Dans « Encyclopédie de l'informatique et des systèmes d'information », (J. Akoka and I. Comyn-Wattiau, Eds.),
Partie 1 : La dimension technologique des systèmes d'information - Section 2 : L'architecture et les systèmes (M. Banâtre, Ed.),
pp. 240-270 ISBN 27117-4846-4,Vuibert, Paris, France, 2006

Tolérance aux fautes

Jean Arlat, Yves Crouzet, Yves Deswarte, Jean-Charles Fabre, Jean-Claude Laprie, David Powell

Mots-clés : sûreté de fonctionnement, tolérance aux fautes, fautes physiques, fautes de conception, fautes d'interaction, malveillances, détection d'erreur, rétablissement du système, systèmes répartis, architectures tolérantes aux fautes.

Résumé : la tolérance aux fautes est l'aptitude d'un système informatique à accomplir sa fonction malgré la présence ou l'occurrence de fautes, qu'il s'agisse de dégradations physiques du matériel, de défauts logiciels, d'attaques malveillantes, d'erreurs d'interaction homme-machine. Après une vision d'ensemble de la tolérance aux fautes et de sa place au sein de la sûreté de fonctionnement, les techniques de tolérance aux diverses classes de fautes sont présentées, ainsi que leur mise en œuvre. Ces techniques sont illustrées par la description de systèmes opérationnels représentatifs. Le chapitre se conclut par l'examen des défis posés à la tolérance aux fautes par les évolutions des systèmes informatiques.

Différentes formes de redondance ont été employées dès les origines de l'informatique : les composants utilisés dans les premiers calculateurs étaient si peu fiables que le recours à la redondance était essentiel afin de permettre au calculateur de mener à bien un calcul conséquent. En effet, l'UNIVAC 1 (au tout début des années 1950) mettait en œuvre des unités arithmétiques et logiques dupliquées et faisait un usage intensif de contrôles par parité. Toujours dans les années 1950, un autre effort significatif, mais moins connu, concerne le calculateur redondant SAPO développé à l'université de Prague.

Le terme « calculateur tolérant aux fautes » (en anglais, *fault-tolerant computer*) est associé aux travaux pionniers relatifs à la fois aux calculateurs (tels que le calculateur STAR du JPL) des premières expériences spatiales visant à assurer des missions de dix ans ou plus, et aux premiers autocommutateurs numériques des centraux téléphoniques (notamment, la série des ESS de Bell).

Bien sûr, depuis, la fiabilité des composants matériels s'est considérablement améliorée. Néanmoins, compte tenu de l'informatisation croissante des différentes facettes de notre société, il faut tenir compte des risques potentiels majeurs pouvant résulter de la défaillance des calculateurs, comme de nombreux exemples l'attestent : pertes de vies humaines, atteinte à la santé, dommages à l'environnement, pertes économiques. Aussi, les techniques de tolérance aux fautes constituent un moyen essentiel pour permettre de faire confiance aux calculateurs mis en jeu dans le cadre d'applications critiques. Actuellement, des mécanismes de tolérance aux fautes sont couramment employés afin de protéger les systèmes informatiques critiques, non seulement vis-à-vis des fautes physiques des composants matériels, mais aussi des fautes de conception du matériel et du logiciel, des fautes des opérateurs lors des interactions homme-machine, y compris les fautes intentionnelles (les malveillances).

Ce chapitre reprend et actualise les points principaux contenus dans (Laprie *et al.* 1996, Arlat *et al.* 1999). Il présente les principes liés à la conception de systèmes informatiques tolérants aux fautes. Nous introduisons tout d'abord les *concepts de base et la terminologie associée* à la sûreté de fonctionnement informatique. La section **2** présente successivement les techniques clés pour la *détection d'erreur* et le *rétablissement du système*. La section **3** met l'accent sur les aspects spécifiques à la tolérance aux fautes dans le cadre des systèmes répartis. Les spécificités relatives à la prise en compte des catégories de fautes tout particulièrement sensibles font respectivement l'objet des sections **4**, **5** et **6** : les *fautes de développement*, les *fautes d'interaction accidentelles homme-machine* et les *malveillances*. La section **7** présente des *exemples de systèmes opérationnels tolérant les fautes accidentelles*. Enfin, la section **8** identifie quelques *défis et tendances* en tolérance aux fautes informatique.

1. Concepts de base

Cette section a pour objectif d'introduire le concept de sûreté de fonctionnement au sein duquel la tolérance aux fautes joue un rôle majeur. Il vise à donner des définitions précises caractérisant les principes qui entrent en jeu dans la sûreté de fonctionnement des systèmes informatiques. Ce texte résulte d'un résumé de (Avižienis *et al.* 2004, Laprie 2004) ; le lecteur est invité à se reporter à ces références pour une vue plus complète et plus détaillée.

La section est composée de sept parties : définitions de base sur les notions de fonction et de service, puis de la sûreté de fonctionnement, les attributs et les entraves de la sûreté de fonctionnement, les techniques de la tolérance aux fautes, sa mise en œuvre et son analyse. Les définitions données sont suffisamment générales pour couvrir le spectre complet des systèmes informatiques, des portes logiques aux réseaux d'ordinateurs, y compris leurs opérateurs et utilisateurs humains.

1.1. Fonction et service d'un système

Un *système* est une entité qui interagit avec d'autres entités, donc d'autres systèmes, y compris le matériel, le logiciel, les humains, et le monde physique avec ses phénomènes naturels. Ces autres systèmes qui interagissent avec le système considéré constituent l'*environnement* du système considéré. La *frontière* du système est la limite commune entre le système et son environnement.

La *fonction* d'un système est ce à quoi il est destiné. Elle est décrite par la *spécification fonctionnelle***,** qui inclut les performances attendues du système. Le *comportement* d'un système est ce que le système fait pour accomplir sa fonction, et est représenté par une séquence d'états.

Le *service* délivré par un système est son comportement tel que perçu par ses utilisateurs (séquence d'états externes) ; un *utilisateur* est un autre système, éventuellement humain, qui interagit avec le système considéré.

Nous avons utilisé jusqu'à présent le singulier pour fonction et service. Un système généralement accomplit plusieurs fonctions, et délivre plusieurs services. Fonction et service peuvent donc être vus comme constitués de *fonctions élémentaires* et de *services élémentaires*.

1.2. Sûreté de fonctionnement

La *sûreté de fonctionnement* d'un système est son aptitude à délivrer un service de confiance justifiée. Cette définition met l'accent sur la justification de la *confiance*, cette dernière pouvant être définie comme une dépendance acceptée, explicitement ou implicitement. La *dépendance* d'un système d'un autre système est l'influence, réelle ou potentielle, de la sûreté de fonctionnement de ce dernier sur la sûreté de fonctionnement du système considéré.

Selon la, ou les applications auxquelles le système est destiné, l'accent peut être mis sur différentes facettes de la sûreté de fonctionnement, ce qui revient à dire que la sûreté de fonctionnement peut être vue selon des propriétés différentes, mais complémentaires, qui permettent de définir ses *attributs* :

- le fait d'être prêt à l'utilisation conduit à la *disponibilité ;*
- la continuité du service conduit à la *fiabilité* ;
- l'absence de conséquences catastrophiques pour l'environnement conduit à la *sécurité-innocuité* ;
- l'absence de divulgations non autorisées de l'information conduit à la *confidentialité* ;
- l'absence d'altérations inappropriées de l'information conduit à l'*intégrité* ;
- l'aptitude aux réparations et aux évolutions conduit à la *maintenabilité*.

L'association, à la confidentialité, de l'intégrité et de la disponibilité vis-à-vis des actions autorisées, conduit à la *sécurité-immunité*[1]**.**

Un *service correct* est délivré par un système lorsqu'il accomplit sa fonction. Une *défaillance du service*, souvent simplement dénommée *défaillance***,** est un événement qui survient lorsque le service délivré dévie du service correct. Le service défaille soit parce qu'il ne respecte plus la spécification fonctionnelle, soit parce que la spécification fonctionnelle ne décrivait pas de manière adéquate la fonction du système. Une défaillance du service est donc une transition de service correct à *service incorrect*, c'est-à-dire ne remplissant pas la fonction du système. La délivrance d'un service incorrect est une *panne du service*. La transition de service incorrect à service correct est la *restauration du service*. La déviation du service correct peut prendre plusieurs formes, qui sont les *modes de défaillance* ; les conséquences des défaillances sur l'environnement du système sont classées et ordonnées selon la *gravité des défaillances*. Lorsque la fonction du système comporte un ensemble de fonctions élémentaires, la défaillance d'un ou plusieurs services remplissant ces fonctions peut laisser le système dans un *mode dégradé*, qui offre encore un sous-ensemble de services à l'utilisateur.

