UNIVERSITÉ DE SHERBROOKE

Département d'informatique
Faculté des sciences

IFT 630 - Processus concurrents et parallélisme

Chapitre 9

Fiabilité

Gabriel Girard[1]

Sherbrooke

18 octobre 2022

[1] Gabriel.Girard@usherooke.ca

Table des matières

1 Fiabilité **5**
1.1 Objectifs et terminologie 5
1.2 Éviter les fautes (Fault avoidance) 8
1.2.1 Éviter les fautes matérielles 8
1.2.2 Éviter les fautes logicielles 8
1.2.3 Fautes des usagers 10
1.2.4 Conclusion 11
1.3 Détecter les erreurs 11
1.3.1 Détection par l'environnement 11
1.3.2 Détection par l'application 11
1.3.3 Mécanismes de protection 14
1.4 Traitement des fautes 14
1.5 Reprise 15
1.5.1 Reprise arrière 16
1.5.2 Reprise avant 18
1.6 Traitement d'erreurs à plusieurs niveaux 21
1.7 Étude de cas 23

Appendices **25**

Annexe A Raid et mémoire stable **25**
A.1 RAID 25
A.1.1 RAID 0 - Entrelacement par bandes 26
A.1.2 RAID 1 - Disques miroirs 28
A.1.3 RAID 2 - Code correcteur d'erreurs 28
A.1.4 RAID 3 - Entrelacement par bandes (octets) et bits de parité 29
A.1.5 RAID 4 - Entrelacement par bandes (blocs) et bits de parité 31
A.1.6 RAID 5 32
A.1.7 RAID 6 32
A.2 Mémoire stable 33
A.3 Conclusion 35

Annexe B Autostabilisation **37**
B.1 Introduction . . . 37
B.2 Définition de l'auto-stabilisation . . . 37
B.2.1 Exemple : anneau à jeton . . . 39
B.3 Prémisses à l'utilisation de l'auto-stabilisation . . . 40
B.4 Environnements propices à l'auto-stabilisation . . . 40
B.5 Conception d'un algorithme auto-stabilisateur . . . 41
B.6 Exemple d'algorithme : anneau à jeton . . . 42
B.7 Conclusion . . . 45

Chapitre 1

Fiabilité

L'une des caractéristiques les plus recherchées d'un système d'exploitation est sans surprise sa fiabilité. Il est impératif qu'elle soit prévue dès les premières étapes de la conception du système. La fiabilité n'est pas un «ajout», elle est conçue à même le système.

1.1 Objectifs et terminologie

La fiabilité est définie par le *degré avec lequel un système logiciel (et en particulier le système d'exploitation sous-jacent) rencontre ses spécifications du point de vue du service à l'usager, même en présence de conditions non prévues et hostiles.*

Ainsi le concepteur d'un système doit s'assurer que les fonctions fournies par celui-ci sont celles prévues et qu'elles s'exécutent correctement. La tâche du concepteur est rendue difficile par le fait que le système opère dans un monde imparfait. Ce dernier rencontrera un grand nombre de circonstances qui affecteront son bon fonctionnement :

- un mauvais fonctionnement du matériel ;
- des erreurs humaines dans les procédures ;
- des demandes illégales ou insensées de la part des usagers ;
- etc.

Un système doit être outillé de manière à pouvoir continuer à fournir ses services, même dégradés, lorsqu'il fait face à ce genre de problèmes.

Bien sûr, **la fiabilité est un concept relatif**. Il faut être conscient qu'une fiabilité parfaite (tout continue à fonctionner à 100% dans toutes les situations) est pratiquement impossible (prenez comme exemple, le cas extrême où TOUS les composants d'un système sous-jacent se révèlent défectueux au même moment). Un système très fiable continuera à rencontrer ses spécifications même en présence de plusieurs pannes matérielles ou autres. Un système peu fiable s'éloignera de ses spécifications à la moindre erreur.

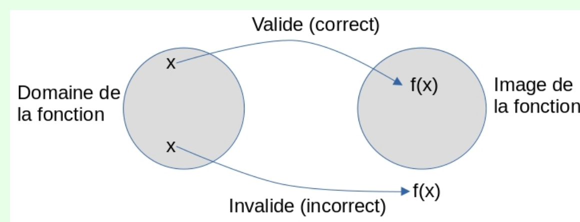
De plus, la **fiabilité dépend des besoins**. Certains systèmes requièrent un très *haut niveau de fiabilité*. C'est le cas des moniteurs des hôpitaux qui vérifient la condition des patients, des navettes spatiales, des systèmes téléphoniques, etc. D'autres systèmes ne nécessitent qu'un *faible niveau de fiabilité* (voire nul) tels les environnements de bureautique servant à la préparation et la recherche de documents.

La questions a son importance puisqu'un haut niveau de fiabilité implique des coûts plus élevés (duplication du matériel ou du logiciel et redondance des données).

Une distinction importante est à noter dès à présent. Celle entre un système dit **correct** et celui qualifié de **fiable**. Il est **correct** lorsque les résultats produits sont ceux attendus selon les données (valides) fournies en entrée. Ainsi «être correct» est un attribut désirable pour un logiciel mais c'est une condition insuffisante pour assurer la fiabilité. En effet, pour montrer qu'un système est correct, par tests ou preuves, on se fie sur des a priori sur l'environnement notamment, qui ne s'avèrent pas toujours vrais. Par exemple, on suppose que les entrées sont exemptes d'erreurs (i.e **correctes** ou **valides**) ou que le matériel ne sera jamais affecté par une panne.

Définition : Programme correct ou valide

Un programme (ou logiciel) est dit valide (exact ou correct) si pour des données valides en entrée il produit un résultat valide. La figure suivante illustre ce concept.



En fait, être un système correct n'est pas seulement une condition insuffisante à sa fiabilité mais elle n'est pas nécessaire. Il est fort possible que certaines parties du logiciel soient incorrectes, en ce sens que les algorithmes gouvernant certains processus ne produisent pas l'effet désiré, mais qu'il continue à fonctionner correctement. Il est donc fiable même si certains de ces composants sont incorrects.

Exemple de système incorrect mais fiable

Supposons un système de fichiers pour lequel l'opération de fermeture (*close*) «oublie», dans certaines circonstances, d'enregistrer la longueur du fichier dans le répertoire. Si le fichier termine par une marque de fin de fichier, les routines d'entrée/sortie sont tout de même aptes à lire sans problème les fichiers. Le système fonctionnera donc fiablement. Cet exemple illustre bien que l'information redondante peut servir à cacher des fautes ou à assurer la reprise lors de fautes.

Ainsi, un système peut être incorrect, mais fiable (ainsi qu'être correct mais non fiable). Ceci dit, développer des systèmes corrects a son importance. Même si cette condition n'est ni nécessaire ni suffisante pour assurer la fiabilité, elle aide. En effet, il est évident qu'un système correct sera probablement plus fiable qu'un qui ne l'est pas. De plus, ce serait là une mauvaise pratique pour un concepteur que de dépendre des mécanismes de fiabilité pour cacher l'effet de déficiences ou même justifier le manque d'efforts pour produire un système correct. Le bon sens exige que le système soit conçu pour être correct et fiable.

Le terme **erreur** désigne toute déviation d'un système de son comportement spécifié. Une erreur est donc un événement détectable telle l'allocation d'une ressource non partagée à plus d'un processus ou la destruction de l'entrée d'un fichier encore utilisée. Une erreur peut être due à un mauvais fonctionnement du matériel, à une action imprévue d'un usager ou à une déficience (bug) d'un ou plusieurs programmes du système.

Définition : erreur

Le terme **erreur** désigne toute déviation d'un système de son comportement spécifié. Une erreur est donc un événement détectable.

La cause d'une erreur est appelée une **faute**. Celle-ci identifie un état, alors qu'une erreur est un événement. Ainsi, le mauvais fonctionnement d'un programme (faute) entraînera souvent une erreur. Il est possible cependant qu'une faute n'entraîne aucune erreur. Elle est donc indétectable.

Définition : faute

Une faute désigne la cause d'une erreur.

Lors d'une erreur, il est fort probable que certains éléments d'information du système soient corrompus. Le **dommage** d'un système est la corruption causée par une erreur. Les dommages à leur tour entraînent généralement des fautes qui elles produiront d'autres événements erronées (erreurs). On verra donc qu'une façon d'obtenir une haute fiabilité consiste à éviter les dommages qui peuvent être causés par une erreur et, du même coup, limiter la propagation de l'erreur.

Définition : dommage

Le dommage d'un système est la corruption causée par une erreur.

Donc, afin de produire un système fiable, nos efforts doivent se porter sur un des domaines suivants :

1. **éviter les fautes** ;

   Cela consiste en l'élimination des fautes aux étapes de conception et d'implantation (produire un système correct).

2. **détecter les erreurs** ;

   Cela consiste à prévoir des mécanismes de détection d'erreur efficaces, i.e. qui détectent les erreurs le plus tôt possible afin d'éviter les dommages.

3. **traiter les fautes** ;

   Lorsque des erreurs sont détectées, il faut identifier et éliminer les fautes qui produisent l'erreur.

4. **effectuer la reprise**.

   Lors d'une erreur, on doit évaluer et réparer les dommages causés afin de permettre à l'exécution de reprendre correctement.

1.2 Éviter les fautes (Fault avoidance)

Une faute peut provenir de trois différentes sources, soit matérielle, logicielle ou humaine.

1.2.1 Éviter les fautes matérielles

Il est relativement difficile d'éviter une faute matérielle. On choisit plutôt d'intégrer des mécanismes pour les masquer et ainsi obtenir des composants plus fiables. Voici certains procédés utilisés à cette fin :

- la mémoire ECC ;
- les opérations répétées ;
- les sommes de contrôle (checksum) ;
- les codes correcteurs ;
- le vote majoritaire (majority polling).

1.2.2 Éviter les fautes logicielles

Éviter les fautes logicielles impliquent le recours à un ensemble de techniques conçues pour produire des programmes sans faute.

Les différentes approches afin d'éviter les fautes logicielles ne sont pas mutuellement exclusives. Au contraire, l'application combinée de celles-ci permet de renforcer la fiabilité du logiciel. Les trois principales sont basées sur (1) des méthodes structurées pour la gestion, la conception et l'implantation, (2) des méthodes de preuve de programme et (3) l'application systématique de tests.

Approches de gestion, de conception et d'implantation

Des logiciels de grande qualité résultent souvent d'une démarche rigoureuse et ce, à chacune des étapes du processus de création. Ainsi une meilleure approche de gestion (impliquant par exemple de petites équipes dont les rôles sont bien définis), des méthodes de programmation appropriées (programmation structurée, orientée objet, ...), l'utilisation d'outils d'aide au développement (modèle objet, outils CASE, gestion de source, langage évolué) et l'application de méthodes formelles ou semi-formelles de spécification lors de la conception se révèlent toutes des approches dictant une véritable discipline aidant à la conception de système de qualité. Précisons celles-ci.

La **méthode de gestion** utilisée lors du développement du logiciel influence sa fiabilité. Au milieu des années 60, on croyait, dans certains milieux, que pour assurer la production d'un logiciel complexe, il suffisait de lui affecter un grande quantité de ressources humaines en charge du développement. L'expérience d'IBM, qui a fait appel à une armée de personnes pour le développement du système d'exploitation OS/360, a mis fin à ce mythe. Il est maintenant reconnu qu'un usage sans discrimination de la puissance humaine crée plus de complications qu'elle n'en résout. Le regroupement des personnes en charge du développement en équipes réduites ayant des rôles mieux déterminés est une solution maintenant considérée comme nettement supérieure pour la production d'un logiciel. Dans cette organisation, chaque équipe est responsable d'un module muni d'une interface spécifique et chacun des membres de l'équipe se voit aussi confier une tâche bien définie.

Les différentes techniques mises de l'avant par les équipes de développement pour la production du code influencent considérablement la qualité du produit fini (en terme de cohérence, clarté et

absence de faute). Parmi toutes les **méthodes de programmation** proposées, la plus connue (et la plus ancienne) est celle dite «structurée» et ses dérivées. Précisons que, plus récemment, la méthode de «programmation par objets» est devenue aussi des plus populaires.

Les **outils** les plus courants et utiles pour produire des logiciels sans fautes sont nombreux et en constante évolution : éditeurs, IDE, langages évolués et spécialisés, bibliothèques, compilateurs, macro-assembleurs, gestion de versions, les "case tools", ... Il faut se rappeler que l'écriture d'un logiciel (dont en particulier les systèmes d'exploitation eux-mêmes) dans un langage évolué ainsi que l'usage de bibliothèques, a permis d'éliminer les fautes et d'augmenter la vitesse de production. Seulement quelques composantes logicielles peuvent exiger le recours à des outils ou langages moins évolués. Ainsi, seules les parties d'un système d'exploitation intimement liées au matériel doivent être écrites en assembleur.

De plus, il ne faut plus croire que tous les langages de haut niveau ne sont pas adaptés à l'écriture de logiciels nécessitant un haut niveau de performance (tels les systèmes d'exploitation). On a en effet longtemps sous-entendu que ceux-ci étaient incapables de produire du code efficace. En vérité, les compilateurs modernes produisent généralement du code optimisé souvent plus rapide et efficace que celui que bien des personnes sont aptes à produire. Dans le cas contraire et au besoin, il y a toujours la possibilité de les rendre optimal «manuellement». Finalement, il est important de comprendre que la fiabilité est au moins aussi importante que l'efficacité.

