plus haut. Au cours de celles-ci, des informations sont sauvegardées dans un fichier spécial appelé "journal"; elles seront exploitées par le processus de reprise. La disponibilité du système est d'autant meilleure que la durée moyenne entre deux défaillances est élevée et que les durées d'immobilisation qu'elles entraînent sont réduites.

1.4 Atomicité et transaction

La notion de cohérence d'un système d'information est très liée à celle d'intégrité. Nous connaissons les concepts de Contraintes d'Intégrité développés lors de l'analyse conceptuelle. Pour une Base de Données, par exemple, nous disons qu'elle reste intègre si les contraintes d'intégrité sont respectées [BAYE80]. Nous disons également qu'elle se trouve dans un état d'intégrité totale si elle représente le dernier état cohérent ayant existé du réel perçu [HAIN87]. Ces concepts sont applicables à l'ensemble du système d'information.

Pour réaliser cette cohérence, nous exigeons certaines propriétés des opérations que nous pouvons exécuter. Ainsi, dans un programme, nous utilisons des primitives (d'accès à la BD, par exemple).

Ces primitives doivent être atomiques. Une primitive atomique , alors qu'elle peut parfois nécessiter plusieurs instructions ou opérations élémentaires, forme un tout indivisible, ce qui signifie qu'une primitive atomique est exécutée entièrement ou pas du tout [JALO86], [RAND78], [SINH85b], [TAYL86].

De plus, à l'intérieur de ce même programme, certaines séquences de primitives sont considérées comme formant un tout. Ces portions de programme s'appellent des transactions [ESWA76]. Une transaction se délimite par des primitives spéciales, à savoir "début-transaction" pour signaler le commencement d'une transaction, "fin-transaction" pour signaler la fin de la séquence considérée comme une transaction. Le "fin-transaction" correspond à un point de non retour. Avant ce point, une autre primitive, "avorte-transaction", permet d'avorter la transaction en cours, à savoir revenir à l'état du système avant le "début-transaction". Une fois le "fin-transaction" exécuté, la transaction, considérée comme définitive ("commit"), doit être absolument sauvegardée dans le système, ce qui signifie qu'elle doit avoir été exécutée entièrement. Certains auteurs [DAVI80] parlent de "Transaction-Atomique".

Pour assurer cette indivisibilité, une transaction doit respecter quatre propriétés, communément appelées l' "ACID Test" [HAER83].

Atomicité : garantit l'exécution entière ou nulle de la transaction.

Cohérence : une transaction terminée (fin-de-transaction) préserve la cohérence du système. Une transaction transforme donc le système d'un état cohérent à une autre état cohérent.

Isolation : une transaction ne doit pas interférer avec une transaction concurrente. Le terme de "synchronisation" [SCHL78] couvre l'ensemble des techniques qui assurent cette isolation. Le paragraphe 1.5 ci-dessous décrit les problèmes de la concurrence et apporte certaines solutions.

Durabilité : une fois qu'une transaction a été confirmée, le système doit garantir que ses résultats survivent à n'importe quelle défaillance qui pourrait se produire.

1.5 Reprise

Nous ne voulons pas, dans cette section, donner une description détaillée des différentes techniques de reprise. Des chapitres entiers ultérieurs y sont consacrés (Chapitre 2 et 3 pour les programmes et chapitre 4 pour les bases de données). Nous voulons seulement donner un aperçu général du concept de reprise dans un environnement transactionnel afin de pouvoir introduire des notions fondamentales nécessaires pour la suite de l'exposé.

Dans un premier temps, nous montrons comment une défaillance peut perturber un système et nous proposons des opérations de restauration (1.5.1). Ensuite, nous donnons une technique, la journalisation, qui permet de mémoriser de l'information redondante pour la restauration (1.5.2). Dans un troisième temps, nous décrivons les grandes techniques d'exploitation du journal en fonction des types d'incidents (1.5.3). Nous terminons par une simple énumération des différents domaines de reprise (1.5.4).

1.5.1 Perturbation par défaillance et restauration

La figure 1.6 [GRAY79] représente les différentes situations dans lesquelles peuvent se trouver des transactions lors de la détection d'un incident. Ces situations sont fonction de la position des transactions par rapport au point de reprise et à la détection de l'incident.

Nous envisageons deux opérations de restauration [GRAY81]:

- DEFAIRE ("UNDO") : Annule l'effet d'une transaction sur le système d'information.

- REFAIRE ("REDO") : Refait une transaction.

Restauration de la base de données à un état cohérent (correspond aux pointillés de la figure ci-dessus) par "réajustement" des transactions:

T1: Rien (OK)
T2: Comme le point de reprise et les différentes opérations des transactions (dont les "début-transaction" et "fin-

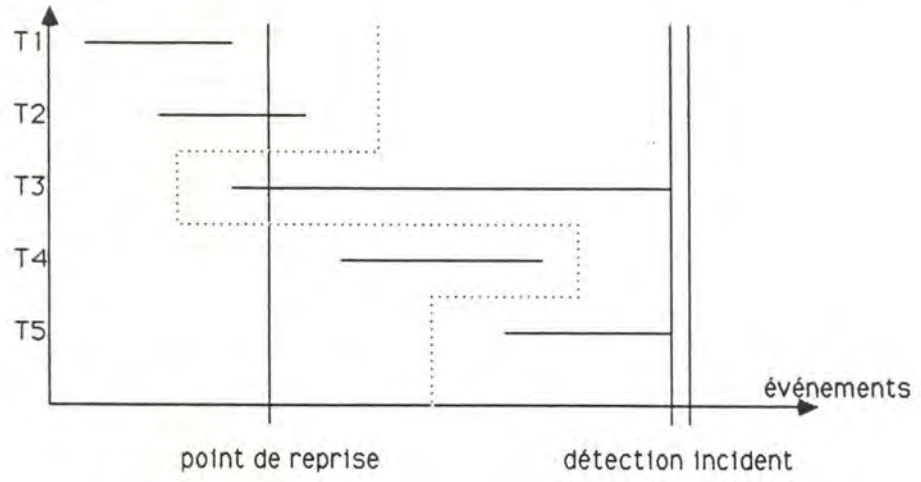

Figure 1.6: Diagramme incident / transaction

transaction") ont été enregistrées dans un journal (cfr section suivante), nous pouvons repartir du point de reprise, s'apercevoir que T2 se clôture avant la détection de la défaillance, et ainsi REFAIRE la transaction.
T3: Transaction non clause ==> DEFAIRE ce qui a été fait.
T4: Cas similaire à T2 ==> REFAIRE.
T5: Cas similaire à T3 mais si nous repartons du point de reprise, il n'y a plus rien à faire.

1.5.2 Le journal

Le journal est un fichier séquentiel dans lequel une série d'informations redondantes est sauvegardée. Il contient entre autres :

- des images des données (ou programmes) AVANT modification,
- des images des données (ou programmes) APRES modification,
- des points de reprise
  * de transaction (correspondant aux "début-transaction" et "fin-transaction")
  * forcés (point de cohérence forcé).

La taille du journal dépend de la granularité de l'image. Cette granularité varie de l'image la plus fine, à savoir une valeur d'item ou de variable, à la plus grande, à savoir tout le fichier ou programme, en passant par la plus grossière et pourtant la plus utilisée, la page physique. La taille de ce journal, prenant la page physique comme granularité, croît très rapidement. Cependant, la facilité de gestion des pages physiques permet une réinstallation très rapide des données en cas de défaillance [HAIN87], [RIES77].

N.B. : ce fichier journal doit être très protégé et spécialement pourvu d'un mécanisme de sauvegarde en cas de problème. En effet, nous utilisons ce journal lorsqu'une défaillance a été détectée, par exemple une coupure de courant provoquant l'effacement des tampons en mémoire centrale. Nous devons donc prévoir que ce fichier séquentiel risque de ne pas être fermé par un label de fin de fichier ("EOF"). Nous pouvons envisager une technique où la fin du fichier est signalée par des pointeurs se trouvant dans des petits fichiers sur des supports différents.

Le journal doit toujours être "plus frais" que le système d'information mis à jour. Nous devons donc toujours écrire dans le journal avant la répercussion des modifications ("Write Ahead Protocol"), sauf le "fin-transaction" qui s'écrit après l'exécution de l'opération [GRAY79].

1.5.3 Les grandes techniques d'exploitation du journal

Le principe est de ramener le système d'information dans un état cohérent.

1. Reprise avant ("GLOBAL REDO").

   Deux cas sont à envisager suivant la gravité de l'incident détecté :

   - Pour un incident majeur, on parle de REPRISE A FROID.

   Ce genre d'incident, comme la perte d'un support, nous oblige à revenir loin en arrière, à savoir de repartir d'une copie de sauvegarde ("backup"). Nous revenons alors au dernier état cohérent en utilisant les "images après" du journal.

   Deux techniques peuvent être envisagées:

   1. La première méthode est dite brutale.

      Elle consiste à parcourir le journal de façon séquentielle en reprenant les opérations dès le début. Pour s'arrêter au dernier point de cohérence, soit celui-ci est signalé par un pointeur, soit alors nous allons jusqu'à la fin du fichier journal, puis nous revenons un peu en arrière, le but étant de revenir au dernier état cohérent. On se rend compte de la faiblesse de ce procédé si l'on possède un vieux journal.

2. La seconde méthode est meilleure.

   Nous partons de la fin du fichier. Seule la dernière modification de chaque page est effectuée, c'est-à-dire la première rencontrée. Nous devons donc maintenir une liste des pages déjà parcourues.

Cette reprise à froid est relativement simple à gérer. Cependant, tout le fichier journal doit être lu. On parle de reprise à FROID car tout le système doit rester inactif (plus ou moins paralysé) tout au long de la restauration de la base de données.

- pour un incident mineur, on parle de REPRISE A CHAUD.

Dans ce cas, la reprise s'effectue assez rapidement. Nous devons seulement, puisque l'incident n'est pas grave, DEFAIRE les transactions non terminées au moment de l'incident, et REFAIRE les transactions terminées. Nous sentons ici le besoin d'avoir des transactions les plus petites possibles afin d'avoir des reprises extrêmement rapides.

1. Reprise arrière ("GLOBAL UNDO").

   Nous voulons retrouver en arrière un point de cohérence. Les raisons de ce retour en arrière peuvent être de deux sortes:

   1. Défaire des transactions non terminées (Avorte-Transaction).

   2. Défaire une transaction pour annuler un interblocage (voir la section 1.5.2).

Notons que nous avons abordé la question des techniques du seul point de vue de la cohérence. Dans l'optique d'un service continu notre gestion reste incomplète quant aux transactions défaites. Ces transactions pourraient être reprises afin de cacher la défaillance à l'utilisateur final. Le problème de la continuité de service est abordé dans le chapitre trois.

1.5.4 Les différents domaines de reprise

1. Les Bases de Données: gestion détaillée dans le chapitre quatre.
   --> journal-BD : contient des informations sur les

modifications de la BD.
Les problèmes sont liés à la perte des tampons.

2. Les terminaux: gestion semblable et complémentaire aux bases de données.
--> journal-TERM : contient les messages [RAND78].

3. Les programmes:

* En transactionnel, nous avons de petits modules (saisie écran, sans mémoire, ...), ce qui permet une gestion simple des reprises à partir du journal-BD et du journal-TERM.

* Si nous avons des programmes plus complexes, c'est-à-dire des gros programmes avec transfert de données entre programmes (variables), il faut alors reprendre l'exécution du programme à partir de certains points où le programmeur aura mémorisé les variables en question, les labels, et le registre d'instruction ("p-counter") [KORE86]. En COBOL, par exemple, nous possédons une primitive qui permet la création de tels points ("RESTART"). Si nous n'avons pas de tels outils, nous devons alors créer une programmation par les messages.

1.6 Gestion de la concurrence

Dans cette section, nous allons analyser de façon globale les problèmes de concurrence dans une base de données. Notre intention n'est pas de faire une analyse complète et détaillée de tous les problèmes liés à l'accès concurrentiel à des ressources, ce qui pourrait faire l'objet d'un mémoire à part entière, mais simplement de donner une synthèse des problèmes posés (1.6.1) et des solutions (1.6.2) proposées dans la littérature.

Les accès concurrentiels à une base de données centralisée représentent une partie importante de la littérature consacrée au problème de la concurrence [BERN79], [CASA81]. Les solutions proposées peuvent d'ailleurs être généralisées à d'autres ressources ou à des bases de données réparties [MENA80], [KOHL81].

De plus, les concepts de contrôle de concurrence et de reprise sont intimement liés. D'une part, pour résoudre le problème des reprises, celui de la concurrence entre processus doit d'abord être résolu. D'autre part, les techniques se basent sur des concepts similaires (atomicité, ...). Agrawal [AGRA85] a intégré ces deux notions pour en analyser les performances.

1.6.1 Les problèmes

Nous distinguons deux classes d'opérations d'accès à la base de données :

- les opérations pour de simples consultations. La littérature parle de lecteur
- les opérations amenant une modification de la base de données. La littérature parle de rédacteur.

Des problèmes, de deux sortes, peuvent survenir quand un ou plusieurs rédacteurs travaillent sur la base de données:

- la destruction de l'intégrité de la base de données par des modifications simultanées de mêmes données (appelée corruption)

- la perception incorrecte de la base de données par des lecteurs parce qu'un (ou plusieurs) rédacteur (s) effectue (nt) simultanément des modifications.

Les problèmes viennent du fait qu'il y a trois niveaux de mémoire:

- la base de données sur support externe
- les données dans les tampons du système de gestion de base de données
- les tampons logiques des programmes d'application (variables),

et que ces niveaux ne sont pas toujours synchronisés.

Pour illustrer les problèmes, nous allons prendre quatre exemples [BERN81] tirés du domaine bancaire, et qui permettent de discerner quatre types de problèmes différents [HAIN87].

1. La Mise à jour Perdue ("Lost Update")

Supposons deux clients qui essayent simultanément de déposer de l'argent sur un même compte. En l'absence de mécanisme de contrôle, ces deux activités peuvent "interférer" (figure 1.7).

Si les clients, ou plus exactement deux transactions associées aux opérations des deux clients, lisent le montant d'un compte plus ou moins au même moment, ils effectuent leurs opérations sur le même montant d'origine puis réécrivent les résultats l'un après l'autre dans la base de données. La deuxième réécriture écrase la première.

2. La Confusion de Courant

Supposons deux sièges d'une banque, l'un à Namur et l'autre à Liège, ayant chacun leurs agents. Les agents d'un siège sont reliés entre eux dans la base de données comme le montre la figure 1.7. Supposons deux transactions parallèles, l'une (T1) demandant la liste des agents d'un siège (Namur), l'autre (T2) mutant un agent (a2) de ce siège vers un autre (Liège).

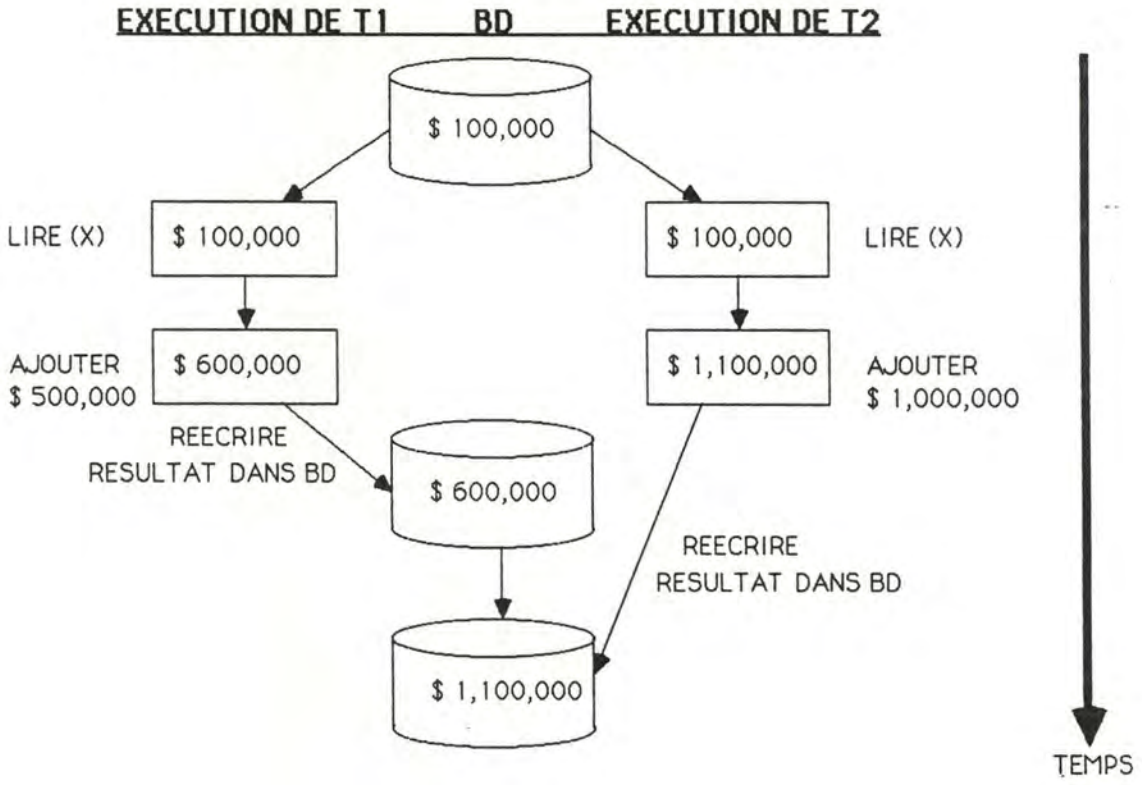

Figure 1.7: Mise à jour Perdue

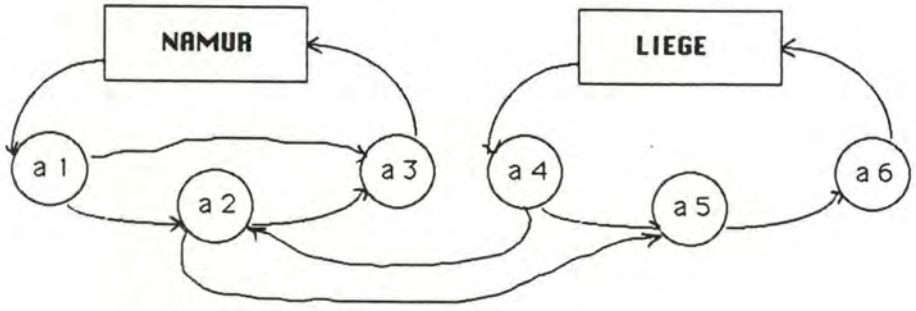

Figure 1.8: Confusion de Courant

Soit l'exécution temporelle suivante:

- T1 accède à a1,
- T2 accède à a1, puis effectue le transfert de a2 (par modification des pointeurs: a1 pointe vers a3, a4 vers a2 et a2 vers a5),
- T1, par son tampon logique, voit pointer a1 vers a2 (ancien pointeur).

Le résultat de T1 est a1, a2, a5, a6 !

3. L' Incohérence Statistique ("Inconsistent Retrieval")

Supposons l'exécution simultanée de deux transactions sur le fichier des comptes. La première, T1, calcule la somme de tous les dépôts, la seconde, T2, fait un transfert d'un compte à un autre. La figure 1.9 donne un exemple de cette situation.

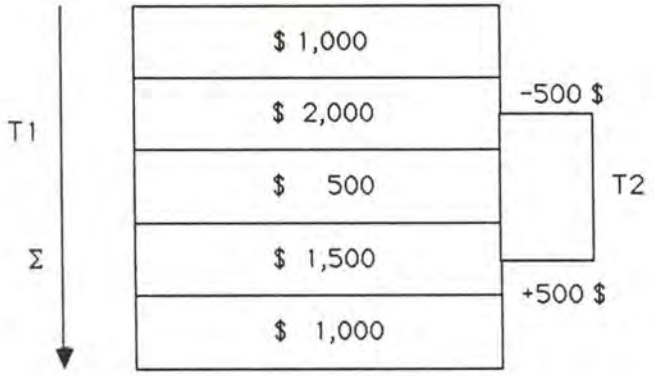

Figure 1.9: Incohérence Statistique

Si la première transaction prend la valeur du deuxième compte avant le transfert; si ensuite, la seconde transaction effectue le transfert; et puis la première transaction prend le montant du compte quatre après transfert, nous n'obtenons pas la bonne somme (excédentaire de 500 $).

N.B. Dans ce cas, la base de données reste cohérente, mais la perception que nous en avons est faussée.

4. Les données instables ("Dirty Data")

Prenons deux transactions, la première lit une donnée "A" puis la modifie (A'). Ensuite, une deuxième transaction lit cette donnée modifiée (A') et continue son exécution. Enfin, la première transaction doit être avortée (et donc être défaite par le principe de l' ACID Test). La donnée A' n'a donc jamais existé. La figure 1.10 représente cette exécution.

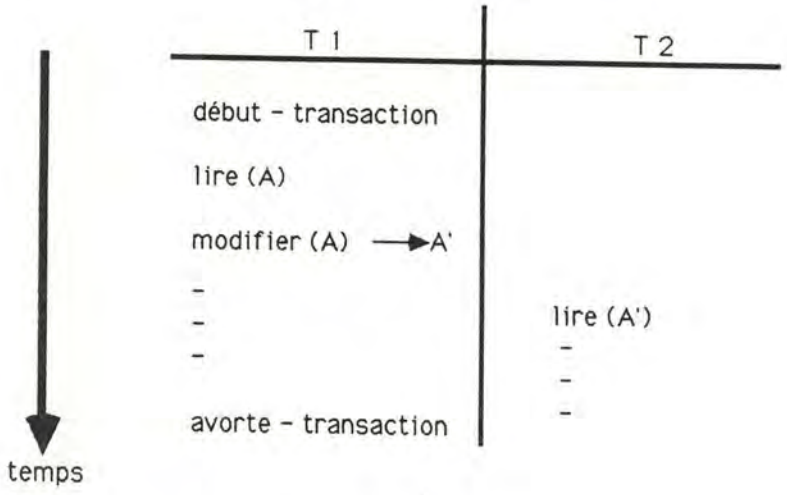

Figure 1.10: Donnée Instable

Ces quatre cas mettent tous en évidence le même problème, celui de la synchronisation de certaines opérations [SCHL78]. En effet, des lecteurs seuls ne posent pas de problèmes de concurrence. Par contre, si un ou plusieurs rédacteurs veulent entrer en jeu, nous devons pouvoir assurer la bonne utilisation des données.

Certains auteurs [BERN79], [BERN81], [CASA80], [CASA81], [ESWA76] ont démontré que si l'exécution simultanée est sérialisable , c'est-à-dire si l'exécution entrelacée produit les mêmes résultats qu' une exécution séquentielle des différents processus (des différentes transactions), alors les problèmes d'accès concurrentiels sont résolus.

1.6.2 Les Solutions

Nous allons décrire deux solutions acceptées par tous dans la littérature [AGRA85], et qui assurent la sérialisabilité décrite plus haut: le verrouillage [BAYE76] et l' estampillage [BAYE82]. Une troisième méthode, dite "optimiste" [KUNG81], est parfois préconisée pour des raisons de performance.

1. Le verrouillage ("locking")

La technique du verrouillage consiste à poser des verrous sur des données afin d'en limiter l'accès, c'est-à-dire faire des réservations exclusives ou non sur

ces ressources [GRAY79].

Afin d'éviter les données instables de l'exemple 4 ci-dessus, nous imposons que le déverrouillage qui restitue la ressource après utilisation ait lieu à la confirmation de la transaction (au "fin-de-transaction"). La littérature [BERN79], [BERN81], parle de protocole de verrouilage de transaction à deux phases ("Two Phase Locking Protocol"), la première pour la pose du verrou, la seconde pour le déverrouillage.

Le verrouillage exclusif assure la séquentialité. Seulement, il supprime tout à fait le parallélisme possible entre diverses opérations de lecteurs. Nous introduisons donc le concept de verrou partagé pour les lecteurs, puisque plusieurs consultations sur le même objet n'entament en rien l'intégrité des résultats. Franaszek [FRAN85] a étudié à l'aide de modèles analytiques les limites de concurrence entre transactions. Nous devons bien sûr prendre en compte la compatibilité des types de verrous lors d'accès concurrentiels.

verrou courant / requête	Partagé	Exclusif
Partagée	Compatible	Conflit
Exclisive	Conflit	Conflit

Figure 1.11: Tableau des Compatibilités

Le tableau des compatibilités (figure 1.12) donne les combinaisons de verrous exclusifs et de verrous partagés [KOHL81]. Dès qu'un rédacteur demande un accès (en mode exclusif), tout autre accès est exclu. Le changement de mode opéré s'appelle une conversion de verrou [MOHA82], [MOHA85]. La figure 1.12 donne le tableau des conversions.

verrou courant / requête	Partagé	Exclusif
Partagée	Compatible (>P)	Compatible (>X)
Exclisive	Conflit (>?)	Compatible (>X)

Figure 1.12: Tableau des Conversions

Si la technique du verrouillage donne une solution aux accès concurrentiels, elle n'en introduit pas moins deux nouveaux problèmes, le verrouillage mortel ou interblocage ("deadlock") et la famine ("starvation").

A. L'interblocage

L'interblocage se caractérise par la mise en attente de deux transactions parce que l'une (T1) attend la libération d'une ressource (B) utilisée (et donc verrouillée) par l'autre (T2) et réciproquement (figure 1.13).

Transaction - 1	Transaction - 2
Verrouillage-exclusif de A	Verrouillage-exclusif de B
-	-
-	-
-	-
-	-
Demande-ver-exclusif de B ==> attend	Demande-ver-exclusif de A ==> attend

Figure 1.13: L'interblocage

Deux optiques peuvent être prises pour résoudre ce problème [MOHA85] :

- soit on ne fait rien et on agit quand cela arrive (correction) [SINH85a]
- soit on empêche que cela arrive (prévention).

Pour les deux solutions, nous devons avoir de toute façon des outils de contrôle. Nous créons un graphe des mises en attente (suite à une demande de réservation de ressource déjà verrouillée en mode exclusif). Nous avons un verrou mortel lorsqu'un circuit est créé [BERN79].

La technique de correction consiste à éliminer ce circuit par la suppression d'un arc de blocage. Un arc de blocage représente une demande de réservation de ressource déjà allouée en mode exclusif. Nous défaisons donc une transaction (UNDO). Le choix de cette transaction (de cet arc) varie suivant la stratégie employée (la plus jeune, la plus vieille, celle qui a fait le moins de mises à jour ce qui accélère le DEFAIRE, etc...) [LEUN79]. Bien entendu, si l'algorithme de choix n'est pas équitable, nous risquons de voir apparaître le phénomène de la famine. Nous le décrivons et le solutionnons plus loin.

La technique de prévention consiste à empêcher l'apparition d'un cycle [RYPK79]. Nous employons alors une stratégie d'ordonnancement, politique plus restrictive puisqu'elle admet moins de concurrence, de parallélisme. Une technique bien connue consiste à empêcher l'entrée de transactions dans une section critique [LAUS85]. Cette section critique est en fait un domaine de valeurs (généralement le numéro des pages modifiées). Nous décrivons plus loin une autre technique, l'estampillage.

B. La famine

La famine se caractérise par la coalition de plusieurs transactions au détriment d'une autre qui ne se voit jamais attribuer la ressource demandée. Le processus (de la transaction) se trouve ainsi bloqué ... indéfiniment [BAYE80].

Pour éviter ce problème, nous devons veiller à une allocation équitable des ressources entre les diverses transations. La technique la plus utilisée consiste à employer des estampilles.

2. L' Estampillage ("Timestamping")

Le mécanisme de l'estampillage consiste à allouer un numéro d'ordre (l'estampille) à chaque transaction. Nous pouvons résoudre les problèmes d'allocation des ressources sans employer de verrous en imposant une exécution séquentielle, celle des estampilles. Des mécanismes de rajeunissement des numéros d'estampille (en défaisant une transaction puis en la refaisant), assurent une allocation équitable des ressources (plus de famine).

3. La Méthode Optimiste ("Optimistic Method")

La technique "Optimiste" consiste à ne rien faire pour les accès concurrentiels. En effet, les mécanismes du verrouillage et de l'estampillage coûtent chers tant pour leur implémentation que pour la suppression parfois inutile du parallélisme [KIES83], [TAY85]. Des études ont été menées pour tenter de diminuer ces coûts en jouant sur la granularité des verrous [MOND85], [RIES77], mais même des coûts optimaux sont encore trop élevés.

Plusieurs auteurs [HINX84], [KUNG81] proposent une méthode basée sur une approche optimiste. Ils font l'hypothèse que les accès concurrentiels aux mêmes ressources sont rares, ou du moins que leur nombre reste peu significatif par rapport à l'ensemble des accès. Cette hypothèse est très souvent vérifiée pour les systèmes répartis [BERN84].

Pour assurer la cohérence du système d'information, nous devons bien sûr veiller à ce que des accès concurrentiels dangereux pour l'intégrité n'aient lieu. Cette vérification se fait "après coup", c'est-à-dire que les effets des transactions ne sont répercutés dans la base de données qu'en fin de transaction ("commit") [HINX84], si et seulement si aucun conflit n'est survenu. Donc, avant l'écriture, une phase de validation est effectuée [KUNG81]. Dans le cas d'un conflit, un DEFAIRE (UNDO) des transactions en concurrence est provoqué et celles-ci sont réessayées. Nous pouvons remarquer que défaire une transaction ne coûte quasi rien puisque celle-ci n'a pas été enregistrée. Les anciennes valeurs sont donc encore disponibles.

Chapitre 2

Techniques de Support

à la

Continuité de Service

2 Techniques de support à la continuité de service

2.1 Introduction

Ce chapitre traite des techniques permettant la continuité de service à travers quatre grandes stratégies décrites par Randell [RAND78]. Mais avant cela il introduit la redondance, base de toutes les techniques de tolérance.

2.2 Redondance

Selon Lee et Anderson dans [ANDE81], la redondance peut être définie comme "l'utilisation d'éléments supplémentaires qui n'auraient pas été nécessaires au système s'il avait été sans faute".

Son utilisation permet de détecter, voire corriger, des fautes matérielles et logicielles du système.

Deux formes de redondance peuvent être distinguées:

- La redondance en composant (matériels & logiciels):

  le système dispose de portes, de cellules mémoires, de bus et de modules fonctionnels supplémentaires.

- La redondance en temps:

  se base sur l'utilisation répétée d'un même composant.

  Nous pouvons donner l'exemple des exécutions multiples de mêmes calculs mais avec des méthodes différentes (suivies du contrôle de la concordance des résultats). Cette redondance est habituellement réalisée par programme.

Il est encore utile de distinguer parmi la redondance en composants :

- la redondance statique :
  destinée à masquer les erreurs dans un composant. Celui-

ci continue à travailler correctement malgré l'occurrence de la faute.

- la redondance dynamique :
  employée dans un composant, uniquement pour détecter s'il y a des erreurs dans ses outputs. Cette forme de redondance doit être complétée par une redondance "externe" au composant.

  Par exemple, l'utilisation des codes auto-correcteurs en Mémoire Centrale peut être vue comme une application de redondance statique, alors qu'un simple code de parité peut être vu comme une redondance dynamique puisque la tolérance d'une faute de mémoire exige une redondance ailleurs (dans le code du système d'exploitation).

La fiabilité gagnée par la duplication d'un composant matériel est facile à calculer.

Rappelons avant tout la formule donnant la disponibilité d'un composant A :

$$D(A) = \frac{MTBF}{MTBF+MTTR} \qquad 0 <= D(A) < 1$$

avec (MTBF: temps moyen entre deux défaillances du composant
(MTTR: " " de réparation du composant

Soient A1 et A2, deux composants identiques disposés en parallèle,
et (D(A1),D(A2)) leur disponibilité respective.

Si A est composé de A1 et A2 (figure 2.1), alors A est "disponible" si au moins UN des deux composants est disponible, ou encore : A est défaillant si à la fois A1 et A2 le sont.

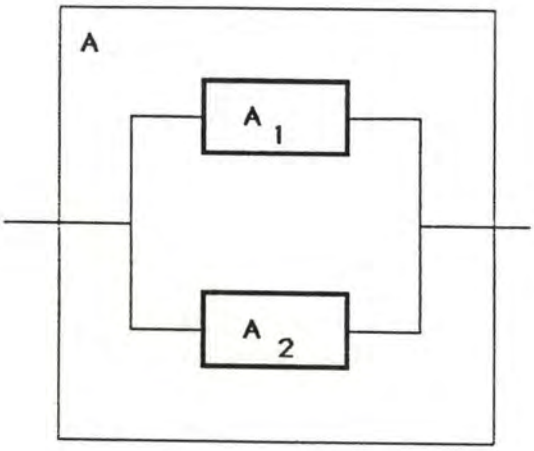

Figure 2.1: Le composant A

Ainsi la disponibilité A :

D(A): $D(A) = 1 - ((1-D(A1)) (1-D(A2)))$
$= D(A1) + D(A2) - D(A1) D(A2)$

La redondance n'a malheureusement pas que des aspects positifs : elle accroît la complexité des systèmes et risque ainsi d'engendrer de nouvelles erreurs. C'est pourquoi il faut l'implémenter avec méthode et prémunir contre les fautes les mécanismes qui la mettent en oeuvre. De plus, un système redondant procure toujours un comportement redondant mettant en jeu un délai supplémentaire.

Dans les systèmes de gestion, la redondance est généralement déployée par une simple duplication.

2.3 Stratégies

Selon Randell [RAND78], les techniques permettant la continuité de service peuvent être classées en quatre stratégies. Celles-ci sont tout d'abord présentées succintement ci-dessous, puis sont reprises en détail dans les sections suivantes.

1. La détection des erreurs :

   Une faute ne peut être tolérée que si le système détecte ses effets c'est-à-dire l'état erroné qu'elle engendre.

   La réussite de tout système tolérant aux fautes dépend fortement de l'efficacité des techniques de détection des erreurs.

2. L'évaluation des dégâts :

   Avant qu'une erreur ne soit détectée, il se passe un certain temps pendant lequel elle se propage, engendrant de nouvelles erreurs. La propagation d'une erreur peut vouer à l'échec les tentatives de récupération relatives à la faute, du fait de l'éloignement entre l'occurrence de l'erreur et sa détection. La propriété qui évite la propagation des erreurs est le "confinement des erreurs".

   Les techniques de détection ne disent rien sur le type d'erreur rencontrée (transitoire/permanente). Cette information ainsi que des précisions sur l'étendue des dommages sont pourtant nécessaires à la définition des opérations à effectuer pour assurer le service continu.

   Cette stratégie regroupe toutes les techniques et les décisions de conception permettant une évaluation des dégâts.

3. Le recouvrement d'erreur ("recovery"):

   Après avoir été détectées, les erreurs doivent être supprimées, de manière à remettre le système dans un état cohérent. Celui-ci est

généralement un état qui a précédé la détection de l'erreur et est appelé point de reprise (checkpoint). C'est l'opération qui consiste à passer d'un état erroné à un état sans erreur que l'on appelle le recouvrement.