[1] L'association des qualificatifs *innocuité* et *immunité* permet de lever l'ambiguïté associée à *sécurité*. Il est à noter que cette ambiguïté n'existe pas en anglais, qui dispose de 'safety' pour la sécurité-innocuité et de 'security' pour la sécurité-immunité, sûreté de fonctionnement étant 'dependability'.

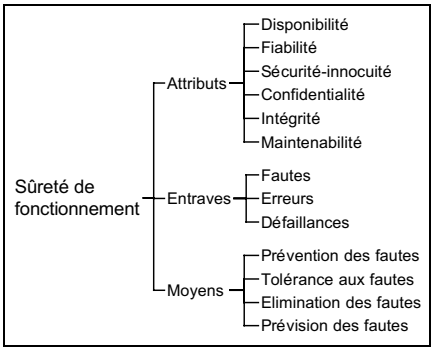


Figure 1.1 - L'arbre de la sûreté de fonctionnement

Plusieurs de ces modes dégradés peuvent être identifiés, tels que service ralenti, service restreint, service d'urgence, etc. Dans ce cas, le système est dit avoir souffert *de défaillances partielles*.

Le service délivré étant une séquence d'états externes, une défaillance du service signifie qu'au moins un état externe dévie du service correct. La déviation est une *erreur*. La cause adjugée ou supposée d'une erreur est une *faute*. Les fautes peuvent être internes ou externes au système. La présence antérieure d'une *vulnérabilité*, c'est-à-dire d'une faute interne qui permet à une faute externe de causer des dommages au système, est nécessaire pour qu'une faute externe entraîne une erreur, et, éventuellement, une défaillance. Généralement, une faute cause d'abord une erreur dans l'état interne d'un composant, l'état externe du système n'étant pas immédiatement affecté. Il s'ensuit la définition d'une *erreur* : partie de l'état total du système qui est susceptible d'entraîner sa défaillance, qui survient lorsque l'erreur affecte le service délivré à l'utilisateur. Il est à noter que nombre d'erreurs n'affectent pas l'état externe du système, et donc ne causent pas de défaillance.

La *spécification de sûreté de fonctionnement* d'un système décrit ce qui est requis pour les attributs de la sûreté de fonctionnement en termes de fréquence et de gravité des défaillances du service pour un ensemble de fautes donné, pour un environnement opérationnel donné. Certains attributs peuvent ne pas être requis pour un système donné.

Il s'ensuit une définition alternative de la *sûreté de fonctionnement*, qui complémente la définition initiale en procurant un critère pour décider si la confiance peut être ou non accordée au service: aptitude à éviter des défaillances du service plus fréquentes ou plus graves qu'acceptable. Des défaillances du service plus fréquentes ou plus graves qu'acceptable manifestent une *défaillance de la sûreté de fonctionnement*.

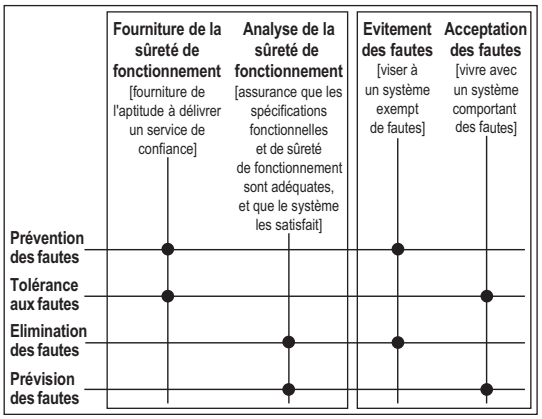


Figure 1.2 - Groupements des moyens pour la sûreté de fonctionnement

Le développement d'un système sûr de fonctionnement passe par l'utilisation combinée d'un ensemble de méthodes qui peuvent être classées en :

- *prévention des fautes* : comment empêcher l'occurrence ou l'introduction de fautes ;
- *tolérance aux fautes* : comment fournir un service à même de remplir la fonction du système en dépit des fautes ;
- *élimination des fautes* : comment réduire la présence (nombre, sévérité) des fautes ;
- *prévision des fautes* : comment estimer la présence, le taux futur, et les possibles conséquences des fautes.

Les principales notions qui ont été introduites peuvent être résumées par l'arbre de la sûreté de fonctionnement, comme indiqué par la figure **1.1**. La figure **1.2** illustre les différents groupements des moyens pour la sûreté de fonctionnement

Les paragraphes suivants traitent des attributs et les entraves à la sûreté de fonctionnement, puis de la mise en œuvre et l'analyse de la tolérance aux fautes qui est le point central de ce chapitre. Une analyse plus fouillée de ces définitions ainsi que des rôles et des liens entre tolérance aux fautes, élimination de fautes et prévision de fautes peut être trouvée dans (Laprie *et al.* 1996, Avižienis *et al.* 2004, Laprie 2004)

1.3. Attributs de la sûreté de fonctionnement

Les attributs de la sûreté de fonctionnement qui ont été définis au § **1.1** peuvent voir leur importance différer selon l'application à laquelle est destinée le système informatique considéré : disponibilité, intégrité, et maintenabilité sont généralement requis (à des degrés variables selon l'application), alors que fiabilité, sécurité-innocuité, confidentialité peuvent ou non être requis. La mesure dans laquelle un système possède les attributs de la sûreté de fonctionnement doit être

considérée de façon relative, probabiliste, et non de façon absolue, déterministe : du fait de l'inévitable présence ou occurrence de fautes, un système n'est jamais totalement disponible, fiable ou sûr.

La définition donnée pour la maintenabilité déborde intentionnellement la maintenance curative ou préventive, pour s'étendre aux autres formes de maintenance, adaptative et perfective.

Nous n'avons pas fait apparaître la sécurité-immunité comme un attribut en tant que tel de la sûreté de fonctionnement. Ceci est en accord avec les définitions usuelles, qui la présentent comme une notion composite : "combinaison de confidentialité, prévention de la divulgation non autorisée de l'information, d'intégrité, prévention d'une modification non autorisée de l'information, et de disponibilité, prévention d'un déni non autorisé d'accès à l'information ou à des ressources".

Au-delà des attributs définis au paragraphe **1.1**, que l'on peut qualifier d'attributs primaires, d'autres attributs peuvent être définis en tant qu'attributs secondaires, qui affinent ou spécialisent les attributs primaires. Un exemple d'attribut secondaire spécialisé est la *robustesse* : la sûreté de fonctionnement par rapport aux fautes externes. La notion d'attribut secondaire est particulièrement pertinente pour la sécurité-immunité, lorsque l'on distingue différents types d'information :

- la *responsabilité*, qui est la disponibilité et l'intégrité de l'identité de la personne qui a effectué une opération ;
- l'*authenticité*, qui est l'intégrité du contenu et de l'origine d'un message, et éventuellement d'autres informations, comme l'instant d'émission ;
- la *non-réfutabilité*, qui est la disponibilité et l'intégrité de l'identité de l'émetteur d'un message (non réfutation de l'origine), ou du destinataire (non réfutation de la destination).

Les variations dans l'accent mis sur les attributs de la sûreté de fonctionnement ont une influence directe sur le dosage des techniques à mettre en œuvre pour que le système résultant soit sûr de fonctionnement. Ceci est un problème d'autant plus délicat que certains attributs sont antagonistes (par exemple, disponibilité et sécurité-innocuité, disponibilité et sécurité-immunité), d'où la nécessité de compromis.

1.4. Entraves à la sûreté de fonctionnement

1.4.1. Fautes

Les fautes susceptibles d'affecter un système peuvent être classées selon huit points de vue, permettant de définir les classes de fautes élémentaires, comme indiqué sur la figure **1.3**.