Les **méthodes formelles** ou semi-formelles sont basées sur les mathématiques et servent particulièrement à la spécification des applications. Elles vérifient (prouvent) aussi que l'implantation est correcte.

La preuve de programmes

Une seconde approche pour réduire l'apparition de fautes logicielles consiste à tenter de prouver, par des moyens plus ou moins formels, que tous les programmes ainsi que leurs interactions sont corrects.

Cela représente une tâche colossale, pour l'instant quasi inapplicable, du moins, pour des logiciels imposants de par leur taille. Certains auteurs ont cependant montré qu'en perfectionnant et automatisant la méthode, celle-ci pourrait éventuellement être envisageable.

Dès maintenant cependant, le bon fonctionnement de certains petits composants critiques est vérifiable et assure un degré de confiance supérieur sans le système.

Une approche qui s'apparente à la preuve et qui est couramment utilisé est le «model checking». Nous en parlerons dans le chapitre sur le formalisme.

Application systématique de tests

L'application de tests systématiques dans le but de détecter et éviter les fautes (notre troisième approche pour améliorer la qualité des logiciels) est fort bien connue et pratiquée par de toute équipe de développement. En fait, il existe aujourd'hui une multitude d'outils qui permettent d'automatiser cette méthode.

Il existe deux grandes approches pour effectuer des tests :

- l'approche boite noire [22, 20]
  Dans bien des situations, on voudrait (ou on doit) tester un programme sans avoir à se préoccuper de sa conception interne (on ne la connaît pas ou on n'y a pas accès). L'approche boite noire consiste à générer des données de tests sans se preoccuper de la structure du logiciel

afin de s'assurer que les sorties obtenues sont bien celles prévues pour des entrées données. Celui-ci est donc vu comme une boite noire.

Cette approche sert à tester la fonctionnalité d'un logiciel.

Idéalement, il faudrait tester toutes les entrées possibles d'un programme. Mais l'étendue de ces tests rend ce travail pratiquement impossible. Étant donné cette situation, il est donc essentiel d'identifier les données critiques du système, i.e., les données qui permettront de détecter le plus grand nombre de fautes (de générer le plus grand nombre d'erreurs). Il n'existe aucune méthode générale qui permet choisir les données critiques pour les tests. Cependant, certaines directives ont été émises pour choisir des données, tel que celles qui se situent à l'extrême limite de l'acceptable pour le programme.

- l'approche boite blanche (boite transparente) [23, 21]

  À l'opposé de l'approche boite noire, il existe une autre méthode de tests qui vise à analyser tous les «chemins» et structures internes d'un programme. Les tests sont donc conçus selon la connaissance que l'on possède du fonctionnement interne du programme. Une analyse ou une connaissance détaillée du code source est nécessaire.

  Les données en entrées sont donc choisies pour tester tous les chemins d'un programme tels que les deux conditions d'un énoncé de sélection et toutes les itérations possibles d'un énoncé d'itération.

- l'approche boite grise [24]

  Cette approche combien les approches boites noire et blanche.

Peu importe la méthode privilégiée, les tests s'appliquent à différentes étapes de la conception d'un logiciel :

- les test unitaires ;

  Ce sont des tests servant à valider le comportement d'un unique module.

- les tests d'intégration ;

  Même si chaque module est valide individuellement, cela ne signifie pas que leur intégration lorsque cela est requis se passera bien. L'application de tests devient beaucoup plus complexe lorsque les modules à tester interagissent entre eux.

  Les tests d'intégration servent donc à valider que tous les modules travaillent bien ensemble. Dans cette situation particulière, non seulement chaque programme doit-il être testé, mais aussi les interfaces entre les programmes. Pour faciliter les tests, il faut garder les interfaces les plus simples possibles.

  Ces tests d'intégration sont encore plus difficiles à établir lorsque les différents modules en interaction s'exécutent en parallèle.

- la validation (valide que le logiciel répond bien aux spécifications) ;
- la vérification (vérifie que les spécifications sont correctes).

1.2.3 Fautes des usagers

Ces fautes ne peuvent être éliminées, on les réduit par une éducation appropriée des usagers.

1.2.4 Conclusion

Aucune des méthodes présentées ne permet de produire des systèmes sans faute. Ces techniques se raffinent et vise à ce qu'éventuellement leur utilisation conjointe permettra d'augmenter le degré de fiabilité d'un système.

1.3 Détecter les erreurs

La détection des erreurs s'accomplie soit par l'environnement (matériel), soit par l'application (logiciel).

1.3.1 Détection par l'environnement

Parmi les erreurs détectées et détectables par le matériel, certaines peuvent être masquées par ce dernier par la répétition d'opérations, la redondance ou la comparaison. Celles qui ne le sont pas sont rapportées au système d'exploitation par des alarmes (*trap*) au niveau du gestionnaire des interruptions. Par exemple, les débordements arithmétiques, la violation des accès en mémoire et la violation de la protection sont toutes des erreurs qui sont signalées au système d'exploitation. Celui-ci entreprendra alors certaines actions dont nous reparlerons plus tard. Notons que l'ajout d'alarmes rend possible seulement la détection des erreurs prévisibles.

1.3.2 Détection par l'application

Grâce à diverses techniques, il est aussi possible pour une application (que ce soit le système d'exploitation ou tout autre logiciel) d'identifier elle-même certaines erreurs. À ce niveau, l'aspect essentiel de la plupart des techniques de détection des erreurs est la redondance. Celle-ci est constituée d'une provision d'informations «superflues» qui permet de valider l'information principale. Le terme «redondance» fait référence au fait que l'information disponible utilisée pour la vérification est redondante en ce qui concerne les algorithmes principaux. Nous avons déjà présenté quelques exemples d'utilisation de la redondance pour masquer les erreurs au niveau du matériel.

La redondance

L'aspect essentiel de la plupart des techniques servant à la détection des erreurs est la redondance. Celle-ci est constituée d'une provision d'informations «superflues» qui permet de valider l'information principale.

La redondance est un concept appliqué à plusieurs techniques de détection d'erreurs telles :

- l'**encodage** (*coding checks*) ;
  L'encodage de l'information sert à détecter les erreurs et, dans certains cas, à la reprise suite à une erreur. Généralement, l'encodage fait appel à la redondance. Les codes détecteurs et correcteurs d'erreurs et les sommes de vérification (*checksum*) constituent des exemples d'encodage.
- la **vérification de cohérence** ;

Exemple d'utilisation de la redondance

Un exemple d'utilisation directe de la redondance pour le logiciel est la programmation en N-versions. Selon cette technique, N versions d'un programme s'exécutent sur les mêmes données et un nombre majoritaire de ces versions doivent fournir les mêmes résultats. On détecte les erreurs lorsque les résultats ne concordent pas.

Un logiciel peut valider la cohérence des résultats de ses opérations. Ainsi une forme usuelle d'une telle validation requiert que les processus contrôlent la **cohérence des structures de données** (*structural checks*) qu'ils utilisent. Parmi les méthodes qui le permettent, l'une d'elle consiste à employer des structures structures de données solides. Celles-ci ont recourt à de l'information redondante pour ne pas perdre d'information.

Exemple de structures de données solides

Les systèmes d'exploitation, se voulant fiable, font souvent appel à des listes doublement chaînées pour implanter les différentes listes critiques (processus prêts ou en attente, demandes d'entrées/sorties, ...). Il a alors la possibilité de parcourir la liste vers l'avant ou l'arrière pour retracer et vérifier les liens. L'emploi d'un second pointeur (vers l'élément précédant dans la liste) constitue l'information redondante.
Les systèmes de fichiers se servent aussi de structures de données solides. Ainsi, le système de fichiers de *PrimeOS* est plus fiable que celui de Unix car il a recourt à une liste triplement chaînée pour relier les différents blocs de données d'un fichier.

- les **tests d'acceptation** ;
  Un autre méthode pour s'assurer de la cohérence des informations consiste à associer à chaque action majeure d'un processus un test d'acceptation. Celui-ci sert de critère pour déterminer si une action a été proprement exécutée. En fait, c'est une expression booléenne qui est évaluée lorsque l'action est complétée. Si le résultat est positif alors l'action s'est déroulée correctement, sinon une erreur s'est produite. Le test d'acceptation peut être aussi rigoureux que nécessaire. Il est cependant formulé pour identifier les erreurs que le concepteur juge les plus probables.
  Les tests d'acceptation tirent aussi parti des informations redondantes. Une première possibilité consiste à dupliquer l'information. Dans ce cas, une comparaison des états suite à chacune des actions permet la vérification. Un autre option est d'ajouter des informations spécifiques pour la validation. Ainsi, pour valider les modifications faites sur les entrées d'une table ou sur certaines informations de longueur connue, il est possible de leur associer une somme de contrôle (*checksum*). Ainsi, si celle-ci, suite à une opération, ne correspond pas à la prédiction, c'est donc qu'une erreur s'y est produite.

Toutes ces techniques de vérification/validation, permettant la détection des erreurs, sont puissantes mais complexes à mettre en place (développement), coûteuses (en espace, en temps et en équipement) et, en fin de compte, peu pratiques. On doit pouvoir justifier ces coûts par une fiabilité

Exemple de test d'acceptation

Soit l'action du planificateurconsistant à modifier périodiquement les priorités des processus et à réordonner la liste des processus. Le test d'acceptation pour cette opération pourrait simplement être de vérifier que les descripteurs des processus sont vraiment en ordre de priorité. Un test plus rigoureux requiert de l'information redondante. Ainsi, supposons une contrainte additionnelle telle que le nombre de descripteurs dans la liste après l'opération soit le même qu'avant celle-ci. Cela requiert une information redondante sous la forme d'un compteur qui ajoute une protection contre la perte ou la duplication d'un descripteur.

accrue.

D'autres méthodes ne font pas ou peu appel à la redondance (de données). Parmi celles-ci on retrouve,

- la **vérification temporelle** (*timing check*) qui s'assure que les opérations s'exécutent dans le temps prescrit. Les horloges de garde en sont un exemple.

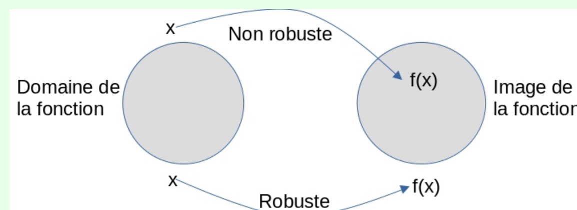
- la **vérification par inversion** (*reversal checks*) qui n'utilise aussi que peu de redondance (juste en temps d'exécution). Ce type de test applique l'opération inverse au résultat obtenu afin de valider la donnée originale. Ce procédé convient bien pour des opérations mathématiques telles l'élévation au carré ou l'extraction de la racine carré.

Les méthodes que nous venons de présenter fonctionnent bien lorsque les erreurs se produisent à l'intérieur d'un composant matériel ou logiciel. Par contre, la détection de celles qui surviennent aux interfaces entre les composants s'avèrent plutôt complexes.

Il y a donc un avantage certain à vérifier la crédibilité de toute l'information circulant par les interfaces. Ainsi, certains paramètres de procédure seront validés en testant s'ils sont contenus dans un intervalle de valeurs approprié. La conformité au protocole établi des messages véhiculés entre les processus doit aussi être testé. Dans tous les cas, il s'agit de s'assurer que les programmes sont **robustes**.

Définition : Robustesse

Un programme (logiciel ou processus) est dit robuste si, pour des données invalides en entrée (n'appartenant pas au domaine de la fonction), il produit un message d'erreur ou un résultat invalide (qui n'est pas dans l'image de la fonction). La figure suivante illustre ce concept.



1.3.3 Mécanismes de protection

Détecter les erreurs le plus tôt possible représente la meilleure façon de limiter les dommages et le temps perdu pour la réparation de ceux-ci de même que pour le traitement de la faute. Cela est particulièrement vraie lorsqu'une erreur devient elle même une faute qui provoque une autre erreur, ainsi de suite.

La capacité d'un logiciel (application ou même les système d'exploitation) de réagir rapidement aux erreurs dès leur apparition est grandement amélioré si des mécanismes de protection appropriés sont présents. Ceux-ci seront implantés par l'application elle-même ou, plus probablement, par le système d'exploitation sous-jacent, afin de limiter la propagation des erreurs et augmenter la fiabilité du logiciel.

1.4 Traitement des fautes

Le traitement d'une faute consiste bien sûr en premier lieu à sa localisation, puis à sa correction. On doit se rappeler le fait que la détection d'une erreur et la localisation de la faite correspondante sont deux opérations bien distinctes. En effet, une erreur peut résulter de plusieurs causes, localisées dans le matériel ou le logiciel, sans qu'aucune d'elles ne soit pour autant apparente.

Dans le but d'identifier une faute, il est d'une importance capitale de détecter rapidement les erreurs avant que leurs causes ne soient obscurcies par des dommages conséquents et par d'autres erreurs.