Le recouvrement est aussi bien utilisé pour le service continu des processus en Mémoire Centrale que pour préserver l'intégrité de la Base de Données. Dans le cas du service continu, le recouvrement permet de remettre le système dans un état tel qu'il puisse traiter la faute. C'est de ce recouvrement qu'il est question dans ce chapitre.

Le recouvrement de l'intégrité de la Base de Données est étudié en détail dans le chapitre 4 sous la dénomination "Techniques de reprise BD". L'action de reprise y signifie plus que le simple retour à un état cohérent.

4. Le traitement des fautes :

Cette stratégie regroupe les techniques permettant au système de continuer ou de redémarrer le service interrompu.

L'ordre dans lequel ces stratégies sont appliquées varie, mais le point de départ reste toujours la détection d'une erreur.

De plus, ces stratégies ne sont pas forcément présentes dans tous les systèmes existants. L'évaluation des dégâts, par exemple, nécessite des techniques exploratoires difficiles à implémenter. Leur présence dans le système dépend des décisions de conception et des objectifs fixés.

Chacune de ces stratégies peut être mise en oeuvre par une série de techniques, mais ce regroupement des techniques en stratégies n'est pas rigoureux. Certaines de ces techniques, comme les tests de diagnostic, se retrouvent en effet dans plusieurs stratégies.

Ces techniques peuvent, pour la plupart, être implémentées physiquement dans les composants matériels, ou logiquement par des programmes. Dans certains cas il y a même les deux possibilités. Ce sont alors des critères d'efficacité et des décisions de conception qui orientent le choix.

La figure 2.2 schématise les opérations effectuées par le système suite à l'apparition d'une faute simple. Le scénario est simplifié à l'extrême en ne reprenant que les stratégies décrites ci-dessus.

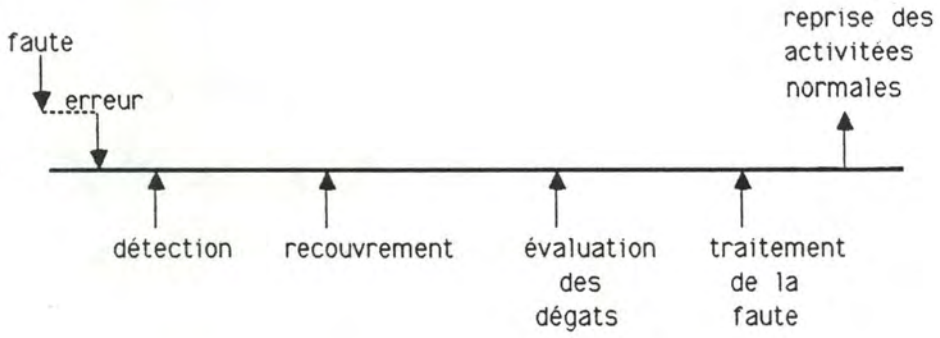

Figure 2.2: Scénario simplifié

2.4 La détection des erreurs

2.4.1 Introduction

L'objectif de la détection des erreurs est de donner au système la possibilité de réagir à l'apparition d'une faute avant qu'elle ne provoque une défaillance. Cet objectif est atteint grâce à des techniques permettant de détecter des états erronés.

Ces techniques sont détaillées en (2.4.4), mais, avant cela, nous précisons les lieux, dans le système, où elles sont déployées (2.4.2) ainsi que leurs principales propriétés (2.4.3). Nous terminons ce point dédié à la détection par une classification de ces techniques en trois catégories (2.4.5).

2.4.2 Où placer les tests ?

Un composant fautif doit, soit se détecter en erreur, de manière à faire jouer ses mécanismes de tolérance ou simplement signaler ses erreurs au système auquel il appartient, soit être détecté en erreur par ce dernier.

Les tests sont situés au début, pendant ou juste après le service du composant, que celui-ci soit matériel ou logiciel.

Les tests déployés au début (à l'entrée) d'un composant ont pour but de valider ses paramètres d'appel. Ces tests, dénommés "tests d'interface" , sont surtout utilisés par le logiciel. Les tests fournis par l'interpréteur (division par 0, instruction invalide, saut en dehors de la zone mémoire ...) sont aussi des tests d'interface.

Dans la littérature [RENN84], les composants matériels sont regroupés en deux catégories extrêmes selon qu'ils exécutent ou NON des tests pendant leur service. Une distinction est ainsi faite entre :

- les circuits qui détectent leurs erreurs parallèlement à leurs opérations normales, par exemple en alternant des cycles de tests avec des cycles d'opérations normales. Ils sont dits auto-contrôlés ("Self-Checking (SC)

module").

- les autres composants, ceux qui n'ont pas la possibilité de détecter leurs fautes, sont dits non auto-contrôlés ("NonSelf-Checking (NSC) module").

Comme nous l'avons défini précédemment, le comportement d'un système doit être conforme à celui décrit dans sa spécification sinon il s'en suit une défaillance. Une bonne manière de vérifier cela est de tester les résultats à la sortie du système.

Idéalement, ce test doit être uniquement basé sur la spécification du système puisque l'architecture de celui-ci peut cacher des fautes de conception. Le test traite alors le système comme une "boîte noire" laissant de côté tout ce qui se passe à l'intérieur de la boîte. De plus, ce test doit être indépendant du système lui-même pour éviter qu'une faute simple n'affecte à la fois le système et le test, empêchant ainsi la détection de l'erreur qui en résulte.

En pratique, des tests si rigoureux ne sont que difficilement applicables parce qu'ils entraînent un accroissement du coût et de la complexité des systèmes. De plus, l'indépendance entre un système et ses tests ne peut être absolue car, de toute manière, les tests doivent avoir accès à l'information à tester et ils ont alors la possibilité de la corrompre. C'est pourquoi la majorité des systèmes utilisent des tests basés sur la conduite supposée du système et de son environnement ainsi que sur l'identification des fautes possibles. Ces tests, dit d'acceptabilité, visent à détecter la majorité des états erronés, mais ne garantissent pas l'absence d'erreur.

Ces tests sont généralement placés à la sortie du système de manière à pouvoir porter sur toute son activité. Néanmoins, dans certains cas, il est intéressant de les placer à l'intérieur du système de manière à réduire l'activité inutile pendant laquelle, d'ailleurs, les erreurs se propagent. L'inconvénient de cette deuxième possibilité reste la dépendance du test vis-à-vis du système.

De toutes manières, une erreur non détectée au niveau du matériel l'est généralement au niveau du logiciel à travers l'état erroné qu'elle engendre.

2.4.3 Propriétés des méthodes de détection

Avant de détailler chacune de ces techniques, il est bon de définir deux propriétés des méthodes de détection importantes pour l'évaluation des dégâts et le recouvrement ainsi que pour déterminer la confiance à placer dans le système.

- La couverture d'une procédure de détection ("detection coverage") est la probabilité qu'une faute apparue soit détectée par cette procédure. Il est difficile d'avoir une couverture complète parce qu'il est très coûteux de concevoir des mécanismes de détection recouvrant tous les types de fautes et de construire des mécanismes de test physiquement indépendants du module à tester.

- Le temps de latence d'une procédure de détection ("detection latency") est la durée entre l'occurrence d'une faute et sa détection par cette procédure.

  Trois valeurs sont généralement attribuées à cette durée:

  instantanée :

  la faute est détectée dès qu'elle apparaît, avant même qu'elle n'engendre un état erroné. C'est peu réaliste. En pratique ce n'est jamais le cas.

  simultanée :

  la faute est détectée avant que l'erreur qui en résulte ne se propage au delà de son composant d'origine. C'est le cas lorsque, par exemple, l'erreur est détectée dans le même top horloge ("clock cycle") que son apparition. La détection s'opère parallèlement aux opérations 'normales' du composant. Cette qualité ne peut appartenir qu'à des composants auto-contrôlés.

  non-simultanée :

  c'est le cas des composants non auto-contrôlés. L'erreur a tout le temps de se propager avant d'être détectée. Si le temps de latence des procédures de détection d'un système est significatif, il est alors possible que celui-ci fournisse des résultats incorrects. En effet, certaines erreurs peuvent apparaître dans la phase de sortie et avoir des chances de traverser les tests qui s'y trouvent.

2.4.4 Techniques de détection

Lee et Anderson [ANDE81] identifient sept grandes méthodes de détection.

Tests de
- duplication
- de minutage
- d'inversion
- de codification
- de "raisonnabilité" (du caractère raisonnable des résultats)
- de structure
- de diagnostic

Celles-ci sont reprises en détail dans les points 2.4.4.1 à 2.4.4.7 .

2.4.4.1 Les tests de duplication

Ce type de test est une application pure et simple de la redondance en temps et en composants, s'appliquant aussi bien à des composants matériels qu'à des composants logiciels. Ces tests procèdent par duplication de l'activité du système de manière à procurer à celui-ci une implémentation alternative avec laquelle il peut comparer ses résultats.

Le premier point discuté s'adresse aux tests de duplication réalisés au moyen de la redondance en composants matériels. Il s'agit là de l'utilisation la plus habituelle de ce type de test, c'est-à-dire la détection des fautes de matériel, comme celles qui peuvent apparaître dans les composants critiques (CPUs, contrôleurs...). Dans ce cas, le test d'acceptation est souvent réalisé par dédoublement de l'élément critique suivi d'un test de cohérence des résultats. Il ne faut pas perdre de vue le fait que le mécanisme de test peut lui aussi souffrir d'une faute. C'est pourquoi la majorité des systèmes se prétendant fiables ont un module de comparaison SC ou bien doublent leurs mécanismes de comparaison.

Lorsque plus de deux copies du composant critique sont utilisées pour la détection des erreurs, le mécanisme de comparaison est appelé vote. Dans ce système, le nombre de composants total (original et répliques) doit être impair.

Lorsqu'il est fait référence aux systèmes à vote, il s'agit de systèmes à (2N+1) processeurs exécutant simultanément la même application au bout de laquelle il y a vote majoritaire sur les résultats. L'avantage de ceux-ci est de pouvoir identifier facilement l'exemplaire du composant qui est dans un état erroné, ce qui n'était pas le cas pour le dédoublement.

La figure 2.3 représente, dans sa partie supérieure, deux exemples de dédoublement (respectivement avec et sans système de comparaison SC), et dans sa partie inférieure, un système à vote.

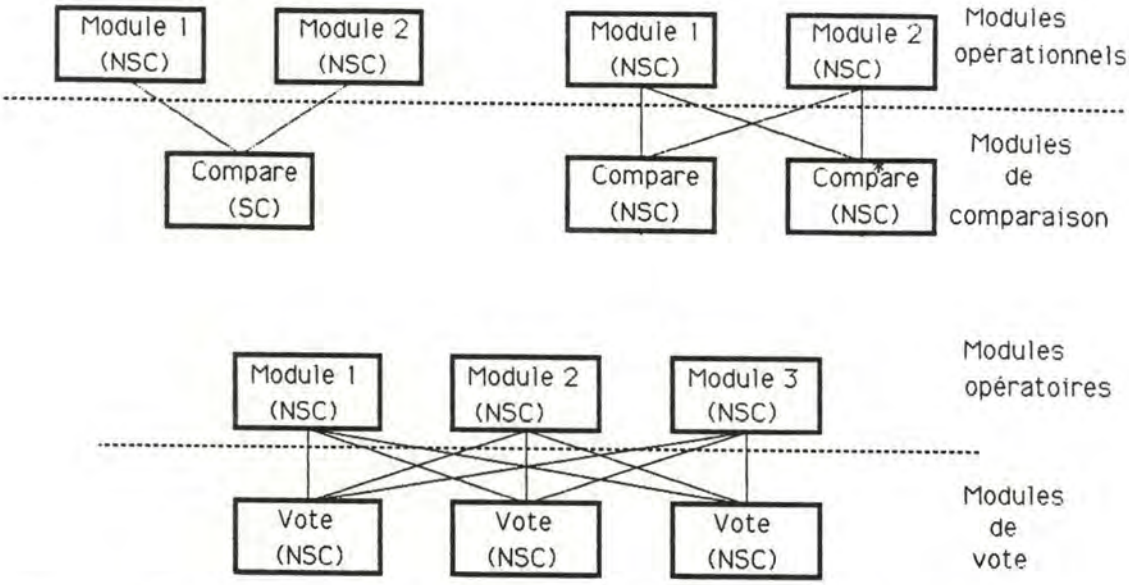

Figure 2.3: Redondance pour des composants matériels

La redondance en composants logiciels implémente ces tests d'acceptabilité de la même manière que la redondance en composants matériels; excepté qu'à la place d'avoir plusieurs composants d'architecture identique, nous avons plusieurs programmes (modules) vérifiant les mêmes spécifications mais construits de manières différentes. Cela tient à l'objectif même de cette implémentation qui vise la détection des erreurs résiduelles de conception dans les logiciels.

Les tests de duplication peuvent enfin être réalisés par l'utilisation de la redondance en temps. Celle-ci permet la détection d'erreurs transitoires par l'utilisation successive d'un composant matériel ou logiciel suivi par une comparaison des résultats.

Comme nous le verrons plus tard, la redondance déployée pour réaliser les tests de duplication permet aussi le masquage des fautes. Le surcoût important que nécessite cette technique de détection est donc justifié quand il doit y avoir recouvrement "ON-LINE", ce qui est le cas du service continu.

Cette technique permet la détection des fautes non-anticipées, parce que peu de suppositions sont faites sur les fautes qui peuvent apparaître. C'est la raison pour laquelle ce type de test est le plus puissant.

La figure 2.4 reprend les différentes implémentations possibles des tests de duplication, en terme de redondance en temps et en composants (mat/log), et y associe les classes de fautes visées.

matériel ou logiciel / redondance	matériel	logiciel
en composant	fautes - d'environnement - de vieillesse	fautes intermittentes résiduelles de conception
en temps	fautes transitoires d'environnement	fautes intermittentes résiduelles de conception

Figure 2.4: Tests de duplication

2.4.4.2 Tests de minutage ("Timing checks")

Les tests de minutage sont utilisés pour les composants matériels ou logiciels ayant des contraintes temporelles dans la spécification de leurs services.

- Si l'un de ces composants ne signale pas la fin de son service dans les temps respectant sa spécification, le système peut en déduire qu'il y a un problème.

- Par contre, le fait qu'un composant respecte sa contrainte temporelle ne signifie pas qu'il ait rendu un service en tout point conforme à sa spécification.

  C'est pourquoi les tests de minutage sont généralement utilisés en supplément à d'autres tests.

Les tests de minutage sont souvent utilisés dans les systèmes logiciels à partir de "watchdog timers (WT)" (littéralement traduit: "minuteries contrôleurs de séquences"). Ceux-ci permettent de détecter un contrôle de flux erroné(a) souvent causé par l'apparition d'une faute transitoire (erreur de codification, défaillance du matériel ou 'mutilation' d'instructions dans le programme).

Par exemple, il peut être imposé à un programme de réinitialiser périodiquement un compteur. Si, suite à une boucle infinie, ce compteur n'est pas réinitialisé, alors le WT travaillant en parallèle s'en rend compte et déclenche une procédure de traitement d'erreur.

Une autre utilisation des WTs [JUN82] nécessite que le programme soit partitionné en intervalles et qu'une base de données contenant des informations sur les chemins dans chacun de ces intervalles soit créée (à partir de l'architecture du programme). Lors de l'exécution, le chemin réellement parcouru dans chaque intervalle traversé est enregistré, puis comparé aux informations de la base de données. Tout écart est ainsi détecté. Le WT est donc un processus indépendant effectuant un test de contrôle de flux concourant. Il peut être exécuté soit sur le même processeur que les programmes à tester, soit sur un processeur différent.

Quand le processeur n'a pour seule fonction que de supporter le WT, il est appelé "watchdog processor (WP)". Celui-ci est un processeur indépendant qui surveille l'exécution des programmes sur le processeur principal en observant les transferts entre celui-ci et la mémoire (figure 2.5) [SHAM86]. Il détecte les fautes transitoires résultant en un contrôle de flux erroné ou une "mutilation" d'instructions dans le programme.

(a) Pour rappel, le contrôle de flux dans un programme structuré est déterminé principalement par quatre constructions:
1) la concaténation (BEGIN...END)
2) la sélection (IF...THEN...ELSE)
3) la répétition (REPEAT...UNTIL)
4) l'abstraction (CALL).

Tout ce qui empêche de mettre à jour correctement le registre d'instruction peut résulter en un contrôle de flux erroné. Les fautes matérielles et logicielles sont toutes deux concernées.

L'utilisation d'un WP pour l'implémentation de tests consacrés aux structures du contrôle de flux dans le programme est détaillé en [JUN82], [YAU80]

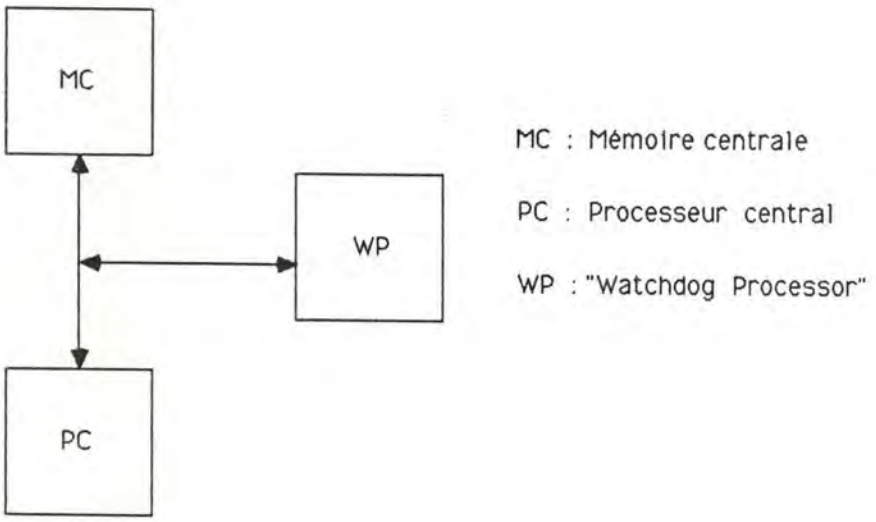

Figure 2.5: Watchdog Processor

La figure 2.5 montre un WP connecté au bus reliant le processeur central à la mémoire. Le WP contrôle l'exécution du programme indépendemment du processeur central.

2.4.4.3 Tests d'inversion

Les tests d'inversion sont utilisés par des systèmes dans lesquels il y a une relation de un à un entre les données et les résultats.

Après les opérations normales du système, pendant lesquelles celui-ci calcule des résultats à partir de données, ces tests font le chemin inverse en déduisant de nouvelles données à partir des résultats. Ces nouvelles données sont alors comparées aux données initiales.

Ces tests sont utilisés pour des composants matériels et logiciels. Par exemple, ils sont parfois implémentés pour contrôler les accès à la mémoire en s'assurant que les mots sélectionnés correspondent bien aux adresses désirées.

Une variante de ces tests d'inversion peut être utilisée dans les systèmes où il existe une relation fixe entre les

données et les résultats à tester. C'est sur cette relation que porte le test. Prenons, par exemple, le cas d'un programme calculant l'inverse d'une matrice donnée. Le test consiste alors à multiplier la matrice donnée en entrée par son inverse calculé par le programme, et à vérifier ensuite que le résultat est bien la matrice unité.

Ces tests ont l'avantage d'être indépendants du système et de pouvoir être exécutés au dernier moment.

2.4.4.4 Tests de codification

Les tests de codification sont basés sur une redondance dans la représentation des objets utilisés par le système. Les données redondantes maintiennent dans chacun de ces objets une relation fixe avec les données représentant la valeur de cet objet. Toutes les erreurs modifiant ces relations sont alors détectées.

Ces tests sont surtout utilisés pour le stockage de données en Mémoire Centrale et, puisque une grande partie des défaillances est due à des erreurs de mémoire, beaucoup d'études ont été réalisées à leur sujet.

Nous reprenons ci-dessous quelques exemples de codification.

- "parity check":
  un simple bit redondant est associé à un mot (plusieurs bits). Ce bit prend comme valeur le modulo 2 de la somme des bits du mot.
- "Hamming's codes":
  plusieurs bits redondants permettent de corriger un bit erroné dans le mot et de détecter des erreurs portant sur plusieurs bits.
- "checksum":
  sommation sur un bloc logique de données.
  Par exemple : au lieu de vérifier rigoureusement qu'un grand ensemble de nombres est la permutation d'un autre, il est plus simple de s'assurer que la somme des nombres de ces deux ensembles est identique.

La codification peut être réalisée par du matériel ou du logiciel.

Comparée à d'autres formes de tests, tel que la duplication, les tests de codification ont une couverture plus petite mais sont plus économiques. C'est pourquoi ils sont tant utilisés pour s'assurer de l'acceptabilité de grandes quantités de données.

2.4.4.5 Tests de "raisonnabilité"

Ces tests contrôlent le caractère raisonnable des résultats obtenus. Ils sont basés sur la connaissance de l'architecture du système. Ces tests sont souvent appliqués aux logiciels sous forme d'assertions.

Une forme habituelle de ce test est le "test de domaine" qui détermine si la valeur d'un objet particulier est dans un domaine acceptable (par exemple entre une borne inférieure et une borne supérieure).

La valeur d'un objet peut aussi être contrôlée par rapport aux valeurs d'autres objets, afin d'assurer une cohérence.

2.4.4.6 Les tests de structures

Les tests de structures permettent de vérifier l'intégrité de structures de données complexes telles que des listes, des files d'attente ou des arbres. Ces tests contrôlent la cohérence des chemins tracés aux travers de ces structures.

2.4.4.7 Les tests de diagnostic

Les tests de diagnostic sont situés aussi bien dans la stratégie de détection des erreurs que dans l'évaluation des dégâts.

Le principe du diagnostic est de faire travailler un composant sur un ensemble de données pour lesquelles les résultats 'corrects' sont connus, et de comparer les résultats

effectivement produits avec ceux attendus.

Ces tests sont exécutés périodiquement pour contrôler les composants matériels. Ils permettent d'activer, donc de détecter des fautes potentielles.

2.4.5 Classification des techniques de détection

Nous trouvons en [SHIN84a] une classification en trois catégories des techniques de détection basées sur l'endroit où elles sont déployées, leurs performances et les méthodes de recouvrement qui leur succèdent.

1. Les mécanismes de détection au niveau du signal.

   Ce sont des mécanismes implémentés par des circuits SC intégrés. Si une erreur est générée par une faute anticipée, le circuit la détecte immédiatement. Il s'agit d'une détection 'simultanée', il n'y a donc pas de propagation de l'erreur en dehors du composant.

   Les techniques classiques utilisées dans cette catégorie sont la codification et la duplication.

2. Les mécanismes de détection au niveau de la fonction.

   Ces mécanismes testent les informations à un plus haut niveau que la catégorie précédente. Ils supervisent les opérations du système par l'utilisation d'assertions sur le temps de réponse, les champs modifiés... et cela au moyen de tests d'acceptabilité implémentés en composants matériels et logiciels. Ils se placent comme des 'barrières' autour des composants NSC pour détecter leurs erreurs et les empêcher ainsi de se propager.

3. Les diagnostics périodiques.

   Cette catégorie correspond aux diagnostics vus au point précédent.

   Remarquons que le temps nécessaire à couvrir entièrement les composants à tester est en général

très long. C'est pourquoi les diagnostics sont généralement utilisés de manière "grossière" pendant les opérations normales du système, et sont périodiquement déployés en détail quand le système est à l'arrêt.

2.4.6 Conclusion

Les techniques décrites ci-dessus s'appliquent aussi bien aux composants matériels que logiciels. Souvent, une faute d'environnement ou de vieillissement non détectée par le matériel, l'est par le logiciel.

Dans tout système tolérant aux fautes, la détection des erreurs a une importance vitale. En principe, plus elle est implantée, plus fiable est le système. C'est le surcoût et la diminution des performances qu'elle engendre qui limitent son déploiement.

2.5 Evaluation des dégâts

2.5.1 Introduction

Lorsqu'une erreur est détectée, l'état du système peut être davantage suspecté que ce qui est réellement découvert erroné. Cela, parce que le délai entre l'occurrence de la faute et sa détection est suffisant pour que des informations erronées se soient étendues dans le système pouvant même aller jusqu'à gêner toutes les tentatives futures de recouvrement. C'est pourquoi, avant de tenter un recouvrement de l'erreur, il peut être utile d'évaluer l'étendue des dégâts causés au système.

L'évaluation des dégâts est toujours basée sur une plus ou moins grande anticipation des événements suivant l'apparition d'une faute. Dans un cas extrême, cette stratégie est seulement basée sur un raisonnement a priori du concepteur, tandis que dans l'autre, elle est implémentée sous forme de techniques exploratoires.

Cette stratégie repose sur une confiance dans la structure supposée du système opérationnel pour déterminer comment celui-ci réagit à l'occurrence d'une faute. La structure, quant à elle, dépend des contraintes imposées sur le flux d'information lors de la conception du système.

Le détail des techniques d'évaluation des dégâts est donné en (2.5.3), mais avant cela, il est indispensable de traiter du confinement des erreurs (2.5.2). Celui-ci, comme nous le disions au début de ce chapitre, permet de limiter la propagation des erreurs.

2.5.2 Confinement des erreurs

La propriété "d'isolation" d'une "action atomique" définie précédemment, permet de structurer l'activité d'un système et ainsi de confiner les erreurs qui y apparaissent.

Considérons la figure 2.6. Elle représente trois processus composés d'actions atomiques encore incomplètes au temps 't'.

Nous constatons que :

- le processus P1 est complètement "isolé" depuis C1
- les processus P2 et P3 ont peut-être échangé des informations depuis qu'ils sont passés en C2 et C3, mais que cette interaction a du cesser en D3

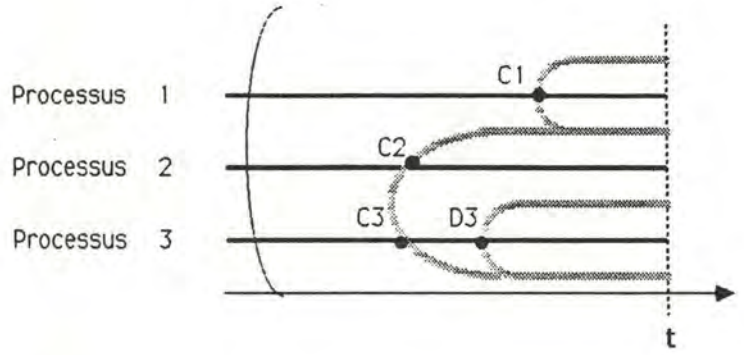

Figure 2.6: Exemple d'interactions entre actions atomiques

Supposons maintenant qu'une erreur soit détectée en "t". Plusieurs cas sont possibles, selon que

- c'est le processus P1 qui est détecté en erreur, nous pouvons suspecter P1
- c'est le processus P2 qui est détecté en erreur, nous pouvons suspecter tout ce qui a été fait depuis C2 et C3
- c'est le processus P3 qui est détecté en erreur, nous pouvons suspecter ce qui a été fait après D3.

Comme nous l'avons dit, la structure que le concepteur suppose présente dans le système est d'une importance vitale pour l'évaluation des dégâts. Il est ainsi nécessaire de disposer de mécanismes permettant de "forcer" la structure dans le système opérationnel, en lui donnant la possibilité de contraindre ses flux d'information. Ainsi, dans l'exemple précédent, le système doit disposer d'un mécanisme permettant à un ou plusieurs processus d'entrer, d'exécuter et de sortir d'une action atomique.

Un flux d'information entre deux processus est une conséquence de l'accès à un objet commun à ces deux processus, soit directement par manipulation d'un objet partagé, soit indirectement par un message. Dans le deuxième cas, où la communication entre processus ne peut s'effectuer qu'au travers l'échange de messages, les actions atomiques peuvent être implémentées par un mécanisme n'autorisant l'envoi de messages qu'aux processus participant à la même action

atomique. Lorsque, comme dans le premier cas, la communication entre processus s'effectue au moyen de variables partagées, l'implémentation du mécanisme est réalisée par l'imposition de contraintes d'accès à l'objet. La gestion des accès concurrents est assurée par des mécanismes de verrouillage ou d'estampillage décrits dans le chapitre 1. Dans les systèmes orientés BD, cela s'inscrit dans une problématique plus large qui est liée à la cohérence de la base de données.

La notion d'atomicité et ses avantages sur le confinement des erreurs ne se limitent pas uniquement aux logiciels. Pour les systèmes matériels, la séparation physique, l'isolation des composants, le fait qu'ils ne soient pas interconnectés forcent l'atomicité. L'avantage des systèmes matériels sur les logiciels, c'est que leur structure visible laisse plus facilement apparaître les flux d'information possibles et ceux qui ne le sont pas.

Pour terminer, notons qu'un bon moyen de confiner les erreurs est tout simplement de les détecter le plus rapidement possible et d'utiliser des tests d'interface.

2.5.3 Techniques d'évaluation des dégâts

Le point de départ de l'évaluation est une estimation initiale des dégâts possibles sur base de la structure supposée du système. Si aucune supposition n'est faite, alors les techniques d'évaluation des dégâts doivent suspecter l'ensemble du système.

En réponse à la détection d'une erreur, la technique d'évaluation la plus pessimiste consiste à suspecter tous les objets de l'évaluation initiale. Il s'agit là d'une estimation "statique" puisque réalisée exclusivement lors de la conception. Cette technique est souvent utilisée pour les systèmes matériels où les fautes sont supposées provoquer des erreurs bien caractéristiques. Mais, la complexité croissante du matériel réduit la "faisabilité" de cette évaluation statique.

Les techniques moins pessimistes pour l'évaluation des dégâts tentent "dynamiquement" de raffiner l'évaluation initiale. Elles peuvent être comprises comme "recherchant à identifier des actions atomiques" plutôt que "se fiant à une atomicité imposée lors de la conception". Ces techniques explorent les structures de données du système afin de détecter des erreurs. Les tests de diagnostics sont particulièrement utilisés à cet effet.

En pratique, l'évaluation des dégâts est fortement liée au recouvrement et au traitement de la faute. Elle constitue

généralement une stratégie incertaine et incomplète. En effet, il est plus facile d'empêcher l'expansion des erreurs que de déterminer, après leur détection, l'étendue des dégâts qui ont été causés.

2.6 Le Recouvrement

2.6.1 Introduction

La stratégie de recouvrement des erreurs a pour objectif de remplacer un état erroné par un état valide à partir duquel le système peut continuer à travailler. Il s'agit seulement là d'éliminer les erreurs, et pas de traiter les fautes originales ni de redémarrer l'application suspendue (ceci fait l'objet de la dernière stratégie: "le traitement des fautes").

Deux techniques de recouvrement peuvent être identifiées selon le degré d'anticipation des erreurs qu'elles visent. Si les erreurs peuvent être anticipées, il est possible d'implémenter un recouvrement rapide, effectuant une correction très précise de l'état du système. Cette technique se base sur des informations pertinentes sur l'état du système et est appelée recouvrement avant. Lorsque les erreurs ne peuvent être anticipées, dans le sens où elles sont difficilement prévisibles ou même évaluables, elles exigent une technique de recouvrement plus générale ne nécessitant pas d'information précise. Celle-ci, dite technique de recouvrement arrière consiste à remplacer l'état erroné du système par un état valide ayant existé. Nous parlons de "restauration" à un état valide.

La notion de "recouvrement" traitée dans cette section est apparentée à la notion de "reprise" décrite dans le chapitre 1. Cette dernière implique plusieurs "actions", dont un recouvrement arrière. Elle concerne plutôt le maintient de la cohérence de la base de données et est abordée en détail dans le chapitre 4.

2.6.2 Recouvrement avant

Cette technique part d'un état erroné du système et tente d'en corriger les erreurs à partir d'une estimation de leurs positions. Ainsi, cette technique est inséparable de la stratégie d'évaluation des dégâts et ne s'applique pas aux fautes pour lesquelles aucune information précise ne peut être obtenue (en l'occurence: les fautes non-anticipées).

Cette technique permet un recouvrement rapide et, pour cela, est utilisée le plus souvent possible là où la

continuité de service est primordiale.

Il n'existe pas de mécanisme général de mise en oeuvre de cette technique, aussi son implémentation dépend fortement du système où elle est déployée.

Elle est souvent utilisée pour le recouvrement des fautes de matériel (puisqu'elles sont anticipables) et quelques fois pour les fautes de logiciel.

L'implémentation la plus classique du recouvrement avant est réalisée au moyen de comparaisons et de votes. En effet, comme nous l'avons vu en 2.4.4.1, les tests de comparaison et de vote nécessitent la présence d'éléments redondants. Ceux-ci peuvent aussi être utilisés pour le recouvrement avant. Pour le vote, par exemple, si trois composants identiques effectuent les mêmes opérations et qu'un des composants est fautif, il suffit de continuer le service avec le résultat obtenu des deux autres. La technique du vote est souvent utilisée par redondance de matériel, mais rarement par redondance de logiciel.

Les codes auto-correcteurs permettent le recouvrement d'erreurs dûes au stockage ou à la transmission d'information. Ce type d'erreurs est tout-à-fait fréquent et il est aisé de localiser les endroits où elles se produisent. Il s'agit d'un recouvrement avant puisqu'il y a correction des bits erronés.

Dans certains systèmes, une redondance structurelle est parfois utilisée, sous forme de pointeurs et de champs supplémentaires, afin de donner à un programme la possibilité d'effectuer des tests de structure et de reconstruire la structure si elle est incohérente.

2.6.3 Recouvrement arrière

Comme nous l'avons décrit dans l'introduction, cette technique procède par restauration d'un ancien état (supposé valide) du système. Elle est ainsi directement liée à la notion de point de reprise définie précédemment. Son grand avantage est de procurer un mécanisme de recouvrement général permettant de tolérer un grand nombre de fautes, même non-anticipées. Ceci, parce qu'elle ne fait aucune supposition sur la nature de la faute et ne nécessite donc pas d'évaluation des dégâts.

Cette technique s'applique aux composants matériels à travers le "RESET". Celui-ci, présent dans tous les systèmes informatiques, permet de réinitialiser le système. Il constitue un cas particulier du recouvrement arrière où le point de reprise est l'état initial.

La technique du recouvrement arrière est principalement appliquée aux programmes par la technique des blocs de reprise ("recovery blocks") et par le principe des processus primaires-secondaires ("primary-backup process").

Dans le cas particulier d'applications transactionnelles, le fait qu'une transaction est "courte" par nature nous permet de la relancer (la rejouer) en cas de panne. Le point de reprise est alors simplement constitué des paramètres d'appel (d'exécution) de la transaction.