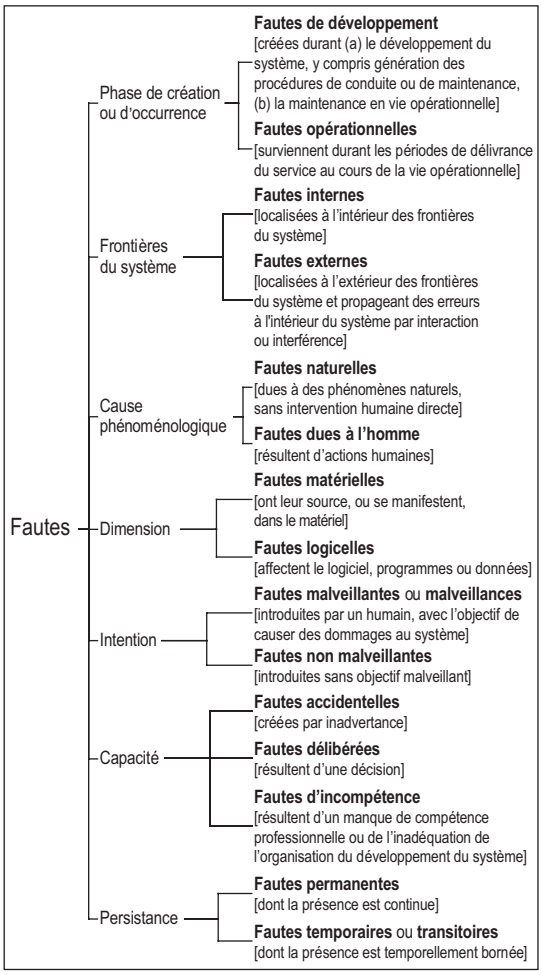


Figure 1.3 - Classes de fautes élémentaires

La figure **1.4** donne les classes de fautes combinées, résultant de la combinaison des huit classes de fautes élémentaires. Il est à noter que si toutes les combinaisons des classes de fautes élémentaires étaient possibles, il y aurait 192 classes de fautes combinées ; la figure n'en recense que 25, certaines combinaisons n'étant pas pertinentes. Comme l'indique la figure **1.4**, les classes de fautes combinées peuvent être regroupées en trois grandes classes non exclusives :

- fautes de développement, qui rassemblent les fautes pouvant survenir durant le développement ;
- fautes physiques, qui rassemblent les fautes affectant le matériel ;
- fautes d'interaction, qui rassemblent les fautes externes.

La figure **1.5** donne des exemples illustratifs des classes de fautes combinées, les numéros étant ceux des feuilles de l'arbre de la figure **1.4**.

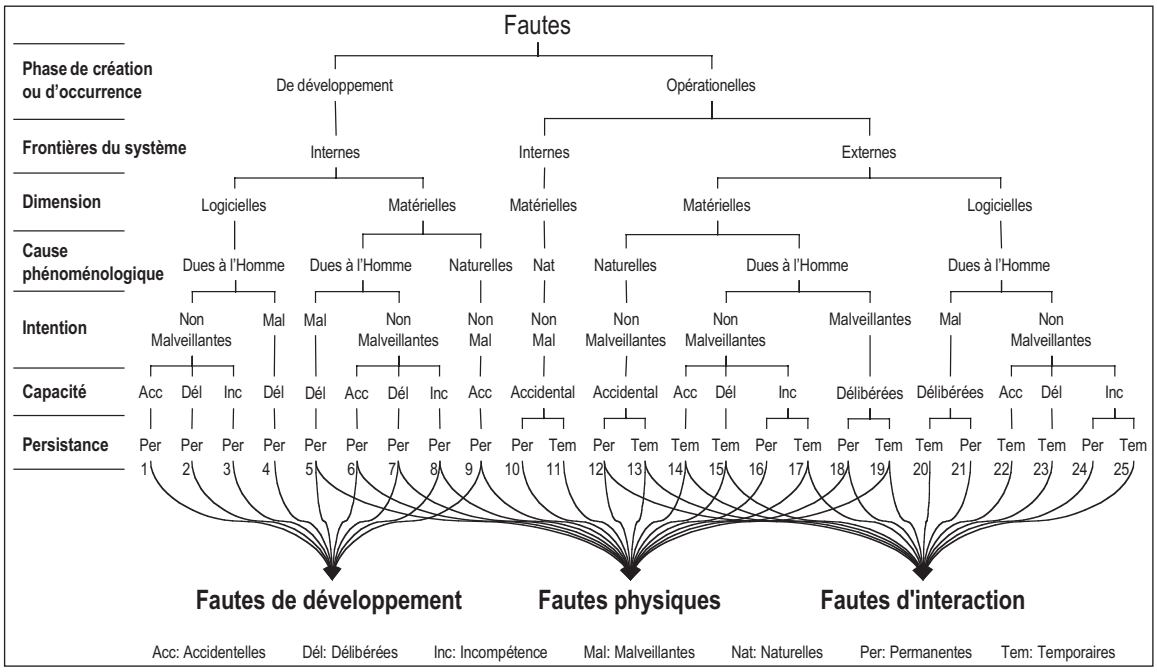


Figure 1.4 - Classes de fautes combinées

Fautes de développement											Fautes d'interaction														
				Fautes physiques																					
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25
Défauts logiciels			Logiques malignes		Errata matériels / Défauts de production				Dégradat. physiques		Agressions physiques								In-trus.	Vi-rus	Erreurs d'entrée			

Figure 1.5 - Exemples de classes de fautes

1.4.2. Défaillances du service

L'occurrence d'une défaillance a été définie par rapport à la fonction d'un système, et non par rapport à sa spécification ; en effet, si un comportement inacceptable est généralement identifié comme une défaillance en raison d'une déviation de la conformité à la spécification, il peut se faire qu'il satisfasse la spécification tout en étant inacceptable pour les utilisateurs du système, révélant ainsi une faute de spécification. Dans ce dernier cas, la reconnaissance que l'événement est indésiré (et constitue en fait une défaillance) peut ne s'effectuer qu'après son occurrence, par exemple via ses conséquences.

Un système ne défaille généralement pas toujours de la même façon, ce qui conduit à la notion de mode de défaillance, qui peut être caractérisée selon quatre points de vue, conduisant aux classes de défaillance du service, comme indiqué par la figure **1.6**.

Trois commentaires sur les modes de défaillance :

- une défaillance non signalée résulte d'une défaillance des mécanismes de détection d'une délivrance d'un service incorrect ; un autre mode de défaillance de ces mécanismes est de détecter et signaler une défaillance alors qu'elle n'a pas eu lieu, donc d'émettre une *fausse alarme* ;
- un système dont toutes les défaillances sont, dans une mesure acceptable, des défaillances par arrêt est un *système à arrêt sur défaillance* ; un système dont toutes les défaillances sont, dans une mesure acceptable, des défaillances bénignes est un *système sûr en présence de défaillance* ;
- les malveillances peuvent affecter la disponibilité, sous la forme de *déni de service.*

1.4.3. Pathologie des défaillances du service: relation entre fautes, erreurs et défaillances

Les mécanismes de création et de manifestation des fautes, erreurs, défaillances peuvent être résumés comme suit :

- une faute est *active* lorsqu'elle produit une erreur. Une faute active est soit une faute interne qui était préalablement *dormante* et qui a été activée par le processus de traitement, soit une faute externe qui a profité d'une vulnérabilité. Une faute interne peut évoluer cycliquement entre états dormant et actif ;