Dans certains cas, il est possible d'ignorer complètement la faute. Cette attitude sous-entend cependant des hypothèses sur la fréquence des erreurs qu'elle générera probablement et sur l'étendue de leurs dommages. Par exemple, il est parfois (très rarement) inutile de tenter de localiser une faute matérielle provoquant des erreurs intermittentes tant que la fréquence ces dernières reste inférieure à une limite acceptable. Toutefois, il demeure extrêmement important que la règle générale soit toujours de localiser et de corriger la faute.

La recherche d'une faute est généralement dirigée par la compréhension ou la connaissance qu'a la personne investigatrice de la structure du logiciel (ou programme). Si sa connaissance est incomplète alors sa tâche en est évidemment complexifiée.

Des outils sont disponibles pour faciliter la localisation d'une faute tels les ***debugger*** (lors du développement) ou les **traces** (après son déploiement). Une trace, ou journal (*log*), est un fichier dans lequel le logiciel ou le système sous-jacent enregistre toutes les activités récentes (ou événements) telles que des activations de processus, des appels de procédure, des transferts d'entrée/sortie, etc. Puisqu'on désire un journal utile, on doit y enregistrer une grande quantité d'information. Cette surcharge (*overhead*) de travail sera toutefois moins lourde si on rend la trace optimale, i.e. si on permet de l'activer seulement au besoin ou encore si on choisit des traces sélectives. Cette dernière option présuppose cependant l'aptitude à détecter autant un éventuel mauvais fonctionnement (début de la trace) que l'endroit où celui-ci se manifeste (sélection).

Une fois qu'une faute est localisée, son traitement exige une forme quelconque de correction. Pour palier aux **fautes matérielles**, on remplace généralement le composant fautif manuellement ou automatiquement, selon l'habileté du matériel à localiser ses fautes puis à déconnecter l'élément défectueux. Le remplacement manuel peut impliquer, quant à lui, la mise hors service du système pour un temps ou, préférablement, se faire en parallèle avec un service continu mais possiblement dégradé. Par exemple, il est possible de remplacer un disque défectueux sans arrêt de service quand

d'autres disques sont disponibles. Le remplacement de l'UCT (le processeur) ou d'un disque, s'il est unique, doit se faire quand le système est hors service.

Les **fautes logicielles** sont généralement produites par des déficiences dans la conception et l'implantation, ou par une corruption causée par des erreurs antérieures. En effet, contrairement au matériel, le logiciel n'est pas sujet à des fautes conséquentes à l'usure. La correction de la faute de conception ou d'implantation nécessite le remplacement d'un certain nombre de lignes de code du programme.

Une corruption est rectifiée en substituant à la copie erronée une copie de sécurité intacte. Une révision des dommages est possiblement nécessaire même à la suite de pannes matérielles (panne d'un disque entraîne parfois la corruption de fichiers, ce qui requiert une intervention pour rétablir les données).

Le remplacement du logiciel doit pouvoir s'effectuer sans imposer l'arrêt complet du système ou de l'application. Pour un système d'exploitation en particulier, autant, par exemple, pour procéder à sa mise à jour (nouvelles versions), qu'à la réparation d'erreurs critiques (*patch*, *service pack*, ...), ni l'arrêt, ni le redémarrage ne doivent être exigé.

1.5 Reprise

La reprise est un mécanisme qui permet de reprendre un traitement fautif suite à une tentative de correction. Avant même d'implanter des méthodes de reprise, il faut être en mesure d'identifier les erreurs, d'estimer les dommages qu'elles ont causés et tenter de les réparer. L'évaluation des dommages se repose soit entièrement sur un raisonnement à priori de la personne en charge de l'analyse de l'erreur, soit sur l'exécution par l'environnement d'un certain nombre de vérifications afin de déterminer les dommages produits.

Dans les deux cas, l'examen est guidé par une relation de cause à effet définie par la structure du logiciel, ainsi que sur une grande connaissance de ce dernier.

Exemple d'analyse en vue d'une reprise

Une erreur survenue lors de la mise à jour d'un répertoire ou d'un fichier est susceptible d'endommager les système de fichiers mais non pas la structure des processus, ni des descripteurs de périphériques. Dans cette situation, on exécute un programme qui vérifie et répare les dommages sur le système de fichiers (tel `fsck` sur UNIX ou Linux).

En fait, il est tout de même possible que la structure du système soit elle-même endommagée si des mécanismes de protection appropriés ne sont pas en place pour empêcher la propagation de tels dommages.

Deux techniques permettent de réparer les dommages causés par une erreur et d'effectuer une reprise : la reprise avant et la reprise arrière. Celles-ci tentent de remettre le système dans un état à partir duquel le traitement peut continuer tout en évitant la panne (l'erreur qui vient de se produire).

Exemple de dommages imprévisbles

Voici plusieurs années, les personnes inscrites à mon cours développaient leur solution sur Windows ou DOS. Ces systèmes, à l'époque, ne protégeait pas la mémoire centrale utilisée par le système lui-même.
Il arrivait donc fréquemment qu'une erreur de pointeur produise une corruption aléatoire dans le système d'exploitation. Les dommages étant imprévisibles, un redémarrage du système devenait parfois nécessaire.

1.5.1 Reprise arrière

La reprise arrière est l'approche la plus générale pour effectuer une reprise. C'est une approche simple qui répare les dommages en retournant les processus affectés vers un état (que l'on sait correct) qui existait avant que l'erreur ne se produise. Un fois ce retour en arrière fait, on tente de reprendre les opérations normales.

Points de reprise (*checkpoint*)

Cette approche est basée sur l'existence de points de sauvegarde ou de reprise (*checkpoints* ou *recovery points*) initiés à intervalles réguliers pendant l'exécution du processus. Lorsque l'exécution d'un processus arrive à un «point de sauvegarde», on enregistre suffisamment d'informations sur l'état du processus afin de lui permettre de redémarrer son exécution à partir ce point au besoin. Les renseignements requis pour cela comprennent son environnement volatile, une copie de son descripteur (PCB) et une copie de son espace d'adresses.

Trace (*audit trail*)

Lors d'une reprise (arrière), il y aura perte de données entre le point qui a causé l'erreur et le point de sauvegarde d'où s'effectuera la reprise. L'intervalle entre les points de sauvegarde représente donc la pire perte possible. Celle-ci peut être réduite par une trace («audit trail») où sont notées au fur et à mesure qu'elles se produisent, toutes les modifications effectuées et qui ont affectées l'état du processus à partir du dernier point de sauvegarde. Cet outil constitue ainsi la liste des opérations à réappliquer lors de la reprise, quand on doit retourner au dernier point de sauvegarde.

Copies de sécurité incrémentales

La technique de trace est analogue aux prises de copies de sécurité incrémentales. Une copie de sécurité incrémentale sauve une copie seulement des fichiers ou données qui ont été modifiés pendant la période indiquée. Ainsi, plusieurs entreprises effectuent des copies de sécurité complètes toutes les semaines, en plus de quotidiennement réaliser une copie de sécurité de tous les fichiers modifiés dans la journée (copie de sécurité incrémentale). Ces copies de sécurité incrémentales constituent un cas particulier de trace (par rapport à la copie de sécurité complète).

Blocs de reprise

L'implantation des procédés de points de sauvegarde et de traces requièrent l'enregistrement de toute l'information d'état et de toutes les altérations à cette information. Cela s'avère coûteux en temps et en espace. Une alternative intéressante est offerte par la technique des blocs de reprise dans lesquels la seule information d'état enregistrée est celle modifiée. Cette technique inclut des éléments de détection des erreurs et de traitement des fautes. Nous en faisons donc ici une description complète.

Un bloc de reprise est une section de programme ayant la structure suivante :

```
ensure «test d'acceptation»   by       «première alternative»
                              elseby   «seconde alternative»
                              elseby   «seconde alternative»
                              ...
                              elseby   «dernière alternative»
```

Programme 1.1 – Blocs de reprise

Le première alternative correspond à la portion de code du programme qui est normalement exécutée. Les alternatives suivantes sont utilisées seulement si la première échoue. Le succès ou l'échec d'une alternative est déterminée par un test d'acceptation associé à ce bloc. Ainsi, l'exécution d'un bloc de reprise débute par l'exécution de la première alternative suivie par celle du test d'acceptation. Si le résultat de ce test est positif (le cas normal), le bloc entier est jugé comme s'être exécuté avec succès. Si le test échoue, l'état du processus est ramené à son état d'avant l'exécution de la première alternative puis la seconde alternative est exécutée. Le test d'acceptation est de nouveau appliquer pour déterminer le succès ou l'échec de la seconde alternative. Les autres alternatives seront considérées au besoin (si le test échoue pour l'alternative précédente). Si toutes les alternatives échouent, alors le bloc entier est déclaré comme ayant échoué. Dans ce cas, la reprise doit être tentée en appelant un bloc alternatif, s'il y en a un (les blocs de reprise peuvent être imbriqués à n'importe quel niveau), ou en redémarrant le processus, s'il n'y a pas d'autres alternatives.

Les alternatives à l'intérieur d'un bloc de reprise se perçoivent comme des logiciels de rechange auxquels ont fait automatiquement appel lorsque l'alternative principale échoue. Contrairement au matériel de rechange, les alternatives ne sont pas de conception identique et emploient habituellement des algorithmes différents. Généralement, l'alternative principale emploie l'algorithme le plus efficace ou le plus approprié pour remplir la fonction du bloc. Les autres alternatives ont recourt à des algorithmes moins performants ou qui satisferont la fonction d'une façon partielle mais tolérable.

L'échec d'une alternative est considéré comme un événement exceptionnel et celle-ci ne sera remplacée que pour l'exécution courante du bloc (contrairement aux remplacements dans le matériel). Il est cependant fortement suggéré de noter dans un journal tous les échecs rencontrés de manière à ce que les erreurs de conception et de programmation soient reconnues et éliminées.

Pour ses opérations, la technique du bloc de reprise se base sur la possibilité de récupérer l'état d'un processus lorsque le test d'acceptation échoue. La facilité avec laquelle on récupère l'état d'un processus dépend des interactions que le processus maintient avec d'autres processus lors de l'exécution de son bloc. S'il n'y a aucune interaction, la récupération de l'état est réduite à ramener les valeurs précédentes (au bloc) de chacune des variables du programme (et des registres) qui ont été modifiées. Cela est réalisable grâce à des «**caches de reprise**» (*recovery cache*). Ce sont dans

ces caches, localisées en mémoire (un petite portion de la mémoire centrale), que sont emmagasinées les valeurs initiales des variables avant que celles-ci ne soient modifiées. Seules celles qui subiront une altération nécessitent une sauvegarde dans ces caches. Les valeurs révisées plus d'une fois n'y sont enregistrées qu'une seule fois. Ainsi, l'échec d'une alternative cause la récupération des valeurs sauvées dans la cache avant l'essai de l'alternative suivante, mais sa réussite entraîne leur destruction car les modifications sont devenues caduques. Les caches de reprise servent aussi à implanter les blocs de reprise imbriqués. Dans ce cas, les caches sont empilées (le bloc le plus récent est au sommet de la pile) telles les variables locales lors d'appels de fonctions.

Lorsqu'il y a interaction avec d'autres processus dans un bloc de reprise, le retour en arrière est moins évident. Une approche consiste à traiter toutes les interactions entre deux processus comme une conversation. Si l'un des processus doit revenir en arrière lors d'une conversation, l'autre devra aussi reculer car l'information transmise pendant la conversation peut être erronée. Ainsi, aucun processus ne peut aller au-delà de la fin d'une conversation tant que tous deux n'ont pas satisfait le test d'acceptation. Un échec de l'un des deux tests d'acceptation ramène les deux processus au début de la conversation. Cette façon de faire se généralise aux conversations entre plusieurs processus.

Ainsi, dans le bloc de reprise :

- la détection des erreurs provient d'un test d'acceptation ;
- le traitement de la faute est réalisée par la provision d'alternatives ;
- la reprise est basée sur un mécanisme de «caches de reprise».

Conclusion

La reprise arrière possède l'avantage d'être une méthode relativement simple à appliquer car elle ne dépend pas de l'erreur. Cela signifie que la méthode utilisée pour continuer à fournir le service spécifié est indépendante de la méthode choisie pour évaluer les dommages (et les réparations à effectuer). De plus l'évaluation des dommages ne tient virtuellement aucun compte de la nature de la faute. En effet, l'exécution est arrêtée puis entièrement reprise en «récupérant» l'état du dernier point de sauvegarde. On ne se préoccupe donc pas de «comment» poursuivre l'exécution.

Aussi, il est envisageable de mettre en place un mécanisme de reprise arrière généralisé. Par exemple, un mécanisme de point de contrôle et de redémarrage fourni par un système d'exploitation ou l'environnement s'avérerait un atout pour développer une variété d'applications fiables.

1.5.2 Reprise avant

La reprise avant est une technique plus complexe que la reprise arrière car, dans ce type de reprise, l'identification de la faute et la reprise sont intimement liées. En effet, la reprise ici est plus dépendante de la nature de la faute ou du moins de ses conséquences. Aussi, un tel mécanisme doit être conçu telle une partie intégrale du système qu'il sert (de l'application).