En un premier temps, nous présentons le recouvrement arrière appliqué à la reprise des programmes par la technique des blocs de reprise. En quelques mots, celle-ci consiste à structurer les programmes séquentiels sous forme d'actions atomiques emboîtées, et à placer des tests d'acceptabilité à la sortie de chacune de ces actions. Si les résultats d'une action atomique sont détectés erronés, alors une version alternative (une autre programmation) de cette action atomique est exécutée avec les mêmes valeurs d'entrée. Cette technique reprend les quatre stratégies de la continuité de service (détection, confinement, recouvrement et traitement de la faute), mais son principe de base est le recouvrement arrière. La section (2.6.3.1) y est entièrement consacrée.

En un deuxième temps (2.6.3.2), nous présentons le principe des processus primaires-secondaires. En quelques mots, ceux-ci sont deux "images" d'un même programme résidant sur deux processeurs indépendants. L'une de celles-ci est "l'image primaire" et est exécutée. C'est elle que l'utilisateur voit. L'autre, c'est "l'image secondaire". Elle ne sert que s'il y a défaillance de l'image primaire, auquel cas elle est exécutée à sa place. En fait, elle continue le programme à partir de l'endroit où l'image primaire est défaillante.

2.6.3.1 Les blocs de reprise

Chaque composant logiciel est ici conçu en plusieurs versions respectant la même spécification. Chacune de ces versions représente une programmation différente d'une même fonction.

Un bloc de reprise est composé de l'ensemble des versions d'un même composant et d'un test d'acceptation [SHIN84b]. Celui-ci est une expression booléenne dont l'évaluation ne produit pas d'effets de bord. Il a pour objectif de vérifier, à l'aide d'un test d'acceptabilité du type 'raisonnabilité', 'codification' ou 'minutage', que l'exécution de la version courante produit le résultat escompté. Le bloc de reprise constitue ainsi une 'unité de détection et de reprise'. Son

point d'entrée est un point de reprise.

La figure 2.7 représente un simple bloc de reprise avec ses deux versions alternatives. Lorsque ce bloc est exécuté, la première version est lancée et ses résultats sont vérifiés par le test d'acceptation. Si les résultats sont erronés et s'il reste une version alternative de ce bloc, alors cette version est exécutée avec les mêmes paramètres d'entrée que la première version. Les résultats de la version alternative sont testés à leur tour et, si nécessaire, une nouvelle version est exécutée jusqu'à ce que le test d'acceptation confirme les résultats obtenus, ou qu'il n'y ait plus de versions alternatives disponibles. Dans ce cas, l'entièreté du bloc de reprise a échoué. Si les résultats d'une version sont confirmés par le test, alors les versions restantes sont ignorées.

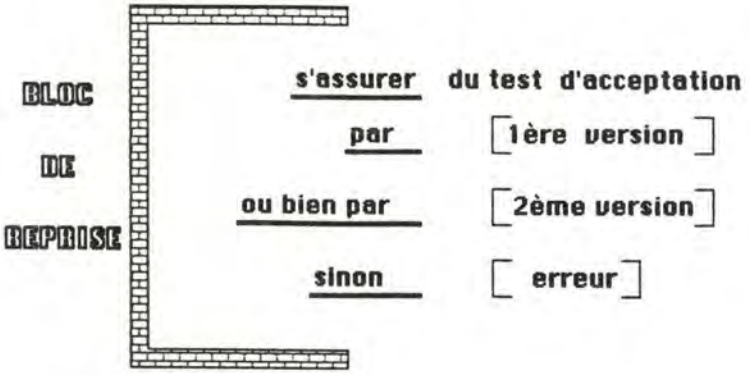

Figure 2.7: Exemple simple de bloc de reprise

La raison pour laquelle l'exécution de versions alternatives a de fortes chances de fournir des résultats corrects tient à l'objectif même de cette technique qui vise à tolérer les fautes intermittentes (transitoires) d'architecture logicielle. Rappelons que ces fautes n'apparaissent que dans des circonstances exceptionnelles. Il suffit, en général, de faire varier quelques caractéristiques de l'architecture ou de l'environnement logiciel du composant pour que l'exécution se passe à bien. Dans le cas des blocs de reprise, c'est l'architecture que l'on modifie à travers l'exécution des versions alternatives. Notons que ces versions sont même souvent programmées par différentes personnes afin de favoriser les différences d'architecture.

Considérons à présent un exemple plus compliqué, celui de la figure 2.8 repris de [RAND75]. Nous remarquons que chaque version a son code sous forme de blocs de reprise. L'exécution du bloc de reprise (A) est tout d'abord tentée à travers la 1ère version (A1). Celle-ci a son code sous la forme du bloc de reprise (B). A l'entrée dans celui-ci, toutes

les variables locales et globales de (A1) deviennent globales. La première version (B1) est lancée et ses résultats sont vérifiés par le test d'acceptation (BT). Si les résultats sont satisfaisants, alors le bloc de reprise (B) a réussi et est quitté. Sinon, les variables globales de (B), c'est-à-dire les variables constituant l'état du système lors de l'entrée dans (B), sont réinstanciées aux valeurs d'entrée dans (B) et la version alternative (B2) est exécutée. Si (B2) et (B3) échouent le test à leur tour, le bloc de reprise (B) a échoué en entier et, avec lui, échoue aussi (A1), la première version de (A). L'état du système est alors restauré plus en arrière encore dans le temps, aux valeurs d'entrée dans (A), et l'alternative (A2) est exécutée.

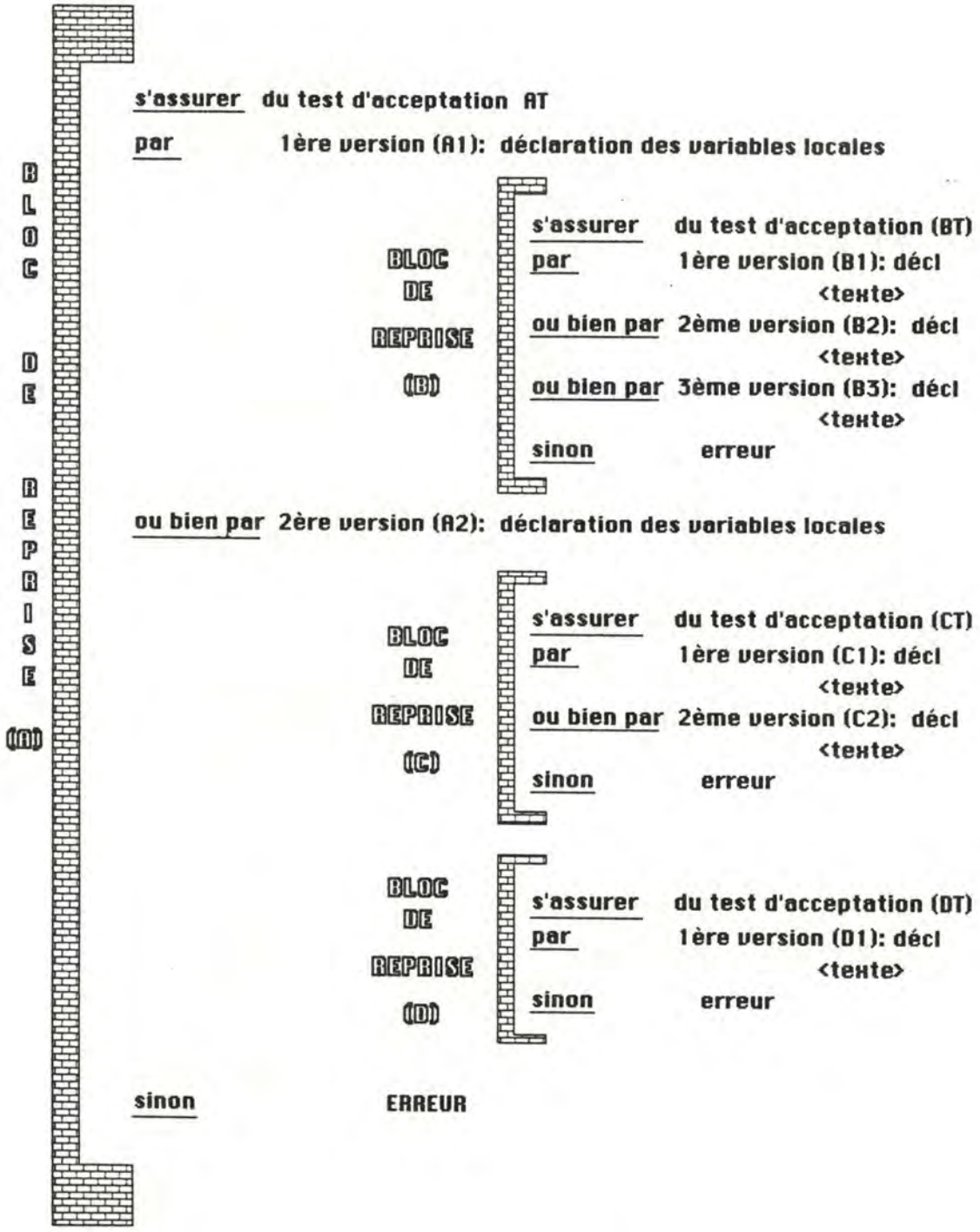

Figure 2.8: Exemple de blocs de reprise imbriqués

Le mécanisme de recouvrement arrière doit permettre "l'établissement", ou encore la sauvegarde, de plusieurs points de reprise. Il doit aussi donner la possibilité d'abandonner les informations relatives à un point de reprise

devenu inutile (lorsque le test d'acceptation a accepté les résultats d'une version, aucune reprise de ce bloc n'est plus envisageable). C'est ce mécanisme d'établissement et d'abandon des points de reprise que nous allons voir à présent.

Un point de reprise est actif entre le moment où il est établi et le moment où il est abandonné. La période pendant laquelle il est actif est appelée la zone de reprise. Lors de l'établissement d'un point de reprise, il n'est pas toujours nécessaire de sauvegarder l'entièreté de l'état du système, c'est-à-dire l'ensemble des variables globales au bloc de reprise entré. En effet, seulement certaines de ces valeurs globales seront modifiées et il est superflu de sauvegarder les autres. C'est pourquoi la méthode utilisée consiste à sauvegarder la valeur des variables globales au fur et à mesure qu'elles sont modifiées. Mieux encore, ces valeurs sont sauvées la première fois que la variable est modifiée après le dernier point de reprise établi. Le mécanisme réalisant la sauvegarde des valeurs utilise une "pile", appelée "cache" de reprise, dans laquelle chaque point de reprise actif dispose d'une zone de sauvetage. La figure 2.9 reprend l'exemple du bloc de recouvrement représenté sur la figure 2.8 et schématise les zones de reprise et de sauvetage associées. Lorsqu'un point de reprise est établi, le mécanisme prépare la zone de sauvetage dans la pile ("cache"). Chaque fois qu'un objet global est modifié, le mécanisme détermine si c'est sa première modification depuis l'établissement du point de reprise courant. Si c'est le cas, alors la zone de sauvetage ne contient pas encore de données de recouvrement pour cet objet. Une entrée y est alors créée et le nom ainsi que la valeur de l'objet avant modification y sont placés. En cas de non respect du test d'acceptation, les variables globales au bloc de reprise courant retrouvent leur valeur d'entrée aux valeurs sauvegardées dans la pile.

Le mécanisme s'occupant de la gestion de la "cache" de reprise doit être supporté par le matériel pour des raisons de fiabilité et d'efficacité. La gestion du contrôle de flux peut être supportée par l'interpréteur. Celui-ci prend alors ses tests d'interface tout autant en considération que les tests d'acceptation. Lorsqu'il détecte une erreur d'interface dans un bloc de reprise, il effectue le même branchement que si le résultat de ce bloc ne satisfaisait pas le test d'acceptation.

La technique des blocs de reprise regroupe, à elle seule, les quatre stratégies de support à la continuité de service:

- la détection des erreurs par le test d'acceptation et les tests d'interface fourni par l'interpréteur

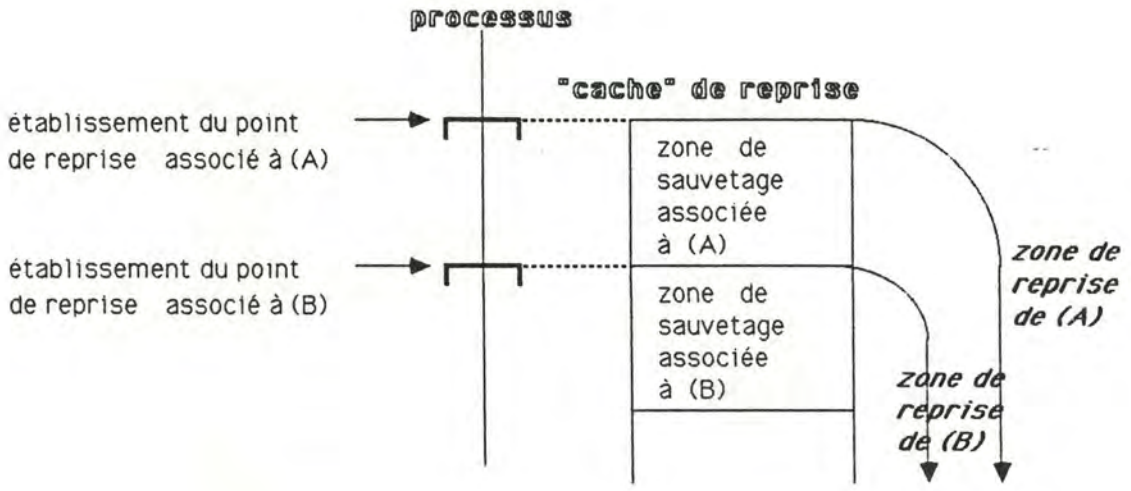

Figure 2.9: 'Cache' de reprise

- le confinement des erreurs par la structure atomique des différentes versions d'un bloc de reprise (une erreur apparaissant dans une version ne s'étend pas aux autres)
- le recouvrement (arrière), par la restauration de l'état en cas d'erreur dans une version
- le traitement de la faute par l'utilisation d'une version alternative pour fournir un résultat correct malgré l'apparition de la faute.

Il n'y a nullement besoin d'évaluation des dégâts puisque l'état du système est restauré, supprimant ainsi toutes les erreurs propagées dans la version précédente du bloc.

2.6.3.2 Processus primaires-secondaires

Une autre utilisation du recouvrement arrière est la technique des processus primaires-secondaires. Celle-ci permet de masquer une faute simple de matériel et de logiciel.

Elle exige la présence d'au moins deux processeurs disposant de leur propre Système d'Exploitation et reliés par un moyen de communication.

La figure 2.10 schématise son principe appliqué à un programme. Le programme a deux "images", une dans chaque processeur. Ces images sont des processus, respectivement le

processus primaire dans le premier processeur, et le processus de "backup" (secondaire) dans le deuxième processeur. Le processus primaire est exécuté mais pas le processus de "backup". Celui-ci est destiné à continuer l'exécution du programme si le processus primaire est défaillant. Pour en être capable, il doit "suivre" l'évolution du primaire en enregistrant régulièrement des informations sur l'état de ce dernier. Celles-ci forment un point de reprise ("checkpoint"). C'est un principe défini précédemment dans un contexte transactionnel, mais qui est applicable dans n'importe quel type de programme séquentiel. Le point de reprise est composé ici de l'ensemble des valeurs des variables du primaire à un moment donné (lors de la prise du "checkpoint"), ainsi que des informations (messages) que le primaire a eu de l'extérieur entre ce moment précis et sa défaillance. Ces dernières permettent au "backup" de se remettre dans l'état où était le primaire lors de la défaillance.

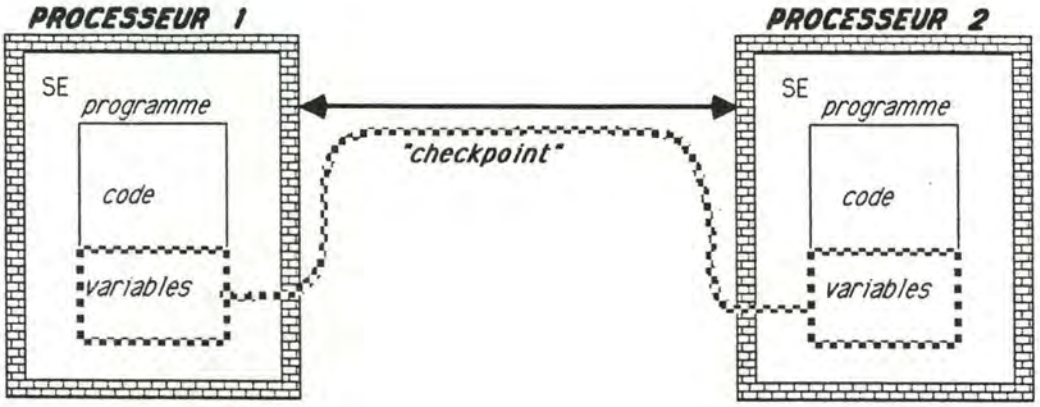

Figure 2.10: Processus primaires-secondaire

La prise du "checkpoint" peut varier d'un système à l'autre. Elle peut se faire soit manuellement, sur l'ordre explicite du programmeur (à l'aide d'une instruction), soit automatiquement, par exemple après un laps de temps précis ou bien avant toute opération sur la base de données.

Il est évident que des prises fréquentes de "checkpoint" consomment du temps de traitement et réduisent les performances du système quand tout va bien, mais améliorent le temps de reprise en cas de défaillance.

La technique du primaire-secondaire permet de tolérer les fautes de matériel et de logiciel apparaissant dans le processeur du primaire. Rappelons que les fautes de logiciel n'apparaissent que dans un environnement logiciel très particulier. Cette technique permet justement de changer d'environnement avec le deuxième processeur.

2.7 Traitement de la faute

2.7.1 Introduction

Les trois stratégies de support à la continuité de service discutées jusqu'ici visent la présence d'erreurs dans le système, mais ne prennent nullement en considération l'origine de celles-ci: la faute. Si cette faute est transitoire, elle ne se reproduira pas et peut alors être ignorée. Un simple réessai de l'opération fautive fournira les résultats espérés. Si la faute est permanente, un traitement à plus long terme est a envisager.

La quatrième stratégie, celle du traitement de la faute, regroupe les techniques visant à éliminer les fautes d'un système, de manière à lui permettre de reprendre le cours normal de ses opérations. Ces techniques sont regroupées en deux catégories correspondant aux deux phases constitutives de la stratégie: la localisation de la faute et la réparation du système. Les techniques décrites dans la section consacrée à la stratégie de détection des erreurs signalent la présence d'une faute, mais ne permettent ni de l'identifier ni de la localiser. Seulement, la réparation du système ne peut se faire sans cette condition. Ici encore, comme dans le cas de l'évaluation des dégâts, la structuration atomique du système nous est d'un grand secours. Elle permet, à elle seule, de localiser avec suffisamment de précision une bonne partie des fautes. Par exemple, dans la technique des blocs de reprise, la structure atomique des différentes versions nous garantit que la faute de conception se situe bien dans la version où l'erreur a été détectée. De même, dans un système matériel à vote, il y a de fortes chances pour que le composant fautif soit celui qui fournisse un résultat différent des autres. Il est ainsi possible d'éliminer la faute. Ce qui est fait, pour les blocs de reprise, en exécutant une version alternative, et pour le système à vote, en ignorant le résultat différent.

L'élimination dont il est question ici n'est que momentanée, dans le sens où la faute n'est que temporairement absente du système opérationnel et se reproduira très probablement dans des conditions analogues à celles de son apparition. La faute reste potentielle (définition d'une faute potentielle dans le chapitre 1 consacré aux concepts). Concrètement dans l'exemple des blocs de reprise, nous comprenons que la version fautive le reste puisque l'algorithme n'est pas corrigé. De même, dans le système matériel à vote, l'erreur peut être attribuée à une faute

transitoire et peut ainsi ne pas exiger la déconnexion brutale du composant fautif (c'est-à-dire le composant où elle est apparue). Rien ne nous assure pour autant que ce n'est pas une faute permanente qui réapparaîtra systématiquement à tous les votes.

Cette notion d' élimination momentanée d'une faute définit un type de traitement des fautes auquel il faut opposer le traitement par élimination permanente et le traitement par élimination définitive.

L' élimination permanente concerne principalement les fautes intermittentes et permanentes du matériel. Dans l'exemple du système à vote, il s'agirait de déconnecter, d'écarter le composant fautif, le vote s'effectuant sur les résultats de certains des composants restants. C'est un traitement durable de la faute qui nécessite auparavant une localisation précise de celle-ci. Généralement, la structuration atomique du système ne suffit pas et la localisation exige la mise en oeuvre de techniques particulières telles que celles décrites dans le point suivant (dédié à la localisation). L'élimination permanente des fautes de logiciel peut être réalisée par la technique des processus primaires-secondaires.

Le traitement permanent d'une faute permet d'assurer momentanément la continuité de service, mais ne suffit pas à longue échéance. En effet, pour pouvoir résister à l'apparition de nouvelles fautes, le système doit récupérer l'entièreté de ses capacités. Il est alors nécessaire d'effectuer un traitement définitif de la faute en réparant ou en remplaçant manuellement le composant écarté. De même, le traitement définitif des fautes de logiciel (traitées momentanément jusqu'ici) est réalisé par une modification manuelle du code des programmes.

Les deux sections suivantes sont respectivement consacrées à la "localisation" et à la "réparation" de la faute.

2.7.2 Localisation de la faute

Avant de pouvoir effectuer l'élimination permanente d'une faute, le système doit la localiser avec précision. Dans certains cas, la relation entre erreur et faute est évidente et les informations procurées par la détection d'une erreur suffisent pour localiser la faute. Dans d'autres cas, où cette relation est moins évidente, la localisation de la faute nécessite l'utilisation de techniques exploratoires similaires à celles déployées lors de l'évaluation des dégâts. La technique exploratoire la plus utilisée pour localiser les

fautes est le diagnostic (tel que décrit précédemment). Pour des raisons d'efficacité, le diagnostic est généralement réalisé par le matériel. Il peut toutefois être programmé (réalisé par logiciel) lorsqu'il doit être utilisé de manière très complète, par exemple dans le cadre d'une maintenance préventive "ON-LINE" ayant pour but de détecter des malfonctions avant qu'elles ne perturbent le système.

Les fautes intermittentes sont, par nature, difficilement localisables. Puisque les diagnostics ne sont d'aucun secours, le système peut toujours garder un compteur représentant le nombre de fois que le composant a été "concerné" par la détection d'une faute (plusieurs composants à la fois sont concernés par la détection d'une faute). Une fois qu'un compteur a dépassé un seuil défini, le composant correspondant est écarté du système afin de voir si la faute persiste.

La localisation d'une faute doit se faire avec une précision suffisante que pour pouvoir cerner le composant remplaçable le plus petit possible. Plus petit est le composant à identifier, plus difficile est sa localisation. Plus grand est-il, plus coûteuse est sa duplication et plus grande est la probabilité qu'il tombe en panne. Cette deuxième remarque, concernant la limitation du MTBF, oriente la conception des systèmes.

2.7.3 Réparation de la faute

Considérons, pour commencer, le cas du traitement momentané d'une faute. Selon la technique de recouvrement employée, il est plus ou moins facile de repérer la phase de traitement de la faute. En effet, il est aisé de distinguer la phase de traitement suivant un recouvrement arrière, alors qu'il est difficile de le faire pour un recouvrement avant. Prenons, par exemple, le cas des blocs de reprise. La restauration de l'état du système y constitue la phase de recouvrement, et l'exécution d'une version alternative y constitue la phase de traitement. Dans le cas des systèmes à vote, par contre, la phase de recouvrement (avant) est indissociable de celle de traitement.

Revenons à l'exemple des blocs de reprise et concentrons nous sur la phase de traitement de la faute. Nous pouvons dire que la version alternative "remplace" la version initiale. Elle représente un composant, respectant les mêmes spécifications, et resté en attente inactive pour l'éventualité d'un remplacement. Ce composant, appelé pour cela "standby-spare", a ici une architecture différente de l'original puisque l'objectif visé est de tolérer les fautes résiduelles de conception.

Le principe du "remplacement", que nous venons de voir dans le cadre du traitement momentané des fautes de logiciel, est largement utilisé pour le traitement permanent des fautes du matériel. Ici, par contre, le composant de rechange ("standby-spare") remplaçant le composant fautif a une architecture identique à celui-ci, puisque l'objectif est de tolérer les fautes de vieillissement et d'environnement. L'opération de remplacement est réalisée au moyen de commutateurs ("switchs"). Un "switch", comme le montre la figure 2.11, est un composant matériel ayant une interface avec deux composants identiques (utilisés en redondance statique) et le reste du système. L'effet de la commutation ("switching") est le même que si le composant fautif était remplacé.

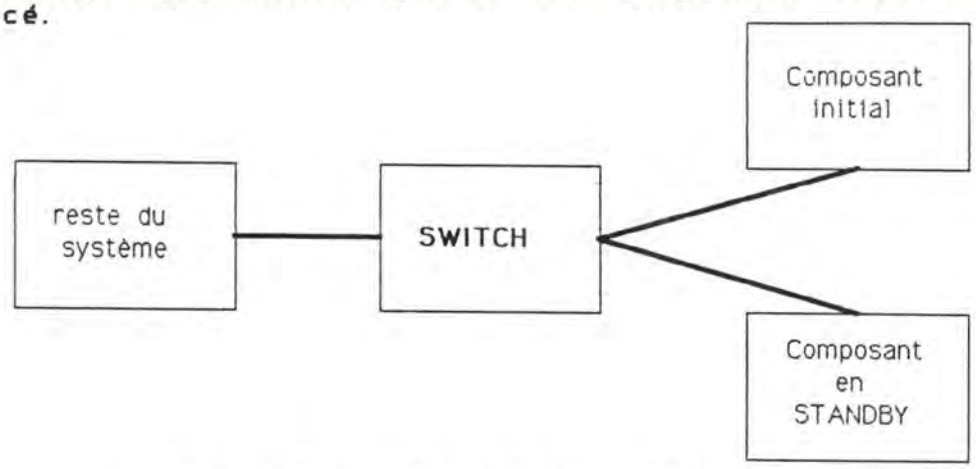

Figure 2.11: Commutateur (Switch)

Le remplacement se base sur une localisation précise du composant fautif, et est exécuté en parallèle avec une reconfiguration du système. L'opération de reconfiguration consiste en une modification automatique des tables de configuration du matériel que le SE consulte. Parfois, le traitement d'une faute n'est constitué que d'une reconfiguration, sans remplacement du composant fautif. Dans ce cas, les fonctions du composant écarté sont généralement prises en charge par des composants semblables mais déjà opérationnels. Le système subit alors une perte de performance. Cette caractéristique de répartition de la charge d'un composant fautif sur des composants semblables s'adresse surtout aux CPUs et aux programmes qui s'y trouvent. Elle va de pair avec la propriété de croissance modulaire qui permet aux systèmes en bénéficiant de faire croître leur puissance en terme de traitements, de capacité de stockage et de capacité de raccordement, sans remplacement de matériel, mais simplement par ajout d'éléments semblables.

Le traitement définitif est réalisé par la réparation ou le remplacement manuel du composant fautif lors d'une opération de maintenance. Cette réparation exige une réintégration du nouveau composant dans le système.

L'entièreté de l'opération, c'est-à-dire la réparation (remplacement) et la réintégration doit se faire "ON-LINE" pour respecter les objectifs de continuité de service. Le traitement définitif des fautes de logiciel est souvent effectué sur base d'informations représentant le contexte du programme fautif juste avant le recouvrement.

Comme nous l'avons vu dans la section consacrée à la localisation des fautes, les concepteurs de systèmes donnent aux composants des tailles suffisamment grandes que pour permettre une localisation aisée des fautes. Un autre facteur influençant la taille des composants est la facilité avec laquelle ils sont manipulés, remplacés.

2.8 Récapitulation

Pour terminer ce chapitre consacré aux techniques de support à la tolérance aux pannes, nous envisageons divers scénarios relatifs à des problèmes et des fautes déjà abordés, et reprenant les techniques décrites ci-dessus.

- Traitement d'une faute transitoire du matériel dûe à l'environnement.

  scénario 1:

  une perturbation électro-magnétique provoque l'apparition d'une erreur simple dans un mot mémoire.

  L'erreur est détectée et corrigée par le code auto-correcteur. Comme nous l'avons vu, c'est un cas de recouvrement avant.

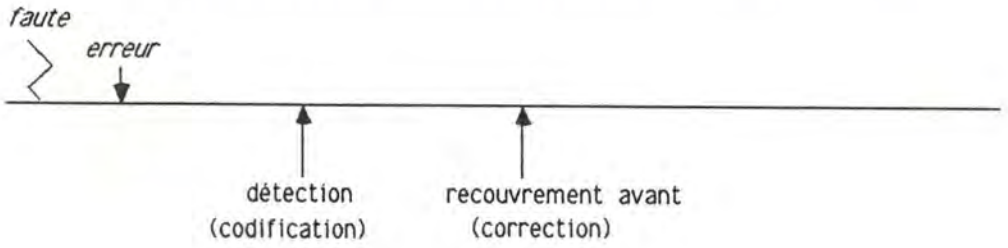

Figure 2.12: scénario 1

scénario 2:

une perturbation électro-magnétique provoque l'apparition d'erreurs alors que trois processeurs disposés en système à vote exécutent une même opération élémentaire.

La détection est la constatation qu'un processeur produit un résultat différent des autres, et le traitement momentané (recouvrement avant) est le fait que le système ignore ce résultat. L'erreur est confinée au seul composant fautif du système grâce à une isolation physique de chacun des composants.

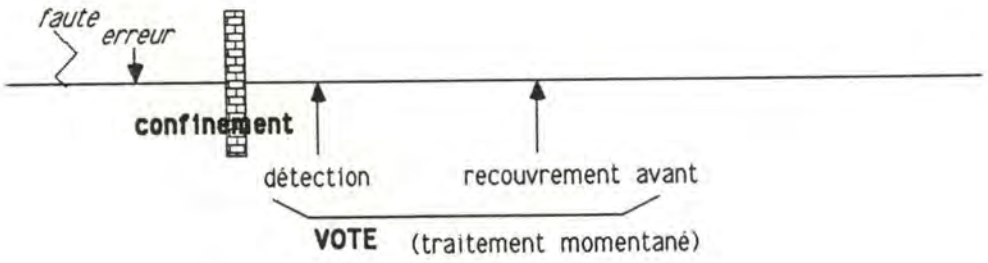

Figure 2.13: scénario 2

- Traitement d'une faute intermittente du matériel (une faute de conception)

  scénario 3:

  sous une forte charge, un contrôleur d'E/S se sature et ne répond plus dans les délais spécifiés.

  Un test de minutage signale que le contrôleur aurait déjà dû répondre. La requête est relancée après quelques secondes (ré-essai). Pour cela, le système doit récupérer les paramètres du service demandé et faire ainsi un recouvrement arrière. Ce recouvrement et le ré-essai qui le suit constitue un traitement momentané de la faute.

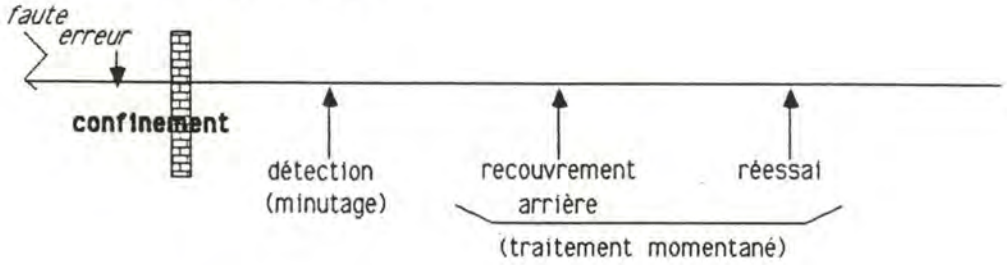

Figure 2.14: scénario 3

- Traitement d'une faute permanente du matériel dûe au vieillissement

  scénario 4:

  supposons qu'au terme du scénario précédent le contrôleur d'E/S n'ait toujours pas répondu. Le système peut alors le considérer comme définitivement hors

d'usage.

Le contrôleur est remplacé par un autre connecté aux mêmes disques et resté en "standby". Le remplacement est précédé respectivement par la détection de l'anomalie dans l'opération d'E/S, et par une localisation du contrôleur d'E/S comme élément défaillant. Le traitement permanent de la faute comprend le remplacement du contrôleur et la reconfiguration du système afin que celui-ci prenne en considération le nouveau composant. Un ré-essai clôture les actions effectuées pour assurer la continuité de service.

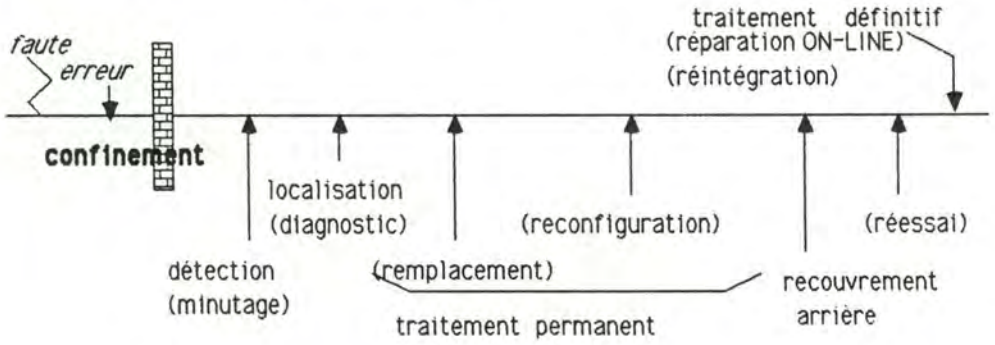

Figure 2.15: scénario 4

Lors de la première maintenance du système, les techniciens traitent définitivement la faute en remplaçant manuellement le composant écarté et en le réintégrantensuite dans le système.

- Traitement d'une faute intermittente du logiciel

  scénario 5:

  considérons un composant construit par la technique des blocs de reprise.

  Le confinement est réalisé par la structure atomique des différentes versions. Le test d'acceptation détecte une erreur dans le résultat de la première version du bloc et l'état du programme est restauré aux valeurs d'entrée dans celui-ci. Le recouvrement arrière vient ainsi d'être effectué. Le remplacement et le ré-essai (traitement momentané) sont réalisés ensuite à travers l'exécution de la version suivante.