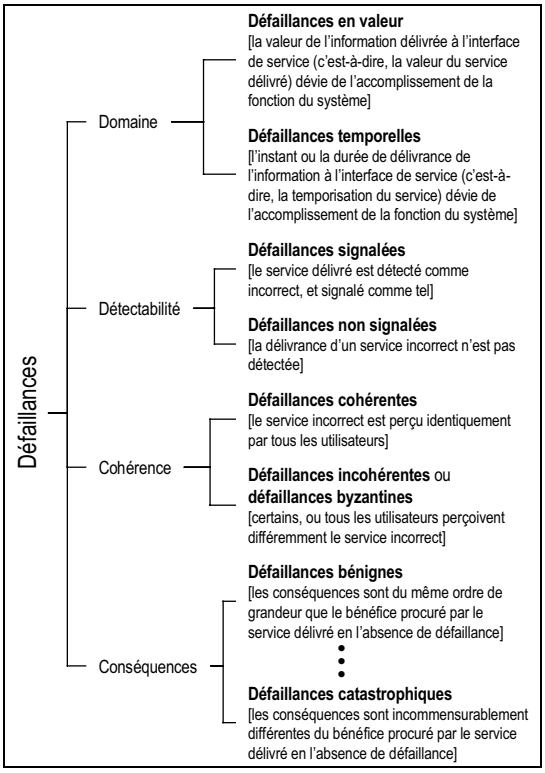


Figure 1.6 - Défaillances du service

- une erreur peut être latente ou détectée ; une erreur est *latente* tant qu'elle n'a pas été reconnue en tant que telle ; une erreur est *détectée* par un algorithme ou un mécanisme de détection qui permet de la reconnaître comme telle. Une erreur peut disparaître avant d'être détectée. Par propagation, une erreur crée de nouvelles erreurs ;
- une défaillance survient lorsque, par propagation, elle affecte le service délivré par le système, donc lorsqu'elle est perçue par le ou les utilisateurs (informellement, lorsqu'elle « passe à travers » l'interface système-utilisateur(s)). La conséquence de la défaillance d'un composant est une faute pour le système qui le contient ou pour le, ou les composants qui interagissent avec lui ; les modes de défaillance d'un composant sont donc des types de fautes pour le système ou pour les composants qui interagissent avec lui.

Ces mécanismes permettent de compléter la *chaîne fondamentale* donnée par la figure **1.7**. Les flèches de la chaîne expriment la relation causale entre fautes, erreurs et défaillances. Elles doivent être interprétées de façon générique : par propagation, plusieurs erreurs sont généralement générées avant qu'une défaillance ne

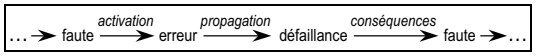


Figure 1.7 - La chaîne fondamentale des entraves à la sûreté de fonctionnement

survienne. Il est à noter que, de par les mécanismes exposés précédemment, la propagation, et donc l'instanciation de cette chaîne, peut survenir dans la création ou la modification d'un système, au-delà des interactions entre composants ou de la composition des composants d'un système.

La possibilité d'identifier les conditions d'activation d'une faute ayant produit une ou plusieurs erreurs est la *reproductibilité d'activation d'une faut*e. La figure **1.8** définit les fautes solides ou furtives selon que leurs conditions d'activation sont reproductibles ou non ; elle introduit également la notion de faute intermittente, en raison de la similitude de manifestation des fautes furtives et des fautes temporaires.

Il est à noter que la plupart des fautes de développement résiduelles dans les logiciels complexes sont des fautes furtives : elles sont suffisamment subtiles pour que leurs conditions d'activation dépendent des combinaisons également subtiles de l'état interne du logiciel et de requêtes externes, combinaisons qui surviennent rarement et sont très difficiles à reproduire.

1.4.4. Défaillances de la sûreté de fonctionnement

Des défaillances trop fréquentes ou trop sévères, qui caractérisent une défaillance de la sûreté de fonctionnement, ont immanquablement pour origine une ou plusieurs fautes de développement, par exemple des fautes logicielles trop nombreuses, ou une trop faible aptitude à tolérer les fautes.

Comme pour les défaillances du service, nous ne définissons pas les défaillances de la sûreté de fonctionnement par rapport à sa spécification, qui peut également être entachée de faute (par exemple : faute d'omission dans la description de l'environnement opérationnel, des classes de fautes à prévenir ou à tolérer).

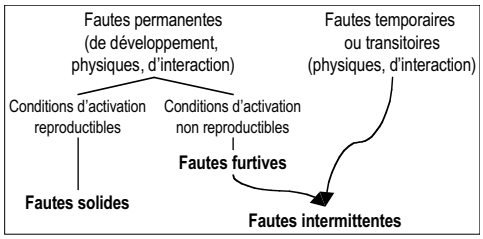


Figure 1.8 - Fautes solides, furtives, intermittentes

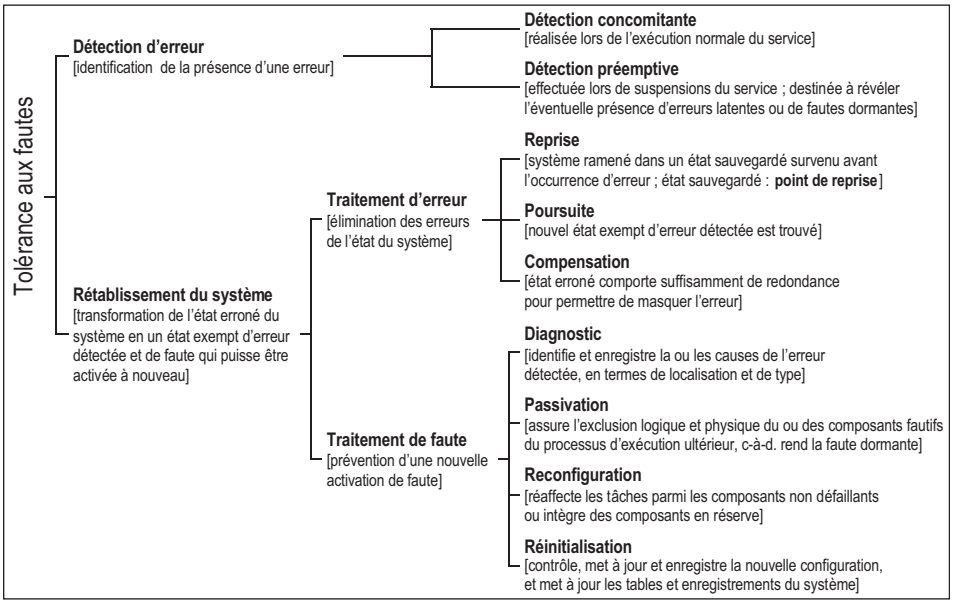


Figure 1.9 - Techniques de tolérance aux fautes

1.5. Techniques de la tolérance aux fautes

La tolérance aux fautes vise à éviter les défaillances et est mise en œuvre par la *détection des erreurs* et le *rétablissement du système*. La figure **1.9** liste les techniques de tolérance aux fautes. Habituellement le traitement de faute est complété par des opérations de maintenance corrective, afin d'éliminer les composants passivés.

Reprise et poursuite sont invoquées à la demande, après qu'une ou plusieurs erreurs aient été détectées, alors que la compensation peut être appliquée à la demande ou systématiquement, indépendamment de la présence ou de l'absence d'erreur. Le traitement d'erreur est, à la demande, suivi du traitement de faute constituent le rétablissement du système, d'où le qualificatif de la stratégie de tolérance aux fautes correspondante : *détection et rétablissement* .

Le *masquage de faute* résulte de l'application systématique de la compensation. Un tel masquage pouvant entraîner une diminution non perçue des redondances disponibles, sa mise en œuvre pratique comporte généralement une détection d'erreur, conduisant au masquage et détection.

Il est à noter que :

- reprise et poursuite ne sont pas exclusives : une reprise peut d'abord être tentée ; si l'erreur persiste, une poursuite peut alors être entreprise ;
- les fautes intermittentes ne nécessitent ni passivation, ni reconfiguration ; identifier si une faute est intermittente ou non peut être effectué par le traitement d'erreur (la récurrence d'une erreur indique que la faute n'est pas intermittente) ou par diagnostic de faute dans le cas de la poursuite.

Détection et traitement d'erreur préemptifs sont des pratiques courantes lors de l'initialisation, d'un système informatique. Elles interviennent aussi en opération sous différentes formes : test d'unités en réserve, « nettoyage » de la mémoire ('memory scrubbing'), programmes d'audit ou « rajeunissement » du logiciel ('software rejuvenation') (Huang *et al.* 1995).

1.6. Mise en œuvre de la tolérance aux fautes

Ce qui vient d'être exposé s'applique tant aux fautes physiques qu'aux fautes de conception ou d'interaction (accidentelles ou malveillantes) : les classes de fautes qui peuvent être réellement tolérées par un système donné dépendent des hypothèses de fautes considérées dans le processus de conception, ce qui est conditionné par l'*indépendance* des redondances par rapport aux processus de création et d'activation des fautes.