Ceci dit, certaines techniques de reprise avant se révèlent plutôt simples et efficaces. Ces caractéristiques «favorables» sont en fait intimement liées à celles de la faute elle-même (fautes simples) et des dommages qu'elle cause. Donc lorsque que la faute et ses conséquences sont «abordables», la reprise avant s'avère souvent mieux adaptée (plus simple et surtout plus efficace) que la reprise arrière.

Il est possible d'appliquer la reprise avant lors des trois situations suivantes :

- une erreur au niveau des composants ;

Ces erreurs sont principalement traitées par des mécanismes de «prise en main des exceptions» (*exception handling*) que nous présentons en détails dans la section suivante.

Notons que des techniques d'auto-stabilisation permettent aussi à des algorithmes de récupérer après une erreur et ce, sans nécessairement devoir recourir à la «prise en main des exceptions». L'annexe B présente en détails le concept d'auto-stabilisation introduit par Dijkstra.

- une erreur au niveau des systèmes en interaction ;

  La reprise lorsque ce type d'erreur se produit est assurée un mécanisme de compensation que nous détaillerons ultérieurement.

- une erreur au niveau des algorithmes ;

  Ce troisième type d'erreur est plus complexe et plus difficile à corriger avec la reprise avant. Le mécanisme de blocs de reprise et la programmation en N-versions offrent possiblement une solution dans ces situations.

  Il est surprenant de parler des blocs de reprise dans la section de la reprise avant. Cependant, cette méthode combine en fait la reprise arrière et avant puisque la reprise arrière est requise pour reprendre l'exécution (cache de reprise) et qu'une série d'alternatives (reprise avant) sont fournies pour corriger la faute dans l'algorithme et que celles-ci dépendent intimement de l'application en jeu.

  La programmation en N-versions tente de maintenir une opération continue en exécutant en parallèle n versions d'un même programme. Les résultats sont ensuite comparés et le plus probable d'entre eux est retenu afin de poursuivre sans délai le traitement. Ce procédé constitue donc une forme de reprise avant.

La prise en main des exceptions

Soit un algorithme qui incorpore des stratégies de reprise à la suite d'une erreur due à des composants en faute. Une conception rationnelle d'un tel algorithme requiert la prédiction des fautes possibles et exige de comprendre la façon dont celles-ci se manifestent hors du composant. Cela consiste à joindre aux comportements normaux décrits lors de la spécification du composant, un certain nombre de comportements indésirables. Aussi longtemps qu'un composant exécute une des activités spécifiées (y compris les les indésirables), il ne sera pas, selon sa spécification, considéré en erreur.

Cependant, du point de vue du système, il y aura une faute lorsqu'une des activités indésirables se produit et certaines parties de l'algorithme seront conçues pour les traiter.

Exemples de reprise avant : codes correcteurs

Il est possible d'intégrer dans le matériel la reprise avant par le biais de codes correcteurs d'erreurs afin de traiter des informations fautives en mémoire ou transmises lors d'une communication. Un code correcteur donné n'est cependant adapté qu'à une classe particulière d'erreurs et nettement restreinte n'impliquant, par exemple, qu'au plus n bits successifs. Ainsi, le code de Hamming permet de corriger les erreurs impliquant un seul bit.
Cette capacité doit être intégrée dans la spécification du composant.

Généralement, on souhaite avoir certains moyens de distinguer la partie principale d'un algo-

rithme de celles vouées à fournir la reprise avant pour les différents types de fautes envisagés dans chacun des composants. Les concepteurs de système d'exploitation et de langage de programmation ont pourvu à ces besoins par certaines facilités permettant la «prise en main des exceptions». Celle-ci est assurée par différents types d'énoncés selon le système ou le langage utilisé :

- C++, Java, C#, Python : «`try`», «`throw`» et «`catch`»
- PL/1 : «`on condition`»
- Ada : «`on exception`»
- Unix (Linux) : «`signal`»
- OS/MVS : «`stae`» et «`spie`»

Attention :

La «prise en main des exceptions» est une méthode permettant d'implanter la reprise avant. Elle n'est cependant pas considérée appropriée pour le traitement des fautes (bug) résiduelles dans les programmes.

La compensation

La compensation est une forme de reprise avant qui répond à un besoin très différent mais très important et qui ne peut être traité par la reprise arrière. Elle permet de traiter la situation où une erreur est détectée lorsque que le système est en communication avec un environnement qui ne peut retourner en arrière.

La compensation est un acte par lequel un système fournit de l'information supplémentaire dans le but de corriger les effets d'informations qu'il a déjà expédiées à un autre système. Cela requiert que les deux systèmes (ou de manière plus générale, tous les systèmes) en interaction soient conçus de façon à ce que, lorsque l'information erronée est découverte parmi celle transmise et reçue, ceux-ci soient en mesure d'accepter l'information correctrice alors expédiée (soit par le système fautif, celui ayant expédié l'information erronée, soit par un programme de reprise).

Définition : Compensation

Acte par lequel un système fournit de l'information supplémentaire dans le but de corriger les effets d'informations qu'il a déjà expédiées à un autre système

Les algorithmes qui incorporent des stratégies de compensation devraient être conçus en se basant sur des techniques similaires à celles employées pour la «prise en main des exceptions». Cependant la conception d'algorithmes de compensation qui fonctionnent bien pour des systèmes complexes présente un défi majeur qui ne possède aucune solution générale. Clairement, la solution, mais malheureusement utopique, est celle d'éliminer le besoin de compenser en garantissant qu'aucun résultat incorrect ne sera jamais autorisé à quitter le système.

Conclusion

Les méthodes de reprise avant et arrière devraient être considérées comme des techniques potentiellement complémentaires plutôt que compétitrices. En effet, la reprise arrière sert à traiter les erreurs résiduelles ce que ne fait pas la reprise avant.

Exemple de compensation

Soit une base de données servant au contrôle de l'approvisionnement. Cette base de données est mise à jour par un message d'entrée indiquant la sortie de quelques éléments. Si, ultérieurement, on découvre que ce message était incorrect, on doit compenser par un autre message indiquant d'enregistrer l'acquisition des éléments de remplacement.
Une simple compensation comme celle-ci est probablement insuffisante si, par exemple, le système de contrôle de l'approvisionnement a déjà recalculé les niveaux optimums des inventaires et produit des ordres d'achats pour le remplacement des éléments impliqués.
La plupart des bases de données récentes incluent des stratégies de reprise très sophistiquées. Cependant, même sur celles-ci , les données qu'elles détiennent peuvent se retrouver dans un état tel qu'il n'y aucune autre alternative que de suspendre le service et tenter une correction manuelle des dommages. Il faut, pour ce faire, bien appréhender le flot d'informations dans l'environnement du système (en entrée et en sortie) et l'enregistrer.
En pratique, une telle compensation manuelle n'est ni garantie ni complète.

Ainsi, il est possible de combiner la prise en main des exceptions avec la technique de blocs de reprise pour effectuer une reprise arrière. Par exemple, la reprise avant peut traiter les fautes simples (telles les entrées invalides) alors que la reprise arrière (blocs de reprise) serait affectée aux fautes moins probables potentiellement attribuées à une conception inadéquate des gestionnaires d'exceptions.

1.6 Traitement d'erreurs à plusieurs niveaux

Les différentes techniques de traitement des erreurs s'appliquent aussi conjointement dans une architecture par couches. Le but principal est de masquer à chaque niveau du système d'exploitation les erreurs qui se sont produites aux niveaux inférieurs.

Ainsi chaque niveau du système doit être, autant que cela est possible, responsable de la reprise lors d'une erreur, de façon à ce que, pour les niveaux supérieurs, il apparaisse sans faute. L'idée consiste à étendre au système d'exploitation et au applications qu'il dessert, le masquage d'erreurs fait par le matériel.

Lorsque le masquage est impossible, les erreurs d'un niveau inférieur, qui se produisent pendant l'exécution d'une fonction quelconque pour un niveau supérieur, doivent être transmises au niveau supérieur d'une façon méthodique (par exemple, avec un code de retour).

Le niveau supérieur effectue alors une certaine forme de reprise, masquant ainsi l'erreur pour les niveaux supérieurs.

Les erreurs qui ne peuvent être masquées à aucun des niveaux doivent éventuellement être transmises à l'application (ou la personne qui l'utilise) ou à la personne en charge du système.

Au niveau le plus bas du système, les erreurs détectées par l'UCT qui ne peuvent être masquées par le matériel, sont signalées au noyau du système d'exploitation par des interruptions déclenchant l'exécution de routines de traitement des erreurs.

Un constat important est qu'il faut inclure des capacités de détection d'erreurs et de reprise à chacun des niveaux. On évite ainsi la propagation de dommages sérieux.

Exemple de traitement des erreurs à plusieurs niveaux

Considérons un processus usager qui souhaite ouvrir un fichier. Le processus effectue un appel au système de fichiers par l'intermédiaire d'une routine («`open`») qui à son tour appelle le système d'entrée/sortie grâce à la procédure «`doio`» qui permet de lire le répertoire de l'usager. L'opération d'entrée/sortie est ensuite initialisée par le gestionnaire du disque.
Supposons qu'il se produit une erreur de parité pendant l'opération. Dans ce cas, un premier niveau de masquage peut être fourni par le matériel du contrôleur du disque, probablement conçu pour détecter de telles erreurs et pour relire le bloc correspondant. Si, après plusieurs essais, l'erreur persiste, elle est rapportée au pilote du disque grâce à un registre d'état, positionné par le contrôleur à la fin du transfert. Le pilote du disque peut tenter de masquer l'erreur en réinitialisant l'opération au complet. Si cela ne réussit pas, l'erreur doit être rapportée à la routine d'ouverture («`open`») du système de fichiers. Au niveau du système de fichiers, la reprise est possible si une autre copie du répertoire existe ailleurs. Il est alors possible de lire cette copie et d'effectuer l'ouverture du fichier (si aucune autre erreur ne se produit). Si cette alternative échoue ou s'il n'y a pas de copies de sécurité pour le répertoire, alors l'erreur, ne pouvant plus être masquée, est rapportée au processus usager.

Exemple de traitement des erreurs à plusieurs niveaux

Une autre erreur susceptible de se produire pendant l'ouverture d'un fichier est la corruption d'un pointeur dans les listes des demandes du disque, soit causée par un mauvais fonctionnement du matériel, soit par une faute de programmation. Si la liste est doublement chaînée et que la routine de manipulation en tient compte, alors toute perte est évitable. Le dommage au pointeur sera alors réparé par les routines de manipulation masquant ainsi l'erreur au pilote et au système de fichiers.
Il est intéressant de noter les implications lorsque la liste est simplement chaînée. La demande sera perdue et les valeurs des compteurs de la liste seront incohérentes avec le nombre de demandes. Deux événements peuvent alors survenir :

- le pilote assume que le succès de l'opération d'attente implique l'existence d'une demande dans la liste. Il retirera de la liste une demande non-existante, interprétera le résultat comme une opération d'entrée/sortie valide et provoquera d'autres erreurs imprévisibles.
- le pilote vérifie la liste des demandes, constate qu'elle ne contient aucune demande valide et rapporte ce fait à la procédure «`doio`». Dans ce cas, l'erreur sera masquée au processus utilisateur car la demande peut être régénérée.

Cependant, certaines erreurs ne doivent pas être masquées (erreurs fatales) car elles indiquent des fautes grossières dans les programmes. Celles-ci ne sont pas issues de l'environnement mais dr l'application elle-même (et des personnes qui l'ont développé). La responsabilité de l'environnement est de les rapporter à l'application ou aux personnes responsables.

Exemple d'erreurs fatales

Soit un débordement arithmétique détecté par le processeur et rapporté au noyau du système d'exploitation via le pilote d'interruption. La routine d'interruption du noyau identifie rapidement le coupable comme étant le processus courant pour le processeur concerné. Le processus est alors éliminé avec un message approprié.
Un second exemple consiste en la violation de la mémoire et de la protection résultant de l'exécution du programme. Cette erreur entraîne aussi l'annulation du processus courant, mais dans ce cas particulier, on doit impérativement communiquer les violations de la protection car elles sont susceptibles d'être des tentatives pour percer le système.

1.7 Étude de cas

Annexe A

Raid et mémoire stable

Pour obtenir une unité d'emmagasinage d'information fiable, on recourt fréquemment aux concepts de RAID et/ou de mémoire stable. Brièvement, un RAID (*Redundant Array of Independant Disks*) est un regroupement de disques physiques alors que la mémoire stable est un concept logique qui assure l'atomicité des opérations et de la cohérence/disponibilité des données.

A.1 RAID

La technologie appelée RAID [11, 1, 19, 9, 6, 3, 14, 15, 12] permet d'obtenir, à partir du regroupement de plusieurs disques physiques, un seul disque logique plus performant et plus fiable. Le principe de base le plus important d'un RAID est sa transparence car l'usager doit considérer le regroupement comme un seul et unique disque (logique).

Historiquement, le concept de RAID a été introduit dans les années 80 pour obtenir, et ce à faible coût, un disque logique d'une plus grande capacité, un temps d'accès plus court, un meilleur taux de transfert et une disponibilité accrue, par rapport aux disques physiques uniques. À l'époque l'acronyme RAID signifiait «*Redundant Array of Inexpensive Disks*».