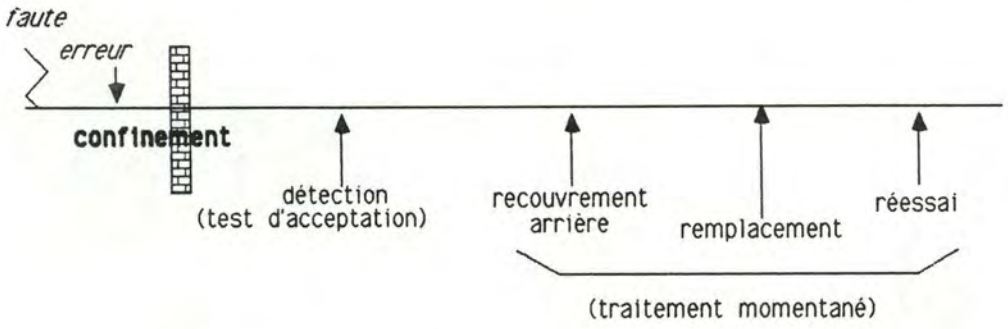

Figure 2.16: scénario 5

Chapitre 3

Implémentation

de la Tolérance aux Pannes

dans le Logiciel

3 Implémentation de la tolérance aux pannes dans le logiciel

Jusqu'à présent, nous avons vu toute une série de techniques permettant la réalisation de systèmes tolérants aux pannes. Leur implémentation sous forme de redondance accroît la complexité des systèmes. Seulement, c'est justement cette complexité qui est la principale source de fautes. Il est donc primordial d'avoir une approche structurée de l'implémentation des moyens de tolérance aux fautes. C'est de cette approche pour les systèmes logiciels dont il est question dans ce chapitre [ANDE79].

La complexité des systèmes provient principalement des interactions entre composants. Il est ainsi nécessaire que le système soit construit avec une structure telle que ces interactions soient bien maîtrisées [LIES86], [SIEW82].

L'élément principal de la majorité des méthodologies de construction de systèmes est la décomposition modulaire et la structure hiérarchique. Celles-ci font l'objet de la première section de ce chapitre.

De plus, puisqu'il est particulièrement nécessaire de maîtriser la complexité issue des intéractions résultant de l'activité de tolérance aux fautes, il faut pouvoir clairement distinguer l'activité habituelle et inhabituelle (activité de tolérance aux fautes) du système. C'est ce qui est fait à travers le modèle d'implémentation d'un composant logiciel idéal décrit dans la deuxième section de ce chapitre.

3.1 Décomposition Modulaire et Structure Hiérarchique

Le principe de base d'une méthodologie en matière d'élaboration de logiciels est de donner des moyens pour faciliter la construction de programmes corrects, clairs, efficaces et dont la structure garantit certaines qualités du logiciel comme la fiabilité, la maintenabilité, la réutilisabilité, la portabilité et la convivialité.

L'approche décrite ci-dessous, à savoir la structuration hiérarchique et la décomposition modulaire, est communément admise par l'ensemble de la communauté informatique [TRAI82], [TRUE83], [WIED83], [WULF75].

Les références à ce sujet ne manquent pas. Nous nous sommes inspirés de [ANDE78], [CAMP86], [LAMS85], [RAND75], et [RAND78].

De façon générale, nous pouvons dire que structurer un système de façon hiérarchique, c'est l'organiser en niveaux différents et ordonnés.

Nous pouvons donner une définition inductive [LAMS85]:

une structure est hiérarchisée si et seulement si il existe une relation R entre ses composants qui permet de définir des niveaux tel que

1) le niveau 0 soit l'ensemble des composants A tel qu'il n'existe aucun composant B vérifiant la relation R(A,B)

2) un niveau i, soit l'ensemble des composants A tel qu'il existe un ou plusieurs composants B de niveau i-1 vérifiant la relation R(A,B), et tel que si une relation R(B,C) existe, cela implique que C soit de niveau i-1 ou de niveau inférieur.

Il existe autant de hiérarchies qu'il y a de relations R vérifiant les conditions ci-dessus.

Une bonne hiérarchie dépend du choix de R. La hiérarchie "utilise", définie par la même définition où tous les R sont remplacés par UTILISE, présente les qualités d'une bonne hiérarchie. Nous disons que "A utilise B" si et seulement si le fonctionnement correct de A dépend de la disponibilité d'une version correcte du composant B.

Nous trouvons trois intérêts fondamentaux pour employer la hiérarchie "utilise":

1. les composants sont beaucoup plus faciles à spécifier du fait qu'on élimine les redondances fonctionnelles

2. la conception est rendue plus aisée par les niveaux d'abstraction. En effet, quand on conçoit A, on ne doit pas savoir comment B fonctionne, mais on doit uniquement connaître les spécifications de B. Quand on conçoit B, on ne doit rien savoir de A (ni fonctions, ni spécifications)

3. l'emploi d'une telle hiérarchie permet de confiner sur un niveau des modifications éventuelles à effectuer par après, ce qui facilite la maintenance.

Remarquons que la découpe modulaire permet d'accroître encore cette localisation, ce confinement.

L'exemple le plus cité dans la littérature d'un gros logiciel construit sur les bases d'une hiérarchie "utilise" est le système d'exploitation "THE".

Nous décrivons brièvement les cinq niveaux de ce système d'exploitation.

Niveau 0. Allocation du processeur à des processus de niveaux supérieurs.

-> abstraction car au dessus de ce niveau, pas de notion de processeur physique

Niveau 1. Allocation de la mémoire virtuelle.

-> abstraction car aux niveaux supérieurs, pas de notion de page physique, mais les informations sont identifiées sous forme de segments

Niveau 2. Interpréteur de messages.

-> abstraction car pas de notion de console physique aux niveaux supérieurs

Niveau 3. Processus séquentiels associés aux Entrées/Sorties.

-> abstraction car, aux niveaux supérieurs, les périphériques utilisés n'ont plus de signification physique

Niveau 4. Programmes utilisateurs.

Cette structure obtenue par la hiérarchie "utilise" nous donne un premier squelette de notre architecture logicielle. Nous devons la compléter, l'enrichir en identifiant les différents modules (ou composants) de chaque niveau, les relations qui existent entre eux et leurs interfaces.

Chaque module doit avoir une forte capacité à cacher de l'information, une forte cohésion interne et un faible degré de couplage.

Le résultat final est un graphe, où à chaque noeud correspond un module et à chaque arc étiqueté correspond une relation

3.2 Modèle du composant logiciel idéal

Nous nous sommes donnés jusqu'à présent une méthode d'identification des composants (modules) du logiciel, mais nous ne nous sommes pas souciés de l'activité de tolérance aux fautes. Celle-ci doit être introduite méthodologiquement dans chaque composant afin de garder les qualités acquises par l'identification de ceux-ci.

Le modèle de composant logiciel idéal dessiné sur la figure 3.1 et décrit ci-dessous est repris de [ANDE81].

Ce modèle marque une distinction entre l'activité "habituelle" et "inhabituelle" d'un composant logiciel. L'activité habituelle est celle que le composant a quand le système ne tolère pas les fautes.

L'activité inhabituelle est celle qui lui permet de traiter les fautes. Son objectif est de remettre le système dans un état permettant la reprise des activités habituelles. Ainsi, les techniques de recouvrement, de diagnostic et de traitement des fautes s'y retrouvent.

Afin de maintenir une séparation claire entre activité habituelle et inhabituelle, le modèle est basé sur la notion d'exception [CRIS82]. Celle-ci peut intuitivement être comprise comme "un événement rare" ou encore "un événement nécessitant un traitement particulier".

Lorsqu'un composant reçoit une requête d'un autre composant, il effectue le service demandé et renvoie une réponse. Si la requête contient des paramètres erronés, le composant appelé s'en rend compte au moyen de tests d'interface et il signale une exception d'interface au composant appelant. D'autre part, si la requête est correcte, mais que le composant détecte une faute pendant ou après son activité habituelle, alors deux cas se présentent:

- soit l'activité inhabituelle du composant dispose de moyens pour traiter la faute. Elle peut alors la cacher au composant appelant.

- soit le composant ne peut traiter la faute à partir de son activité inhabituelle, et il signale une exception de défaillance au composant appelant.

De manière similaire, lorsqu'un composant reçoit un signal d'exception en réponse à une requête ou bien détecte une erreur dans son activité habituelle, il relève une exception et passe la main à son activité inhabituelle afin d'essayer de traiter la faute.

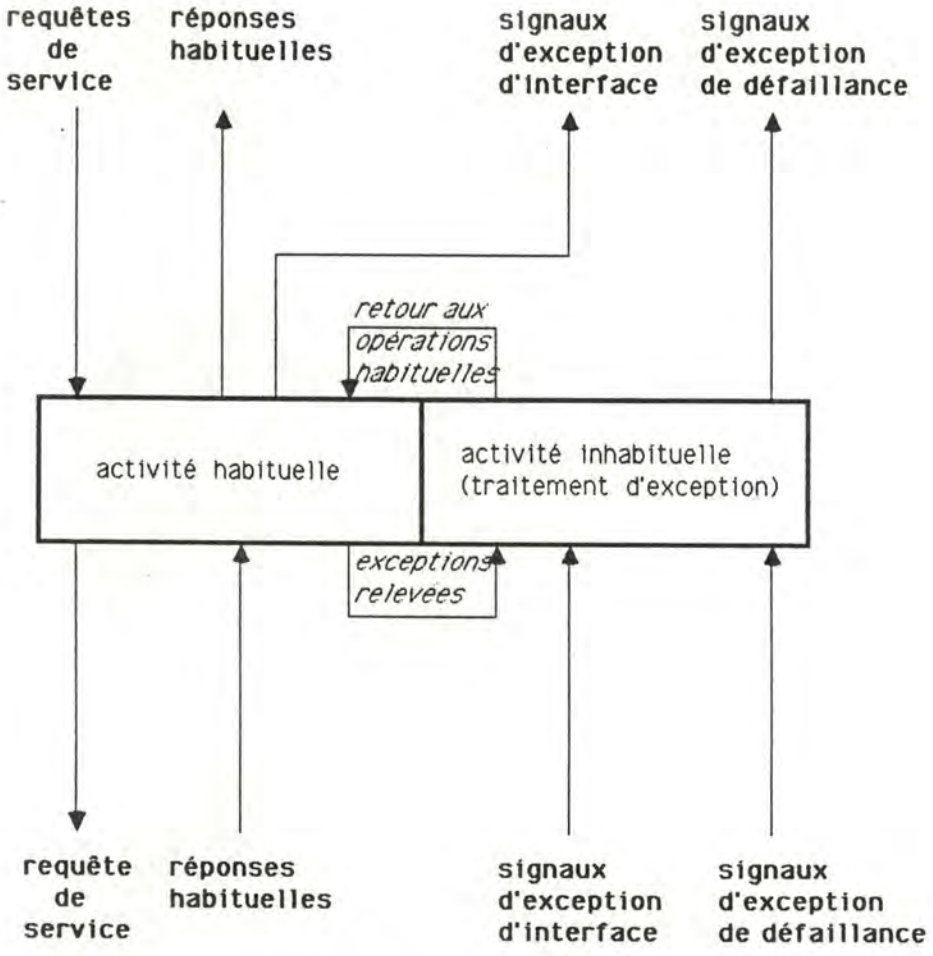

Figure 3.1: Modèle du composant logiciel idéal

Pour chaque type d'exception, l'activité inhabituelle d'un composant dispose d'un gérant d'exceptions particulier. L'interpréteur (1.1.2) supporte, au moyen d'interruptions, un mécanisme permettant d'invoquer automatiquement les gérants lors de l'apparition des exceptions correspondantes. Ces exceptions sont, d'une part détectées par l'interface de l'interpréteur (exemple: division par 0) et, d'autre part, relevées par l'activité habituelle du composant (cf. ci-dessus).

Le principe du traitement des fautes sous-jacent à ce modèle est le recouvrement avant (puisque le traitement de la faute dispose d'informations précises à propos de l'erreur sous forme du type d'exception relevée) mais, comme nous le verrons ci-dessous, ce modèle peut être utilisé avec un recouvrement arrière.

L'association "exception-gérant particulier" peut être spécifiée dynamiquement dans les programmes (composants) au moyen d'instructions spéciales. Par exemple:

- "enable <x:G>" pour que le gérant "G" soit exécuté en réponse à une exception de type "x", (ou encore pour faire "l'association x->G")

- "disable <x:G>" pour défaire cette association

L'association "exception-gérant" se matérialise par un contexte que l'interpréteur sauve sur une pile.

La figure 3.2 illustre un exemple d'apparition d'une exception "x", de son traitement par le gérant associé, et du retour aux opérations habituelles. Le composant logiciel effectue un "enable" du gérant "G" (l'association est représentée par une grande parenthèse ouvrante), puis relève une exception. Le gérant est alors déclenché et il traite la faute. A la fin de celui-ci, l'activité habituelle reprend et "disable" le gérant avant de retourner ses résultats au composant appelant.

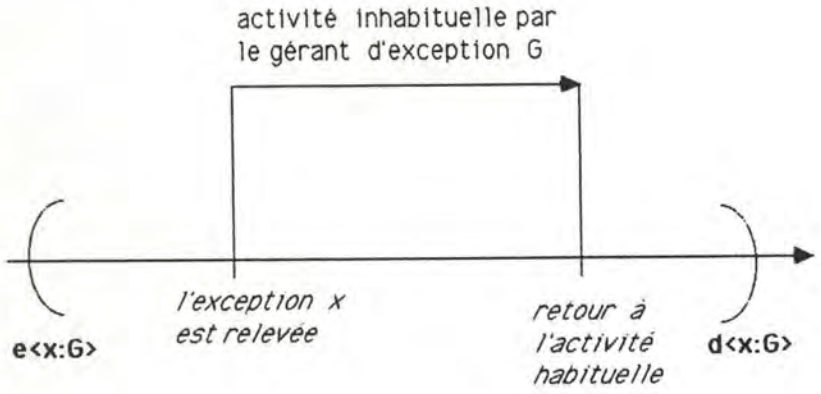

Figure 3.2: Traitement d'une faute - premier scénario

Si un gérant ne parvient pas à traiter l'exception pour laquelle il a été appelé ou, tout simplement, s'il n'y a pas de gérant dédié au type d'exception relevée, alors l'activité inhabituelle du composant courant (appelé) signale une exception de défaillance au composant appelant. L'exemple suivant (figure 3.3) reprend un scénario où un composant C1 "enable" le gérant G1, puis adresse une requête au composant C2. Celui-ci "enable" à son tour le gérant G2. Une exception est alors relevée par l'activité habituelle de C2. Le gérant G2 tente de la masquer, mais n'y arrive pas. Il signale une exception de défaillance à C1. Celui-ci exécute son gérant G1 et masque la faute. L'activité habituelle reprend ensuite son cours normal.

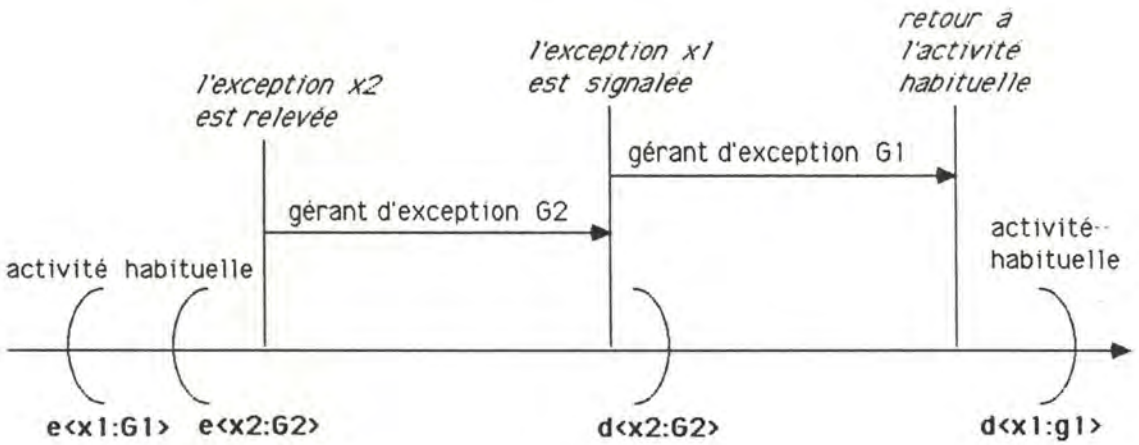

Figure 3.3: Traitement d'une faute - deuxième scénario

Les gérants d'exception sont aussi des composants et peuvent disposer de leurs propres gérants d'exception.

Le mécanisme d'exception que nous venons de décrire peut servir de support à l'implémentation d'une technique de recouvrement arrière. Nous le montrons ci-dessous au moyen des "blocs de reprise" décris au point 2.7.3. Cet exemple est repris de [CAMP86].

La figure 3.4 montre un bloc de reprise avec ses trois versions alternatives. A l'entrée de la première version ("bloc primaire"), l'état des variables globales est sauvé ("initialise-cache"). L'opération "enable" indique à l'interpréteur qu'il doit effectuer un branchement sur le bloc "alternate-1" (qui devient ainsi le gérant d'exception courant) en cas d'exception détectée par l'interpréteur (div par 0) ou relevée par le programme. Ensuite, la première version de l'algorithme est exécutée, suivie du test d'acceptation. Si celui-ci vérifie le résultat, alors l'association exception-gérant précédente est "disable" et les valeurs d'entrée des variables sont effacées ("discard-cache"). Dans le cas où le test d'acceptation échoue, une exception est relevée au moyen de l'instruction "signal" (les opérations "signaler" et "relever" se traduisent par l'instruction "signal" d'appel à l'interpréteur.

Rappelons que c'est l'interpréteur qui supporte le mécanisme d'exception, c'est donc lui qui sait quels sont les gérants et comment les invoquer).

L'interpréteur donne le contrôle à la deuxième version du bloc puisque celle-ci est aussi le gérant d'exeption courant depuis le "enable" précédent. La première opération consiste à restaurer l'état des variables globales ("restore cache"), puis associe aux exceptions un nouveau gérant "alternate-2" qui sera invoqué dans les mêmes circonstances que décrites

ci-dessus. Le dernier gérant signale, au moyen d'une exception de défaillance, que l'exception initiale n'a pu être traitée par les gérants successifs.

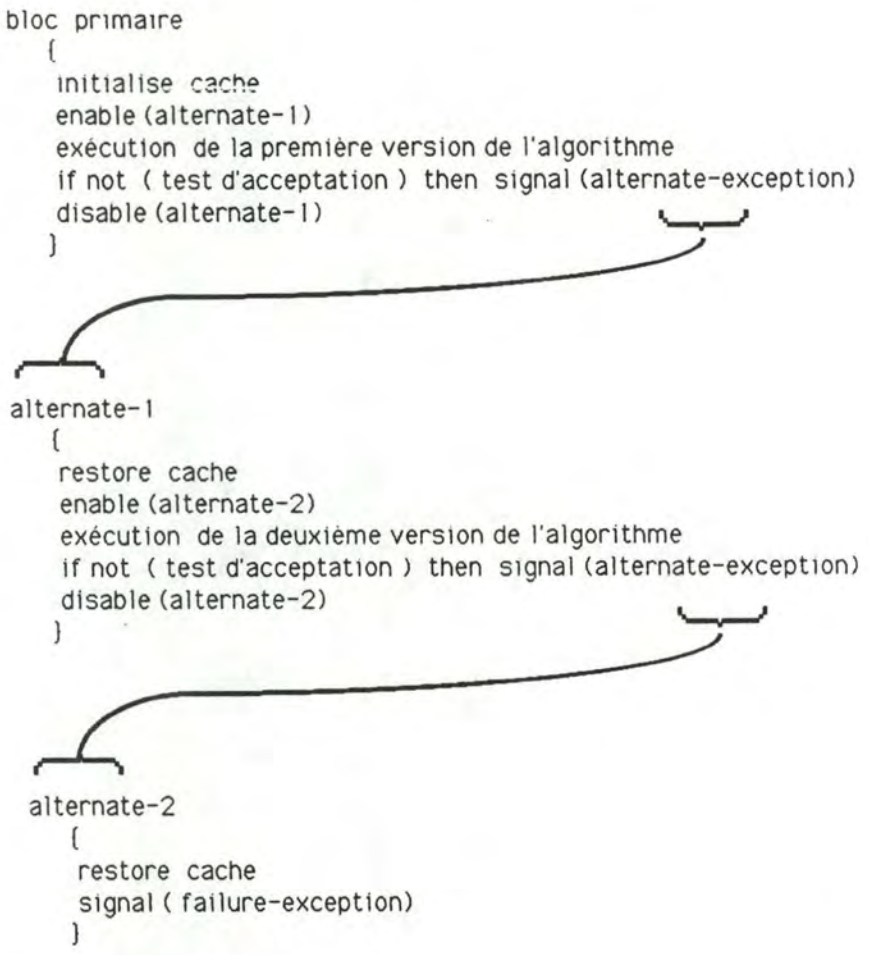

Figure 3.4: Implémentation de la technique des blocs de reprise

Chapitre 4

Les Techniques de Reprise B.D.

4 Les techniques de reprise

4.1 Introduction

Comme nous l'avons dit dans le premier chapitre, le concept de reprise reste nécessaire dans tout système, même dans le plus tolérant aux fautes, car une défaillance peut toujours survenir. Les techniques de reprise servent donc à restaurer les données d'un système dans un état "utilisable", après qu'une défaillance ait pu les altérer.

Les différents types de défaillance entraînent différents types de reprise [CARD83]. Cependant, toutes les défaillances possibles ne peuvent être prévues [RAND78], ce qui peut provoquer une catastrophe. De plus, les données redondantes sur lesquelles se basent ces techniques peuvent être corrompues. Les cas d'écrasement de tête sur disque ou d'erreur dans le logiciel de gestion des écritures sur disque en sont des exemples classiques. Ces données pourraient elles-mêmes être protégées par des techniques de reprise et procéder ainsi une infinité de fois. En pratique, bien sûr, nous ne pouvons mettre en oeuvre de tels systèmes. Les risques, bien que limités, restent donc présents.

Nous devons distinguer deux types de bases de données, celles qui supportent l'approche transactionnelle et les autres.

Dans la première classe, la transaction est l'unité centrale de la cohérence de la base de données. Toutes les techniques tournent autour du concept de transaction [BRAE85]. La littérature parle de "schémas de reprise orientés transaction". Cette approche est la plus répandue aujourd'hui. De plus en plus de systèmes de gestion de base de données se basent sur celles-ci.

La deuxième catégorie reprend en fait tous les autres systèmes de gestion de données, bien souvent plus anciens. De manière générale, nous pouvons dire que cette classe reprend les systèmes de gestion de fichiers qui ont été et sont encore très utilisés pour le traitement par lot.

En fait, cette distinction peut se faire en terme de systèmes [CURT86]. La première classe concerne les systèmes partagés en temps réel , où la notion de transaction permet de gérer les accès concurrentiels à la base de données. La seconde concerne les systèmes en temps différé , où le

traitement par lot élimine les problèmes de concurrence.

Signalons dès à présent que si les systèmes sont fondamentalement différents, un certain nombre de techniques de reprise des systèmes de traitement par lot ont été adoptées par les systèmes transactionnels (parfois avec certaines améliorations).

Nous allons donc dans un premier temps décrire chacune des techniques de reprise pour les systèmes de traitement par lot. Ensuite, nous présentons les techniques de reprise pour les systèmes transactionnels. Nous donnons une typologie des différentes solutions apportées.

4.2 Techniques de reprise pour les systèmes de traitement par lot

Une première approche possible, celle de Gray [GRAY79], consiste à ranger les techniques de reprise, appelées "Recovery Procedures" par l'auteur, en fonction des différentes sortes de défaillances. Suivant l'étendue et la cause d'une défaillance, nous obtenons trois types de défaillances:

- une défaillance d'un programme ou d'une transaction
- une défaillance de tout le système
- une défaillance matérielle.

Nous choisissons une autre approche, celle de Verhofstad [VERH78], qui fait la distinction entre deux différents types de données:

- les données qui sont les valeurs courantes
- les données de reprise permettant la restauration de valeurs précédentes.

Nous allons décrire comment ces données courantes et ces données de reprise peuvent être organisées et manipulées pour permettre une reprise.

Dans un premier temps, nous décrivons les techniques de reprise (4.2.1). Nous citons ensuite les différents types de reprise possibles (4.2.2), à savoir les états dans lesquels la base de données est remise. Puis nous analysons les différentes techniques par rapport à ces états de la base de données (4.2.3).

4.2.1 Les différentes techniques de reprise

1. Programme de sauvetage ("Salvation Program").

Un programme de sauvetage est activé après une panne pour restaurer le système dans un état valide. Il n'emploie pas de données (redondantes) de reprise. Ce programme est utilisé si les autres techniques de reprise ont echoué, si elles ne sont pas employées, ou si aucune résistance aux pannes n'a été prévue. Il sauve les

informations qui sont encore reconnaissables.

2. Copie incrémentale ("Incremental Dumping").

La copie incrémentale implique la copie des fichiers mis à jour sur une mémoire d'archivage (disque ou bande) après qu'un travail ait fini sa tâche ou à intervalles réguliers. Il crée des points de reprise pour les fichiers mis à jour. Les copies de sauvetage des fichiers peuvent être restaurées après une panne pour remettre tous les fichiers dans leur dernier état cohérent. Cependant, toutes les mises à jour effectuées par des travaux tournant au moment de la panne peuvent ne pas être restaurées complètement si certains fichiers actifs à ce moment n'ont pas été recopiés à temps.

3. Fichier journal ("Audit Trail").

Un journal enregistre sur un fichier appelé "Audit Trail" la suite d'opérations effectuées (lecture, écriture, ...). Ce journal contient en plus les heures et dates auxquelles ces opérations ont eu lieu, et les codes d'identification de l'utilisateur (ou des programmes utilisateurs) les ayant effectuées.

Les journaux peuvent être employés à différents usages:

* Reprise de panne (REFAIRE généralisé).

Lorsque les versions de sauvegarde des fichiers sont réinstallées, le journal peut être employé pour effectuer des opérations sur ceux-ci, et ainsi restaurer l'état de la base de données au moment de la panne [CURT77].

* Annulation de certaines opérations (DEFAIRE des opérations, "Global Undo" , "Backing out").

Lorsqu'un système s'effondre sans endommager la mémoire secondaire, les fichiers modifiés par des processus au moment de la défaillance peuvent être restaurés dans les états où ils se trouvaient avant le commencement des processus. Le journal peut être parcouru en arrière. Ainsi, une tâche peut être défaite en cas d'interblocage ("deadlock") ou d'échec. Les données affectées par cette tâche peuvent être restaurées dans l'état où elles se trouvaient avant le commencement de la tâche.

* Certifier l'intégrité du système.

La journalisation permet de vérifier que certaines règles sont suivies (Contraintes d'intégrité). Cette technique s'applique

particulièrement bien comme outil d'intégrité pour les systèmes de base de données en temps réel, où les données sont partagées entre plusieurs utilisateurs (problème de la concurrence). Nous reparlerons de cette technique lors de l'analyse des techniques de reprise pour les bases de données orientées transaction.

Gray [GRAY79] donne une description complète de l'emploi des journaux pour les reprises. Un des apports majeurs de cette étude réside dans la solution apportée à la perte des tampons en cas de panne. En effet, pour des raisons de performance, des tampons concervent quelques pages de la base de données en mémoire centrale afin d'éviter trop d'entrées/sorties. Le moindre incident risque d'effacer la mémoire centrale et de faire perdre les données modifiées de la base de données se trouvant encore dans les tampons. La solution proposée se résume en un protocole d'écriture dans le journal avant de répertorier la modification dans la base de données ("Write Ahead Protocol"). Ainsi, aucune information ne se perd en cours de route.

Un problème, mis en évidence par Randell [RAND75] et Curtice [CURT77], doit être solutionné. Un journal peut permettre de "DEFAIRE" un processus qui peut avoir des interactions avec un second de telle façon que celui-ci doit également être "défait" et ainsi de suite. Cet enchaînement, appelé " l'effet domino " dans la littérature, peut être résolu en rendant les interactions entre processus impossibles. Une technique maintenant traditionnelle consiste à employer des verrous afin de sérialiser certains processus concurrentiels. Nous l'avons détaillée dans le premier chapitre. Supposons ici qu'un point de reprise est enregistré quand aucun processus utilisateur n'est actif, ce qui rend impossible un effet domino au-delà de ce point.

4. Les fichiers différentiels ("Differential Files").

Nous observons que, dans une base de données, les pages accédées par une simple consultation ont peu d'occasions d'être détruites. Les problèmes viennent lors de la mise à jour de pages modifiées [HAIN87]. L'idée de base de la technique des fichiers différentiels est de faire la distinction entre les deux types d'opérations sur la base de données.

Nous avons au départ un fichier de base qui peut être consulté sans problème. Si des mises à jour sont effectuées, nous les enregistrons dans un fichier de différences. En cas d'incident, au pire, nous perdons le fichier de différences, mais le fichier de base reste intact et cohérent. Nous devons donc protéger par d'autres techniques (journalisation) le fichier de

différences.

Au début, ce fichier reste petit, ce qui facilite la mise en oeuvre de protections, par exemple une copie intégrale toutes les demi-heures. S'il devient trop important, nous pouvons le fusionner avec le fichier de base ce qui a pour effet de le vider. Cette fusion peut s'effectuer en batch, mais aussi en parallèle avec la charge normale du système car nous avons un fichier de différences qui fonctionne en extension (chaque modification s'écrit en fin de fichier).

Pour la consultation d'une page, nous regardons d'abord dans le fichier de différences. Si nous ne la trouvons pas, alors elle se trouve dans le fichier de base.

Nous pouvons ainsi calculer le coût en accès physiques. Prenons P la proportion de pages modifiées, nous obtenons comme coût:

```
1 +  P    * 0  = 1        si la page a été modifiée,
                          elle se trouve dans le fichier
                          de différences
1 +  1-P  * 1  = 2 - P    sinon.
```

Donc, pour un petit P, nous doublons presque le nombre d'accès. Nous ne pouvons tolérer une telle perte de performance.

Différentes solutions ont été apportées pour résoudre ce problème.

La plus répandue reste l'emploi d'un prédicteur ou fonction qui nous aide à déterminer où se trouve une page. Un prédicteur exact nous renseigne à coup sûr la localition d'une page. Prenons une table booléenne de taille égale au nombre de pages. Si la page se trouve dans le fichier de différences, le bit correspondant est mis à "1". Pour de grandes bases de données, nous ne pouvons prendre une telle table. Nous employons alors un prédicteur non déterministe. Ce type de prédicteur exécute une fonction de projection de l'adresse de la page sur une table booléenne de taille inférieure au nombre de pages, ce qui amène des risques de collision. Voici comment se présente ce type de prédicteur:

Soit la fonction "F" : un prédicteur non déterministe.
Soit une structure "B" : un tableau de bits.
Nous avons F(R) = i, avec "R" : numéro de page.

```
==> si B[i] = 0 , nous pouvons affirmer que la page R
                  n' appartient pas au fichier de
                  différences
            = 1 , nous ne pouvons rien dire.
```

La table B se modifie comme suit :

En cas de modification de la page R : B[F(R)] := 1.
Au départ : B[i] := 0, pour tout i.

Cette table doit également être bien protégée. Nous l'incluons donc dans le fichier de différences.

De plus, cette propriété des bits à "1" nous aide à déterminer quand fusionner les fichiers. Quand près de cinquante pour cent des bits se trouvent à "1", le prédicteur devient mauvais.

Pour éviter la dégradation trop rapide du prédicteur, nous pouvons effectuer une combinaison de prédicteurs : F1, F2, .., Fn. Si le résultat d'un prédicteur Fi est différent de zéro, alors la page recherchée ne se trouve pas dans le fichier de différences.

Même dans ce dernier cas, le coût de calcul des fonctions Fi est nettement inférieur au coût qui résulte des Entrées-Sorties supplémentaires nécessaires quand nous travaillons de façon aléatoire.

5. Version courante et copie de sauvegarde ("Backup and Current Version").

Les fichiers contenant les valeurs actuelles forment la version courante de la base de données. Les fichiers contenant des valeurs anciennes forment une version cohérente et de sauvegarde de la base de données. Les versions de sauvegarde peuvent être utilisées pour restaurer les fichiers à d'anciennes valeurs. Nous pouvons donner l'exemple de nombreux éditeurs de fichiers qui produisent une nouvelle version complète des fichiers quand l'utilisateur les édite. Les fichiers originaux restent inchangés et en cas de problème avec l'édition en cours, l'utilisateur dispose toujours des anciennes versions des fichiers.

6. Les copies multiples ("Multiple Copies").

La technique des copies multiples comprend en fait deux techniques:

* La garde d'un certain nombre de copies des données.

Si la majorité des copies d'une même donnée a la même valeur, alors cette valeur est considérée comme correcte. Cette technique s'appelle le vote majoritaire.

* La tenue de deux copies avec des drapeaux pour indiquer si une mise-à-jour est en cours.

Curtice [CURT77] appelle ces drapeaux des "Damage Flags". Ils servent à déterminer si une mise à jour de pages modifiées est en cours.

Excepté pendant une mise à jour, les copies multiples doivent toujours avoir la même valeur. Une copie cohérente peut toujours être retrouvée. Elle posséde soit la valeur qu'elle avait avant la mise à jour exécutée pendant la panne, soit la nouvelle valeur. La copie incohérente peut toujours être reconnue comme telle et être écartée.

7. Remplacement prudent ("Careful Replacement").

Le principe du "remplacement prudent" permet de ne pas mettre directement à jour une partie de la structure de données. En fait, les parties modifiées sont mises dans une copie de l'original; cet original n'est détruit qu'après la modification terminée et confirmée.

La différence entre cette méthode et les autres réside dans l'existence de deux copies uniquement pendant la mise à jour.

La technique s'emploie pour permettre une résistance aux fautes puisque l'original reste disponible pendant la mise à jour. Au moindre problème, on conserve l'ancienne valeur.

4.2.2 Les différents types de reprise

Les types de reprise se classifient en terme de "qualité de reprise", ce qui peut être très utile pour une comparaison ou une évaluation des différentes techniques de reprise.

Nous considérons six types de reprise mentionnés ci-dessous.

1. Reprise à l'état correct.

Une base de données est dans un état correct si l'information qu'elle contient correspond aux données les plus récentes entrées par l'utilisateur, et si elle ne contient aucune donnée annulée par celui-ci.

2. Reprise à un état correct qui a existé dans le passé.

Nous pouvons donner la même définition que pour une base de données dans un état correct, mais ici, les données ne sont pas nécessairement les plus récentes. Elles correspondent, par exemple, à celles d'un point de reprise.

3. Reprise à un état passé possible.

Cette reprise consiste à remettre les différents fichiers dans des états qui ont existé précédemment, mais peut-être pas tous au même moment.

4. Reprise à un état valide.

Une base de données est dans un état valide si elle contient une partie des informations d'un état correct. Elle ne contient pas de données erronées, mais certaines peuvent avoir été perdues.

5. Reprise à un état cohérent.

Une base de données est dans un état cohérent si elle est dans un état valide et si les données satisfont les contraintes d'intégrité.

6. Résistance aux fautes.

Une résistance aux fautes assure qu' après une défaillance, le système sera toujours dans un état correct, c'est-à-dire celui précédant la dernière opération (ou série d'opérations).