Un exemple de ce qui précède est donné lorsque l'on considère la tolérance aux fautes physiques et la tolérance aux fautes de conception. Une méthode (largement utilisée) pour la tolérance aux fautes est d'effectuer des traitements multiples par des voies multiples, séquentiellement ou en parallèle. Lorsque la tolérance aux fautes physiques seules est recherchée, les voies multiples peuvent être identiques, en vertu de l'hypothèse selon laquelle des composants matériels défaillent *indépendamment*. Une telle approche s'est avérée adéquate pour les fautes de développement furtives, par reprise (Gray 1986) ; elle n'est cependant

pas adéquate pour la tolérance aux fautes de développement solides, où les multiples voies mettent en œuvre la même fonction via des conceptions et des réalisations séparées, c'est-à-dire en recourant à la *diversification fonctionnelle* (Avižienis & Kelly 1984, Laprie *et al.* 1990).

L'adjonction dans un composant de mécanismes de détection d'erreur à ses capacité de traitement fonctionnel conduit à la notion de *composant autotestable* ; un des principaux intérêts est la possibilité de définir clairement des zones de confinement d'erreur (Siewiorek & Swarz 1992).

Un aspect important dans la coordination des activités de composants multiples est d'éviter que la propagation d'erreurs n'affecte l'activité de composants non défaillants. Cet aspect devient particulièrement important lorsqu'un composant donné doit communiquer à d'autres composants une information qui lui est propre. Des exemples typiques de telles informations issues de sources uniques sont des données locales de capteurs, la valeur d'une horloge locale, la perception locale de l'état d'autres composants, … La conséquence de la nécessité de communiquer d'un composant à d'autres composants des informations issues d'une source unique est que les composants non défaillants doivent s'accorder sur la manière d'utiliser de manière cohérente les informations obtenues, et donc de se prémunir contre des défaillances éventuellement incohérentes. Un attention particulière a été portée à ce problème dans le cas des systèmes répartis (cf. § **3**).

La tolérance aux fautes est (également) un concept récursif : les mécanismes destinés à mettre en oeuvre la tolérance aux fautes doivent être protégés contre les fautes susceptibles de les affecter eux-mêmes. Des exemples sont fournis par la réplication de voteurs, par les contrôleurs autotestables, par la notion de mémoire « stable » pour les programmes et données de reprise (Muller *et al.* 1996).

1.7. Analyse de la tolérance aux fautes

Comme mentionné au paragraphe **1.2** (figure **1.2**), l'analyse de la sûreté de fonctionnement inclut l'élimination de fautes et la prévision des fautes. L'analyse des systèmes informatiques tolérants aux fautes passe par la mise œuvre des mêmes types d'activités que pour les systèmes non tolérants aux fautes. La vérification et l'évaluation sont au cœur de ces activités. Toutefois une différence importante est qu'en plus des entrées fonctionnelles, l'analyse doit aussi être menée en considérant les « entrées » spécifiques — les fautes — que ces systèmes sont destinés à traiter.

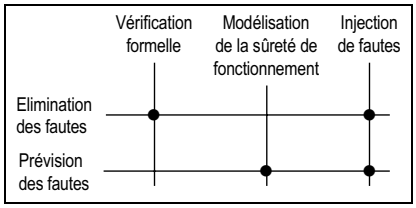


Figure 1.10 - Analyse de la tolérance aux fautes

La figure **1.10** résume les techniques d'analyse de la tolérance aux fautes, et leurs contributions à l'élimination et à la prévision des fautes.

1.7.1. Vérification formelle

Les techniques de vérification formelle, qu'il s'agisse de la vérification de modèle ou de la preuve de théorème, permettent de déterminer si des algorithmes de tolérance aux fautes satisfont les propriétés exprimées dans leur spécification, ainsi que les hypothèses sur lesquelles sont basées leur conception, en particulier vis-à-vis de l'occurrence de fautes multiples et de leur gravité (Rushby 1992).

1.7.2. Modélisation de la sûreté de fonctionnement

La modélisation de la sûreté de fonctionnement a pour objectif d'évaluer le comportement du système par rapport à l'occurrence ou à l'activation de fautes. L'évaluation a deux facettes :

- évaluation qualitative ou ordinale, destinée à identifier, classer, ordonner les modes de défaillances, ou la combinaison d'événements (défaillances de composants ou conditions environnementales) susceptibles d'entraîner la défaillance du système ;
- évaluation quantitative, ou probabiliste, destinée à évaluer en termes de probabilités les conséquences de l'occurrence ou de l'activation de fautes sur les attributs de la sûreté de fonctionnement, vus alors comme des *mesures* de cette dernière.

Les méthodes d'évaluation peuvent être spécifiques (par exemple, analyse des modes de défaillance et leurs effets pour l'évaluation qualitative, ou chaînes de Markov et réseaux de Petri stochastiques pour l'évaluation quantitative), ou peuvent s'appliquer aux deux formes de l'évaluation (par exemple, diagrammes de fiabilité, arbres de fautes).

La modélisation de systèmes tolérant les fautes peut être conduite vis-à-vis des diverses classes de fautes, de développement, physiques, d'interaction.

Dans l'évaluation de systèmes tolérant les fautes, la *couverture* de la tolérance aux fautes, c'est-à-dire l'efficience des algorithmes et mécanismes de tolérance aux fautes, a une très forte influence sur les mesures de

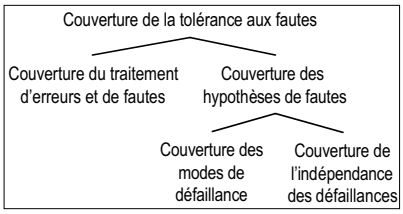


Figure 1.11 - Couverture de la tolérance aux fautes

la sûreté de fonctionnement. Les imperfections de la tolérance aux fautes, donc les défauts de couverture de la tolérance aux fautes, entraînent une limitation à l'accroissement de sûreté de fonctionnement. De telles imperfections (figure **1.11**) sont dues :

- aux fautes de développement affectant les mécanismes de tolérance aux fautes par rapport aux hypothèses de fautes prises en compte dans le développement, avec pour conséquence un défaut de couverture du traitement d'erreurs et de fautes ;
- à des hypothèses de fautes qui diffèrent des fautes réellement rencontrées durant la vie opérationnelle, avec pour conséquence un défaut de couverture d'hypothèse, qui peut à son tour être dû soit à des composants dont le comportement à défaillance est différent des hypothèses, c'est-à-dire un défaut de couverture des modes de défaillance, soit à l'occurrence de défaillances de mode commun alors que des défaillances indépendantes ont été supposées, c'est-à-dire un défaut de couverture de l'indépendance des défaillances.

Il a été montré que les défauts de couverture de traitement d'erreurs et de fautes limitent très sévèrement l'amélioration de sûreté de fonctionnement. Des effets similaires résultent d'un défaut de couverture des modes de défaillances : des hypothèses faibles (par exemple, fautes byzantines) ont comme conséquence une couverture élevée des modes de défaillance, mais nécessitent davantage de redondances et des mécanismes de tolérance aux fautes plus complexes, d'où une éventuelle décroissance globale de la sûreté de fonctionnement (Powell 1992).

1.7.3. Injection de fautes

La complexité des phénomènes mis en jeu par la pathologie des fautes et des techniques de tolérance aux fautes confère une place centrale à l'injection de fautes (Arlat *et al.* 1990). De nombreuses techniques et outils les supportant ont été développés. Ces travaux peuvent être classés en fonction des moyens d'injection de fautes : l'injection de fautes au niveau du matériel et l'injection de fautes par logiciel.

Dans le premier cas, les fautes sont injectées directement au niveau des composants matériels par le biais d'altérations électriques ou radiatives. Les travaux se sont tout d'abord focalisés sur les broches des circuits intégrés. De fait, par l'application de niveaux de tensions spécifiques, cette technique vise à provoquer des erreurs simulant les conséquences des fautes physiques (internes ou externes) survenant en opération. L'application de techniques plus sophistiquées (par exemple : bombardement par ions lourds, rayonnement laser) permet de s'approcher un peu plus de la réalité (problème de représentativité des tests) pour une classe spécifique de fautes causées par les perturbations cosmiques ou SEU ('Single Event Upsets') qui sont d'un intérêt majeur pour les domaines spatial et avionique. En plus du problème de la représentativité des fautes injectées au niveau des broches des circuits, deux autres problèmes importants concernent l'accessibilité et le parasitage. Déjà fortement pénalisants avec les technologies modernes, ces problèmes ne vont pas aller en s'améliorant au cours des années à venir (circuits multibroches à montage de surface, fréquences d'horloge accrues).