Aujourd'hui, la capacité des disques s'étant significativement améliorée, cette technologie sert surtout à accroître le temps d'accès, le taux de transfert et la disponibilité. Même si le coût des disques est moins important, on utilise fréquemment des disques peu coûteux aux performances moyennes pour construire des disques logiques selon l'architecture RAID.

Les principes sous-jacents à la technologie RAID sont de :

1. considérer qu'un ensemble de disques constitue une seule et unique entité logique ;
2. répartir les données (*striping*) sur tous les disques physiques afin de fournir de meilleurs temps d'accès et d'augmenter le taux de transfert ;

   La répartition des donnés est effectuée en subdivisant les agrégats de données par bandes (agrégat par bandes). Une bande est un regroupement de données sur un disque. Ce peut être des bits, un secteur, un bloc (regroupement de secteurs), une piste, etc. Toutes les bandes d'un même agrégat sont positionnées au même endroit sur les disques physiques. Comme elles

se situent sur des disques physiques distincts, elles sont lues et écrites en parallèle. Ainsi, en théorie, si un RAID contient N disques physiques, le taux de transfert pourrait être N fois plus rapide que sur un disque physique unique.

3. se servir d'informations redondantes pour augmenter la fiabilité (haute disponibilité des données).

   La redondance sur un RAID est conçue pour être capable, en cas de panne, de régénérer les données manquantes à partir des données encore disponibles et de l'information redondante. Cette dernière peut toutefois prendre plusieurs formes. Les plus courantes sont les copies multiples et les bits de parité.

   Il y a une différence entre la redondance appliquée sur un disque physique unique et celle appliquée sur un RAID. Sur un disque unique, la redondance prend généralement la forme d'un code CRC emmagasiné à la fin de chaque enregistrement (sur le même disque). Dans le cas d'un RAID, l'information redondante est emmagasinée sur des disques supplémentaires. Ainsi, en cas de panne d'un disque, il est possible de récupérer les informations (ce qui n'est pas le cas sur un disque unique même avec un code CRC). De plus, sur un RAID, l'écriture de l'information redondante ne ralentit pas l'accès puisqu'elle se fait en parallèle.

Le RAID s'oppose fréquemment au SLED (*Single Large Expensive Disk*). Ce dernier est un disque plus coûteux car il offre une plus grande capacité d'emmagasinage et une meilleure fiabilité. Toutefois si le disque tombe en panne, on perd l'accès à toutes les données, alors que sur un RAID, on peut toujours accéder aux données restantes et régénérer les données en utilisant la redondance.

Il existe de multiples façons de combiner entrelacement des données et redondance pour obtenir une architecture RAID. L'industrie s'est donc entendue sur certains standards pour l'entrelacement et la redondance que l'on appelle les niveaux RAID. Le niveaux les plus populaires sont le RAID 5 et des hybrides de RAID 0 et RAID 1. Nous allons donc présenter les niveaux RAID dans les prochaines sections.

A.1.1 RAID 0 - Entrelacement par bandes

Le niveau 0 des architectures RAID est simplement une agrégation par bandes (entrelacement par bandes ou «*striping*»).

Cette architecture requiert au moins deux disques physiques. Selon cette organisation, un fichier est divisé en N bandes, une bande étant idéalement une combinaison de plusieurs secteurs. Si notre disque logique RAID possède K disques, alors K bandes seront écrites sur les disques en parallèle. Cette situation est illustrée à la figure A.1. Dans cet exemple, le fichier est divisé en 16 bandes réparties sur quatre disques. Supposons que la taille des bandes est de 32 K. Les 16 bandes contiendraient alors un fichier dont la taille serait de 500K ($500 \div 32 = 15,625 \rightarrow 16$ bandes)

L'efficacité de cette approche dépend des demandes d'entrées/sorties. Si celles-ci impliquent le transfert d'une grande quantité de données à la fois, alors on gagne en vitesse. Toutefois si elles ne transfèrent que de des petites quantités de données à la fois, par exemple un secteur, alors le gain est minimal sinon nul. Ce type d'organisation est surtout utile pour les applications ayant besoin d'une vitesse de transfert élevée telle que celle requise par les applications de montage vidéo.

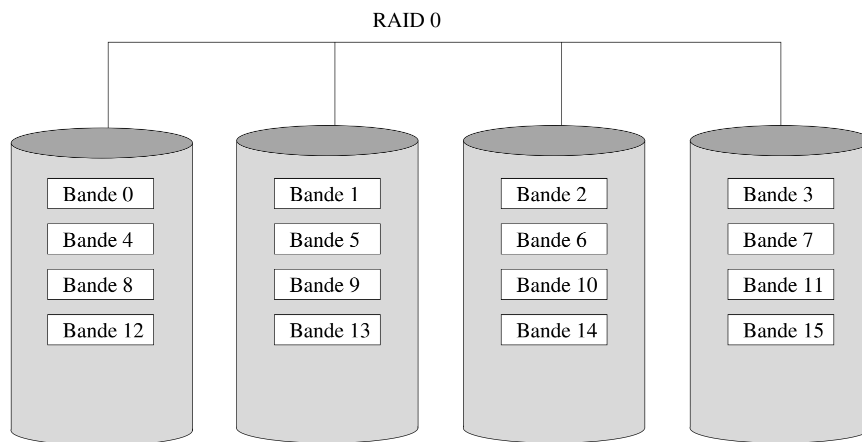


Figure A.1 – Architecture RAID de niveau 0

L'espace total disponible sur une telle architecture dépend de celui du plus petit disque. Ainsi, pour K disques, dont le plus petit est de 500G, la capacité totale est de $K \times 500$G. Donc pour une architecture RAID 0 de deux disques de 500G, son volume total sera de 1000G. Mais, si une architecture RAID 0 se compose d'un disque de 500G et d'un autre de 1000G, sa capacité totale reste de 1000G car l'espace excédentaire du second disque ne sera jamais utilisé. La figure A.2 illustre ce problème de perte d'espace.

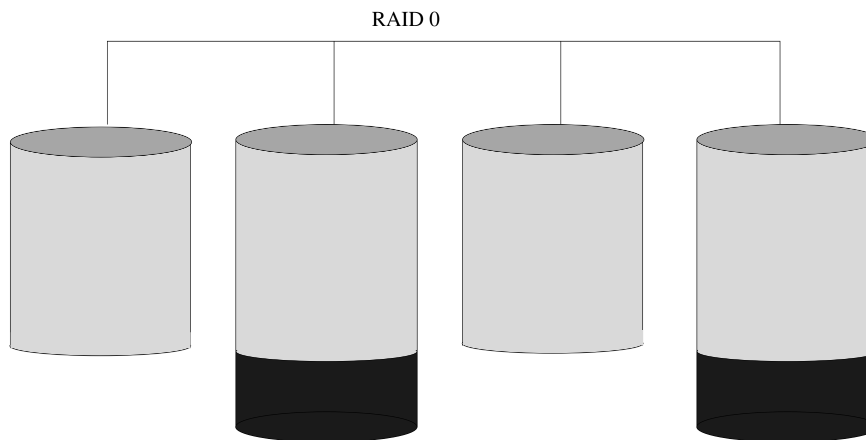


Figure A.2 – Perte d'espace potentielle avec RAID 0

L'inconvénient de cette approche est qu'elle n'offre aucune fiabilité. En effet, s'il y a panne d'un disque, les données deviennent inaccessibles. Si ce disque devient illisible, les données qu'il contenait ne pourront pas être récupérées (sauf si on a une copie de sécurité). En fait, sa fiabilité est moindre que celle d'un «SLED» et même, moindre que celle d'un seul disque (plus grand est le nombre de

disques, plus la probabilité de pannes augmente).

Pour tout dire, l'arrivée des disques SSD semble avoir sonné le glas de cette architecture.

A.1.2 RAID 1 - Disques miroirs

L'architecture RAID de niveau 1 emploie K disques redondants. Selon cette organisation, tous les disques contiennent exactement les mêmes données, d'où le terme miroir. La figure A.3 décrit une telle architecture.

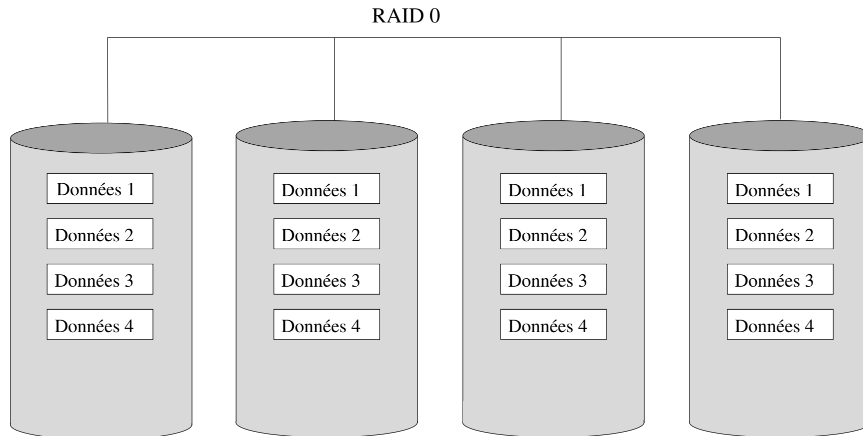


Figure A.3 – Architecture RAID de niveau 1 - Disques miroirs RAID 0

Cette architecture apporte un grande fiabilité mais peu d'accélération concernant l'accès aux données. Les quelques améliorations en performance proviennent du fait qu'une demande d'entrées/sorties a l'avantage d'être dirigée vers le disque physique présentant la plus petite latence (déplacement de la tête de lecture et plus petite rotation du disque)

Le coût de la mise à jour des multiples copies est nul car toutes le écritures sont exécutées en parallèle sur tous les disques en même temps. La reprise après une panne est aussi très simple selon cette approche.

Les inconvénients de cette architecture sont principalement le manque d'accélération lors du traitement des demandes et son coût très élevé (en terme de disques). Pour palier ce problème, on combine généralement les architectures de niveau 0 et 1. On obtient alors une accélération pour l'accès et une fiabilité accrue en cas de panne. Cette architecture mixte est présentée à la figure A.4.

A.1.3 RAID 2 - Code correcteur d'erreurs

L'architecture RAID de niveau 2 applique des techniques de détection et de correction d'erreurs identiques à celles employées pour la mémoire centrale des ordinateurs. Elle fait appel à une technique d'entrelacement par bandes (RAID 0) combinée à une méthode de détection d'erreurs

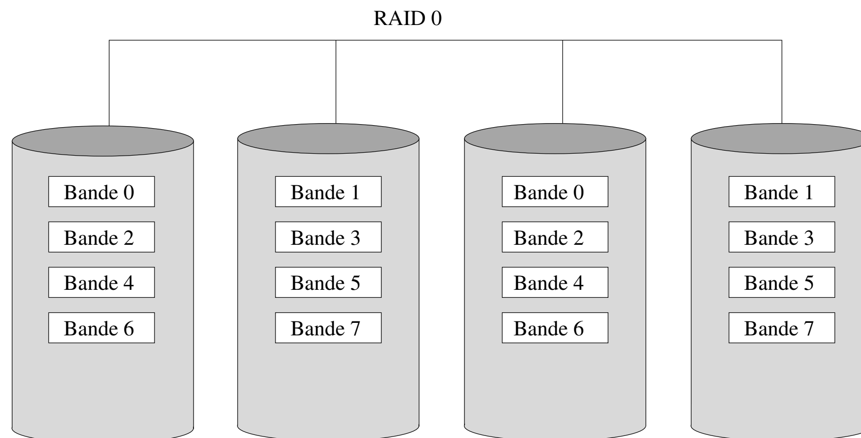


Figure A.4 – Architecture RAID mixte combinant les niveaux 0 et 1

afin d'obtenir une unité rapide et fiable. Notons que le code de Hamming est souvent la méthode privilégiée pour le contrôle des erreurs.

Ce type d'architecture favorise fréquemment l'entrelacement par bandes de 1 bit. L'unité de répartition est très petite, soit un octet ou un mot. Les bandes (de 1 bit chacune) de l'unité de répartition sont emmagasinées sur des disques distincts du RAID et les bandes de contrôle (bits de contrôle), elles, le sont sur d'autres disques. La bande de données en position i contient le $i^{ème}$ bit de chaque unité tandis que les bandes de contrôle contiennent chacune un des bits de contrôle.

Par exemple, avec un code de Hamming appliqué à un octet, on obtient 4 bits de données et 3 bits de contrôle. Sept disques sont donc nécessaires pour emmagasiner tous les bits. Il est aussi possible d'utiliser 8 disques de données (8 bits) pour 4 bits de contrôle. L'ordinateur CM-2, un ordinateur expérimental, employait 32 bits de données (32 disques), 6 bits de contrôle basés sur le code de Hamming (6 disques) et un bit de parité (un autre disque). Le RAID comprenait donc 39 disques.

Le niveau RAID 2 est n'est guère usité de nos jours car il est complexe, coûteux et désormais désuet. En effet, il propose un contrôle d'erreurs qui est maintenant directement intégré dans les contrôleurs de disques durs.