4.2.3 Comparaisons des différentes techniques de reprise

Afin de mieux cerner les relations entre les techniques que nous venons de décrire, nous présentons une table de références croisées (figure 4.1) qui met en évidence les possibilités de solutions offertes par les différentes techniques en fonction des différents types décrits plus haut.

Un programme de sauvetage (1), par définition, ne peut remettre la base de données que dans un état valide. Utilisé si toutes les autres techniques ont échoué, il ne peut remettre la base de données dans un état cohérent. Il récupère seulement ce qui peut l'être. Cependant, un programme de sauvetage peut être employé comme technique de reprise pour les données redondantes de reprise plutôt que pour la base de données elle-même. Par exemple, il est utilisé pour restaurer le journal après une panne du système.

La technique de la copie incrémentale (2) permet de conserver la base de données dans un état correct passé, ou dans un état passé possible, ou encore dans un état valide.

Type de Reprise / Technique	1) état correct	2) état correct passé	3) état passé possible	4) état valide	5) état cohérent	6) résistance aux pannes
Programme de Sauvetage				X		
Débordement Incrémental		X	X	X		
Fichier Journal	X	X	X	X	X	
Fichiers Différentiels		X	X	X		X
Version Cour. et Copie de S.		X	X			
Copies Multiples						X
Remplacement Prudent		X				X

Figure 4.1: Table de références croisées

La technique du fichier journal (3), une des plus utilisées en pratique parce qu'elle assure un état correct de la base de données, permet également de remettre la base de données dans un état correct passé ou dans un état passé possible, ce qui est moins intéressant. Un second avantage, en plus de l'état correct, est d'assurer un état cohérent. Bien sûr, cette technique peut également donner un état valide, mais d'autres techniques moins coûteuses font tout aussi bien l'affaire.

Les fichiers différentiels (4) sont en premier lieu utilisés parce qu'ils assurent une résistance aux fautes. Le fait qu'ils fournissent en plus un état correct passé, ou un état passé possible n'est pas primordial. Souvent, en pratique, les fichiers différentiels sont associés au fichier journal afin d'assurer une résistance aux fautes en conservant l'état correct ou un état cohérent.

La cinquième technique, version courante et copie de sauvegarde (5), a été reprise parce qu'elle est une des plus anciennes. Elle assure un état correct passé ou un état passé possible. Dans la réalité, elle est souvent couplée avec d'autres techniques (le fichier journal et les fichiers différentiels) comme mécanisme d'archivage.

La technique des copies multiples (6) est assez récente. Elle assure une résistance aux fautes, ce qui la rend très attractive pour les systèmes en temps réel. Elle représente une alternative aux fichiers différentiels pour être associée au fichier journal.

Le remplacement prudent (7), lui aussi, est très intéressant puisqu'il assure une résistance aux fautes. Cependant, dans la réalité, cette technique, bien que présente, a connu moins de succès que le fichier journal (3), les fichiers différentiels (4), ou encore les copies multiples (6).

Pour terminer, nous pouvons dire que

- les techniques de copies multiples, de fichiers différentiels et de versions de sauvegarde font partie de la structure de la base de données
- à l'opposé, les techniques de programme de sauvetage, de la copie incrémentale et du journal restent indépendantes de la manière dont les données sont structurées et mises à jour, elles peuvent être vues comme des utilitaires externes qui peuvent être ajoutés sans grandes difficultés à tout système de base de données.

4.3 Techniques de reprise pour les systèmes transactionnels

Dans cette seconde partie du chapitre consacré aux techniques de reprise pour les bases de données, nous allons plus particulièrement analyser les différentes solutions apportées pour des systèmes en temps réel, ou transactionnels [SERL83].

Le concept majeur qui détermine l'ensemble des techniques que nous allons classifier ici, a été défini dans le premier chapitre : la transaction.

La notion de transaction permet en effet de gérer la concurrence, mais aussi, elle permet une gestion efficace des reprises grâce au paradigme de l' "ACID test" déjà décrit dans le premier chapitre.

Les grandes notions de reprise avant et de reprise arrière s'appliquent tout à fait dans cette partie.

La structure de cette section est basée sur une classification des différents concepts utiles pour une reprise dans une base de données, réalisée par Haerder [HAER78], [HAER83] et Reuter [REUT81].

Avant de présenter les différentes techniques (4.3.2), nous devons d'abord choisir une architecture simplifiée (4.3.1) d'un système de gestion de base de données (SGBD).

Dans un premier temps, nous adoptons différents niveaux d'abstraction pour un système de gestion de base de données. Ensuite, nous décrivons une hiérarchie dans les mémoires. Nous présentons alors différentes vues possibles de la base de données. Puis, nous discernons les différents modes de mise à jour des données dans la base de données (propagation atomique ou non). Enfin, nous donnons trois exemples de techniques assurant une propagation atomique.

Une fois ces cinq points terminés, nous posséderons une base d'architecture de système de gestion de base de données qui nous permettra de présenter de façon plus précise les techniques de reprise basées sur le principe de la journalisation.

4.3.1 Architecture simplifiée d'un S.G.B.D.

1. Découpe modulaire par couche d'abstraction

La figure 4.2 montre les étapes majeures d'abstraction utilisées du niveau de stockage physique jusqu'à celui de l'interface utilisateur.

N5. Couche des accès procéduraux.
N4. Couche des accès navigationnels.
N3. Gestion des chemins d'accès.
N2. Contrôle de propagation.
N1. Gestion des fichiers.

Figure 4.2: Abstraction par couches

A chaque niveau d'abstraction, les objets deviennent plus complexes et permettent des opérations plus puissantes [LAMS85]. L'interface la plus élevée supporte un des modèles de données bien connus: le relationnel, celui en réseau ou le hiérarchique [DATE81], [GILL85].

Pour des raisons de performance, cette hiérarchie ne se reflète pas toujours dans la structuration modulaire.

Nous allons brièvement décrire chacun des niveaux.

* Niveau 1 : gestion des fichiers.

La couche la plus basse travaille directement sur des bits stockés sur un support non volatile (disque ou bande) à accès direct et interprétés par le système de gestion de base de données. Cette couche connaît les caractéristiques physiques de chaque type de support.

* Niveau 2 : contrôle de propagation.

Ce niveau n'est pas considéré séparément dans la littérature courante sur les bases de données, mais pour des raisons qui deviendront claires plus loin, nous devons faire la distinction entre pages (niveau logique) et blocs (niveau physique). Une page est une suite de mots de longueur fixe de l'espace adressable et est associée à un bloc physique par la couche "contrôle de propagation". Ainsi, une page peut être stockée dans différents blocs durant son existence, selon la stratégie choisie pour le contrôle de propagation.

* Niveau 3 : gestion des chemins d'accès.

Cette couche implémente des fonctions d'interface beaucoup plus compliquées que celles des niveaux inférieurs. Elle doit maintenir toutes les représentations physiques (enregistrements, champs, ...) et les chemins d'accès correspondants (pointeurs, clés calculées ("hashed"), arbres de recherche, etc...).

* Niveau 4 : couche des accès navigationnels.

Nous trouvons dans cette couche les opérations et les objets qui sont typiques pour un langage procédural de manipulation de données (DML).
Au niveau de cette interface, l'utilisateur navigue d'un enregistrement à un autre au travers d'une hiérarchie, d'un réseau ou le long de chemins d'accès logiques.

* Niveau 5 : couche des accès non procéduraux.

Ce niveau donne une interface non procédurale à la base de données. Nous pouvons donner comme exemple d'abstraction atteint par ce niveau un modèle relationnel avec un langage de haut-niveau comme SQL [DATE81], [GILL85].

2. La hiérarchie des mémoires

Les dépendances entre mémoire volatile et mémoire permanente ont un impact considérable sur les algorithmes de collecte des informations redondantes et sur l'implémentation des mesures de reprise.

Nous allons prendre une structure hiérarchique des mémoires qui ressemble à la plupart des systèmes de gestion de base de données commerciaux (figure 4.3) [CARD87].

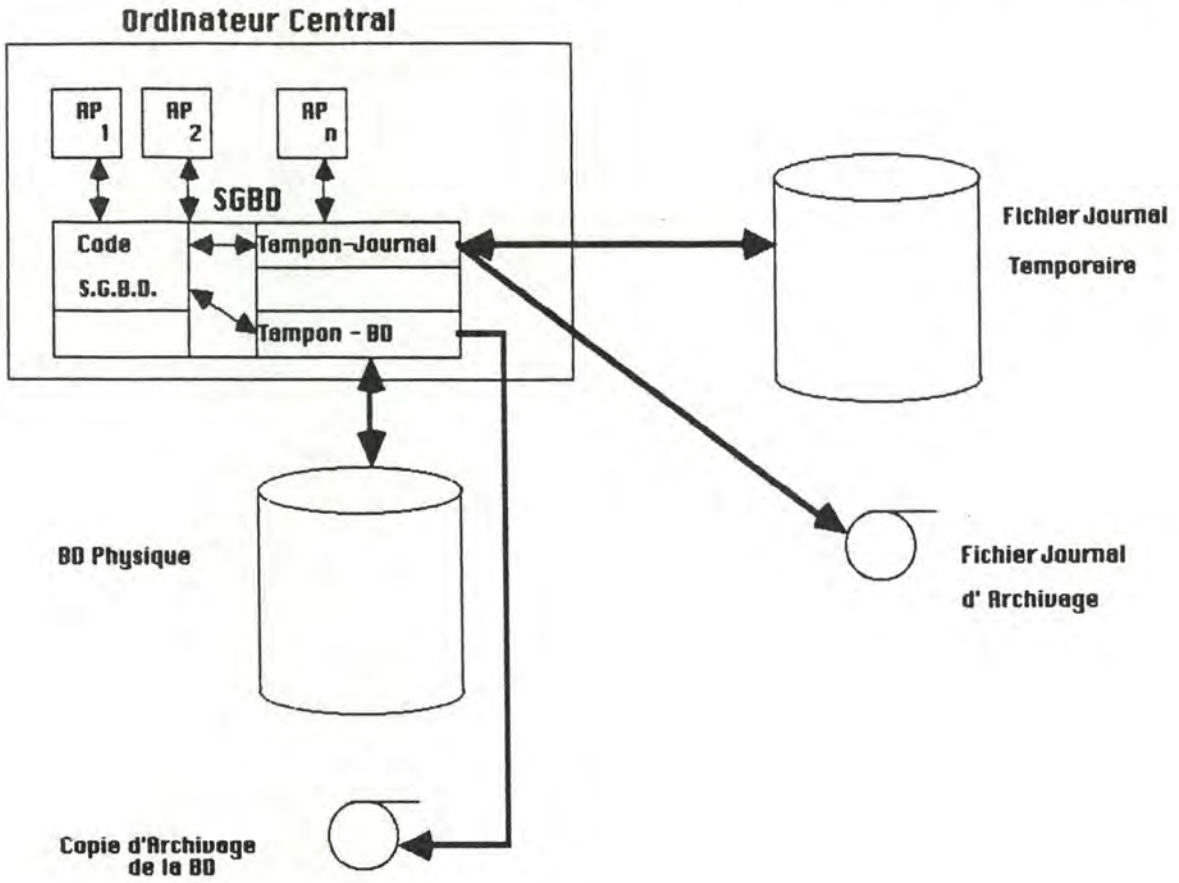

Figure 4.3: Hiérarchie des mémoires

L'ordinateur central, où se situent les programmes d'application (AP1, .., APn) et le système de gestion de base de données (SGBD), possède une mémoire commune d'habitude volatile. Cette mémoire peut donc s'effacer en cas de panne. Dans certaines applications en temps réel, des batteries de secours associées à la mémoire centrale permettent de ne plus perdre les tampons du journal et de la base de données en cas de défaillance. Nous n'envisageons pas ce cas ici, et supposons qu'à tout moment, la mémoire centrale peut s'effacer. Nous considérons donc que les contenus des tampons de la base de données , aussi bien que les contenus des tampons de sortie des fichiers journaux , sont perdus quand le système de gestion de base de données se termine anormalement.

En dessous de la mémoire principale volatile, nous avons une hiérarchie à deux niveaux de copies permanentes de la base de données.

Un niveau contient une version "on-line" de la base de données en accès direct; l'autre contient une copie d'archivage comme prévision de la perte de la copie "on-line".

Comme toutes les techniques pour la tolérance aux fautes, les techniques de reprise se basent sur la redondance.

Nous venons de mentionner un type de redondance, la copie d'archivage , conservée comme point de départ pour la reconstruction d'une version à jour de la base de données (Refaire généralisé).

Pour permettre ceci, nous avons besoin de deux fichiers journaux :

- Journal Temporaire.

  L'information collectée dans le journal temporaire permet la reprise après une panne. Ce fichier contient l'information nécessaire pour reconstruire le tampon le plus récent de la base de données. DEFAIRE (UNDO) une transaction exige un accès aléatoire aux enregistrements redondants. Ce journal temporaire devrait se trouver sur disque.

- Journal d'Archivage.

  Le journal d'archivage permet un REFAIRE GENERALISE (GLOBAL REDO) après une défaillance du support. Il contient toutes les modifications effectuées sur la base de données après la dernière mise à jour de la copie d'archivage. Le journal d'archivage s'emploie toujours de manière séquentielle. Il devrait donc s'écrire sur bande magnétique.

3. Les "vues possibles" de la base de données

En cas de défaillance, nous avons différentes possibilités pour retrouver le contenu d'une page, nous les appellons "différentes vues" de la base de données.

La base de données courante comprend tous les objets accessibles au système de gestion de base de données pendant l'utilisation normale. Les contenus courants de toutes les pages peuvent être trouvés sur disque, à l'exception des pages récemment modifiées qui se trouvent encore dans le tampon de la base de données.

La base de données matérialisée représente la base de données que le système de gestion de base de données trouve au redémarrage après une panne au moment où aucune information journalisée n'a encore été réexécutée. Les tampons sont effacés, et certaines modifications de pages (même de transactions réussies) peuvent ne pas avoir été enregistrées. De plus, un nouvel état d'une page peut

avoir été écrit sur disque, sans pour autant avoir mis à jour la structure de contrôle qui fait le lien entre les pages et les blocs. Une référence à une telle page donnera l'ancienne valeur. Cette vue de la base de données doit être transformée par le système de reprise pour remettre la base de données dans l'état logique cohérent le plus récent.

La base de données physique se compose de tous les blocs de la copie "on-line" contenant les images de pages. Cette vue doit normalement être utilisée par les procédures de reprise.

Avec ces vues de la base de données, nous distinguons trois types de mises à jour.

D'abord, nous avons la modification du contenu des pages par un module de niveau supérieur. Cette opération prend place dans le tampon de la base de données et n'affecte donc que la base de données courante.

Ensuite, nous avons l' opération d'écriture qui transfère la page modifiée sur disque. Cette opération n'affecte que la base de données physique. L'information sur le bloc de la nouvelle page se trouve sur une mémoire volatile. Elle risque d'être perdue en cas de défaillance et ainsi, le bloc ne serait plus accessible. La nouvelle page ne fait pas encore partie de la base de données matérialisée.

Pour éviter cela, nous devons effectuer un troisième type d'opération, la propagation. Cette opération écrit les données courantes mises à jour dans une mémoire non volatile, ce qui assure la disponibilité après une panne.

4. Différents modes de mise à jour

Dans cette section, nous allons définir un certain nombre de concepts liés au transfert des modifications de la base de données d'une mémoire volatile à une mémoire non volatile.

Toute modification d'une page (qui change la base de données courante) prend place dans le tampon de la base de données et se trouve donc en mémoire volatile. Pour sauver ce nouvel état, la page correspondante doit être copiée en mémoire non volatile, c'est-à-dire dans la base

de données physique.

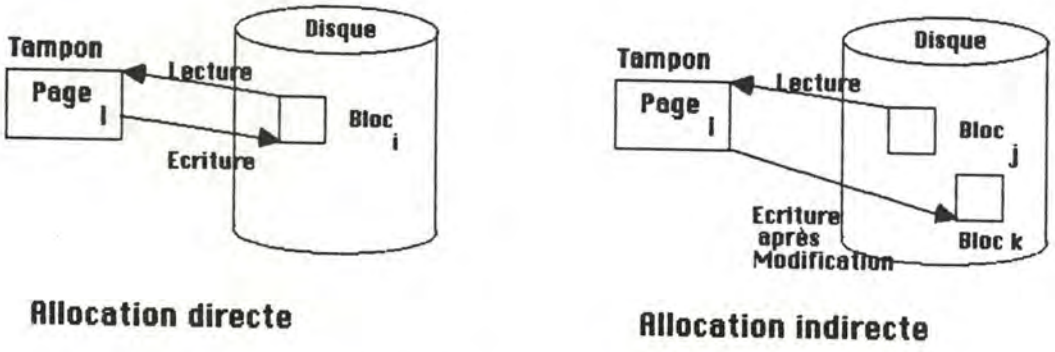

Figure 4.4: Types d'allocation

La figure 4.4 montre deux types d'allocation.

En allocation directe , chaque page est reliée exactement à un bloc du fichier correspondant. Toute écriture d'une page modifiée écrase l'ancienne. Dans ce cas, l'écriture et la propagation coïncident. Chaque propagation (écriture physique) peut être interrompue par une panne du système, et ainsi laisser la base de données matérialisée, et peut-être aussi la base de données physique, dans un état incohérent.

Le principe d' allocation indirecte permet de ne pas écraser l'ancienne valeur et d'écrire la page modifiée dans un nouveau bloc. Nous pouvons aussi garder plusieurs versions successives d'une page. Nous pouvons déterminer par un critère approprié (au point de vue de la cohérence) le moment où une version plus récente remplace définitivement une plus ancienne; celui-ci n'est plus lié à l'écriture. Ce schéma de mise à jour comprend certaines propriétés très intéressantes dans le cas d'une reprise. Nous pouvons toujours revenir à un état passé. La propagation d'un nombre arbitraire de pages peut être rendue ininterruptible par une défaillance du système.

Sur base de cette observation, nous pouvons distinguer deux types de stratégies de propagation: la propagation atomique et la non-atomique.

PROPAGATION ATOMIQUE. Tout groupe de pages modifiées peut être propagé comme un tout, de telle sorte que toutes ou aucune des mises à jour ne se répercute dans la base de données matérialisée.

PROPAGATION NON-ATOMIQUE. Les pages sont écrites dans les blocs en écrasant les anciennes valeurs. Aucun groupe de

pages ne peut être écrit de façon indivisible, et même une simple écriture peut être interrompue. La propagation est vulnérable à une panne du système.

5. Exemples de techniques assurant une propagation atomique

Nous allons décrire trois techniques qui assurent une propagation atomique.

1. La technique de la "Page Fantôme" ("Shadow Page").

Cette technique est présentée en détail par Lorie [LORI77].

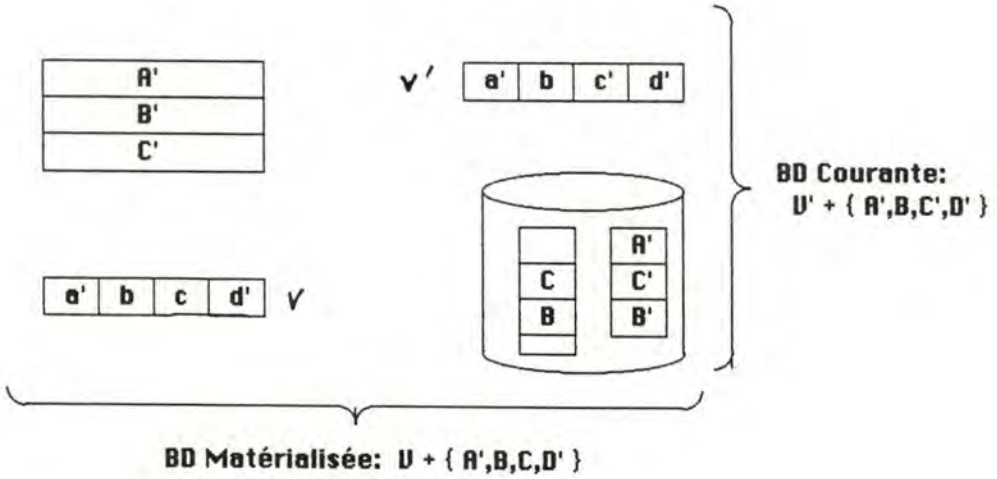

Figure 4.5: Le Mécanisme de la page fantôme

La figure 4.5 montre comment le lien entre pages et blocs se fait à l'aide de tables de correspondance. Ces tables (V et V') ont une entrée par page donnant le numéro du bloc où se trouve la page.

Les pages fantômes, accédées par la table V, préservent l'ancien état de la base de données matérialisée. La version courante est définie par la table V'. Toutes les pages modifiées sont écrites dans leur nouveau bloc.

Si une défaillance se produit lors d'une écriture, la base de données revient à son ancien état. Quand toutes les pages ont été écrites, la propagation atomique peut avoir lieu en changeant un enregistrement sur disque (qui pointe maintenant sur V' plutôt que sur V), ce qui peut être fait sans perturbation même si une panne se produit.

2. La technique de la double copie ("Twist Storage Structure")

La technique de la double copie, schématisée à la figure 4.6, garde deux versions récentes d'une page. Pour chaque accès à une page, les deux versions doivent être lues dans le tampon.

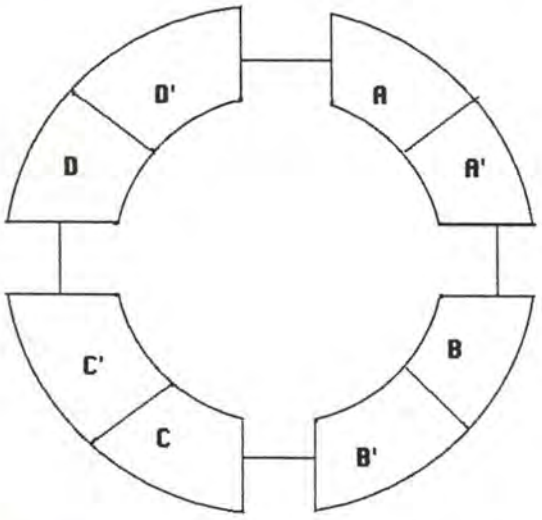

Figure 4.6: Le mécanisme de la double copie

Nous pouvons diminuer le surcoût d'une telle technique (deux fois plus d'accès) en les stockant sur des blocs adjacents. Reuter [REUT80] propose même de prendre des blocs de la taille de deux pages afin d'avoir les deux versions sur un et un seul bloc ce qui supprime le surcoût d'accès.

La dernière version, reconnue par le mécanisme de l'estampillage [BAYE82], reste dans le tampon, l'autre est immédiatement écartée. Une page modifiée remplace sur disque la version la plus ancienne.

La propagation atomique est assurée par l'incrémentation d'un compteur spécial qui relate l'estampillage d'une page.

Cette technique présente bien entendu la faiblesse d'être très vulnérable en cas d'écrasement de tête sur disque, puisqu'alors les deux copies sont détruites.

3. La technique du fichier différentiel ("Differential File").

La technique des fichiers différentiels (figure 4.7) a déjà fait l'objet d'une description dans le paragraphe précédent sur les techniques de reprise dans les systèmes de traitement par lot [HAIN87], [VERH78].

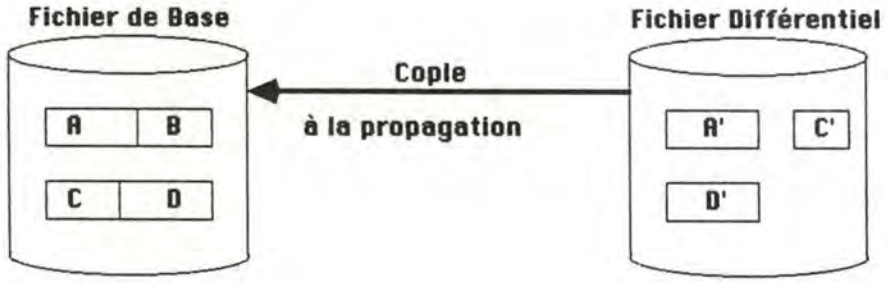

Figure 4.7: Le mécanisme du fichier différentiel

Les pages modifiées sont écrites sur un fichier séparé, appelé fichier de différences. La propagation des mises à jour dans la base de données principale (le fichier de base), ce qui revient à la fusion des fichiers, n'est pas atomique en elle-même, mais une fois que les modifications ont été écrites dans le fichier de différences, la propagation peut être répétée plusieurs fois si nécessaire.

Maryanski [MARY80] donne une description complète de cette technique couplée avec celle du remplacement prudent afin de créer une architecture de système de gestion de base de données tolérante aux fautes.

4.3.2 Les techniques de reprise

Maintenant que nous avons notre système de gestion de base de données, nous pouvons présenter de façon détaillée la reprise de panne.

Agrawal [AGRA85] a comparé et intégré les trois techniques de reprise les plus utilisées dans les systèmes transactionnels (fichier journal, copies multiples et fichiers différentiels) avec les mécanismes de concurrence décrits dans le premier chapitre (verrouillage, estampillage et méthode optimiste). Il a montré les interactions possibles et a évalué les différentes solutions à l'aide d'équations de coût.

Nous allons nous concentrer sur une technique de reprise de panne, la plus répandue pour les systèmes transactionnels [CURT77], à savoir l'emploi d'un fichier journal. Bien sûr, d'autres techniques de reprise sont parfois, et même souvent, implémentées, mais leur fonctionnement ne diffère guère de

celui décrit pour les systèmes de traitement par lot. Nous n'allons pas nous répéter et nous incluons ces techniques comme part entière de cette section. Généralement, ces techniques sont combinées avec celle du fichier journal afin d'assurer une plus forte protection des données et des informations redondantes. Une défaillance matérielle (écrasement d'une tête sur disque), ou logicielle (mauvaise écriture sur disque), peut rendre tout le système de reprise inopérationnel si le fichier journal est détruit. Nous devons bien entendu être capable de retrouver l'information "détruite". Pour cela, nous employons les techniques d'archivages décrites dans la section précédente sur les techniques de reprise pour les systèmes de traitement par lot.

Nous allons donc décrire ici la technique de reprise par l'emploi d'un fichier journal [HAER83]. En fait, cette technique regroupe un ensemble de techniques classées selon les différentes "journalisations" possibles.

Dans un premier temps, nous analysons les états possibles d'une base de données après une panne. Nous en retirons les informations redondantes nécessaires pour remettre la base de données dans un état cohérent. Pour rendre efficace ces techniques, nous introduisons le concept de point de reprise. Enfin, nous donnons une évaluation qualitative des différentes solutions.

1. Etat de la base de données après une panne

Après une catastrophe, le système de gestion de base de données doit redémarrer en appliquant toutes les actions nécessaires décrites dans le premier chapitre (REFAIRE les transactions).

Nous avons perdu le tampon de la base de données et ainsi, la base de données courante qui était la seule vue de la base de données à contenir l'état le plus récent des opérations effectuées.

Nous avons donc la base de données matérialisée et le fichier journal pour commencer la reprise.

Nous n'avons pas encore discuté du contenu des fichiers journaux pour la bonne raison que le type et le nombre de données à journaliser sont dépendants de l'état de la base de données matérialisée après une panne. Cet état dépend de la méthode de propagation employée.

Dans le cas d'une allocation directe, c'est-à-dire d'une propagation non-atomique, chaque opération d'écriture affecte la base de données matérialisée. La décision d'écrire des pages est prise par le gestionnaire des tampons selon la capacité des tampons, et donc à des moments qui paraissent arbitraires [EFFE84]. Nous ne pouvons donc pas prévoir l'état de la base de données matérialisée après une panne. Quand des modifications

récentes ont été répercutées sur la base de données matérialisée, nous ne pouvons plus savoir (sauf informations supplémentaires) quelles pages ont été modifiées par des transactions complètes (dont les contenus doivent être reconstruits par l'action REFAIRE), et quelles pages ont été modifiées par des transactions incomplètes (dont les contenus doivent être remis à leurs valeurs précédentes par l'action DEFAIRE). Comme nous ne pouvons rien dire sur l'état de la base de données, nous la qualifions de "chaotique".

Nous ne pouvons pas nous attendre à lire des images valides pour toutes les pages de la base de données matérialisée après une défaillance. La base de données matérialisée devient incohérente au niveau propagation (Niveau 3), et toute abstraction d'une couche supérieure échouera.

Dans le cas d'une allocation indirecte, c'est-à-dire d'une propagation atomique, nous en savons beaucoup plus sur l'état de la base de données matérialisée après une panne. La propagation atomique garantit une répercution totale ou nulle des pages modifiées dans la base de données matérialisée. Nous trouvons donc la base de données matérialisée exactement dans l'état produit par la dernière propagation réussie.

Nous possédons une base de données matérialisée dans un état cohérent (mais pas nécessairement à jour) au moins jusqu'au niveau trois de la hiérarchie, permettant ainsi l'exécution d'opérations d'un niveau supérieur.

2. Information à journaliser

Le journal doit contenir toute l'information nécessaire pour transformer la base de données matérialisée dans l'état cohérent le plus récent.

Comme nous l'avons montré, la base de données matérialisée peut être dans un état plus ou moins défini. Elle peut ou non remplir des contraintes de cohérence. Le contenu de la base de données matérialisée au début d'une reprise détermine la quantité et la qualité de l'information à journaliser.

Nous pouvons être sûr du contenu de la base de données matérialisée dans le cas d'une propagation atomique. Par contre, nous ne pouvons le prévoir avec une propagation non atomique. Cependant, des mesures supplémentaires permettent de réduire quelque peu le degré d'incertitude.

Dans un premier temps, nous exposons la gestion des tampons pour permettre des actions de reprise de type DEFAIRE. Ensuite, dans la même optique, nous l'analysons

pour le type REFAIRE. Enfin, nous donnons une classification des informations à journaliser.

1. Actions de reprise de type DEFAIRE

Pendant l'exécution normale, les pages modifiées sont écrites sur disque par un algorithme de gestion des tampons de la base de données. De façon idéale d'un point de vue des performances [EFFE84], cette écriture se produit à des moments déterminés uniquement par l'occupation des tampons. En général, des données instables, c'est-à-dire des pages modifiées par des transactions incomplètes, peuvent avoir été écrites dans la base de données physique. Les opérations de type DEFAIRE doivent restaurer la base de données matérialisée, à savoir éliminer les données instables.

Nous pouvons éviter cela par la modification du gestionnaire des tampons pour empêcher l'écriture et la propagation de données instables. Les pages ne sont plus écrites ni propagées tant que la transaction correspondante n'a pas été confirmée ("commit").

Dans ce cas, DEFAIRE peut être considérablement simplifié:

- si aucune donnée instable n'a été propagée, nous n'avons plus besoin de l'action "DEFAIRE GENERALISE"

- si aucune donnée instable n'a été écrite, l'action de DEFAIRE une transaction peut se limiter aux opérations sur la mémoire centrale à savoir sur les tampons

Malheureusement, l'emploi de cette méthode devient généralement impossible pour des systèmes réels qui demandent de longues transactions, et donc de larges tampons. Cette restriction semble cependant s'amoindrir à l'heure actuelle, vu les énormes progrès de ces dernières années sur l'augmentation des capacités des mémoires centrales.

Nous observons donc deux méthodes typiques pour la gestion des tampons en fonction des actions de reprise de type DEFAIRE:

- la méthode "dérobée": les pages modifiées peuvent être écrites ou propagées à tout moment

- la méthode "non-dérobée": les pages modifiées sont conservées dans les tampons au moins jusqu'à la fin de la transaction

2. Actions de reprise du type REFAIRE

Par le principe de la durabilité d'une transaction, lorsqu'une transaction est terminée, tous ses résultats doivent survivre à n'importe quelle défaillance. Les mises a jour terminées, non encore propagées dans la base de données matérialisée, pourraient être perdues en cas de panne (effacement des tampons, ...).

Nous devons donc prévoir assez d'informations redondantes dans le fichier journal pour reconstruire ces résultats en vue d'une remise en route du système.

Nous pouvons éviter ce genre de reprise par la technique suivante. Pendant la première phase de l'exécution d'un "fin de transaction", toutes les pages modifiées par cette transaction sont propagées dans la base de données matérialisée. Nous forçons donc leur écriture et leur propagation. Ainsi, nous pouvons affirmer qu'en cas de transaction terminée, toutes les modifications ont été répercutées, et qu'en cas de panne pendant la propagation, la transaction n'a pas été confirmée et donc doit être "défaite".

Nous observons également deux méthodes pour la gestion des tampons en fonction des actions de reprise de type REFAIRE :

- la méthode "forcée": toutes les pages sont écrites et propagées pendant l'exécution du "fin-de-transaction"

- la méthode "non-forcée": la propagation n'est pas déclenchée par l'exécution du "fin-de-transaction".

3. Classification des informations à journaliser

Suivant les techniques d'écriture et de propagation employées, l'information journalisée peut servir à :

- enlever des données invalides de la base de données matérialisée
- compléter la base de données matérialisée avec les mises à jour des transactions terminées non encore répercutées au moment de la panne.

Nous allons maintenant décrire quels types d'informations nous pouvons journaliser et comment les employer en cas de panne.

Les données journalisées sont des informations redondantes, collectées dans le seul but de permettre la reprise en cas de défaillance. Dans la littérature, nous retrouvons plusieurs approches pour classifier les différentes sortes d'informations à journaliser. La plupart du temps, les auteurs parlent de granularité [RIES77], [HAIN87].

D'après Haerder [HAER83], nous pouvons classer ces données journalisées selon deux critères.

Le premier se rapporte au type d'objet journalisé. Nous appelons "journalisation physique" l'opération qui écrit une représentation physique des données (souvent la page) dans le journal, et "journalisation logique" l'opération qui écrit une représentation logique, c'est-à-dire d'un niveau supérieur (souvent l'enregistrement).

Le second se rapporte au type d'information journalisée. Nous pouvons enregistrer dans un journal les états de la base de données avant et après une modification, ou bien la transition de ce changement.

	ETAT	TRANSITION
LOGIQUE	—	ACTIONS (DML)
PHYSIQUE	IMAGES AVANT - IMAGES APRES	DIFFERENCES (EXOR)

Figure 4.8: Classification des informations à journaliser

La figure 4.8 reprend les types d'information à journaliser suivant les critères décrits ci-dessus. Nous obtenons donc les cas suivants :

*) La technique la plus utilisée dans les systèmes de gestion de base de données commerciaux prend la page physique comme unité d'information journalisée. Chaque fois qu'une modification a lieu, l'entièreté de la page est écrite dans le journal. Pour les opérations du type DEFAIRE, nous avons besoin des images avant modification des pages. Pour celles du type REFAIRE, nous avons besoin des images après modification.

*) Une autre technique, basée également sur les pages, écrit dans le journal la différence entre le nouvel et l'ancien état. L'opération qui calcule cette différence doit être commutative et associative pour permettre de retrouver l'image avant ou après modification suivant les cas. La fonction OU-EXCLUSIF répond à ces exigences.