Ils peuvent toutefois être contournés par l'emploi de la stratégie alternative agissant explicitement au niveau de l'information manipulée ou mémorisée ; il s'agit de la techniques d'injection de fautes par logiciel (SWIFI : 'Software Implemented Fault Injection'). Dans ce cas, l'injection correspond à la corruption de variables logiques, de cellules de registres ou de mémoires (Arlat *et al.* 2002). Elle agit donc dans le domaine des erreurs, qui est tout à fait adapté pour analyser la représentativité des tests effectués. En pratique, cette approche permet de simuler les conséquences d'un large spectre de fautes (fautes physiques et aussi dans une certaine mesure, les fautes logicielles).

Les techniques qui viennent d'être mentionnées s'appliquent à des systèmes déjà réalisés, éventuellement sous forme de prototypes. Une autre tendance naturelle est de conduire des tests par injection de fautes dès les premières étapes de la conception, en s'appuyant sur un modèle de simulation du système cible.

La plupart des travaux et outils visent soit l'évaluation de la couverture de la tolérance aux fautes, soit la caractérisation des modes de défaillance de logiciels COTS (Arlat *et al.* 2002). Le plus souvent, l'élimination des fautes (de conception ou de mise en œuvre) des mécanismes de tolérance aux fautes n'est qu'un « produit secondaire » des expériences d'évaluation ayant révélé un comportement anormal. Aussi, le problème de la définition d'expériences d'injection de fautes à fin explicite d'élimination de défauts au sein des mécanismes de tolérance aux fautes reste encore un domaine de recherche ouvert.

2. Techniques

Cette section décrit successivement les principales techniques pour la *détection d'erreur* et le *rétablissement du système* (Siewiorek & Swarz 1992, Laprie *et al.* 1996).

2.1. Détection d'erreur

La détection d'erreur est basée sur une redondance qui peut prendre plusieurs formes : redondance au niveau information ou composant, redondance temporelle ou algorithmique. Nous considérons uniquement les techniques de détection concomitante. La forme la plus sophistiquée de détection d'erreur consiste à construire des composants autotestables en adjoignant au composant purement fonctionnel des éléments de contrôle permettant de vérifier que certaines propriétés entre les entrées et les sorties du composant sont satisfaites (Wakerly 1978). Les formes de détection d'erreur les plus couramment utilisées sont les suivantes :

- codes détecteurs d'erreur ;
- doublement et comparaison ;
- contrôles temporels et d'exécution ;
- contrôles de vraisemblance ;
- contrôles de données structurées.

Les trois derniers types de contrôle peuvent être mis en œuvre par des assertions exécutables au niveau logiciel. Une assertion est une expression logique permettant d'effectuer en ligne un contrôle de vraisemblance sur les objets d'un programme. La valeur de l'expression logique est à VRAI si l'état est correct et à FAUX dans le cas contraire ; dans ce dernier cas, une exception est levée.

En pratique, cette approche se concrétise souvent par la technique d'« empaquetage » qui fait l'objet d'une description rapide à la fin de ce paragraphe.

2.1.1. Codes détecteurs d'erreur

Les codes détecteurs d'erreur ciblent plus particulièrement les erreurs induites par les fautes physiques. La détection est basée sur une redondance dans la représentation de l'information, soit en ajoutant des bits de contrôle aux données, soit en utilisant une nouvelle forme de représentation incluant la redondance (Wakerly 1978). La première forme de redondance constitue la classe de *codes séparables* tandis que la seconde correspond à la classe des *codes non séparables*.

Un concept fondamental caractérisant les codes détecteurs d'erreur est la *distance de Hamming* qui, pour deux mots binaires, correspond au nombre de bits par lesquels les deux mots diffèrent. La *distance d'un code* est définie comme la distance de Hamming minimal entre deux mots quelconques valides du code. Pour qu'un code soit capable de détecter une erreur sur e bits, il faut que la distance du code soit supérieure ou égale à $e + 1$.

Le niveau de redondance utilisé dépend des hypothèses d'erreurs qu'il est possible de répartir en trois classes : erreurs simples, erreurs unidirectionnelles, erreurs multiples.

Concernant les erreurs simples, le contrôle de parité est la forme la plus connue du codage de l'information permettant de détecter de telles erreurs. La détection d'erreurs affectant une tranche de b bits d'information peut être assurée grâce à des codes *b-adjacents*. Lorsque les données codées subissent des traitements arithmétiques (addition, multiplication), il peut être intéressant d'utiliser des *codes arithmétiques*. Ces codes présentent l'avantage d'être préservés lors des opérations arithmétiques (un code C est préservé par une opération o si $A,B \in C$ implique que A o $B \in C$). Les codes arithmétiques se répartissent en codes séparables et codes non séparables.

Concernant les erreurs unidirectionnelles, les codes peuvent aussi être répartis en codes séparables (par ex., code de Berger) et codes non séparables (par ex., codes *K-parmi-N*). La détection des erreurs multiples nécessite un code 1-parmi-2 (code à complémentation ou double-rail) assimilable à une forme de duplication.

2.1.2. Doublement et comparaison

Le doublement et comparaison, malgré un coût matériel important, est un moyen de détection très utilisé en raison de sa simplicité de mise en œuvre. La mise en œuvre repose sur l'hypothèse que les deux unités redondantes sont indépendantes vis-à-vis du processus de création et d'activation de fautes ; il faut notamment s'assurer que, soit les fautes sont créées ou activées indépendamment dans les deux exemplaires, soit que, si une même faute provoque des erreurs dans les deux exemplaires, ces erreurs sont différentes.

Ainsi, si seules les fautes physiques internes sont considérées, les deux canaux peuvent être identiques dans la mesure où il est possible de faire l'hypothèse que les composants matériels défailleront de manière différente dans chaque canal. Dans le cas de fautes physiques externes, il faut éviter des fautes de mode commun en isolant physiquement les deux unités ou décalant dans le temps le traitement sur les deux unités. Lorsqu'une telle hypothèse d'indépendance ne peut être faite (cas où les fautes de conception du matériel ou logiciel sont prises en compte), il est nécessaire que les deux unités fournissent des services identiques mais au travers de conceptions ou mises en œuvre différentes (diversification).

2.1.3. Contrôles temporels et d'exécution

Le contrôle par « chien de garde » est sans doute, compte tenu de son faible coût, le moyen de détection le plus couramment utilisé pour la détection d'erreur en ligne. Il peut être utilisé dans différentes situations telles que la détection de la défaillance d'un périphérique en contrôlant son temps de réponse qui ne doit pas dépasser une valeur maximale ('time-out') ou la surveillance de l'activité des processeurs. Dans ce dernier cas, le chien de garde doit être périodiquement rafraîchi par le processeur. Ainsi, si le comportement du processeur est altéré de sorte que le chien de garde n'est plus rafraîchi avant qu'il n'expire, une exception est levée. Une telle approche peut être utilisée, pour sortir d'une situation de blocage ou d'une boucle infinie, ou pour détecter l'arrêt d'un processeur distant.

Une amélioration possible de l'efficacité de détection d'un tel mécanisme consiste à vérifier en plus le flot de contrôle du programme exécuté par une unité centrale. Cette méthode, connue aussi sous le nom d'analyse de signature, se base sur un schéma de compression qui produit une signature, qui de manière usuelle correspond à la somme de contrôle de la série d'instructions exécutées. Deux types d'approches peuvent être distinguées suivant qu'elles nécessitent une modification du programme initial (contrôle de signature imbriqué) ou que la surveillance est effectuée de manière totalement externe sans modification du programme (contrôle de signature disjoint). L'analyse de signature peut être mise en œuvre via un moniteur matériel ('watchdog processor').