A.1.4 RAID 3 - Entrelacement par bandes (octets) et bits de parité

L'architecture RAID de niveau 3 emmagasine les informations avec un entrelacement par très petites bandes (un octet) sur plusieurs disques. De plus, elle nécessite un disque supplémentaire pour emmagasiner des bits de parité. Un RAID de niveau 3 comprend donc K disques, $K - 1$ pour les données et un pour la redondance. La figure A.5 illustre une architecture RAID de niveau 3 contenant 4 disques.

Cette architecture offre une très grande rapidité d'accès, un haut taux de transfert, la fiabilité et cela à un coût moindre que les architecture précédentes (RAID 1 et RAID 2). En effet, elle ne

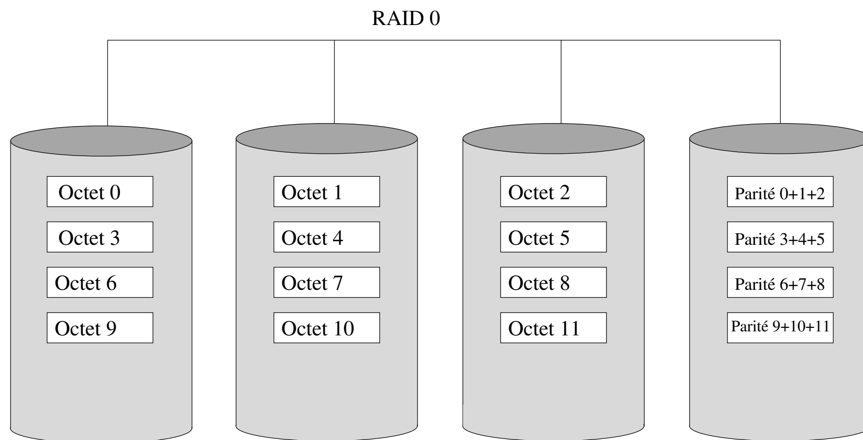


Figure A.5 – Architecture RAID de niveau 3

nécessite toujours qu'un seul disque de redondance et ce, peu importe le nombre de disques de données.

La fiabilité est obtenue par les bits de parité. Ainsi, si un disque subit une défaillance, il est possible de reconstruire aisément l'information à partir des autres disques. Soit un système RAID 3 contenant 5 disques : 4 pour les données (X_0, X_1, X_2 et X_3) et un pour la redondance (X_4). Le bit de parité pour les bits en position i pour toutes les bandes à travers tous les disques est obtenu grâce à la formule suivante :

$$X_4[i] = X_3[i] \oplus X_2[i] \oplus X_1[i] \oplus X_0[i]$$ [1]

Si l'un des disques tombe en panne (supposons X_1), la récupération du bit en position i pourra se faire par la calcul suivant :

$$X_1[i] = X_4[i] \oplus X_3[i] \oplus X_2[i] \oplus X_0[i]$$

Ainsi, tout le contenu initial du disque X_1 sera régénéré. Par contre, cette architecture ne tolère qu'une seule panne.

La rapidité d'accès est assurée ici par la lecture en parallèle de toutes les bandes de données (haut taux de transfert). Comme celles-ci sont petites, les données seront systématiquement réparties sur la totalité des bandes. Toutefois ce fait implique que tous les disques doivent être synchronisés en permanence et participent tous à chaque demande d'entrées/sorties. Cet avantage s'accompagne de l'inconvénient : le traitement d'une seule demande à la fois (moins d'entrées/sorties par seconde).

Notons en terminant que les écritures sur disque subissent un certain ralentissement dû au calcul de parité. Par contre, comme tous les disques sont synchronisés en permanence, l'écriture elle-même sur les disques ne souffre d'aucun délai.

1. Le symbole $\oplus$ représente un «ou exclusif»

A.1.5 RAID 4 - Entrelacement par bandes (blocs) et bits de parité

L'architecture RAID de niveau 4 est similaire à celle du RAID 3 sauf qu'il privilégie un entrelacement par bandes dont la taille est beaucoup plus grande, d'au moins un secteur. Ainsi, un RAID 4 de K disques, emmagasinera toutes les bandes de données (blocs) sur $K-1$ disques et la bande de contrôle (bits de parité) sur le dernier disque. La parité est calculée de la même façon que pour le RAID 3 sauf qu'elle s'applique sur des bandes de tailles supérieures. La figure A.6 décrit l'architecture du RAID niveau 4.

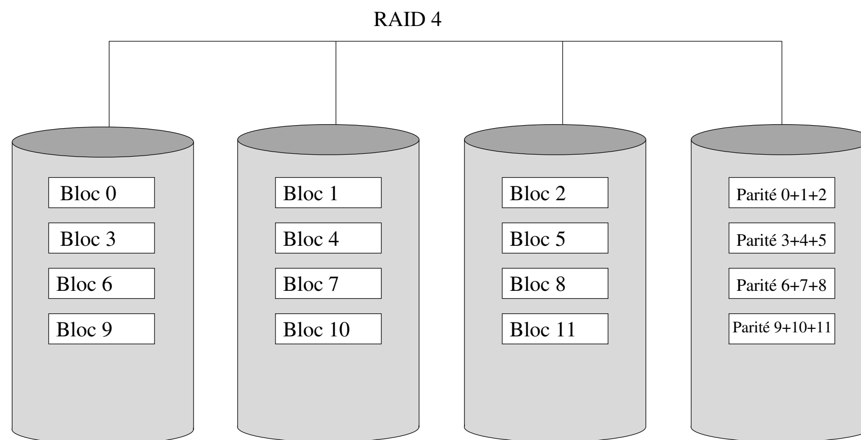


Figure A.6 – Architecture RAID de niveau 4

Comme le RAID 3, cette architecture offre la rapidité d'accès, un haut taux de transfert et la fiabilité (au même coût que le RAID 3). Le RAID 4 est cependant plus performant dû principalement à la taille de ses blocs (par rapport aux octets). Ainsi, lors de l'écriture de grandes quantités de données, celles-ci peuvent être emmagasinées en parallèle sur tous les disques. Cependant, contrairement au RAID 3, il n'est pas nécessaire de synchroniser tous les disques en permanence. En effet, les bandes (blocs) étant de grandes tailles, elles peuvent être indépendantes (i.e. les lectures ou écritures n'ont pas à accéder à toutes les bandes sur tous les disques en même temps pour manipuler les données). Ainsi, lors de l'écriture d'une quantité relativement faible d'informations (qui entrent dans une seule bande), plusieurs demandes d'E/S pourront être acheminées concurremment et ce, tant qu'elles opèrent sur des disques distincts (plus d'entrées/sorties par seconde).

Comme le RAID 3, cette approche souffre d'une certaine pénalité à l'écriture due au calcul de parité. Toutefois, dues à l'indépendance des blocs (les disques ne sont pas synchronisés), lors de la mise à jour d'une bande unique, la lecture et l'écriture de la bande de parité associée sont requises, afin de recalculer les nouveaux bits de parité. Donc une mise à jour, même mineure, exige deux lectures et deux écritures (bande de données et bande de parité).

A.1.6 RAID 5

En ce qui concerne l'architecture RAID de niveau 5, c'est la même que celle du RAID de niveau 4 à l'exception que dans cette version, les bandes de contrôle (bits de parités) sont réparties sur tous les disques. Une architecture de type RAID 5 de K disques, utilise $K-1$ bandes de données réparties sur les K disques et une bande de contrôle (aussi répartie sur les K disques). La figure A.7 illustre une architecture RAID 5 de 4 disques.

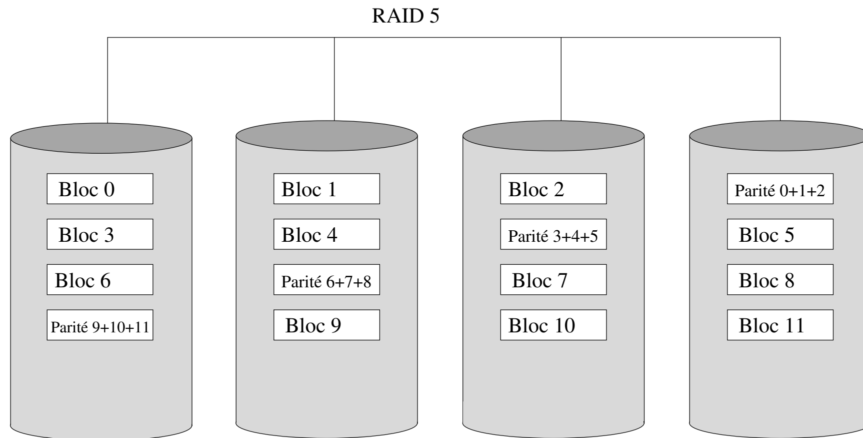


Figure A.7 – Architecture RAID de niveau 5

Cette architecture est plus efficace que celle du RAID 4 car, dans ce dernier, le disque de parité est susceptible de devenir un goulot d'étranglement. Cette situation ne peut survenir sur un RAID 5 car les écritures des informations de contrôle (bits de parité) sont mieux distribuées. De plus, cet avantage est obtenu sans coût supplémentaire.

Ce type d'architecture est l'un des plus populaires.

A.1.7 RAID 6

L'architecture RAID de niveau 6 utilise une double parité. C'est cette seule particularité qui la distingue de la version précédente, RAID 5. Ce fait permet de tolérer des pannes sur deux disques. Cette architecture exige un minimum de quatre disques (deux disques de données et deux disques de parité). Les deux bandes de contrôle nécessitent des techniques de calcul de parité différentes. Soit P et Q, ces parités respectives. La parité P peut être obtenue par un simple «ou exclusif». Quant à la parité Q, on le calcule à l'aide d'un autre algorithme tel que le code de Hamming ou le code de Reed-Solomon. La figure A.8 expose une architecture RAID 6 de 5 disques.

Cette architecture est plus fiable que les autres versions RAID mais plus lente que le RAID 5 étant donné les calculs supplémentaires de parité qui doivent être effectués. Elle est aussi plus coûteuse.

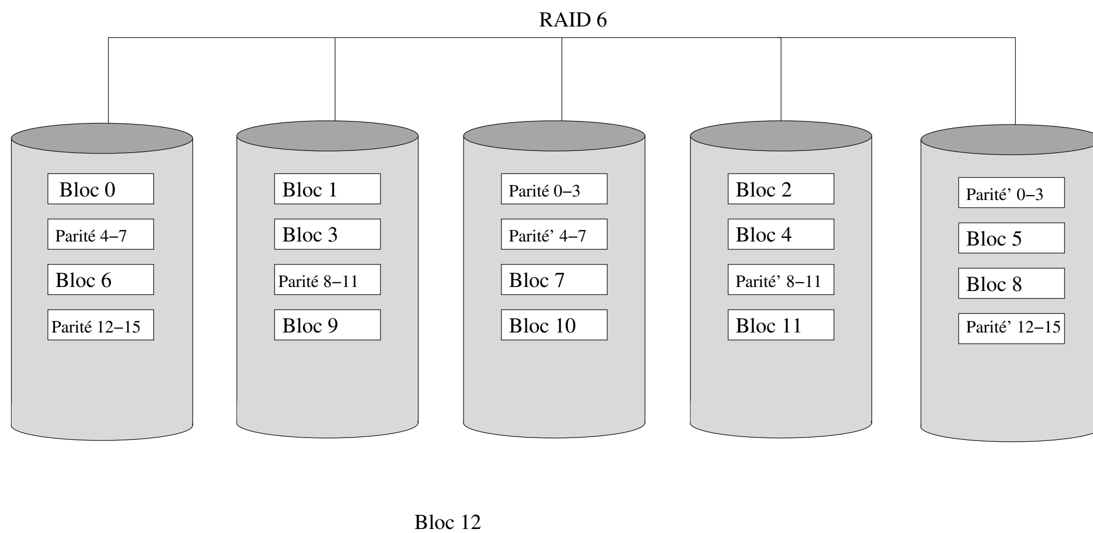


Figure A.8 – Architecture RAID de niveau 6

A.2 Mémoire stable

La mémoire stable [16, 12, 15] est un concept par lequel on tente de garder en permanence la mémoire (disque ou journal) dans un état cohérent. Ainsi, ce qui réside en mémoire stable n'est jamais, par définition, ni perdu ni corrompu même en présence de pannes matérielles ou logicielles.

Pour y parvenir on doit dupliquer l'information et rendre les opérations atomiques. Les données sont copiées sur plusieurs disques afin de tolérer les pannes matérielles reliées à un disque. Les écritures sont rendues atomiques afin de garantir qu'une éventuelle panne, lors d'une mise à jour, ne provoquera aucune corruption.

Pour bien comprendre les problématiques reliées à l'implantation de la mémoire stable, il faut analyser chacun des événements susceptibles de se produire lors d'une opération sur disque. Soit :

1. l'opération se termine avec succès ;

   Dans ce cas, toutes les copies des données sont mises à jour correctement. Les modèles considèrent généralement qu'une copie même correctement écrite peut spontanément se corrompre. Toutefois, ces événements sont si peu fréquents, que l'on considère comme nulle la probabilité que toutes les copies soient affectées simultanément, même s'il n'y en a que deux.