Le gain de place constitue un des avantages majeurs de cette technique par rapport à la précédente. En effet, l'enregistrement des états comme unité de journalisation nécessite deux pages, alors que celui de la différence n'en demande qu'une. De plus, vu le faible taux de modifications par page, nous pouvons aussi employer une technique de compactage des zéros obtenus par différence de deux pages "presque identiques" [KAUN84].

*) Nous pouvons aussi stocker l'information d'un niveau supérieur, à savoir l'enregistrement, ou la transaction amenant cet enregistrement.

La réduction de la taille du journal par rapport aux techniques précédentes donne un avantage certain à cette technique.

*) Nous avons enfin le type de journalisation le plus élevé dans la hiérarchie. La journalisation logique au niveau des accès navigationnels. A ce niveau, nous pouvons représenter les modifications exécutées par le programme (transactionnel) de façon très compacte, en enregistrant seulement les actions de mises à jour avec leurs paramètres dans le langage de manipulation de données (DML).

Le journal contient seulement des opérations d'INSERTION, de MODIFICATION, et de SUPPRESSION sur des enregistrements. Les interfaces s'occupent de faire le lien avec les autres niveaux (quelles pages modifier, etc...).

La reprise se résume à réexécuter certaines expressions du langage de manipulation de données. Dans le cas d'une action de type DEFAIRE, nous devons bien entendu réexécuter l'inverse de l'expression qui doit être défaite.

3. Optimisation par introduction de points de reprise

En général, le nombre de données journalisées pour refaire certaines opérations croît avec l'intervalle de temps entre deux défaillances successives; ce qui veut

dire que le coût de reprise augmente avec la disponibilité du système. Pour des raisons de performance et d'applicabilité, nous devons introduire des mesures pour rendre le coût de reprise indépendant du temps moyen entre défaillances (MTBF). Une technique largement employée consiste à mettre des points de reprise afin de limiter le nombre d'opérations à refaire après une panne.

Le concept de point de reprise a été décrit dans le premier chapitre. Cependant, il a été analysé au sens large (en dehors du contexte de transaction) ce qui lui conférait une autre signification : la recherche d'un état passé de la base de données ou du système pour effectuer une restauration. Ici, nous possédons un environnement transactionnel qui nous fournit ces états passés de la base de données. Nous n'utilisons donc les points de reprise qu'au point de vue performance.

De nombreuses études ont été menées pour calculer l'endroit où mettre ces points de reprise afin d'optimiser les temps de reprise en fonction du surcoût provoqué [CHAN75], [GELE78], [GRAY79], [KRIS84], [REUT84], [TONI75].

Nous allons reposer le problème en terme de transaction, et nous trouvons quatre critères pour déterminer où effectuer un point de reprise [HAER83].

1. Points de reprise orientés transaction

La figure 4.9 schématise les points de reprise orientés transaction.

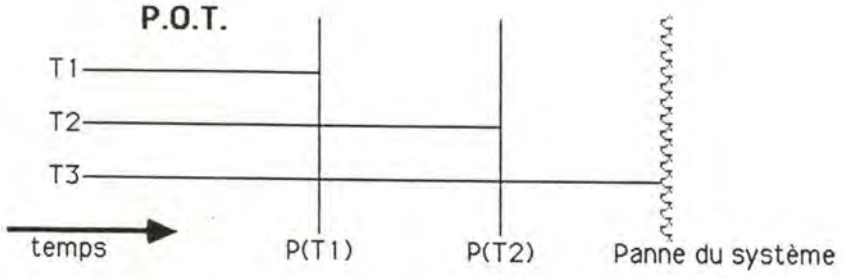

Figure 4.9: Points de reprise orientés transaction

A la fin de chaque transaction, un point de reprise est provoqué, ce qui correspond en fait à la discipline "FORCE" vue plus haut où les effets d'une transaction terminée doivent être enregistrés de

façon permanente (propagation des pages modifiées avant l'écriture du "fin de transaction" dans le journal). La réduction du coût de reprise, induite par la discipline "FORCE" qui évite de devoir DEFAIRE une transaction après une panne, provoque cependant un surcoût sur la charge normale du système. Pour de grandes applications sur des systèmes ayant de larges tampons, le coût d'écritures non nécessaires de pages fréquemment utilisées rend ce procédé peu efficace et donc peu attractif.

2. Point de reprise correspondant à un point de cohérence

A la figure 4.10, nous trouvons comment générer un point de reprise qui, en même temps, assure la cohérence de la base de données telle que nous l'avons définie dans le premier chapitre. Le point de reprise correspond à un point de cohérence.

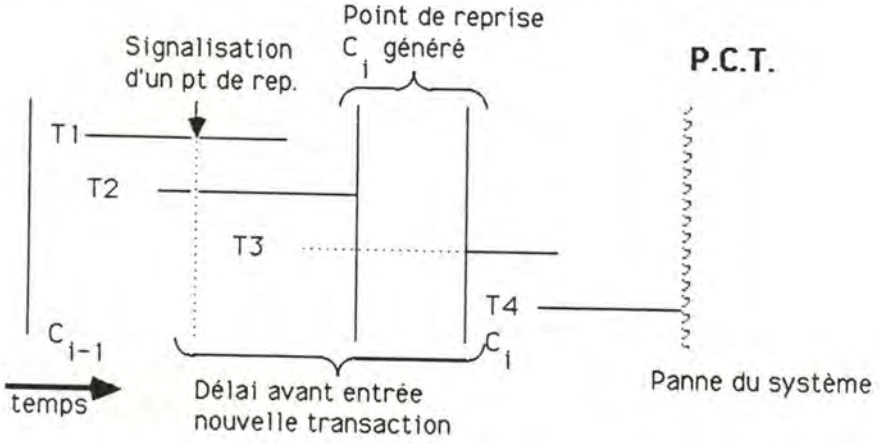

Figure 4.10: Points de reprise avec cohérence

Lorsqu'un signal de demande de point de reprise est généré, par intervalles réguliers ou pour des raisons de besoin de place en mémoire centrale, les transactions en cours continuent jusqu'à leur fin mais toute nouvelle transaction est mise en attente jusqu'à ce que ce point de reprise soit généré, ce qui est fait à la terminaison de la dernière transaction encore en cours.

Le gain se situe ici au niveau des opérations à refaire en cas de panne. Sur la figure 5.10, seule la transaction T3 doit être réexécutée. La transaction T4, elle, doit être défaite.

Cependant, cette technique ne peut être admise dans de nombreux systèmes, surtout multi-utilisateurs. D'abord, la mise en attente du système pour la génération d'un point de reprise pendant que les transactions se terminent peut représenter un délai intolérable pour l'arrivée de nouvelles transactions. Ensuite, le coût d'exécution d'un point de reprise peut être élevé dans le cas de tampons larges, où de nombreuses modifications de pages peuvent être accumulées. Le temps nécessaire à la propagation peut ne pas être acceptable (plusieurs secondes d'indisponibilité).

3. Points de reprise en dehors de toute action

Chaque transaction est considérée comme une séquence d'actions élémentaires affectant la base de données. Des points de reprise peuvent être générés quand aucune action de mise à jour n'est en cours.

Ce type de point de reprise met aussi le système en attente, mais uniquement au niveau des actions, ce qui paralyse bien moins le système que la technique précédente. La figure 4.11 montre un scénario de création d'un tel point.

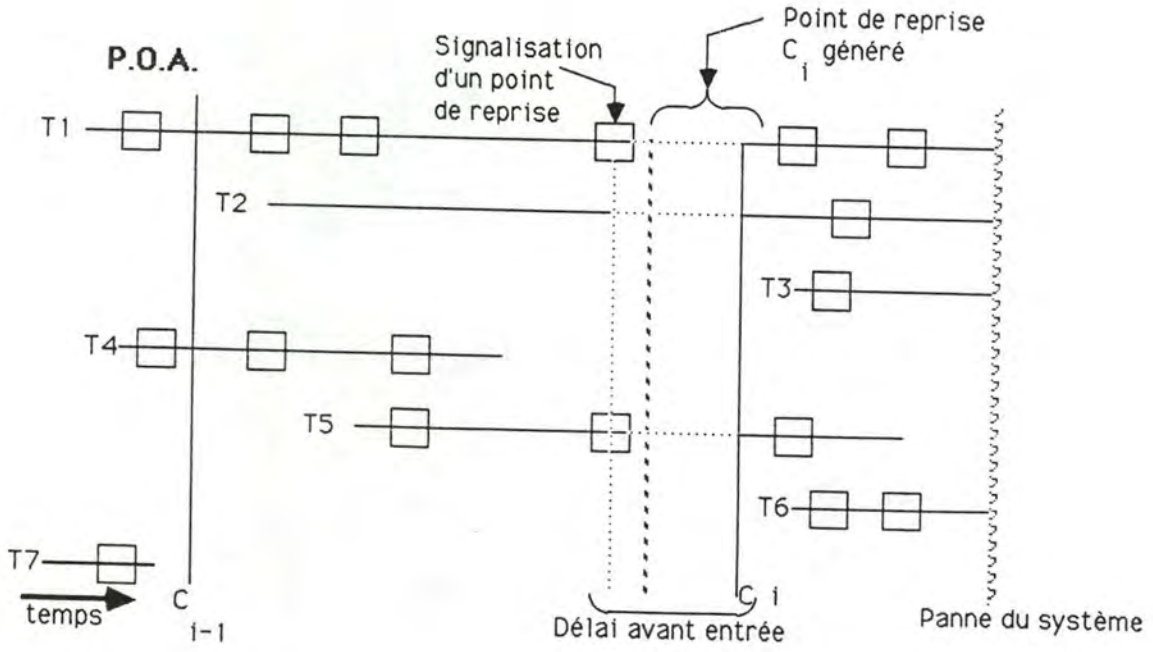

Figure 4.11: Points de reprise orientés action

La reprise dans ce cas effectue un "Défaire Généralisé" pour les transactions T1, T2 et T3. La dernière action de la transaction T5 et toutes celles de T6 doivent être refaites. Les modifications des transactions T4 et T7 font partie de la base de données matérialisée par effet des points de reprise.

Cette technique, bien que plus réaliste que la précédente parce qu'elle évite les longs délais pour l'entrée de nouvelles transactions, reste toujours coûteuse pour la propagation si nous employons de grands tampons.

4. Points de reprise "flous"

Le principe du point de reprise "flou" consiste à réduire le coût trop élevé d'un point de reprise en évitant la propagation au moment du point de reprise. Lors d'un point de reprise, au lieu d'écrire les pages dans le fichier journal, nous écrivons des informations sur le contenu,

l'occupation du tampon, ce qui peut être fait en deux ou trois opérations [GRAY79]. Nous pouvons ainsi déterminer quelles pages du tampon contenaient des données définitives au moment de la panne.

Cependant, cette technique ne rend pas le "Refaire Partiel" indépendant du temps moyen entre défaillances puisqu'une page fort utilisée peut rester longtemps en mémoire centrale. Nous pouvons, pour combler cette lacune, faire une analyse sur les pages du tampon lors d'un point de reprise. Si une page se trouvait déjà en mémoire centrale lors du dernier point de reprise, elle est sauvée, c'est-à-dire propagée.

De plus, cette méthode ne peut s'appliquer qu'avec une propagation non atomique. Une propagation atomique prend toujours effet à un moment bien défini, à savoir au point de reprise. Les pages qui n'ont pas encore été écrites (ou propagées) sont perdues après une panne.

4. Evaluation qualitative des différentes solutions

Nous avons fait le tour des différentes classifications possibles pour les systèmes transactionnels. Nous pouvons donner une figure récapitulative (figure 4.12) des concepts vus [HAER83]. Certaines combinaisons contradictoires n'ont pas été reprises.

Stratégie de propagation	NON ATOMIQUE							ATOMIQUE				
Remplacement du tampon	DEROBE				NON DEROBE			DEROBE			NON DEROBE	
Fin de transaction	FORCE	NON FORCE			FORCE	NON FORCE		FORCE	NON FORCE		FORCE	NON FORCE
Type de point de reprise	P.O.T.	P.C.T.	P.O.A.	FLOU	P.O.T.	P.C.T.	FLOU	P.O.T.	P.C.T.	P.O.A.	P.O.T.	P.C.T.
Etat de la BD Matérialisée après un panne	CH	CH	CH	CH	CH	CH	CH	CT	CT	CA	CT	CT
Coût de DEFAIRE une transaction	+	+	+	+	--	--	--	-	-	+	--	--
Coût du REFAIRE partiel lors du redémarrage	--	-	-	+	--	-	-	--	-	-	--	-
Coût du DEFAIRE généralisé lors du redémarrage	+	+	+	+	--	--	--	--	--	+	--	--
Surcoût pendant l'exécution normale	--	--	--	--	--	--	--	+	+	+	+	+
Fréquence des points de reprise	+	-	-	-	+	-	-	+	-	-	+	-
Coût des points de reprise	+	++	++	-	+	++	+	+	++	++	+	++

Légendes: CH = chaotique
CT = cohérent point de vue transaction
CA = cohérent point de vue action
P.O.T. = point de reprise orienté transaction
P.C.T. = point de reprise avec cohérence
P.O.A. = point de reprise orienté action

Figure 4.12: Taxonomie et évaluation des concepts de reprise

En reprenant la classification de la figure 5.12, nous pouvons entamer une évaluation des différentes techniques. Certains critères de notre taxonomie permettent de diviser les différentes reprises en techniques clairement distinctes.

* La propagation atomique assure une base de données matérialisée cohérente lors de la survenance d'une panne. Les techniques de journalisation au niveau physique ou logique peuvent être appliquées. Les avantages décrits sont cependant compensés par un surcoût considérable pendant l'exécution normale dû au maintien d'informations redondantes.

* La propagation non atomique amène une base de données chaotique lors de la survenance d'une panne, ce qui rend obligatoire la journalisation au niveau physique.

* Les points de reprise orientés transaction et les points de reprise au niveau des actions sont coûteux.

Pour des études plus détaillées par modèles analytiques, nous renvoyons à la littérature [CHAN75], [EAST78], [GRIF85], [REUT84].

Une étude comparative de systèmes de gestion de base de données (SGBD) existants est donnée en [SCHE82].

PARTIE 2

Cas Concrets :

Les Systèmes

TANDEM

et

STRATUS

Cette deuxième partie est consacrée à l'étude des systèmes à tolérance de panne les plus implantés sur le marché du transactionnel: les systèmes "NonStop" de TANDEM et "Stratus" d'IBM.

Nous analysons tout d'abord leurs aspects de tolérances aux pannes selon trois grands axes:

- le matériel
- le logiciel
- l'intégrité des données

Ensuite, nous situons ces deux systèmes par rapport à un ensemble de critères.

Chapitre 5

Le Système

NonStop

de TANDEM

5 Le système NonStop de Tandem

La description qui suit du système NonStop de Tandem est basée sur un exposé et de la documentation donnés par le constructeur [TAND87].

5.1 Aspects de la tolérance aux pannes dans le matériel

Chaque système NonStop est constitué d'au moins deux CPUs, de plusieurs chemins d'accès entre les CPUs et les contrôleurs d'E/S, et de plusieurs sources d'alimentation. De plus, comme nous le voyons sur la figure 5.1, les CPUs sont connectés entre eux via une paire de bus ("Dynabus") fournissant deux chemins de communication indépendants.

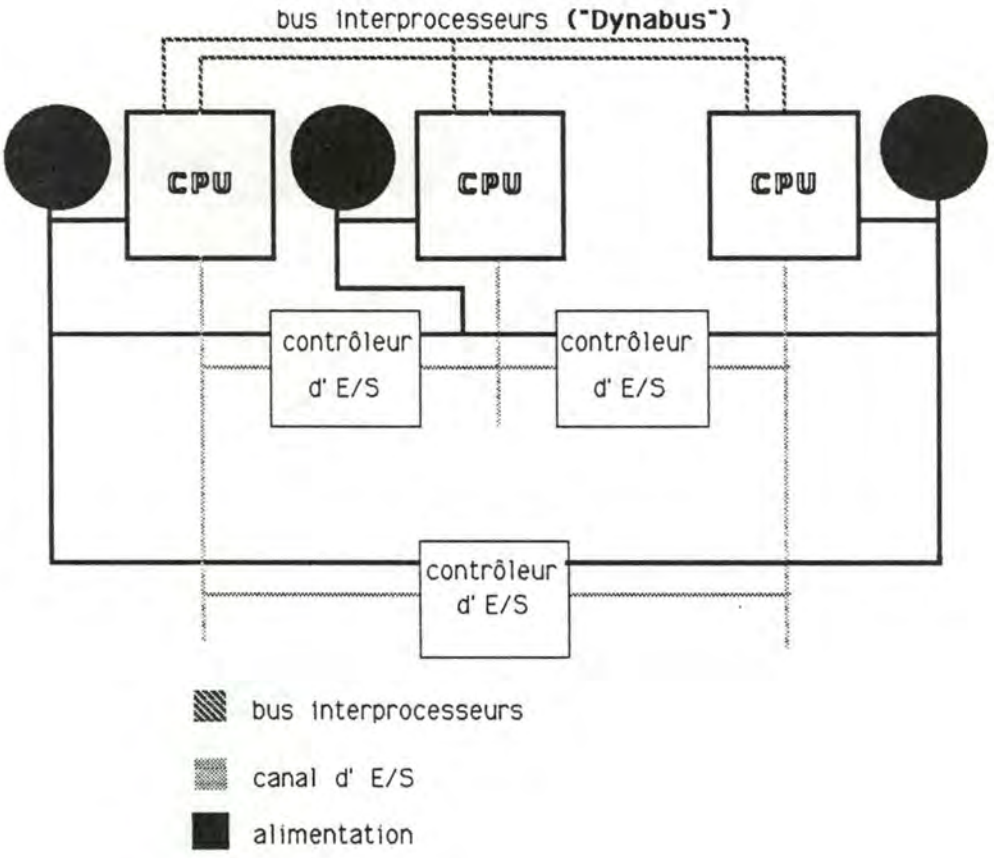

Figure 5.1: Le système NonStop de TANDEM

- Les CPUs et le Dynabus

Un système NonStop est composé de 2 à 16 modules CPUs, chacun de ceux-ci comprenant une UT (Unité de Traitement), une mémoire, un canal d'E/S et une interface avec le bus interprocesseur ("Dynabus"). La figure 5.2 représente deux modules CPUs connectés ensembles via le Dynabus. Le fait que les CPUs communiquent par un bus (et non pas par une mémoire commune) permet de confiner la majorité des erreurs au sein de chaque CPU.

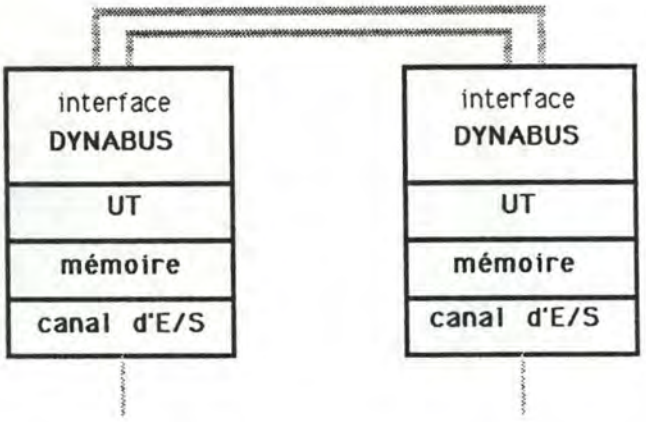

Figure 5.2: Deux modules CPUs

Le dynabus est composé d'une paire de bus autonomes travaillant simultanément. En cas de défaillance de l'un de ces bus, les communications destinées à celui-ci sont redirigées vers le bus restant, provoquant ainsi une dégradation de performance [KIM84].

Chaque CPU est doté d'une mémoire centrale avec correction d'erreur simple et détection d'erreurs multiples. En cas de panne de la source d'alimentation, un accumulateur maintient la mémoire.

Le canal d'E/S effectue un test de parité sur chaque donnée transférée et détecte, à l'aide d'un test de minutage ("Watchdog timer")(2.4.4.2), si un contrôleur ne répond plus dans les temps spécifiés.

- Contrôleurs et canaux d'E/S

Les disques sont reliés aux canaux d'E/S par des contrôleurs à double porte. Ces contrôleurs sont connectés aux canaux d'E/S de deux CPUs différents, de manière à ce que le système dispose de deux chemins d'accès distincts à chaque périphérique (figure 5.3). Chaque contrôleur est géré par un seul CPU, l'autre restant capable de prendre le relais si le premier venait à défaillir.

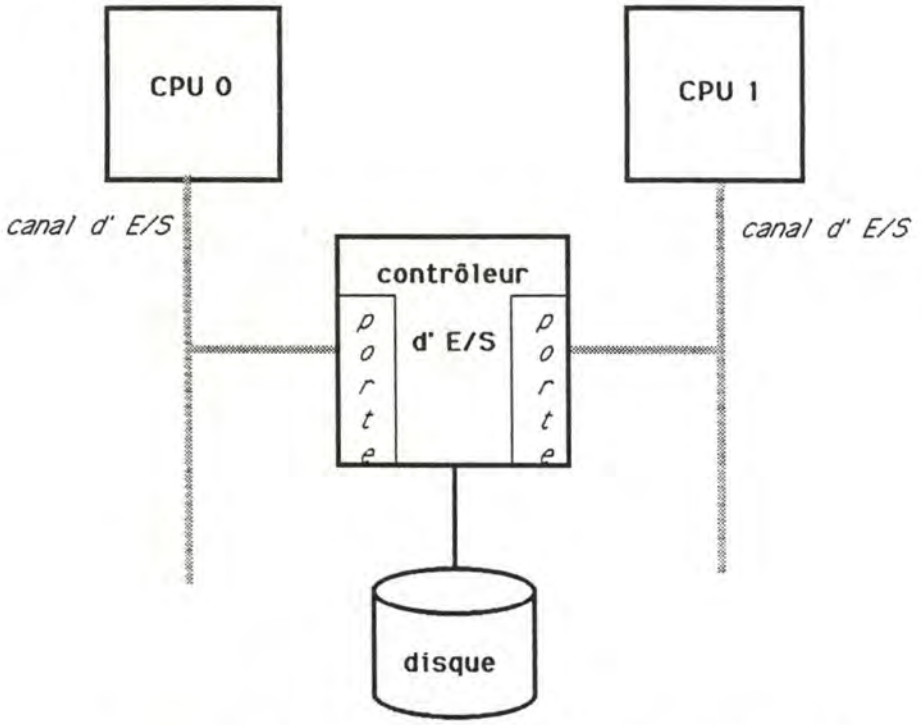

Figure 5.3: Contrôleurs à double porte

- Configuration des disques

Pour permettre de résister à une défaillance du disque, Tandem autorise l'utilisation de disques miroirs. Ceux-ci, comme le montre la figure 5.4, consistent en des paires de disques physiquement indépendants, et généralement attachés à des contrôleurs séparés. Les programmes accèdent à ces paires comme s'il ne s'agissait que d'un seul et même périphérique. Toutes les opérations d'écriture sont répercutées sur chacun des deux disques, et toutes les données peuvent être lues indistinctement de l'un ou l'autre de ceux-ci. Cela permet, non seulement de prémunir le système contre une défaillance des disques, mais encore d'accroître les performances en lecture car le disque choisi pour effectuer l'opération est celui qui minimise le déplacement du bras nécessaire à l'accès de la page demandée.

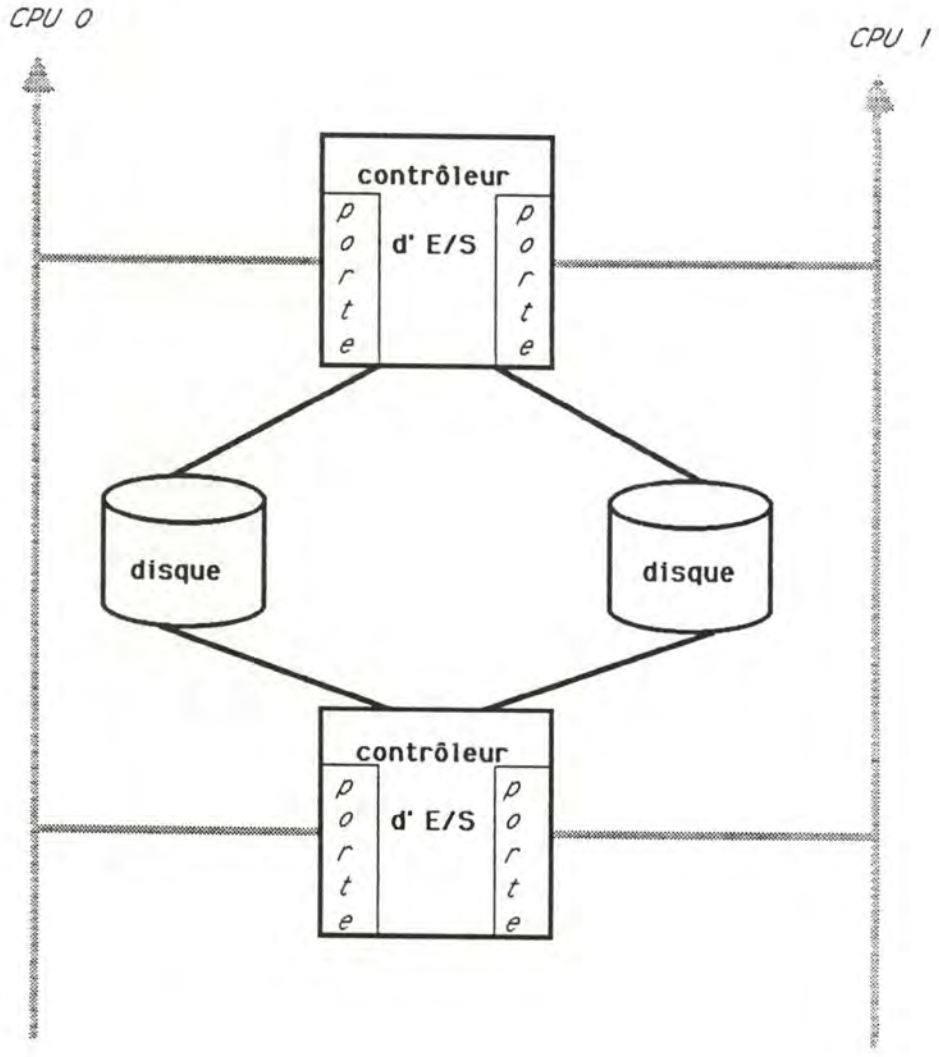

Figure 5.4: Disques miroirs

- De multiples sources d'alimentation

Comme nous le voyons sur la figure 1 de ce chapitre, la distribution d'électricité a été conçue telle que, d'une part, chaque module CPU dispose de sa propre source d'alimentation et, d'autre part, chaque contrôleur d'E/S reçoive de l'énergie de deux sources distinctes (les mêmes sources que les CPUs auxquels il est relié). Ainsi, lorsqu'une de ces sources défaille ou bien est coupée pour réparer le CPU qu'elle approvisionne, aucun contrôleur d'E/S n'est affecté.

Le système a la possibilité d'effectuer la réparation "ON-LINE" sans interruption de n'importe quel composant cité ci-dessus (CPUs, contrôleurs, bus, ...).

5.2 Aspects de la tolérance aux pannes dans le logiciel

Le Système d'Exploitation de Tandem assure toutes les opérations nécessaires à l'exploitation continue.

Tandem adopte la technique des processus "primaires-secondaires" décrite précédemment (2.6.3.2). Lorsqu'un programme exigeant un service continu est lancé, le SE met en route une tâche de secours (tâche secondaire) dans un autre processeur. Le contexte d'exécution (l'état) de celle-ci est mis à jour lors des "checkpoints". En cas d'arrêt de la tâche du primaire, la tâche de secours reprend le travail, mettant en place à son tour une autre tâche de secours. Lorsque l'erreur initiale est corrigée, le SE procède aussitôt à la répartition initiale des charges.

Tous les processus ne sont pas dédoublés. La continuité de service est assurée pour les processus critiques du SE et pour les programmes d'application quand l'utilisateur le désire. Nous reprenons séparément ci-dessous ces deux cas particuliers.

Le SE de Tandem exécute tous les processus d'Entrée/Sortie sous forme de primaire-secondaire. Nous avons vu, dans l'architecture matérielle, qu'il existe en permanence deux chemins d'accès physiques aux données. En cas de défaillance du processeur, du canal d'E/S ou du contrôleur d'E/S supportant les E/S d'un disque particulier, l'autre chemin (CPU, canal, contrôleur) reliant ce disque prend le relais. Pour que cette opération soit possible, il est nécessaire que les processus s'occupant des E/S dans les CPUs soient dédoublés sous forme de primaires-secondaires. Ces derniers sont placés dans les deux CPUs supportant les deux chemins d'accès. Si une défaillance se produit après une écriture sur disque mais avant que le périphérique ait pu signaler sa terminaison, alors le "relais" du Secondaire est suivi d'une vérification que l'enregistrement n'ait pas été copié deux fois de suite.

Le "checkpoint" d'un processus d'E/S contient des informations sur les fichiers qui sont ouverts et fermés. Le "backup" ouvre et ferme ces fichiers alors que le primaire est encore actif, de manière à pouvoir prendre le relais le plus vite possible, sans perdre le temps d'ouvrir les fichiers manipulés.

C'est bien sûr à l'utilisateur de déterminer les programmes d'application qui doivent être dédoublés. Selon qu'il s'agisse d'applications transactionnelles ou non,

l'utilisateur peut disposer d'un environnement lui cachant le mécanisme du "primary-backup". Ce cas particulier est détaillé ci-dessous. Pour assurer la continuité de service, les autres applications (non transactionnelles) doivent explicitement être programmées pour supporter la gestion du Backup. Les langages de programmation fournis aux utilisateurs leur offrent des primitives permettant de programmer explicitement la création d'un "Backup"(Secondaire), son exécution et les prises de "checkpoints".

L'utilisateur a la possibilité d'exécuter ses applications transactionnelles dans l'environnement "PATHWAY". Un langage, le "Screen-Cobol", y est mis à la disposition de l'utilisateur.

Ce langage:

- donne au programmeur Cobol le moyen de concevoir simplement des dialogues par écran,

- permet le déroulement simultané de ses programmes sur un grand nombre de terminaux, en assurant leur interprétation par un processus spécial de Contrôle du Terminal ("TCP") qui garantit le déroulement en continu.

  TCP contrôle les données saisies et assure leur rangement dans l'espace de données internes des programmes utilisateurs.

Afin de comprendre l'apport de TCP pour la continuité de service, il nous faut expliquer le principe des processus "Demandeur et Serveur" de Tandem.

Après avoir lancé une transaction à un terminal, deux types de processus coopèrent pour accomplir les opérations demandées sur la Base de Données: les processus demandeurs et les processus serveurs (figure 5.5). Ces processus communiquent avec les autres via des messages.

- Un processus demandeur gère l'interface avec les terminaux en acceptant des requêtes d'utilisateurs à plusieurs terminaux.

- Un processus serveur gère la Base de Données en servant des requêtes provenant des processus demandeurs.

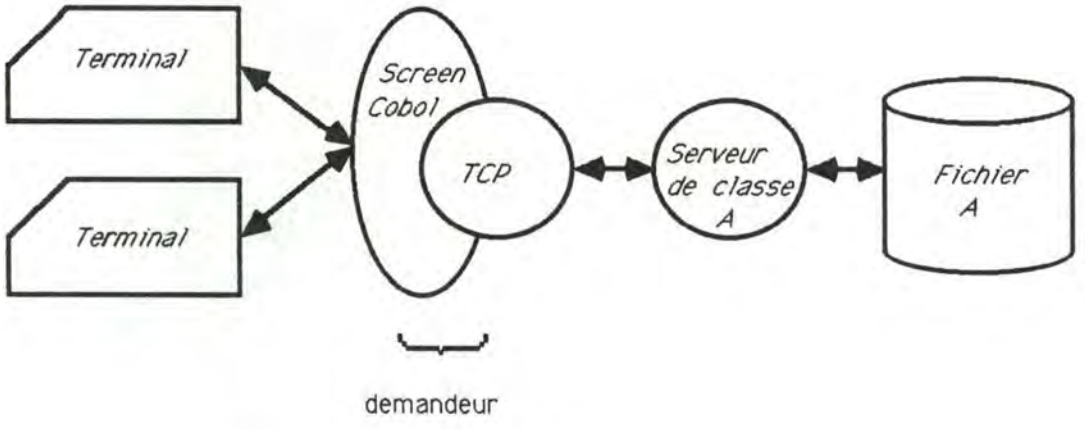

Figure 5.5: Demandeurs-Serveurs dans un environnement PATHWAY

Quand un programme, écrit en Screen-Cobol, initialise une nouvelle transaction ("Begin-Transaction"), le TCP et les processus serveurs coopèrent pour accomplir les actions désirées sur la Base de Données. TCP supporte l'exécution des programmes de tous les terminaux en acceptant leurs messages et en les envoyant aux serveurs appropriés (une transaction peut exiger plusieurs serveurs).

Nous parlons de "processus demandeurs" par abus de langage. En fait, le processus TCP supporte l'exécution des programmes demandeurs (il les interprète). En d'autres termes, il n'y a qu'un seul et gros demandeur: TCP. Ainsi, pour l'exécution de plusieurs programmes demandeurs, nous n'avons besoin que d'un seul backup, celui de TCP.

Les serveurs sont "Context-Free", c'est-à-dire qu'après leur exécution, ils se remettent dans le même état qu'avant leur service. Il n'est donc pas nécessaire de les dédoubler.

L'utilisateur programme les demandeurs et les serveurs sans se soucier de TCP qui, pourtant, lui assure la continuité de service de ses programmes transactionnels.

Le primaire de TCP envoie comme "checkpoint" les données extraites par les programmes écrits en Screen-Cobol.

Afin de détecter une défaillance d'un des CPUs, chacun de ceux-ci envoie toutes les secondes un message "I-AM-ALIVE" aux autres CPUs du système. Réciproquement, chaque CPU vérifie toutes les deux secondes s'il a bien reçu ce message de tous les autres CPUs. Il s'agit d'un test du type "watchdog

timer"(2.4.4.2). Lorsqu'un CPU constate qu'il n'a pas reçu de message "I-AM-ALIVE" d'un autre processeur, il le signale aux autres et déclenche les opérations de traitement de la panne. Les "backups" des processus primaires que supportait le CPU défaillant sont exécutés. D'une manière générale, la charge que supportait ce processeur est distribuée sur les CPUs restants. Cette technique permet de détecter des erreurs de matériel et de logiciel. En effet, dès qu'un CPU détecte une faute de matériel (défaillance), il coupe ses communications avec l'extérieur (les autres CPUs). Il n'envoit donc plus de message "I_AM_ALIVE". De même, une faute de logiciel dans le SE empêche ce dernier d'envoyer le message.