La surveillance du flot d'exécution peut être appliquée à des niveaux d'abstraction élevés tels que, par exemple, un modèle à base de réseau de Petri pour un protocole de communication. Dans ce cas, l'exécution en parallèle du programme normal et de son modèle abstrait permet de vérifier que les états successifs du protocole sont cohérents.

Une réalisation industrielle qui mérite d'être mentionnée est le processeur codé qui est utilisé dans plusieurs métros automatisés. Elle repose sur l'utilisation conjointe de deux techniques :

- code arithmétique pour détecter les erreurs liées au stockage ou transfert des données ainsi qu'à leur traitement ;
- signature pour détecter les erreurs dans le séquencement du programme et ses itérations ou dans la sélection des données traitées.

2.1.4. Contrôles de vraisemblance

Les contrôles de vraisemblance présentent l'avantage de ne nécessiter généralement qu'un surcoût faible par rapport aux éléments fonctionnels du système. Ils permettent de faire face à des erreurs induites par un large spectre de fautes, mais la couverture associée peut parfois être limitée. Les contrôles de vraisemblance peuvent être mis en œuvre par :

- du matériel spécifique pour détecter des valeurs erronées (instruction illégale, adresse mémoire inexistante) ou des violations de protections de segments mémoire ;
- du logiciel spécifique pour vérifier la conformité des entrées ou des sorties du système à des invariants.

Les contrôles de vraisemblance par logiciel peuvent être intégrés au niveau système, et donc mis en œuvre pour tout programme d'application, (par ex., contrôle dynamique de type, vérification des indices de tableaux, etc.) ou être spécifiques du programme d'application (par ex., intervalle de valeurs possibles, écart maximal par rapport au résultat précédent dans un contrôle de processus continu, etc.).

2.1.5. Contrôles de données structurées

Deux types de contrôle peuvent être appliqués aux structures complexes de données d'un système informatique. Les contrôles peuvent porter sur l'*intégrité sémantique* des données ou sur l'*intégrité structurelle* de la structure de données.

Les contrôles de l'intégrité sémantique consistent à vérifier la cohérence des informations contenues dans la structure de données en utilisant des tests de vraisemblance décrits dans le paragraphe précédent.

Les contrôles structurels sont particulièrement applicables à des données structurées dont les différents éléments sont liés par des pointeurs. La redondance présente dans ces structures peut prendre les trois principales formes suivantes :

- comptabilisation du nombre d'éléments contenus dans la structure ;
- utilisation de pointeurs redondants incorporés dans la structure (double chaînage) ;
- ajout d'indicateurs sur le type des éléments de la structure.

Compte tenu de ces redondances, une modification valide de la structure nécessite la modification de plusieurs éléments de la structure qui doit être réalisée de manière atomique. La détection des erreurs repose sur le fait, qu'en présence d'un comportement erroné, la modification ne sera pas atomique. Il faut souligner que plus le nombre de changements nécessaires sera important, plus le pouvoir de détection sera élevé.

2.1.6. Empaquetage

La technique d'empaquetage ('wrapping') permet de mettre en œuvre des mécanismes de détection d'erreur (notamment les contrôles de vraisemblance, la

vérification de propriétés, etc.). Cette notion est historiquement issue du monde de la sécurité-immunité, sa vocation première étant d'équiper des composants logiciels de mécanismes de protection. L'idée de base est d'associer à un composant des mécanismes additionnels, externes au composant lui-même, permettant d'améliorer le confinement des erreurs. La protection vise soit des fautes pouvant survenir dans l'environnement opérationnel (fautes physiques ou d'interaction) soit au sein du composant (par exemple, fautes logicielles résiduelles).

En pratique, la couche d'empaquetage d'un composant correspond à un ensemble d'assertions exécutables (les 'wrappers'). Typiquement, un 'wrapper' se présente comme une commande gardée, de la façon suivante :

<condition> : <assertion> : <action>

La *condition* exprime l'état et les événements qui déclenchent l'évaluation de l'assertion. L'*assertion* est un prédicat qui correspond à une propriété. L'*action* correspond généralement au signalement de l'erreur qui a été détectée par l'assertion, si tel est le cas. La mise en œuvre de ces différents éléments d'un 'wrapper' nécessite deux types de mécanismes fondamentaux :

- la capture d'information d'état ou d'événements (appel de fonctions à l'interface du composant, valeur de paramètre ou de variables internes ou externes) ;
- la capacité d'agir sur le composant empaqueté pour réaliser le traitement d'erreur (par exemple, pour placer le composant dans un état sûr).

Les capacités d'observabilité et de commandabilité évoquées ici dépendent fortement de la complexité des propriétés que l'on souhaite vérifier à l'exécution. Leur mise en œuvre dépend, quant à elle, du degré de visibilité associé au composant : « boîte blanche » ou « boîte noire ». Dans le premier cas, les mécanismes nécessaires peuvent être facilement adjoints au composant. Dans le second, seules des propriétés faisant intervenir des événements ou informations externes (appel de fonction à l'interface, données d'entrée-sortie) peuvent être vérifiées simplement. Les techniques réflexives (Maes 1987) permettent de contourner cette difficulté, en associant des mécanismes d'observation et de commande au composant cible, et ce de façon peu intrusive. Ces techniques peuvent notamment s'appliquer à des « composants sur étagère », soit en source libre, soit commerciaux (COTS), et dans ce cas par le biais d'une interaction avec le fournisseur. Des travaux récents ont montré l'intérêt des 'wrappers', au-delà de la détection d'erreur, vis-à-vis de la tolérance aux fautes (Rodríguez *et al.* 2002). En effet, dans certains cas, l'action de traitement d'erreur peut permettre de corriger l'état erroné, tout en restant transparent pour l'utilisateur.

2.2. Rétablissement du système

Suivant le moyen utilisé pour reconstruire un état correct, trois formes de rétablissement du système ont été identifiées : la reprise, la poursuite et la compensation d'erreur.

2.2.1. Reprise

La reprise représente de loin la technique la plus fréquemment utilisée pour assurer le rétablissement du système. Elle consiste en la sauvegarde périodique de l'état du système de façon à pouvoir, après voir détecté une erreur, ramener le système dans un état antérieur, appelé « point de reprise » .

La sauvegarde périodique de l'état du système doit s'effectuer au moyen d'un mécanisme de mémorisation, ou « support stable » qui protège les données contre les effets des fautes (Muller *et al.* 1996). La sauvegarde peut être facilitée par des mécanismes matériels ou logiciels permettant de sauvegarder automatiquement les données modifiées entre deux points de reprise. Si la couverture de détection n'est pas totale (cas général), les points de reprise peuvent être contaminés par une erreur avant qu'elle ne soit détectée. Dans ce cas, la reprise ne pourra être efficace que si est possible de restituer un état exempt d'erreur. Cela implique qu'il existe plusieurs points de reprise successifs ou que la structure de l'application permette de conserver des points de reprise *emboîtés*, comme dans le cas des blocs de recouvrement.

Les techniques de reprise ont quelques inconvénients. Tout d'abord, elles sont généralement incompatibles avec des applications ayant des contraintes temps réel strictes. De plus, la taille des points de reprise et le surcoût temporel nécessaire à leur établissement imposent souvent des contraintes structurelles qui doivent être prises en compte dans le développement de l'application, avec un support spécifique du système d'exploitation. Cela interdit généralement l'usage de systèmes d'exploitation généraux tels qu'Unix et de progiciels qui n'auraient pas été développés spécialement pour l'architecture considérée. Notons, toutefois, que cette structuration permet d'améliorer la qualité (et donc la fiabilité des logiciels correspondants.

2.2.2. Poursuite

Le rétablissement par poursuite constitue une approche alternative ou complémentaire à la reprise — après avoir détecté une erreur, et après avoir éventuellement tenté une reprise, la poursuite consiste en la recherche

d'un nouvel état acceptable pour le système à partir duquel celui-ci pourra fonctionner (éventuellement en mode dégradé).

Une approche simple de poursuite consiste à réinitialiser le système et à acquérir un nouveau contexte à partir de l'environnement (par ex., relecture des capteurs dans un système de contrôle-commande).