2. une panne pendant l'opération d'écriture ;

   La conséquence d'une panne lors de l'opération d'écriture est une potentielle corruption des données sur une copie ou une mise à jour partielle de seulement un sous-ensemble des copies des données. Toutefois, on considère que les erreurs sont toutes détectables (la probabilité qu'une erreur indétectable se produise est considérée comme négligeable [15]).

3. une panne avant l'opération d'écriture.

   Dans ce cas, la panne s'étant produite avant l'opération, les données antérieures demeurent intactes.

En supposant que l'on conserve deux copies des données, les opérations en mémoire stable sont implantées de la façon suivante :

- Écriture stable ;

  Lors d'une écriture dite stable, on effectue d'abord l'écriture des données sur un premier disque physique (copie primaire) et, seulement lorsque cette opération est terminée avec succès, on reprend l'écriture des données sur le second disque.

  Avant de déclarer une écriture (l'originale ou la copie) terminée avec succès, l'information écrite est relue à des fins de validation. Si cette lecture n'est pas conforme, les opérations d'écriture et de lecture sont ré-exécutées jusqu'à ce qu'elles le deviennent ou aient atteint un certain nombre d'essais maximum. Si cette dernière éventualité se produit, il y aura tentative d'écrire des données à un endroit différent du disque.

  S'il n'y aucune panne, les données sont considérées correctement écrites sur les deux disques.

- Lecture stable ;

  Une lecture stable s'effectue d'abord sur la copie située sur le premier disque. S'il y a une erreur, la lecture est relancée. Si après n essais le résultat est toujours erroné, la lecture est lancée sur la seconde copie. En supposant que l'écriture est toujours correcte et que les deux copies ne peuvent pas se corrompre simultanément, cette seconde lecture terminera avec succès. Par contre, si la corruption simultanée est possible, la création de plus de deux copies doit être envisagée.

- Reprise après une panne.

  Lorsqu'une panne se produit, une opération de récupération est lancée pour vérifier la validité des informations présentes sur les deux disques. Si les deux copies sont identiques, aucune action n'est entreprise. Si l'une des copies est corrompue, une nouvelle copie est créée avec l'information valide disponible sur la seconde copie. Si les deux copies contiennent des informations différentes, la contenu de la première est retranscrit sur la deuxième ou vice versa selon les auteurs [15, 12]).

  Cette façon de faire nous assure qu'une écriture termine avec succès ou qu'aucune modification n'est faite.

En l'absence de panne totale du processeur, cette implantation fonctionne très bien car l'écriture stable produit toujours deux copies valides. Évidemment, cela suppose que notre hypothèse, à savoir que les deux blocs ne peuvent spontanément se corrompre en même temps, tient la route.

Les implantations se généralisent aussi à plus de deux copies de sauvegarde mais, même si cela diminue encore le risque de pertes de données, le fait est que deux copies suffisent généralement pour obtenir un système offrant une très bonne stabilité.

Dans le but d'améliorer la performance de la mémoire stable, des optimisations sont présentées par Tanenbaum[15] et Peterson[12].

A.3 Conclusion

Enfin, il existe d'autres architectures RAID [19, 1, 11, 9, 12], principalement des cas hybrides issus de celles présentées dans ce document.

Notons de plus que les architectures RAID s'implantent aussi bien au niveau logiciel, matériel ou en une combinaison des deux [19, 9, 11].

Annexe B

Autostabilisation

B.1 Introduction

Une des propriétés les plus importantes d'un système parallèle ou distribué est la tolérance aux fautes. Être apte à résister à une très grande partie des comportements déviants (fautes ou pannes) ou malicieux (Byzantine) quand les différents composants d'un système tentent d'atteindre un consensus est d'une très grande importance pour construire des systèmes fiables. Toutefois ce type de protocole exige que la majorité des participants soit sans faute en tout temps. Il est important de noter que plus le nombre de participants à un système augmente plus les pannes sont communes.

Est-il donc possible de concevoir un système distribué qui pourrait survivre à des pannes intermittentes même si tous les nœuds sont temporairement en panne ?

Il existe plusieurs approches pour fournir la tolérance aux fautes (ou pannes) : les approches optimistes ou pessimistes, et celles avec ou sans masquage. Avec une approche pessimiste, on implante des algorithmes robustes protégés contre toutes les pannes possibles. Avec une approche optimiste on peut utiliser des algorithmes auto-stabilisateurs qui offrent la garantie, après une panne, d'automatiquement atteindre un état légal en un temps fini.

Une approche avec masquage s'assure que la couche application ne voit pas les fautes (elle utilise la redondance et la réplication). Une approche sans masquage implique que, lorsqu'une déviation comportementale se produit, celle-ci soit détectée et corrigée (reprise arrière ou avant).

La notion d'auto-stabilisation, introduite en 1974 par Dijkstra[4], est une technique de tolérance aux fautes (ou pannes) sans masquage et optimiste. Optimiste car elle ne vise pas à tolérer toutes les pannes et avec marquage car la reprise après une panne implique le passage par un certain nombre d'états illégitimes.

B.2 Définition de l'auto-stabilisation

Un système distribué est auto-stabilisateur [17, 13, 4, 18, 10, 2, 5, 7] si, à partir d'un état arbitraire (mal initialisé ou perturbé), il est garantie de converger vers un état légitime en un temps

fini et cela par lui-même sans aucune intervention externe. Si le système est dans un état légitime, il est garantie qu'il demeurera dans cet état, si aucune nouvelle faute ne se produit. Un état est légitime s'il satisfait les spécifications du système distribué.

L'habileté de ces algorithmes de pouvoir effectuer une reprise à partir de n'importe quel état implique qu'aucune phase d'initialisation n'est requise. L'auto-stabilisation assure donc une reprise après une panne. Cependant le système peut produire de mauvais résultats pendant une période finie après la ré-initialisation.

Note sur les algorithmes auto-stabilisateur

Un algorithme auto-stabilisateur n'a pas besoin d'être initialisé «correctement».

Plus formellement un système distribué comprend un certain nombre de machines que nous identifions plus simplement comme des processus. Un système comprend deux composants : des processus et des canaux de communication. La topologie du système est un graphe orienté, les nœuds étant les processus et les arcs les canaux de communications entre les processus. Chaque composant possède un état local. Nous définissons un état global comme l'union de l'état de tous les composants (processus et canaux de communication). Le comportement du système consiste en un ensemble d'états, une relation de transition entre les états et un ensemble de critères d'équité sur la transition de relation.

Un système est donc une paire $S = (C, \rightarrow)$ où C représente l'ensemble des états globaux du système S et $\rightarrow$ est une relation de transition binaire dans C. Une exécution de S est une séquence non-vide d'états globaux $(c_1, c_2, ...)$ telle que $\forall i \geq 0 : c_i \in C$ et $c_i \rightarrow c_{i+1}$.

Definition 1 *On défini l'auto-stabilisation pour un système S par rapport à un prédicat P sur un ensemble d'états globaux où P sert à identifier les exécutions valides. S possède la propriété d'auto-stabilisation par rapport à P s'il satisfait les deux propriétés suivantes :*

- *Fermeture : P est fermé sous l'exécution de S. Cela signifie qu'une fois que le système atteint un bon comportement (satisfaisant P) il reste dans l'ensemble des états légitimes (P reste vrai) en l'absence de nouvelles fautes.*
- *Convergence : Quand une faute se produit qui laisse le système S dans un état global arbitraire, S est garantie d'atteindre un état global satisfaisant P au bout d'un nombre fini de transitions d'états.*

Tous les états du système satisfaisant un prédicat P sont appelées des états légitimes. Les autres sont appelés des états illégitimes.

Definition 2 *Alternativement, il est aussi possible de définir l'auto-stabilisation en terme de séquence d'exécution valide. Une séquence d'exécution est dites valide si chaque état global dans cette séquence est légitime. Ainsi, un système est auto-stabilisateur s'il existe un sous-ensemble $L \subset C$ de tous les états légitimes telle que :*

- *chaque exécution commençant par un état dans L est valide ;*

- *chaque exécution de S contient un état de L (convergence).*

Pour respecter notre première définition, on exige aussi que le sous-ensemble L soit fermé, i.e. qu'en l'absence de fautes chaque transition partant d'un état légitime amène le système dans un nouvel état légitime.

Cette dernière définition a permis de mieux présenter une nouvelle approche de conception d'algorithmes auto-stabilisateurs. Elle permet aussi de visualiser l'auto-stabilisation en terme de suffixe valide. Soit un état arbitraire c_a provoqué par un faute intermittente (message perdu ou corrompu, panne d'un processeur, ...). Soit SE l'ensemble des exécutions valides pour le système. Un état c_s est considéré sûr si toutes les exécutions admissibles à partir de cet état sont dans SE. La figure B.1 «illustre» cette définition de l'auto-stabilisation.

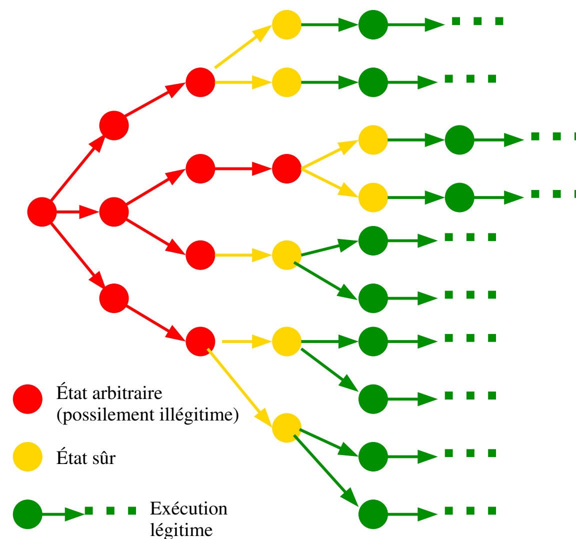


Figure B.1

B.2.1 Exemple : anneau à jeton

Pour bien comprendre ce concept, prenons comme exemple l'algorithme auto-stabilisateur développé par Dikstra[4]. L'algorithme de Dijkstra assure l'exclusion mutuelle sur une anneau sur lequel circule un jeton. Le jeton circule grâce à de la mémoire partagée entre les nœuds voisins. La possession du jeton donne accès à la section critique. Chaque nœud de l'anneau possède un des deux états suivants :

- le nœud possède le jeton ;
- le nœud ne possède pas le jeton.

L'état global est l'union de tous les états des nœuds. Un état est donc légitime s'il satisfait les contraintes (prédicats) suivantes :

1. Il doit y avoir au moins un jeton en circulation ;
2. Il doit y avoir un seul jeton en circulation (exclusion mutuelle) ;
3. Toute action à partir d'un état légitime doit amener le système dans un nouvel état légitime (fermeture) ;
4. Pendant une exécution (infinie), chaque machine doit pouvoir posséder le jeton un nombre infini de fois (pas de famine) ;
5. Soit deux états légitimes, il y a une série d'actions qui amènent les système d'un état légitime vers l'autre

Il est intéressant de noter que tous les nœuds peuvent se trouver dans un état local valide même si l'état global est invalide. Par exemple, si tous les nœuds sont dans l'état «le nœud ne possède pas de jeton», alors l'état global ne sera pas légitime (ne satisfait pas la contrainte 1). Un autre exemple est que si plusieurs nœuds sont dans l'état «le nœud possède le jeton», alors l'état global ne satisfait pas la contrainte 2.

B.3 Prémisses à l'utilisation de l'auto-stabilisation

L'auto-stabilisation fonctionne quand les fautes sont intermittentes, peu fréquentes et qu'un mauvais fonctionnement temporaire est acceptable. On suppose aussi que la faute intermittente peut affecter l'état du système mais pas le comportement. Cela signifie qu'elle peut seulement modifier l'état local d'un processus (emmagasiné en mémoire ou dans des registres) ou le contenu des messages. Elle n'affecte jamais le code du programme.

Si une faute se produit trop fréquemment, les systèmes auto-stabilisateurs ne sont plus efficaces. De plus, ils ne supportent pas les pannes impliquant l'arrêt complet de certains composants.

Toutefois, un avantage de l'auto-stabilisation est qu'elle traite toutes ces erreurs de la même façon. En effet, ces algorithmes n'ont besoin d'aucune information sur la panne (type, durée, étendue et même si elle s'est vraiment produite). La seule condition est que la panne ne soit pas permanente et qu'elle cessera éventuellement d'affecter le système. de plus, même si une panne amène le système dans un état non légitime, les prochaines étapes d'exécutions du même code vont ramener automatiquement le système dans un état stable. Traditionnellement, ces erreurs sont toutes gérées séparément (par un système de prise en main des exceptions par exemple). Pour plus de détails, vous pouvez consulter l'article de Schneider[10].

B.4 Environnements propices à l'auto-stabilisation

Les systèmes auto-stabilisateurs sont particulièrement utiles dans les environnements ou domaines où l'intervention d'un humain pour rétablir le système après une défaillance est impossible

ou dans lesquels il est préférable de s'en passer : les réseaux informatiques, les réseaux de capteurs et les systèmes critiques, tels que les satellites. Jusqu'à maintenant, on retrouve des exemples de systèmes auto-stabilisateurs en réseau et en robotique. Wikipedia[17] donne une exemple détaillé d'en environnement propice à l'utilisation de l'auto-stabilisation (réseau de capteurs sans fil). De plus, cette approche peut être très utile dans des environnements qui ne peuvent pas être facilement initialisés correctement. Ainsi, il est difficile d'exiger qu'un système multi-processeurs puisse être initialisé correctement. De même, il est difficile d'imaginer devoir arrêter et ré-initialiser un réseau à chaque fois qu'on lui ajoute ou retire un nœud.