La détection des erreurs s'effectue par le matériel et le logiciel. Pour le matériel, par exemple, une chute de tension est détectée dans un CPU. Le logiciel, quant à lui, utilise un test de minutage entre les CPUs ("I-AM-ALIVE"), mais aussi lors de l'attente des résultats d'une opération sur périphériques. De même, des tests de codification (2.4.4.4) sont utilisés sur chaque chemin de communication et des tests de structure (1.4.4.6) sont régulièrement appliqués sur les structures de contrôle du SE.

Le confinement des erreurs est assuré par le fait que le système est composé de processeurs indépendants disposant de leur propre mémoire et que les processus communiquent par messages.

Le principe de recouvrement est clairement le recouvrement arrière, avec la technique du primaire-secondaire pour les processus, et le recouvrement avant (correction) pour les fautes de mémoire.

Le traitement permanent des fautes s'effectue par reconfiguration sans remplacement, donc avec une dégradation de performances.

5.3 Intégrité des données

L'intégrité des données est basée sur le concept de transaction tel que défini dans la première partie du mémoire.

L'utilitaire de gestion de transaction, "TMF" ("Transaction Monitoring Facility") maintient la cohérence de la base de données à partir du concept de transaction.

Dans un premier temps, nous décrivons comment se comporte une telle transaction dans le système Tandem et ce, sous les trois phases d'une transaction (figure 5.6), à savoir

- phase 1: la transaction est générée et identifiée comme unité de reprise
- phase 2: le corps de la transaction s'exécute, et le TMF place les verrous (demandés, bien sûr, par l'utilisateur dans le programme) pour empêcher des interactions possibles
- phase 3: la transaction est confirmée.

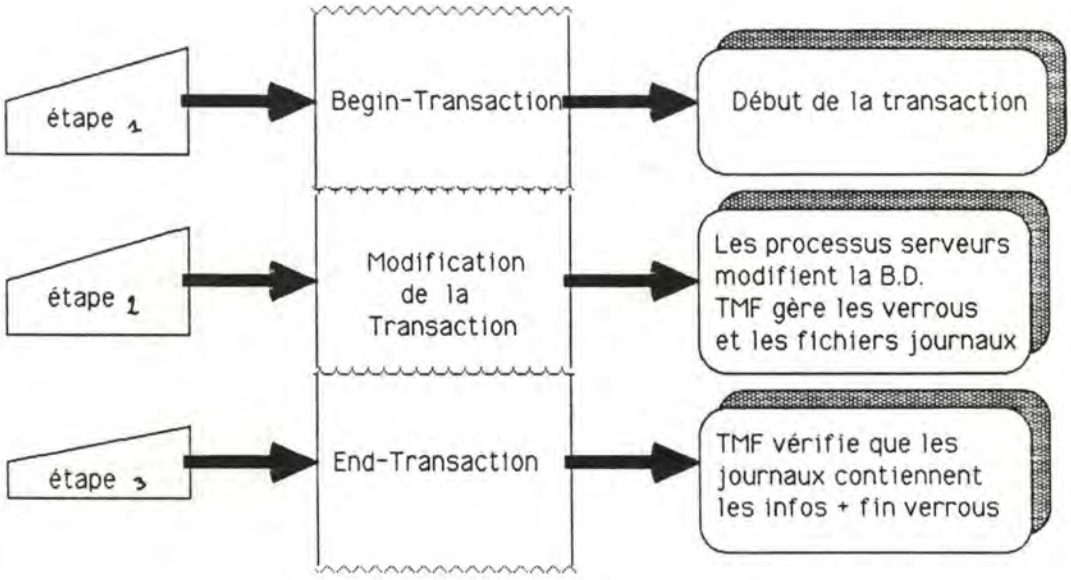

Figure 5.6: Opérations de TMF

Dans un second temps, nous donnons les caractéristiques principales de TMF.

5.3.1 Comportement d'une transaction

5.3.1.1 Identification d'une transaction

Chaque transaction est identifiée de façon unique dans le système et reçoit pour cela un identifiant de transaction, souvent appelé un "TRANSID".

Une routine PATHWAY décèle le début d'une transaction par une instruction spéciale écrite par l'utilisateur dans son programme. En Cobol, par exemple, le verbe "Begin-Transaction", placé dans la "Procedure Division", annonce le début d'une transaction.

Un nouvel identifiant de transaction est alors créé, le terminal où s'exécute le programme se commute en "mode transaction", et le nouvel identifiant est associé au terminal.

5.3.1.2 Transactions et processus

Dans un système TANDEM, un processus est l'unique entité d'exécution créée lorsque le code objet d'un programme est activé.

On peut donc voir un processus comme un programme s'exécutant. Chaque fois que l'on exécute un programme, un processus séparé est créé.

Le système identifie chaque processus par son numéro de CPU et son numéro d'identification (PIN: "Process Identification Number").

L'exécution d'une transaction peut impliquer plusieurs processus, le système doit donc associer à chaque processus l'identifiant de la transaction courante.

Sur la figure 5.7, nous pouvons voir comment le processus de contrôle du terminal (TCP: "Terminal Control Process") et les processus serveurs coopèrent pour effectuer les modifications demandées par la transaction.

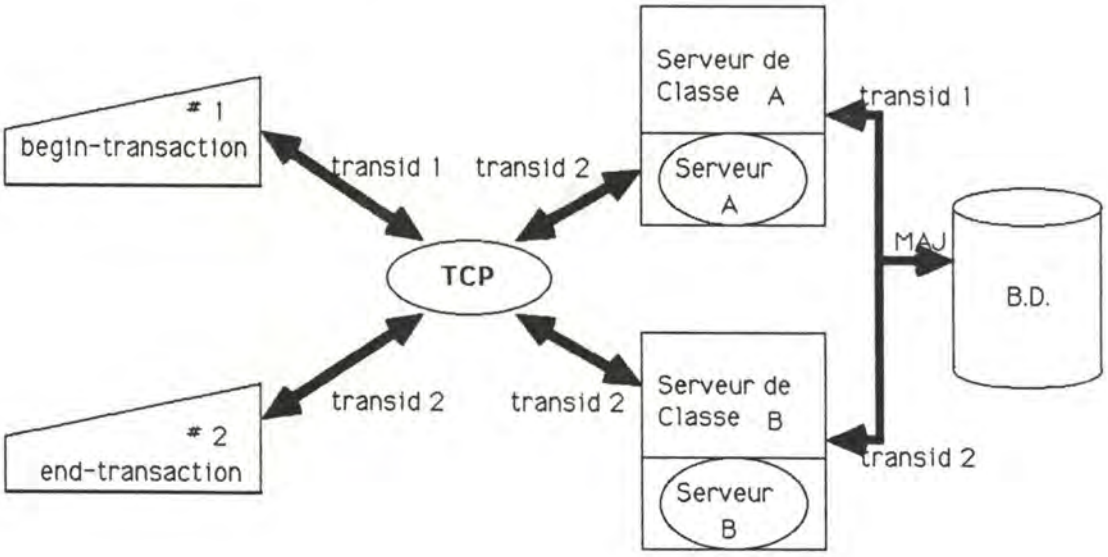

Figure 5.7: Relation entre TCP et processus serveurs

Le TCP et les serveurs communiquent via des messages interprocessus.

Pour assurer que chaque action est identifiée par une transaction spécifique, l'identifiant de transaction est attaché à chaque message envoyé par tout processus. Quand un processus lit le message, l'identifiant de transation du message devient l'identifiant de la transaction courante pour ce processus.

Chaque processus serveur implémente une fonction spécifique (de modification de la base de données, par exemple).

5.3.1.3 Empêchement d'interaction entre transactions

Pour empêcher des interactions entre transactions, TMF impose que l'on suive certaines règles de verrouillage des enregistrements.

1. Une transaction doit verrouiller tous les enregistrements qu'elle met à jour. Les verrous peuvent être posés pour un enregistrement ou pour tout un fichier.

   Chaque verrou est associé à l'identifiant de la

transaction courante.

2. Quand une transaction a supprimé un enregistrement, TMF place un verrou jusqu'à la fin de la transaction (commit), de façon à éviter qu'une autre transaction ne vienne insérer ou modifier un enregistrement ayant la même valeur de clé.

3. TMF verrouille tous les nouveaux enregistrements insérés et tient ces verrous jusqu'à la fin de la transaction.

4. Les enregistrements lus et non modifiés peuvent être verrouillés si le programmeur le désire.

5.3.1.4 Confirmation d'une transaction

La routine PATHWAY identifie la commande de fin de transaction ("End-Transaction" en Cobol).

Quand le fin de transaction est exécuté, TMF confirme les changements effectués par la transaction en deux phases pour assurer la permanence des changements dans le système.

La phase 1 écrit dans les fichiers journaux toutes les images avant et après associées à la transaction confirmée. A ce moment, un enregistrement de confirmation de transaction est écrit dans le fichier journal principal. Cet enregistrement indique que les modifications sont permanentes et doivent donc être reprises après une défaillance du système.

La phase 2 libère tous les verrous associés à la transaction.

5.3.2 Caractéristiques principales de TMF

TMF permet de préserver la cohérence de la base de données. Les images des pages modifiées par une transaction sont sauvegardées dans des fichiers journaux. Ces fichiers journaux permettent de DEFAIRE ("BACKOUT" dans la terminologie TANDEM, ce qui correspond à notre UNDO théorique) les transactions qui ont échoué. TMF assure aussi la copie incrémentale de la base de données (DUMP) tout en autorisant l'utilisation normale de la base de données. En cas de défaillance, TMF restaure la base de données dans le dernier état cohérent du système.

Nous allons donc décrire TMF en analysant cinq de ses fonctions principales, à savoir:

1) la tenue des journaux ("Audit Trails")

2) le DEFAIRE d'une transaction ("Transaction Backout")

3) la reprise arrière automatique ("Autorollback")

4) les copies "on-line" ("Online Dump")

5) la reprise avant ("Rollforward").

5.3.2.1 La tenue des journaux

Comme déjà expliqués, les fichiers journaux contiennent de l'information sur les status des transactions et sur les modifications effectuées sur la base de données (figure 5.8).

La base de données et les fichiers journaux se trouvent sur des supports différents afin d'augmenter la sécurité des données.

Dans le système NonStop de TANDEM, nous avons un fichier journal principal avec un certain nombre d'autres fichiers journaux de données.

Ce journal principal contient des enregistrements numérotés (estampillés) sur

- la confirmation d'une transaction ("Commit")

- l'abandon en cours d'une transaction ("Abort")

- la relâche des verrous

- et les enregistrements de données, c'est-à-dire les images avant et après des pages modifiées (ce type d'enregistrement se trouve aussi dans les fichiers journaux de données).

L'ensemble est géré par le moniteur de transaction, le TMP (Transaction Monitor Process), qui crée les fichiers quand cela est nécessaire, qui les vide, etc ...

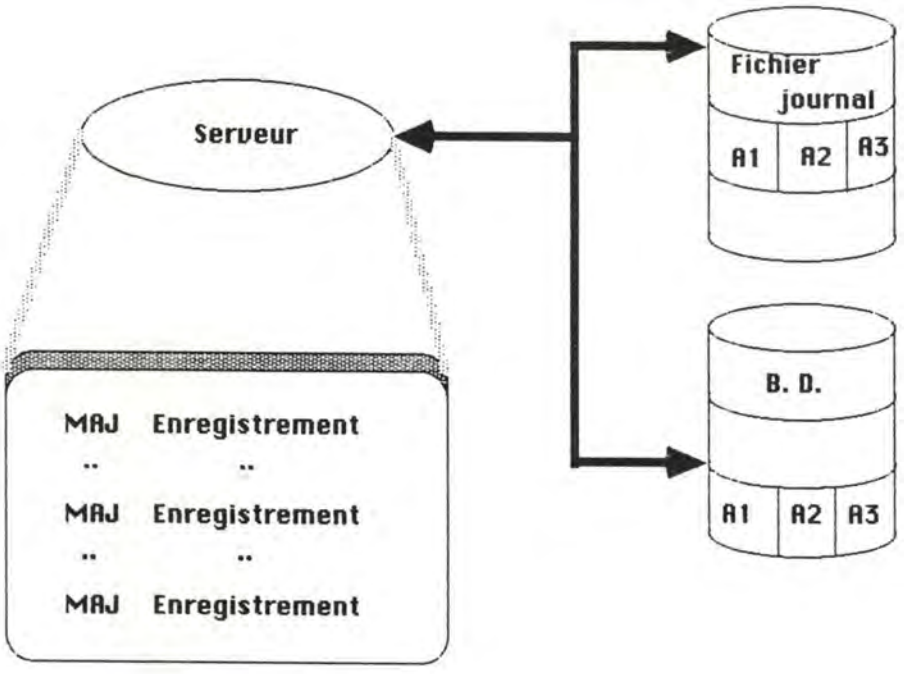

Figure 5.8: Fichiers journaux

5.3.2.2 Le DEFAIRE d'une transaction

Le DEFAIRE d'une transaction ("transaction backout") utilise les images avant des fichiers journaux pour annuler les effets d'une transaction avortée.

TMF permet de défaire une transaction tout en préservant la cohérence de la base de données grâce au respect de trois facteurs:

(1) le mécanisme de contrôle de concurrence qui empêche les interactions entre transactions,

(2) le mécanisme du "Two Phase Commit Protocol",

(3) les fichiers journaux.

Une transaction est défaite lorsqu'un événement arrête la transaction avant que l'enregistrement de confirmation ne soit écrit sur le fichier journal principal.

Cet événement peut être une décision du programme d'application, une défaillance d'un composant, ou une intervention d'un opérateur.

Un processus de remise en état ("Backout Process") de la transaction réécrit les images avant sur les fichiers qui ont été affectés par la transaction en question. Après la restauration des fichiers, les verrous de la transaction sont libérés.

5.3.2.3 La reprise arrière automatique

La reprise arrière automatique de TANDEM ("Autorollback") correspond au Défaire Généralisé de la théorie.

Elle s'applique pour des incidents mineurs (les plus fréquents) et permet une reprise de la base de données relativement rapide (le temps de reprise dépendant du nombre de fichiers à restaurer, de la longueur des transactions, ...).

Quand TMF est déclenché après une défaillance du système, la reprise arrière est initialisée automatiquement pour tous les fichiers ouverts au moment de la défaillance.

Les fichiers non concernés, ou ceux déjà repris, peuvent aussitôt être utilisés par d'autres transactions.

Quand la reprise arrière automatique est achevée, TMF fournit une liste de fichiers pour lesquels une reprise arrière n'a pas été possible. Ces fichiers doivent être restaurés par la reprise avant ("Rollforward").

5.3.2.4 Les copies On-Line

La copie "on-line" permet en fait de copier des fichiers journaux sur disque ou sur bande tout en continuant l'application.

Cette fonction permet donc de faire des copies de sauvegarde contre des défaillances totales du système.

5.3.2.5 La reprise avant

Lorsqu'un incident majeur se produit, ou que la reprise arrière automatique échoue pour certains fichiers, nous devons effectuer une reprise avant.

Les étapes à suivre sont les suivantes:

1. TMF demande à l'opérateur de monter les bandes ou les disques pour la reprise, c'est-à-dire les bandes ou les disques qui contiennent la base de données dans un état cohérent déterminé.

2. Dans un deuxième temps, le processus de reprise avant remet les fichiers dans le dernier état sauvegardé ("online dump").

   Si la copie s'est fait sur disque, le fait de le monter suffit pour restaurer les fichiers du disque. Pour la copie sur bande, par contre, TMF détermine quels fichiers journaux étaient ouverts au moment de la panne et copie ces fichiers, remplaçant les fichiers "endommagés".

3. L'étape suivante permet d'appliquer les images avant et après des fichiers journaux. Sont réexécutées les transactions terminées au moment de la panne, les autres sont ignorées.

4. Quand un fichier est repris, un message donne son statut et indique qu'il peut de nouveau être ouvert.

Chapitre 6

Le Système :

STRATUS

d' IBM

6 Le système STRATUS d'IBM

6.1 Aspects de la tolérance aux pannes dans le matériel

Dans le système Stratus [STRA87], la tolérance aux fautes est quasi entièrement réalisée par des moyens matériels.

Un système Stratus est un réseau composé de 1 à 32 modules duplex de traitement, où chaque module est composé de deux CPUs, deux mémoires, deux contrôleurs de bus interprocesseurs ("StrataLINK"), deux contrôleurs de disques et deux contrôleurs de communications avec les terminaux. La figure 6.1 schématise un module duplex de traitement.

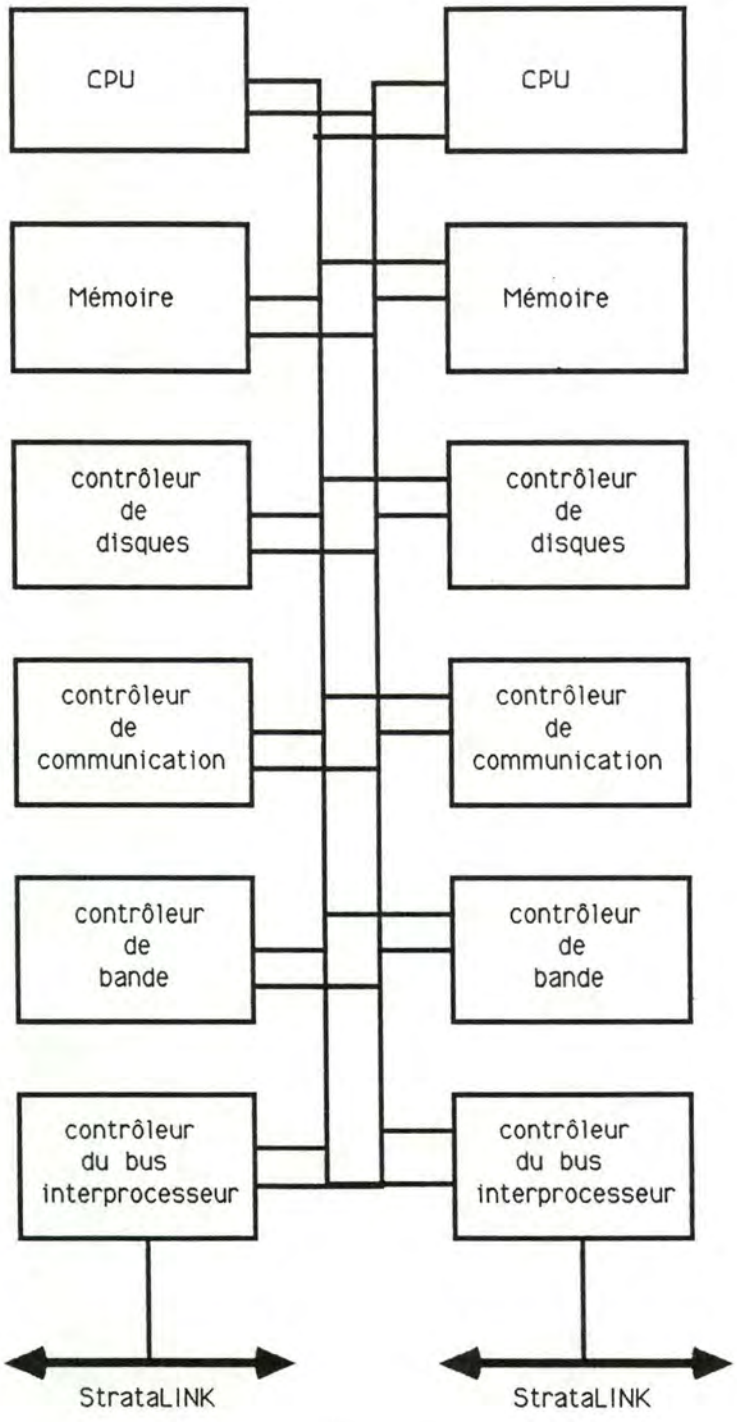

Figure 6.1: module duplex de traitement

Chacun des composants cités ci-dessus est "Self Checking"(2.4.2), c'est-à-dire qu'il s'auto-teste. La figure 6.2 modèlise la relation qui existe entre composants identiques (par exemple deux CPUs).

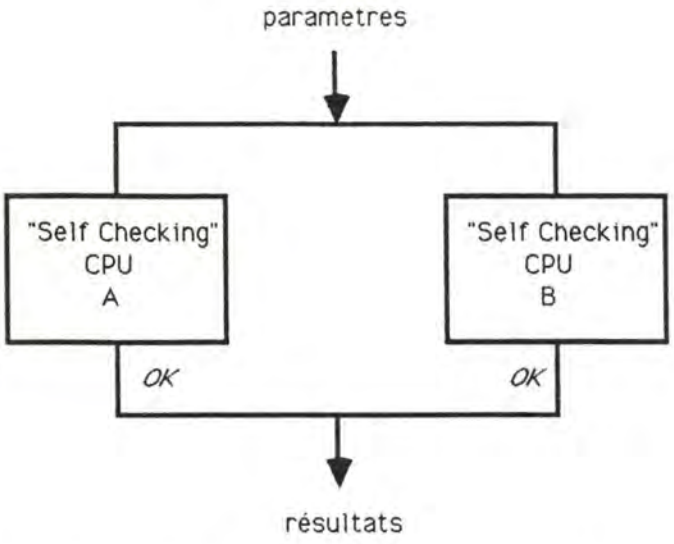

Figure 6.2: Modèle de relation entre deux composants identiques

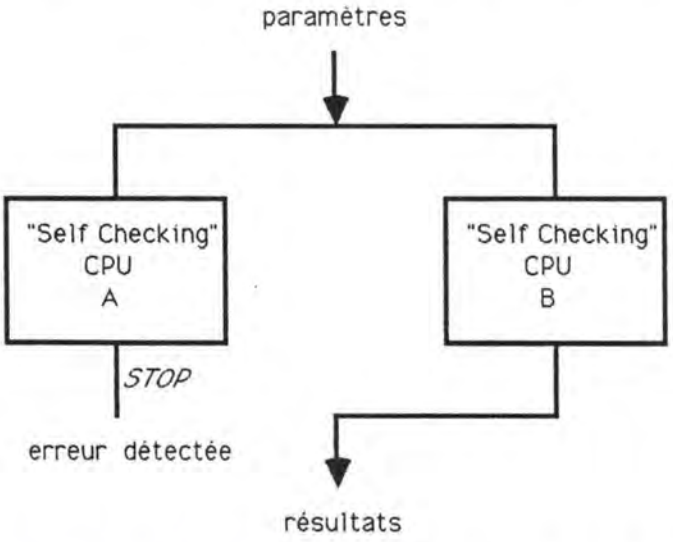

Figure 6.3: même modèle quand une erreur est détectée

Comme le montre la figure 6.3, lorsqu'un composant se détecte en erreur, il se déconnecte du système et laisse l'autre composant envoyer seul ses résultats.

La capacité de "Self_Checking(SC)" repose sur l'utilisation de redondance dynamique (2.2) et d'un circuit de comparaison. Un composant SC est constitué de deux circuits logiques identiques effectuant chaque opération en parallèle, et d'un circuit de comparaison vérifiant que les résultats concordent.

La figure 6.4 illustre comment s'effectue ce test dans un contrôleur de disque, de manière à ce qu'aucune donnée erronée ne soit écrite sur le disque. Nous y remarquons la

"passerelle" qui déconnecte le contrôleur en cas d'erreur détectée sur le circuit de comparaison.

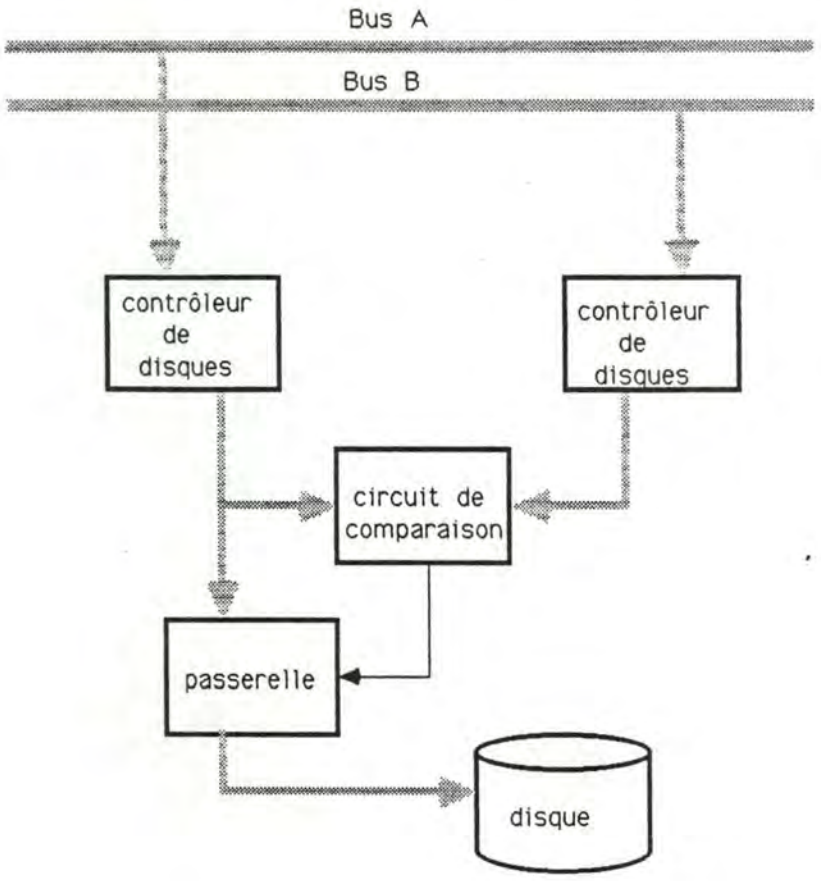

Figure 6.4: Contrôleur de disque SC

En résumé, les systèmes Stratus sont construits autour de deux concepts [KIM84]:

- les composants matériels SC pour détecter les fautes (cas de redondance dynamique)
- le dédoublement des composants matériels pour permettre la tolérance aux pannes (cas de redondance statique).

Toutes les opérations de calcul, de stockage et d'E/S sont réalisées en parallèle par la duplication matérielle. Chaque composant (CPU, mémoire, contrôleur)

d'un module se teste tous les cycles de l'horloge à l'aide de deux circuits identiques et d'un circuit de comparaison. Si la comparaison note une différence entre les résultats, le circuit de comparaison arrête instantanément le composant. Il n'en résulte aucune dégradation du temps de réponse puisque les résultats sont alors pris du composant restant et que celui-ci continue son travail.

La figure 6.5 illustre avec un exemple très simple, ce mécanisme appliqué aux CPUs.

Toute erreur est rapportée à un logiciel de maintenance de SE ("VOS") qui détermine, à l'aide d'un diagnostic, la cause et la nature de la faute. S'il s'avère que la faute est transitoire, le composant est remis en service. Si, par contre, la faute est permanente, les résultats du diagnostic sont conservés sur un fichier pour la maintenance manuelle, et un message est envoyé à l'opérateur (ou même à un "Centre d'Assistance Stratus").

Les systèmes Stratus permettent de faire de la maintenance à distance (à partir d'un Centre d'Assistance par exemple).

Chaque module, tel qu'on l'a défini ci-dessus, possède plusieurs sources d'alimentation.

Stratus supporte optionnellement des "disques miroirs"(5.1.1). Il permet aussi la réparation "ON-LINE" des composants matériels défectueux. Lors d'une opération de maintenance, si un composant dédoublé est remplacé, alors le nouveau composant est automatiquement mis dans le même état que le composant restant. Dans un système de disques miroirs par exemple, le nouveau disque est mis-à-jour alors que les opérations d'E/S s'adressent au disque restant.

Les seules erreurs de logiciel dont Stratus tient compte sont celles détectées par l'interpréteur (1.1.2).

Par sa solution basée sur le matériel, les concepteurs de Stratus pensent procurer un service continu et assurer l'intégrité de la Base de Données.

Pour situer cette solution matérielle par rapport aux différentes phases du traitement des fautes identifiées dans le chap 2:

- La détection est réalisée par le "Self Checking"
- Il y a un recouvrement avant par la technique du vote.

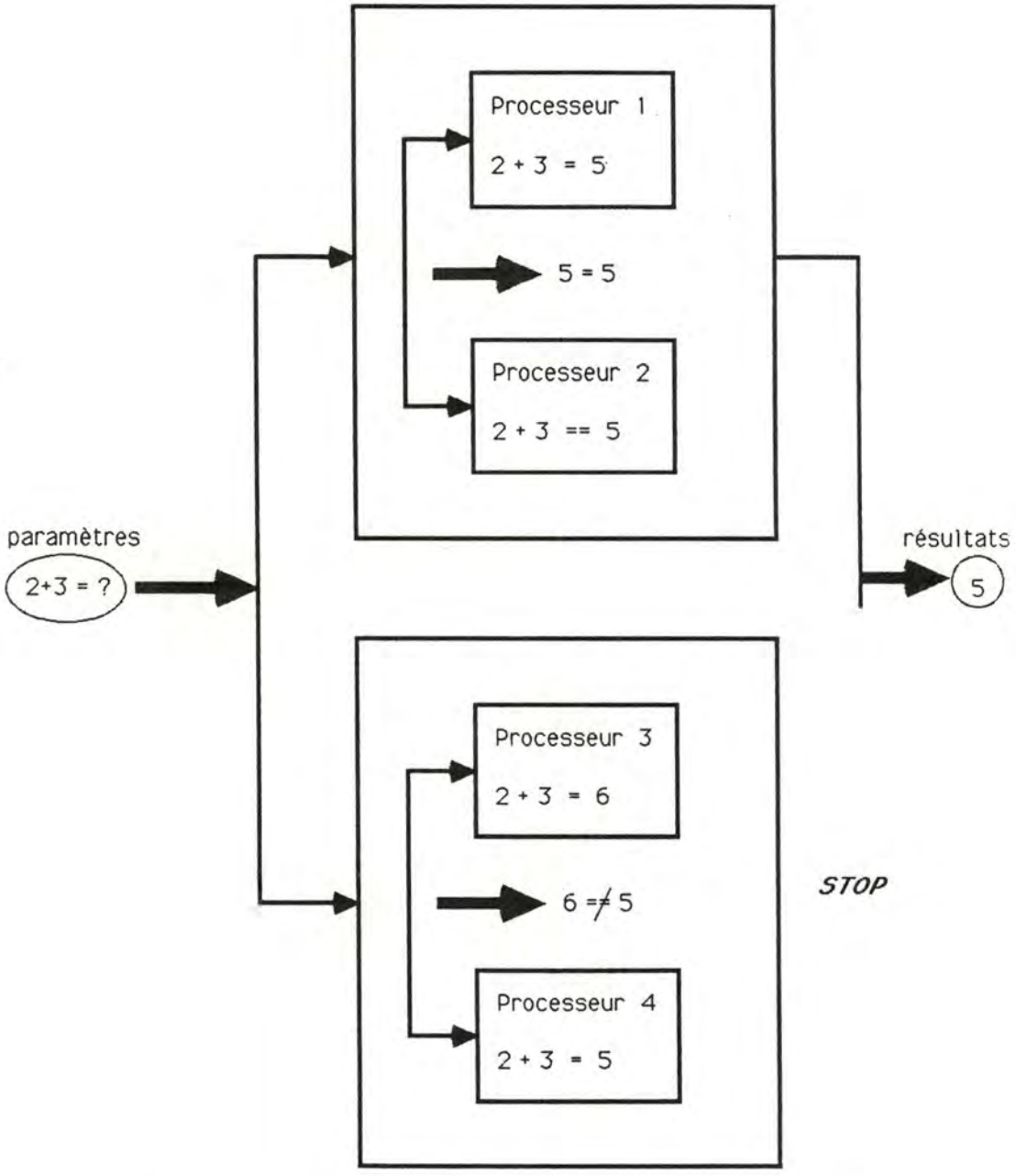

Figure 6.5: Mécanisme de comparaison de STRATUS

Le traitement de la faute est du type momentané ou permanent, en fonction des résultats de la localisation de la faute qui précède. Cette localisation s'effectue par diagnostic et détermine si la faute est de nature transitoire ou permanente. Dans le premier cas, le traitement est momentané (par l'utilisation du vote qui "ignore" le résultat fautif), et dans le second, le traitement est permanent (par la déconnection du composant fautif).

6.2 Intégrité des données

Comme nous l'avons vu ci-dessus, la continuité de service est assurée uniquement par la duplication du matériel. Il en est de même pour l'intégrité des données. Aucun fichier de sauvegarde ("Backup") n'est entretenu.

Le système STRATUS ne supporte pas directement un Système de Gestion de Base de Données. En fait, les Bases de Données sont gérées via le système des fichiers grâce à l'utilitaire "TPF" ("Transaction Processing Facility").

En cas de défaillance ou d'annulation d'une transaction, celle-ci n'a pas d'effet sur la base de données si elle n'a pas été confirmée (Commit). C'est le principe de l'indivisibilité d'une transaction.

TPF s'occupe aussi de la gestion de verrous pour assurer un certain contrôle de la concurrence dans les accès. Disons simplement que TPF permet de verrouiller des fichiers, des enregistrements ou même des clés et que ces verrous sont libérés au "Commit" de la transaction par le principe du "Two Phase Commit Protocol" déjà décrit.

Comment TPF assure-t-il l'atomicité des transactions ?

Le principe est assez simple, il ressemble à celui de la "Mise à jour retardée". Quand une transaction est entamée, les modifications sont effectuées dans un fichier spécial ("log file"), et les fichiers de la base de données concernés sont non accessibles par les autres processus. A la confirmation de la transaction (Commit) et uniquement à ce moment-là, toutes les modifications sont reportées sur les fichiers "verrouillés" et lorsque cela est terminé, les verrous sont enlevés et les fichiers sont de nouveau accessibles par d'autres transactions.

Si une défaillance ou un abandon de la transaction survient avant le Commit, seul le fichier "Log" est affecté et la base de données reste intègre.

Chapitre 7

Critères

d' Evaluation

7 Critères d'évaluation

Dans ce dernier chapitre, nous identifions un certain nombre de critères d'évaluation des systèmes à haute fiabilité et nous y situons les deux systèmes étudiés.

Voici, selon Won Kim [KIM84], cinq critères que doivent respecter les systèmes orientés BD pour être qualifiés de "hautement fiables".

1. Le système doit assurer la cohérence de la Base de Données, c'est-à-dire:

   - permettre le traitement des transactions
   - contrôler les accès concurrents
   - avoir la possibilité d'effectuer des reprises.

   TANDEM : OUI par TMF

   STRATUS : OUI par TPF

2. Suite à une défaillance, le système doit pouvoir effectuer une reprise automatique de toute la charge par un processus de "backup"

   TANDEM : OUI par la technique du Primaire-Secondaire ("primary-backup")

   STRATUS : NON mais il assure la continuité de service par la technique du Vote

3. Le système doit survivre à une faute simple (défaillance) dans un composant matériel majeur tel que:

   - le processeur
   - le canal d'E/S
   - les disques
   - le moyen de communication interprocesseur

Comme nous l'avons vu, ce critère est satisfait par la duplication de ces composants dans les deux architectures (TANDEM et STRATUS).

4. Le système doit permettre une intégration "ON-LINE" des composants matériels et logiciels nouveaux (ou réparés).

TANDEM : OUI

STRATUS : OUI

5. L'apparition des fautes simples doit être cachée aux utilisateurs.

TANDEM : la défaillance d'un composant se traduit par une dégradation des performances de l'ensemble du système.

STRATUS : il n'y a pas de dégradation des performances

PS: nous remarquons que la partie "intégrité de données" de la tolérance aux pannes fait uniquement l'objet du premier point, alors que la "continuité de service" est reprise dans plusieurs des points suivants.

Voici, pour terminer, un ensemble de critères supplémentaires.

- A quels types de fautes le système peut-il résister ?
  - TANDEM :
    - les fautes de matériel (vieillissement des composants, environnement)
    - les fautes de logiciel (application et SE)
  - STRATUS :
    - les fautes de matériel (vieillissement des composants, environnement)
- La propriété de <u>croissance modulaire</u>
  - TANDEM : OUI

- STRATUS : OUI

- Quelles sont les applications à continuité de service ?
  - TANDEM :
    - les applications transactionnelles (d'office),
    - les autres applications (du SE par exemple) programmées explicitement comme telles
  - STRATUS : toutes

- Le caractère optionnel de la continuité de service
  - TANDEM : OUI
  - STRATUS : NON

- Temps de traitement des fautes (temps de reprise)
  - TANDEM : NON-NUL
  - STRATUS : NUL

- Facilités de maintenance
  - STRATUS :
    - existance d'un Centre d'Assistance aux utilisateurs
    - circuits physiques auto-contrôlés

Conclusion

Conclusion

Les concepts de base à la tolérance aux fautes dans les systèmes informatiques ont été définis dans le chapitre 1.

De nombreuses techniques ont été élaborées pour assurer une continuité de service (chapitre 2).

Nous pouvons remarquer que les réalisations pratiques sont bien au point pour ce qui est du matériel.

Au point de vue du logiciel, des méthodologies sont proposées (chapitre 3); mais la complexité toujours croissante des systèmes logiciels freine la mise en pratique des techniques de construction de logiciels tolérants aux fautes.

Les problèmes d'intégrité des bases de données sont complètement résolus, en long et en large dans la littérature, mais aussi en pratique (chapitre 4).

Nous suggérons comme extension de cette partie "théorique" des études plus "analytiques" sur les performances des différentes solutions proposées, ainsi qu'une mise en pratique des modèles de tolérance aux fautes dans les logiciels. En particulier, nous proposons l'étude des reprises pour des processus coopérants.

Pour ce qui est de la partie "application" de ce mémoire, nous avons décrit deux systèmes, Tandem (chapitre 5) et Stratus (chapitre 6), et nous en avons donné les principales possibilités (chapitre 7).

Cette présentation des deux systèmes les plus connus sur le marché reste quand même fort descriptive et donc assez "théorique".

Nous proposons comme suite à donner au travail une analyse de ces systèmes dans des cas concrets d'utilisation comme dans les banques pour les systèmes de distribution automatique de billets (Bancontact, Mister Cash, ...).

Une évaluation des performances (disponibilité) des deux systèmes dans des applications semblables semble intéressante.

Bibliographie

Annotée

Bibliographie Annotée

[AGRA85] AGRAWAL, R., DEWITT, D.J. "Integrated Concurrency Control and Recovery Mechanism: Design and Performance Evaluation". ACM Transactions on Database Systems, 10, 4 (December 1985), 529-564.

Après une brève présentation des mécanismes de contrôle de concurrence et de reprise, l'interaction entre ceux-ci est analysé. Ensuite, des modèles analytiques sont développés pour l'étude du comportement et pour la comparaison des performances des mécanismes intégrés. Les résultats de l'évaluation des performances sont présentés.

[ANDE78] ANDERSON, T., LEE, P.A., SHRIVASTAVA, S.K. "A Model of Recoverability in Multilevel Systems". IEEE Transactions on Software Engineering, SE-4, 6 (November 1978), 486-494.

Cet article analyse les solutions apportées pour permettre la reprise arrière d'objets abstraits dans une système à multi-couches. Deux schémas de reprise (disjoint et inclusif) sont présentés pour l'implémentation d'objets recouvrables.

[ANDE79] ANDERSON, T., RANDELL, B. "Computing Systems Reliability". Computing Reviews, 1979.

Ce livre reprend un certain nombre d'articles sur la disponibilité des systèmes informatiques. Nous ont plus particulièrement intéressés les chapitres 5 et 6 sur la tolérance aux fautes, et 9 sur la disponibilité des logiciels.

[ANDE81] ANDERSON, T., LEE, P.A. "Fault-Tolerance: Principles and Practice". Prentice Hall Internationnal, 1981.

Ce livre reprend l'ensemble de la problématique de la tolérance aux fautes. Nous nous sommes plus particulièrement intéressés aux chapitres 6, 7, 8 et 9 consacrés aux techniques de tolérances aux fautes.

[BAYE76] BAYER, R. "On the Integrity of Data Bases and Resssource Locking". Data Base Systems, Proc. 5th Int. Symp. IBM Germany, Bad Hombourg. Lecture Notes in Computer Science, 39, 1976, 339-361.

Cet article, après avoir posé le problème de l'intégrité des bases de données, propose des techniques de verrouillage des ressources, et plus particulièrement du verrouillage des objets individuels et des prédicats. Plusieurs stratégies pour éviter l'attente indéfinie ou infinie des transactions sont proposées.

[BAYE80] BAYER, R., HELLER, H., REISER, A. "Parallelism and Recovery in Database Systems". ACM Transactions on Database Systems, 5, 2 (June 1980), 139-156.

Une nouvelle méthode pour accroître le parallélisme dans les bases de données est décrite. Le principe est de prendre avantage de deux versions possibles d'un objet, déjà existantes pour des raisons de reprise. L'accès en lecture est ainsi toujours permis.

[BAYE82] BAYER, R., ELHARDT, K., HEIGERT, J., REISER, A. "Dynamic Timestamp Allocation and its Applications to the BEHR-Method".
Cet article présente une technique de synchronisation pour les systèmes de gestion de base de données distribuées, qui combine l'emploi du mécanisme de l'estampillage avec celui des graphes pour résoudre les verrous mortels.

[BERN79] BERNSTEIN, P.A., SHIPMAN, D.W., WONG, W.S. "Formal Aspects of Serializability in Database Concurrency Control". IEEE Transactions on Software Engineering, SE-5, 3 (May 1979), 203-216.
Un modèle formel, basé sur l'emploi d'un journal des transactions, est développé pour l'analyse et la comparaison des comportements de différents mécanismes de contrôle de cohérence dans une base de données. Un système centralisé simple qui permet une opération de lecture et une opération d'écriture par transaction est présenté pour l'analyse.

[BERN81] BERNSTEIN, P.A., GOODMAN, N. "Concurrency Control in Distributed Database Systems". ACM Computing Surveys, 13, 2 (June 1981), 185-221.
Les problèmes de la synchronisation et de la concurrence pour les bases de données distribuées sont analysés. Plusieurs solutions sont proposées dont l'estampillage et le protocole de verrouillage en deux phases.

[BERN84] BERNSTEIN, P.A., GOODMAN, N. "An Algorithm for Concurrency Control in Replicated Distributed Databases". ACM Transactions on Database Systems, 9, 4 (December 1984), 596-615.
Cet article propose un algorithme de contrôle de concurrence et de reprise pour un certain nombre de défaillances dans les bases de données distribuées. L'utilisateur peut travailler sur les données tant qu'une copie est disponible.

[BRAE85] BRAEGGER, R.P., DUDLER, A.M., REBSAMEN, J., ZEHNDER, C.A. "Gambit: An Interactive Database Design Tool for Data Structures, Integrity Constraints, and Transactions". IEEE Transactions on Software Engineering, SE-11, 7 (July 1985), 574-583.
Gambit est un outil interactif de conception de bases de données. Il est basé sur un modèle Entité/Association. Il aide l'utilisateur à décrire des structures de données et à conserver la cohérence

grâce au concept de transaction.

[CAMP86] CAMPBELL, R.H., RANDELL, B. "Error Recovery in Asynchronous Systems". IEEE Transactions on Software Engineering, SE-12, 8 (August 1986), 811-826.
Cet article présente des techniques de reprise arrière et de reprise avant pour la construction de systèmes logiciels tolérants aux fautes. La notion d'action atomique est introduite pour permettre des interactions entre processus.

[CARD83] CARDENAS, A.F., ALAVIAN, F., AVIZIENIS, A. "Performance of Recovery Architectures in Parallel Associative Database Processors". ACM Transactions on Database Systems, 8, 3 (September 1983), 291-323.
Après une classification de tous les événements indésirables qui peuvent arriver dans un environnement base de données, et après un résumé de l'information redondante nécessaire pour la reprise, un modèle de processeurs parallèles et associatifs orientés base de données est présenté. Pour trois différents types d'architectures de reprise, les charges supplémentaires imposées par les mécanismes de reprise sont analysées.

[CARD87] CARDINAEL, J.P. Notes de cours "Gestion des Ressources Informatiques". 1987 FNDP.
Cours donné en troisième licence et maîtrise en informatique aux FUNDP de Namur présentant les problèmes de gestion des ressources humaines et matérielles d'un grand centre informatique.

[CASA80] CASANOVA, M.A., BERNSTEIN, P.A. "General Purpose Schedulers for Database Systems". Acta Informatica, 14 (March 1980), 195-220.
L'emploi de GPS dans les bases de données permet la sérialisation faible des opérations ce qui assure la cohérence. Il facilite aussi le redémarrage en cas de défaillance.

[CASA81] CASANOVA, M.A. "The Concurrency Control Problem for Database Systems". Lecture Notes in Computer Science, 116 (1981).
Ce livre traite entièrement le problème des accès concurrentiels à une base de données. L'approche transactionnelle est préconisée.

[CHAN75] CHANDY, K.M., BROWNE, J.C., DISSLY, C.W., UHRIG, W.R. "Analytic Models for Rollback and Recovery Strategies in Data Base Systems". IEEE Transactions on Software Engineering, SE-1, 1 (March 1975), 100-110.
Cet article présente des modèles et des techniques qui aident à déterminer les intervalles de temps optimaux pour l'activation des points de reprise. Un point de reprise est un point dans le temps où une copie des données ou des fichiers est

faite sur disque, permettant la reprise. La reprise est assurée par un fichier journal.

[CRIS82] CRISTIAN, F. "Exception Handling and Software Fault Tolerance". IEEE Transactions on Computers, C-31, 6 (June 1982), 531-540.
Après une présentation des concepts de base pour la construction de logiciels tolérants aux fautes, l'article donne une méthode de gestion des exceptions basée sur le recouvrement arrière.

[CURT77] CURTICE, R.M. "Integrity in Data Base Systems". Datamation (May 1977), 64-68.
Cet article propose une synthèse sur l'intégrité des bases de données. Il donne d'abord des solutions pour assurer l'intégrité du système pendant l'exécution normale. Ensuite, il décrit comment résoudre les problèmes en cas de défaillance.

[CURT86] CURTICE, R.M. "Getting the Database Right". Datamation (October 1986), 99-104.
L'auteur pose le problème du choix d'une base de données adaptée à l'environnement, c'est-à-dire à son utlisation.

[DATE81] DATE, c. "An Introduction to Database Systems". Addison Wisley (1981).
Ce livre est un des documents fondamentaux en matière de base de données. Les chapitres 1 sur la reprise, 2 sur l'intégrité, et 3 sur la concurrence sont particulièrement intéressants dans notre contexte.

[DAVI80] DAVIES, D.W., HOLLER, F., JENSEN, E.D. "Distributed Systems Architecture and Implementation". Lecture Notes in Computer Science, 105 (1980).
Nous avons retenu de ce livre le chapitre 11 entièrement consacré à la notion de transaction atomique.

[DENN78] DENNIS MICKUNAS, M., MODRY, J.A. "Automatic Error Recovery for L.R. Parsers". Communications of the ACM, 21, 6 (June 1978), 459-465.
Cet article présente un schéma de détection et de reprise automatique d'erreurs de syntaxe dans les programmes.

[EAST78] EASTON, M.C., FAGIN, R. "Cold-Start vs. Warm-Start Miss Ratios". Communications of the ACM, 21, 10 (October 1978), 866-872.
Etude sur la manière de mesurer le nombre d'Entrées/Sorties de pages des tampons dans une hiérarchie de stockage à deux niveaux (tampons et disques).

[EFFE84] EFFELSBERG, W., HAERDER, T. "Principles of Database Buffer Management". ACM Transactions on Database Systems, 9, 4 (December 1984), 560-595.

Cet article propose d'inclure le gestionnaire des tampons de la base de données comme composant d'un système de gestion de base de données. Les différences principales entre la pagination de la mémoire virtuelle et le gestionnaire des tampons de la base de données sont expliquées.

[ESWA76] ESWARAN, K.P., GRAY, J.N., LORIE, R.A., TRAIGER, I.L. "The Notions of Consistency and Predicate Locks in a Database System". Communications of the ACM, 19, 11 (November 1976), 624-633.

Les notions de transaction, de cohérence, et de verrouillage sont formellement définies par un modèle de données abstrait. Ensuite, il est démontré que la cohérence exige que les transactions soient en deux phases et bien formées. Le verrouillage logique basé sur des prédicats résoud le problème du verrouillage d'enregistrements inexistants ou fantômes dans un environnement base de données.

[FRAN85] FRANASZEK, P., ROBINSON, J.T. "Limitations of Concurrency in Transaction Processing". ACM Transactions on Database Systems, 10, 1 (March 1985), 1-28.

Cet article présente une étude analytique des possibilités de concurrence entre transactions.

[GELE78] GELENBE, E., DEROCHETTE, D. "Performance of Rollback Recovery Systems under Intermittent Failures". Communications of the ACM, 21, 6 (June 1978), 493-499.

Cet article décrit un modèle mathématique d'un système orienté transaction avec des défaillances intermittentes. Des points de reprise et une méthode de reprise arrière assure une continuité de service. Le modèle est employé pour donner des mesures de performance, dont la disponibilité, le temps de réponse et le point de saturation du système.

[GIBB76] GIBBONS, I.T. "Integrity and Recovery" in Computer Systems (1976).

Ce livre reprend l'ensemble des problèmes liés à l'intégrité des données dans un système d'information et présente certains mécanismes de reprise.

[GILL85] GILLENSON, M.L. "Database Step-By-Step". IBM Systems Research Institute. A Wiley-Interscience Publication, (1985).

Ce livre reprend en détail les étapes de construction d'une base de données, sans oublier bien sûr les problèmes de concurrence et d'intégrité.

[GRAY79] GRAY, J.N. "Notes on Data Base Operating Systems". In Operating Systems: An Advanced Course, R. BAYER, R.M. GRAHAM, and G. SEEGMULLER, Eds., Springer-Verlag, New York, 1979, 393-481.

Après une brève présentation de la gestion des données, ce rapport aborde les problèmes de verrouillage et de reprise dans un environnement transactionnel. Une méthode complète et unifiée est présentée et comparée à des approches alternatives. Cette méthode est largement basée sur celle du système R.

[GRAY81] GRAY, J.N., MCJONES, P., BLASGEN, M., LINDSAY, B., LORIE, R., PRICE, T., PUTZOLU, F., TRAIGER, I. "The Recovery Manager of the System R Database Manager". ACM Computing Surveys, 13, 2 (June 1981), 223-242.

Cet article donne une description et une évaluation du gestionnaire de reprise du système R. Le concept de transaction permet aux programmes d'application de confirmer, d'avorter ou de défaire de façon partielle leurs effets. Le protocole DO-UNDO-REDO permet de nouveaux types de reprise. Les programmes d'application peuvent enregistrer des données dans un fichier journal des transactions pour faciliter la reprise d'applications spécifiques. Le mécanisme du point de reprise est basé sur les fichiers différentiels.

[GRIF85] GRIFFETH, N., MILLER, J.A. "Performance Modeling of Database Recovery Protocols". IEEE Transactions on Software Engineering, SE-11, 6 (June 1985), 564-572.

Les modèles de performance décrits dans cet article comparent plusieurs protocoles qui assurent que la base de données peut être remise dans un état cohérent après une défaillance ou une panne du système.

[HAER78] HAERDER, T. "Implementierung von Datenbanksystemen". Carl Hanser Verlag München Wien (1978).

Ce livre, écrit en Allemand, reprend l'ensemble de la problématique de l'intégrité des bases de données, et propose des solutions basées sur l'approche transactionnelle.

[HAER83] HAERDER, T., REUTER, A. "Principles of Transaction-Oriented Database Recovery". ACM Computing Surveys, 15, 4 (December 1983), 287-317.

Le premier objectif de cet article est d'établir un terminologie adéquate et précise. Une architecture terminologique est donnée pour décrire différents schémas de reprise orientés transaction pour les bases de données. Le tout est complété par une classification des techniques de journalisation.

[HAIN87] HAINAUT, J.L. Notes de cours "Technologie des fichiers". 1987 FNDP.

Le cours de technologies des fichiers, donné en troisième licence et maîtrise en informatique aux FUNDP de Namur, présente les problèmes d'intégrité, de reprise et de concurrence dans les bases de données et propose des éléments de solution.

[HINX84] HINXMAN, A. I. "Updating a Database in an Unsafe Environment". Communications of the ACM, 27, 6 (June 1984), 564-566.
Investir des ressources considérables pour maintenir la base de données s'avère parfois superflu. Une simple technique, utilisant un fichier de données partiel, protège la base de données si l'enregistrement immédiat d'une transaction n'est pas essentiel.

[JALO86] JALOTE, P., CAMPBELL, R.H. "Atomic Actions for Fault-Tolerance Using CSP". IEEE Transactions on Software Engineering, SE-12, 1 (January 1986), 59-68.
CSP est utilisé pour implémenter des actions atomiques ce qui assure la tolérance aux pannes par les mécanismes de reprise avant et reprise arrière.

[JUN82] JUN LU, D. "Watchdog Processors and Structural Integrity Checking". IEEE Transactions on Computers, C-31, 7 (July 1982), 681-685.
Cet article décrit l'emploi de processeurs "Chien-de-garde" dans l'implémentation de tests d'intégrité structurelle. Un modèle est donné.

[KAUN84] KAUNITZ, J., VAN EKERT, L. "Audit Trail Compaction for Database Recovery". Communications of the ACM, 27,7 (July 1984), 678-683.
Cet article présente une technique de compactage du journal par utilisation d'une table binaire afin d'accélérer la reprise en cas de défaillance.

[KIES83] KIESSLING, W., PFEIFFER, H. "A Comprehensive Analysis of Concurrency Control Performance for Centralized Databases".
Cet article analyse et évalue au moyen de modèles de simulation le comportement des différentes méthodes de contrôle de concurrence dans les bases de données. On peut ainsi évaluer le surcoût amené par ces mécanismes de synchronisation.

[KIM84] KIM, W. "Highly Available Systems for Database Applications". ACM Computing Surveys, 16,1 (March 1984), 71-98.
Cet article présente une classification des techniques utilisées pour rendre les systèmes tolérants aux fautes et décrit certains systèmes existants comme TANDEM et STRATUS.

[KOHL81] KOHLER, W.H., "A Survey of Techniques for Synchronization and Recovery in Decentralized

Computer Systems". ACM Computing Surveys, 13, 2 (June 1981), 149-183.
Cet article présente différentes techniques de synchronisation (verrouillage, estampillage, ..) pouvant s'appliquer aux problèmes d'accès concurrentiels à une base de données. Ensuite, il donne des mécanismes de reprise en cas de défaillance.

[KORE86] KOREN, I., KOREN, Z., SU, S.Y.H. "Analysis of a Class of Recovery Procedures". IEEE Transactions on Computers, C-35, 8 (August 1986), 703-712.
Cet article présente une procédure génerale de reprise qui réessaie des instructions et reprend le programme au dernier ou plus proche point de reprise. A cause de la complexité de la procédure générale, seuls trois cas particuliers ont été analysés complètement.

[KRIS84] KRISHNA, C.M., SHIN, K.G., LEE, Y.H. "Optimization Criteria for Checkpoint Placement". Communications of the ACM, 27, 10 (October 1984), 1008-1012.
Cet article présente certaines formules mathématiques pour maximiser le temps de reprise en fonction du surcoût engendré par le placement de points de reprise.

[KUNG81] KUNG, H.T., ROBINSON, J.T. "On Optimistic Methods for Concurrency Control". ACM Transactions on Database Systems, 6, 2 (June 1981), 213-226.
Cet article propose deux méthodes dites optimistes pour le contrôle de concurrence aux accès dans une base de données. Ces techniques se révèlent moins coûteuses que celles du verrouillage.

[LAMS85] van LAMSWEERDE, A. Notes de cours "Méthodologie de développement de logiciels". 1985 FNDP.
Le cours de méthodologie de développement de logiciels est donné en deuxième licence et maîtrise en informatique aux FUNDP de Namur. Il présente différents concepts méthodologiques pour la conception de gros logiciels, dont la structuration hiérarchique (UTILISE) et la décomposition modulaire.

[LAUS85] LAUSEN, G., SOISALON-SOININEN, E., WIDMAYER, P. "Pre-Analysis Locking: A Safe and Deadlock Free Locking Policy". Proceedings of VLDB 85, Stockholm, 270-281.
Cet article présente une technique de pose de verrous de façon à éviter les verrous mortels. Cette technique consiste en fait à construire un graphe géométrique des verrous. Une pré-analyse permet de déceler à l'avance le risque d'un interblocage.

[LEUN79] LEUNG, J.Y.T., LAY, E.K. "On Minimum Cost Recovery from System Deadlock". IEEE Transactions on

Computers, C-28, 9 (September 1979), 671-677.
Trois heuristiques rapides sont proposées pour reprendre le système en cas de verrouillage mortel, et leurs moins bonnes performances sont analysées.

[LIES86] LIESTMAN, A.L., CAMPBELL, R.H. "A Fault Tolerant Scheduling Problem". IEEE Transactions on Software Engineering, SE-12, 11 (November 1986), 1089-1095.
Cet article propose une méthodologie (et les algorithmes) pour structurer des systèmes temps réels et tolérants aux fautes.

[LORI77] LORIE, R.A. "Physical Integrity in a Large Segmented Database". ACM Transactions on Database Systems, 2, 1 (March 1977), 91-104.
Un mécanisme de reprise, basé sur la maintenance de deux versions des pages modifiées, la page courante et la page précédente, est présenté. Le surcoût nécessaire pour maintenir les pages s'avère être relativement peu élevé.

[MARY80] MARYANSKI, F.J., CHAROENPONG, P. "An Architecture for Fault-Tolerance in Database Systems". ACM (1980), 389-398.
Cet article propose une architecture pour la tolérance aux fautes dans les systèmes de gestion de base de données basée sur les concepts de fichiers différentiels et de remplacement prudent. Un modèle de simulation est proposé pour une analyse des performances.

[MENA80] MENASCE, D.A., POPEK, G.J., MUNTZ, R.R. "A Locking Protocol for Resource Coordination in Distributed Databases". ACM Transactions on Database Systems, 5, 2 (June 1980), 103-138.
Cet article présente un protocole de verrouillage pour l'accès à des données dans une base de données distribuées.

[MOHA82] MOHAN, C., FUSSEL, D., SILBERSCHATZ, A. "A Biaised Non-Two-Phase Locking Protocol". Department of Computer Sciences, University of Texas at Austin, 337-361.
Cet article présente un protocole de verrouillage différent du 2 PLP. Il est basé sur un mécanisme de conversion des verrous. Il assure la sérialisabilité mais ne garantit pas l'absence de deadlock. D'autres mécanismes de détection doivent être prévus.

[MOHA85] MOHAN, C., FUSSEL, D., KEDEM, Z.M., SILBERSCHATZ, A. "Lock Conversion in Non-Two-Phase Locking Protocols". IEEE Transactions on Software Engineering, SE-11, 1 (January 1985), 15-22.
Cet article donne un protocole de verrouillage différent du 2 PLP qui se base sur l'établissement

d'un graphe. Les verrous mortels sont décelés par l'apparition d'un cycle.

[MOND85] MOND, Y., RAZ, Y. "Concurrency Control in B+ Trees Databases Using Preparatory Operations". Proceedings of VLDB 85, Stockholm, 331-334.
Le principe de contrôle de concurrence présenté ici repose sur la structure en arbre de la base de données. Seuls le noeud concerné et son parent sont verrouillés ce qui garantit plus de parallélisme que d'autres méthodes.

[RAND75] RANDELL, B. "Systeme Structure for Software Fault Tolerance". IEEE Transactions on Software Engineering, SE-1, 2 (June 1975), 220-232.
Une méthode pour atteindre des logiciels tolérants aux fautes est proposée. Le but est d'assurer la détection des erreurs et de donner des techniques de reprise en utilisant une technique de structuration des systèmes appelée "bloc de reprise" permettant de spécifier des tests d'acceptation et des composants de programme alternatifs. Des systèmes à haute disponibilité sont implémentés par la construction en couches de machines virtuelles.

[RAND78] RANDELL, B., LEE, P.A., TRELEAVEN, P.C. "Reliability Issues in Computing System Design". ACM Computing Surveys, 10, 2 (June 1978), 123-165.
Cet article présente les différents problèmes rencontrés pour atteindre un haut degré de disponibilité pour des systèmes informatiques complexes. Il discutte aussi des relations existantes entre les techniques de structuration des systèmes et les techniques de tolérance aux pannes. Trois systèmes existants sont alors étudiés.

[RENN84] RENNELS, D.A. "Fault-Tolerant Computing -- Concepts and Examples". IEEE Transactions on Computers, C-33, 12 (December 1984), 1116-1128.
Après une brève historique des ordinateurs tolérants aux fautes, l'article suit par une étude des différentes architectures de systèmes tolérants aux fautes et par une description des concepts de base.

[REUT80] REUTER, A. "A Fast Transaction-Oriented Logging Scheme for UNDO Recovery". IEEE Transactions on Software Engineering, SE-6, 4 (July 1980), 348-356.
L'algorithme TWIST ("Twin Slot") est décrit. Dans cet algorithme, un segment logique correspond à deux blocs physiques. L'espace disque est doublé, mais le UNDO est facilité par le mécanisme de la page fantôme et demande un minimum d'opérations Entrées/Sorties en plus.

[REUT81] REUTER, A. "Fehlerbehandlung in Datenbanksystemen". Carl Hanser Verlag München Wien (1981).
Ce livre écrit en Allemand décrit entièrement les mécanismes de reprise des bases de données centralisées orientées-transaction.

[REUT84] REUTER, A. "Performance Analysis of Recovery Techniques". ACM Transactions on Database Systems, 9, 4 (December 1984), 526-559.
Différentes techniques de journalisation et de reprise pour des bases de données centralisées orientées transaction sont décrites et leurs performances sont analysées à l'aide de modèles analytiques eux-mêmes étudiés en fin d'article.

[RIES77] RIES, D.R., STONEBRAKER, M. "Effects on Locking Granularity in a Database Management System". ACM Transactions on Database Systems, 2, 3 (September 1977), 233-246.
Cet article tente d'analyser les effets de la granularité des verrous sur un système de gestion de base de données dans des conditions simples et normales.

[RUSH85] RUSHINEK, S., RESHINEK, A. "Backup and Recovery in Accounting Information Systems". Data Processing, 27, 3 (April 1985), 42-46.
Cet article examine la nécessité des techniques de sauvegarde et de reprise dans les systèmes d'information financiers. L'importance des contrôles de sécurité est mise en évidence.

[RYPK79] RYPKA, D.J., LUCIDO, A.P. "Deadlock Detection and Avoidance for Shared Logical Resources". IEEE Transactions on Software Engineering, SE-5, 5 (September 1979), 465-471.
Des modes sont définis pour spécifier la manière dont les processus peuvent accéder concurremment à des ressources. Un mode compatible est employé pour spécifier la détection d'un interblocage.

[SCHE82] SCHEUERMANN, P. Improving Database Usability and Responsiveness (1982).
Ce livre compare un certain nombre de systèmes de gestion de base de données (ADABAS, IDMS, IMS, ROBOT, TOTAL, SYSTEM 2000) selon certains critères dont ceux qui nous intéressent sont l'intégrité et la reprise.

[SCHL78] SCHLAGETER, G. "Process Synchronization in Database Systems". ACM Transactions on Database Systems, 3, 3 (September 1978), 248-271.
Cet article décrit de façon systématique les problèmes de synchronisation (sérialisation) dans les bases de données.

[SERL83] SERLIN, O. "Exploring the OLTP Realm". Datamation (1983), 60-68.
Après la description des applications transactionnelles, l'article décrit le marché des machines assurant de telles applications dont TANDEM et STRATUS.

[SHAM86] SHAMBHU UPADHYAYA, J., SALUJA, K.K. "A Watchdog Processor Based General Rollback Technique with Multiple Retries". IEEE Transactions on Software Engineering, SE-12, 1 (January 1986), 87-95.
Pour la plupart des défaillances, le simple réessai d'un programme depuis le point de reprise précédent n'est pas suffisant. Cet article propose une stratégie globale de reprise avec essais multiples, avec l'emploi d'un processeur "chien de garde" comme outil de détection d'une erreur pour le lancement de l'action de reprise.

[SHIN84a] SHIN, K.G., LEE, Y.H. "Error Detection Process. Model, Design, and Its Impact on Computer Performance". IEEE Transactions on Computers, C-33, 6 (June 1984), 529-540.
Cet article propose un modèle pour la détection des erreurs, qui permet de mesurer l'impact de technique de détection sur la performance d'un système informatique.

[SHIN84b] SHIN, K.G., LEE, Y.H. "Evaluation of Error Recovery Blocks Used for Cooperating Processes". IEEE Transactions on Software Engineering, SE-10, 6 (November 1984), 692-700.
Cet article présente trois alternatives pour implémenter les blocs de reprise. Ce sont les implémentations asynchrones, synchrones, et avec point de pseudo-reprise. Des modèles probabilistes sont développés pour analyser les trois méthodes sous des hypothèses standards.

[SINH85a] SINHA, M.K., NASTARAJAN, N. "A Priority Based Distributed Deadlock Detection Algorithm". IEEE Transactions on Software Engineering, SE-11, 1 (January 1985), 67-80.
Cet article contribue au développement des techniques pour la gestion des transactions en décrivant un algorithme de détection des interblocages pour les systèmes à bases de données distribuées. Cet algorithme se base sur l'établissement de priorités.

[SINH85b] SINHA, M.K. "Atomic Actions and Resource Coordination Problems Having Nonunique Solutions". IEEE Transactions on Software Engineering, SE-11, 5 (May 1985), 461-471.
Le concept d'action atomique est décomposé en action atomique dépendant de la base de données et

en action atomique dépendant de l'application. Cette approche permet de solutionner les problèmes de coordination des ressources et ce, de différentes façons.

[SIEW82] SIEWIOREK, D.P., SWARZ, R.S. "The Theory and Practice of Reliable System Design". Digital Press, 1982.
La première partie de ce livre décrit une taxonomie des techniques de fiabilité et des modèles mathématiques pour les évaluer. La seconde partie donne des illustrations de techniques de design et une méthodologie de design pour les systèmes fiables.

[STRA87] Stratus.
Notes prises au cours d'un exposé aux FUNDP de NAMUR en avril 1987. + Documentation constructeur donnée par le présentateur.

[TAND87] Tandem.
Notes prises au cours d'un exposé aux FUNDP de NAMUR en avril 1987. + Documentation constructeur donnée par le présentateur.

[TAY85] TAY, Y.C., GOODMAN, N., SURI, R. "Locking Performance in Centralized Databases". ACM Transactions on Database Systems, 10, 4, (December 1985), 415-462.
Un modèle analytique est présenté pour étudier les performances de verrouillage dynamique dans une base de données centralisée. L'article met bien en évidence l'influence de la granularité des verrous.

[TAYL86] TAYLOR, D.J. "Concurrency and Forward Recovery in Atomic Actions". IEEE Transactions on Software Engineering, SE-12, 1 (January 1986), 69-78.
Cet article décrit des méthodes de gestion de la concurrence et de reprise avant à partir des actions atomiques. L'intérêt est la présentation de nombreux exemples.

[TONI75] TONIK, A.B. "Checkpoint, Restart, and Recovery : Selected Annotated Bibliography". ACM Sigmod 7, 3-4 (1975), 72-76.
Cet article contient des résumés de 16 articles sur les points de reprise, le redémarrage et la reprise.

[TRAI82] TRAIGER, I.L., GRAY, J., GALTIERI, C.A., LINDSAY, B.G. "Transactions and Consistency in Distributed Database Systems". ACM Transactions on Database Systems, 7, 3 (September 1982), 323-342.
Les concepts de transaction et de cohérence des données sont définis pour un système réparti. La découpe par abstraction est préconisée pour améliorer la transparence du système.

[TRUE83] TRUEBLOOD, R.P., REX HARTSON, H., MARTIN, J.J. "Multisave - A Modular Multiprocessing Approach to Secure Database Management". ACM transactions on Database Systems, 8, 3 (September 1983), 382-409.

Cet article décrit comment une découpe modulaire dans l'architecture d'un système de gestion de base de données peut augmenter la sécurité des bases de données. Un exemple, MULTISAVE, est décrit entièrement.

[VERH78] VERHOFSTAD, J.S.M. "Recovery Techniques for Database Systems". ACM Computing Surveys, 10, 2 (June 1978), 167-195.

Cet article étudie les techniques employées pour implémenter le "défaire", la reprise de panne, la résistance aux pannes et la cohérence dans les bases de données. Les techniques sont classifiées en fonction de l'environnement dans lequel elles sont employées et du but pour lequel elles sont appliquées.

[WIED83] WIEDERHOLD, G. "Database Design". Computer Science Series (1983).

Nous nous intéressons plus particulièrement aux chapitres 11 sur la disponibilité des systèmes informatiques et 13 sur l'intégrité des données. Des méthodes pour augmenter la disponibilité et assurer l'intégrité des données y sont décrites.

[WULF75] WULF, W.A. "Reliable Hardware/Software Architecture". IEEE Transactions on Software Engineering, SE-1, 2 (June 1975), 233-240.

Cet article se propose de donner une méthodologie de système à haute fiabilité tant d'un point de vue logiciel que matériel. Cette méthodologie se base sur la structuration modulaire.

[YAU80] YAU, S.S., CHEN, F.C. "An Approach to Concurrent Control Flow Checking". IEEE Transactions on Software Engineering, SE-6, 2 (March 1980), 126-137.

Cet article décrit un schéma de test de flux de contrôle concourants qui vérifie la cohérence des flux de contrôle des programmes s'exécutant avec ceux prévus lors de leur création. Ainsi, durant l'exécution d'un programme, une erreur du code (logiciel), un malfonctionnement du matériel, ou une erreur de mémoire peuvent être détectés.