Les algorithmes auto-stabilisateur sont intéressants dans d'autres contextes mais il faut se souvenir qu'ils ne peuvent pas servir à tolérer tous les types de fautes. De façon générale, comme les algorithmes auto-stabilisateurs ne nécessitent aucune initialisation, qu'ils peuvent récupérer de tous les types de pannes transitoires et sont insensibles au reconfiguration dynamique, ils sont des candidats intéressants pour de multiples applications tels que la réinitialisation distribuée, les protocoles de routage et de communication, la synchronisation des horloges et la théorie des graphes. Ainsi c'est un candidat idéal pour traiter facilement la panne d'un nœud (ainsi que sa ré-insertion).

Tous ces cas de défaillances sont des cas particuliers d'une perte de coordination. La coordination est dite perdue si, pour un état global particulier d'un système distribué, les états locaux des différents processus du système sont incohérents les uns par rapport aux autres (dans cet état global) et cela même si chacun des états locaux pris individuellement est cohérent. Tous les programmes auto-stabilisateurs peuvent récupérer d'un perte de coordination.

Un exemple de perte de coordination est la perte d'un jeton sur un réseau en anneau. L'état individuel de chaque site est correct quand il indique que le site ne possède pas le jeton. Toutefois l'état global est incorrect car dans l'état global la présence d'un jeton unique est obligatoire.

Les causes d'une perte de coordination sont multiples. Cela peut être une mauvaise initialisation, un changement dans le mode de fonctionnement, une erreur de transmission, une panne d'un processus (suivi d'une reprise) ou une panne de la mémoire.

B.5 Conception d'un algorithme auto-stabilisateur

La conception d'un algorithme auto-stabilisateur est un problème extrêmement complexe. En effet, pouvoir détecter une panne et effectuer une reprise même si un état légitime ne peut être évalué par un seul processus du système distribué (propriété de vérification locale) n'est pas simple. Le cas de l'anneau à jeton est un bon exemple. Dans ce cas, aucun processus ne peut détecter un état global légitime (ou non). Malgré cela, l'algorithme auto-stabilisateur doit être capable de récupérer d'une faute. Cela suggère que l'auto-stabilisation est une sorte d'intelligence collective où chaque composant exécute des actions locales basé sur sa connaissance locale mais qu'éventuellement cela garanti la convergence.

La conception d'algorithmes auto-stabilisateur soulève un certains nombres de défis associés :

- au nombre d'états de chaque sous-système dans le système distribué ;
- à l'uniformité (veut-on un algorithme identique pour tous ou non) ;

- au contrôle centralisé ou non ;
- au coût de l'algorithme (en terme d'étapes avant d'atteindre un état légitime).

Dans la littérature [4, 13, 17, 10, 8, 7], trois techniques ont été proposées pour concevoir des algorithmes auto-stabilisateurs.

1. La première technique consiste à concevoir un algorithme auto-stabilisateur à partir de rien. Avec cette approche, l'algorithme est conçu de façon à ce que le système qui l'intègre exécute les mêmes opérations qu'ils soient dans un état légitime ou non. Cette approche est la plus complexe. Il n'existe pas de méthodes pour aider à concevoir ce type d'algorithme. C'est l'approche utilisé par Dijkstra[4].

2. Katz et al.[7, 13], propose de concevoir un algorithme auto-stabilisateur en ajoutant une extension auto-stabilisatrice à un algorithme standard. Cette approche consiste à surimposer un programme «S» à un programme normal «P» qui fonctionne de la façon suivante :

   - Après chaque «k» étapes de l'exécution de «P», le programme «S» capture l'état global de «P» et vérifie s'il est légitime ;
   - S'il n'est pas légitime, le programme «S» ré-initialise l'état de chaque processus pour obtenir un état global légitime et poursuit l'exécution de «P» ;
   - s'il est légitime, il poursuit l'exécution de «P».

   Cette approche, quoi qu'intéressante, est peu pratique car capturer un état global cohérent est complexe et coûteux. Plus récemment des extensions auto-stabilisatrices ont été développées qui ne nécessitent pas la capture d'un état global cohérent. En effet, elles peuvent détecter une panne en regardant seulement l'état des voisins immédiats.

3. Dans certains cas [10], on peut concevoir un algorithme auto-stabilisateur en combinant plusieurs algorithmes qui possèdent déjà la propriété d'auto-stabilisation. Schneider [10] donne des exemples de conception de tels algorithmes.

Plus récemment, un lien a aussi été établie entre les algorithmes auto-stabilisateurs et la théorie des jeux. Cela permettra possiblement d'aider à la conception de nouveaux algorithmes auto-stabilisateurs.

B.6 Exemple d'algorithme : anneau à jeton

Nous avons déjà présenté cet exemple. Il s'agit de faire circuler un jeton sur un réseau en anneau. La possession du jeton permet d'avoir un accès exclusif à une ressource. Le concept d'état légitime a déjà été défini pour cet exemple.

Le premier algorithme, proposé par Dijkstra, n'est pas uniforme et utilise de la mémoire partagé pour faire circuler le jeton. Par non uniforme, on signifie qu'il y a processus exceptionnel qui exécute un code différent des autres. Le programme 1 montre ce premier algorithme. Dans cet exemple, on retrouve N processus, numérotés de 0 à N-1, formant un anneau unidirectionnel. Chaque processus

$p[i]$ possède un état $e[i]$ qui peut prendre une valeur entière entre 0 et $k-1$. Le processus $p[0]$ est le processus exceptionnel. Le processus qui est couramment en exécution est dénommé PC. Dans cet algorithme, lorsque la condition testée (`e[0] = e[N-1]` ou `e[i] <> e[i-1]`) est satisfaite cela signifie que le processus possède le jeton et peut accéder à la section critique. Le programme 2 montre le code C++ implantant ce même algorithme. Les figures B.2 à B.4 montrent des cas d'exécution de cet algorithme. La figure B.2 montre le résultat d'une exécution démarrant dans un état légitime. La figure B.3 montre le résultat d'une exécution démarrant dans un état illégitime dans lequel il y a deux jetons. Enfin, la figure B.4 montre le résultat d'une exécution démarrant dans un état illégitime dans lequel il y a quatre jetons.

```
if PC = p[0] then

     // Le processus exceptionnel
     if e[0] = e[N-1] then
            // SECTION CRITIQUE
            e[0] := (e[N-1]+1) mod K
     end if
else
     // Les autres processus
     if e[i] <> e[i-1] then
            // SECTION CRITIQUE
            e[i] := e[i-1]
     end if

end if
```

Programme 1 – Algorithme auto-stabilisateur pour un anneau à jeton.

```
_____________________________________
|  P0  |  P1  | P2  |  P3  |  P4 |
|______|______|_____|______|_____|
|      |      |     |      |  *  |
|______|______|_____|______|_____|
|  *   |      |     |      |     |
|______|______|_____|______|_____|
|      |  *   |     |      |     |
|______|______|_____|______|_____|
|      |      |  *  |      |     |
|______|______|_____|______|_____|
|      |      |     |  *   |     |
|______|______|_____|______|_____|
|      |      |     |      |  *  |
|______|______|_____|______|_____|
|  *   |      |     |      |     |
|______|______|_____|______|_____|
|      |  *   |     |      |     |
|______|______|_____|______|_____|
|      |      |  *  |      |     |
|______|______|_____|______|_____|
|      |      |     |  *   |     |
|______|______|_____|______|_____|
|      |      |     |      |  *  |
|______|______|_____|______|_____|
```

Figure B.2 – Résultat d'un exécution à partir d'un état légitime

```
void pcs_exc()
{
    while(true)
    {
        this_thread::sleep_for(chrono::milliseconds(10));
        s1.lock();
        if (etat[0] == etat[4])
        {
            jeton[0] = '*'; // SECTION CRITIQUE
            etat1[0] = (etat[0] + 1 ) % K;
        }
        else  etat1[0]  = etat[0];
        fin_tour(0);
    }
}
void pcs_autres(int i)
{
     while(true)
    {
        s1.lock();
        if (etat[i] != etat[i-1])
        {
            jeton[i] = '*'; // SECTION CRITIQUE
            etat1[i] = etat[i-1];
        }
        else  etat1[i]  = etat[i];
        fin_tour(i);
     }
}
```

Programme 2 – Algorithme auto-stabilisateur en C++ pour un anneau à jeton.

```
-------------------------------------
|  P0  |  P1  | P2  |  P3  |  P4 |
|______|______|_____|___*__|__*__|
|______|______|_____|______|__*__|
|__*___|______|_____|______|_____|
|______|__*___|_____|______|_____|
|______|______|__*__|______|_____|
|______|______|_____|___*__|_____|
|______|______|_____|______|__*__|
|__*___|______|_____|______|_____|
```

Figure B.3 – Résultat d'un exécution à partir d'un état illégitime (2 jetons)

P0	P1	P2	P3	P4
	*	*	*	*
*		*	*	*
	*		*	*
*		*		*
	*		*	
		*		*
			*	
				*
*				
	*			
		*		
			*	
				*

Figure B.4 – Résultat d'un exécution à partir d'un état illégitime (4 jetons)

B.7 Conclusion

L'auto-stabilisation est un outil intéressant pour faire de la reprise avant dans un système comprenant un ensemble de processus parallèles. Il est toutefois complexe de concevoir un algorithme auto-stabilisateur. De plus, les pannes que ce type d'algorithmes peut traiter sont limités.

Toutefois pour des cas bien précis c'est une approche élégante et efficace pour traiter les fautes.

Bibliographie

[1] Sylvain ADAMI : Le raid et ses différents types. https://www.supinfo.com/articles/single/1176-raid-ses-differents-types, 2015.

[2] Jerzy Brzeziński et Michał Szychowiak : Self-stabilization in distributed systems – a short survey. *Foundations of Computing and Decision Sciences*, 25(1):3–22, 2000.

[3] D.M. Dhamdhere : *Operating Systems : A Concept-based Approach*. Tata McGraw-Hill Pub., 2012.

[4] Edsger W. Dijkstra : Self-stabilizing systems in spite of distributed control. *Commun. ACM*, 17(11):643–644, novembre 1974.

[5] Shlomi Dolev : *Self-stabilization*. MIT Press, Cambridge, MA, USA, 2000.

[6] Culture Informatique : C'est quoi le raid ? https://www.culture-informatique.net/cest-quoi-raid, 2017.

[7] Shmuel Katz et Kenneth J. Perry : Self-stabilizing extensions for meassage-passing systems. *Distributed Computing*, 7(1):17–26, novembre 1993.

[8] Ajay D. Kshemkalyani et Mukesh Singhal : *Distributed Computing : Principles, Algorithms, and Systems*. Cambridge University Press, USA, 1 édition, 2008. Voir chapitre 17.

[9] Jean-François Pilou, Galilée et et al. : Le raid c'est quoi ? https://www.commentcamarche.net/faq/159-le-raid-c-est-quoi, 2016.

[10] Marco Schneider : Self-stabilization. *ACM Comput. Surv.*, 25(1):45–67, mars 1993.

[11] Amen School : Le raid c'est quoi ? https://www.amenschool.fr/raid-informatique-quest-ce-que-cest/, 2017.

[12] Abraham Silberschatz, Peter B. Galvin et Greg Gagne : *Operating System Concepts*. Wiley Publishing, 9th édition, 2012.

[13] sss2019 : Self-stabilization. http://www.selfstabilization.org/~selfstab/, 2019.

[14] William Stallings : *Operating Systems : Internals and Design Principles*. Prentice Hall Press, USA, 7th édition, 2011.

[15] Andrew S. Tanenbaum et Herbert Bos : *Modern Operating Systems*. Prentice Hall Press, USA, 4th édition, 2014.

[16] WIKIPEDIA : Stable storage. https://en.wikipedia.org/wiki/Stable_storage, 2018.

[17] WIKIPEDIA : Autostabilisation. https://fr.wikipedia.org/wiki/Autostabilisation, 2019.

[18] WIKIPEDIA : Self-stabilization. https://en.wikipedia.org/wiki/Self-stabilization, 2019.

[19] WIKIPEDIA : Raid (informatique). https://fr.wikipedia.org/wiki/RAID_(informatique), 2020.

[20] WIKIPEDIA : Black-box testing. https://en.wikipedia.org/wiki/Black-box_testing, 2021.

[21] WIKIPEDIA : Boîte blanche. https://fr.wikipedia.org/wiki/Bo%C3%AEte_blanche, 2021.

[22] WIKIPEDIA : Test de la boîte noire. https://fr.wikipedia.org/wiki/Test_de_la_bo%C3%AEte_noire, 2021.

[23] WIKIPEDIA : White-box testing. https://en.wikipedia.org/wiki/White-box_testing, 2021.

[24] WIKIPEDIA : Gray box testing. https://en.wikipedia.org/wiki/Gray_box_testing, 2022.