Une autre approche est celle des *traitements d'exceptions*, en se basant sur des primitives offertes par certains langages de programmation. Dans ce cas, les programmes d'application sont conçus pour prendre en compte des *signaux d'erreur* (issus des mécanismes de détection d'erreur) et passer d'un traitement normal à un traitement d'exception (généralement dans un mode dégradé). En ce qui concerne les systèmes sûrs en présence de défaillance, les traitements d'exception peuvent être limités aux seules tâches critiques. Dans les cas extrêmes, les tâches critiques amènent le système dans un état stable sûr et ensuite arrêtent le processeur. Comme exemple, citons l'arrêt d'un train : un train immobile est dans un état sûr si les passagers peuvent quitter le train (par ex., en cas d'incendie), et si l'arrêt du train est signalé suffisamment tôt aux autres trains circulant sur la même voie.

Il faut souligner que la mise en œuvre du rétablissement par poursuite est toujours spécifique d'une application donnée. De par son principe, et contrairement aux techniques de reprise ou de compensation, la poursuite ne pas servir comme mécanisme de base d'une architecture tolérante aux fautes à usage général.

2.2.3. Compensation

La compensation d'erreur nécessite que l'état du système comporte suffisamment de redondance pour permettre, en dépit des erreurs qui pourraient l'affecter, sa transformation en un état exempt d'erreur.

Avec la compensation, il n'est pas nécessaire de ré-exécuter une partie de l'application (cas de la reprise) ou d'exécuter une procédure dédiée (cas de la poursuite) pour permettre de continuer le traitement fonctionnel. Ce type de recouvrement est donc relativement transparent vis-à-vis de l'application : il n'est pas nécessaire de structurer l'application en vue d'un éventuel traitement d'erreur. Ceci peut permettre l'utilisation de systèmes d'exploitation et de progiciels standard.

Comme nous allons le voir dans ce paragraphe, la compensation d'erreur peut être initialisée par une détection d'erreur (détection et compensation), ou être systématique (masquage). Même dans ce cas, il est important de signaler les erreurs pour initialiser un traitement de faute. En effet, si un traitement de faute n'est pas effectué, la redondance peut se dégrader sans que cela soit perçu conduisant ainsi par la suite à une défaillance quand une autre faute surviendra. Enfin, une dernière possibilité de compensation est fournie par les codes correcteurs d'erreur, notamment pour la transmission et ou le stockage de l'information.

Détection d'erreur et compensation

Un exemple typique de la *détection et compensation d'erreur* est l'utilisation des composants autotestables exécutant en redondance active le même traitement ; en cas de défaillance de l'un d'entre eux, il est déconnecté et le traitement se poursuit sans interruption sur les autres. La compensation, dans ce cas, se limite à une commutation éventuelle de composants.

C'est sur ce principe que se base, par exemple, les architectures des systèmes Stratus S/32 et IBM System/88. C'est aussi le cas du système de gestion de commandes de vol des Airbus 320/330/340 (cf. § **7.2**).

Masquage de faute

Contrairement à la technique précédente, le *masquage de faute* est une méthode de compensation d'erreur où la compensation est effectuée de manière systématique, sans détection préalable d'erreur. Un exemple typique est celui du *vote majoritaire* : les traitements sont exécutés par au moins trois composants identiques dont les sorties sont votées ; les résultats majoritaires sont transmis, les résultats minoritaires (supposés erronés) sont éliminés.

Comme le vote est appliqué systématiquement, le traitement, et par conséquent le temps d'exécution, sont identiques qu'il y ait ou non erreur. C'est ce qui différencie le masquage de la technique de détection et compensation.

L'algorithme de vote peut être simple si les exemplaires sont identiques et synchronisés et si le traitement est déterministe. Si ces hypothèses ne peuvent être garanties, il faut considérer que les exemplaires sont diversifiés et appliquer un algorithme de décision plus ou moins complexe, dépendant généralement du type des informations sur lesquelles il faut voter.

Codes correcteurs d'erreurs

Le principe de codage de l'information, introduite à la section sur les codes détecteurs d'erreur à fin de détection d'erreur, peut également être utilisé pour construire des codes correcteurs d'erreur (Wakerly 1978). La correction d'erreur nécessite cependant une distance de Hamming plus grande entre les symboles (mots) du code. Pour corriger une erreur simple, il faut que la distance de Hamming soit supérieure ou égale à 3 (au lieu de 2 pour la détection). Plus généralement, pour corriger e erreurs, un code

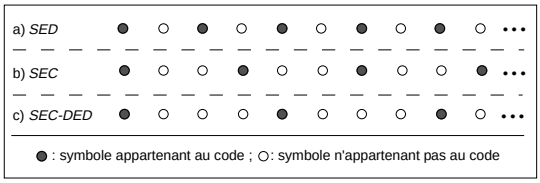


Figure 2.1 - Distance de Hamming et détection - correction d'erreur

avec une distance de $2 \times e + 1$ est nécessaire. Il est important de souligner qu'un code de distance d $(d \geq 3)$ peut être utilisé soit pour détecter $d-1$ erreurs, ou corriger $\lfloor (d-1)/2 \rfloor$ erreurs. Ainsi, un code de distance *d* peut corriger *ec* erreurs et détecter *ed* erreurs supplémentaires, si et seulement si : $d \geq 2 \times ec + ed + 1$.

Le principe de l'utilisation de la redondance pour assurer la détection ou la correction d'erreur est illustré par le schéma de la figure **2.1**.

Le code de Hamming est le code correcteur d'erreur simple le plus répandu. Un tel code est obtenu par adjonction de plusieurs bits contrôlant sélectivement la parité de certains bits d'information. Ces bits de contrôle sont utilisés pour élaborer un *syndrome* permettant d'effectuer sans ambiguïté le diagnostic d'erreur. Pour que la correction soit possible, il faut donc être capable d'identifier, à l'aide des combinaisons binaires du syndrome, les cas d'absence d'erreur et d'occurrence d'une erreur sur un quelconque des bits du mot (y compris les bits de contrôle). Si *k, c* et *n* désignent respectivement le nombre de bits d'information, le nombre de bits de contrôle (et également du syndrome) et le nombre total de bits d'un mot du code ($n = k + c$), cela conduit à la condition suivante : $2^c \geq n + 1$. Par exemple, pour un mot de données codé sur 16 bits, 5 bits de contrôle sont nécessaires : le surcoût en nombre de bits est ainsi de l'ordre de 30 % ; ce coût devient inférieur à 15 % pour des données codées sur 64 bits.

L'extension du code de Hamming à la détection systématique et simultanée des erreurs doubles est obtenue par l'ajout d'un bit de contrôle de parité supplémentaire portant sur l'ensemble des *n* bits.

D'autres codes correcteurs plus puissants ont été développés. Il s'agit en tout premier lieu des codes cycliques tout particulièrement adaptés à la transmission de l'information en série. Ils sont intéressants car les opérations de codage et de décodage peuvent être facilement et économiquement réalisées en utilisant des registres à décalage avec des rebouclages. De plus, ces codes se prêtent bien à la correction d'erreurs « en paquets » (erreurs affectant plusieurs bits adjacents). La classe de codes cycliques binaires la plus connue correspond aux codes BCH (Bose, Chauduri et Hocquenghem). Elle constitue une généralisation du code de Hamming à la correction d'erreurs multiples. Parmi les codes d'ordre supérieur (c'est-à-dire portant sur des symboles non binaires), la classe la plus importante correspond aux codes Reed-Solomon. Ils représentent une extension directe des codes binaires et permettent notamment de développer simplement des codes correcteurs d'erreurs en paquets.

Une manière efficace de définir un code puissant est de combiner deux codes, voire plus. De tels codes, dits codes *produits*, permettent d'obtenir d'intéressantes propriétés à faible coût. Par exemple, un code correcteur d'une erreur simple peut être obtenu en utilisant une forme bidimensionnelle de parité. En plus du bit conventionnel de parité associé à chaque mot (ligne) de la matrice représentant l'espace mémoire, un bit de parité est associé à chaque colonne (incluant la colonne des bits de parité sur les lignes). La matrice représentant la mémoire est par conséquent étendue par un mot de parité horizontale et un mot de parité verticale. Une erreur affectant un seul bit peut être facilement détectée et localisée (et par conséquent corrigée) dans la mesure où elle affecte la parité de la ligne et de la colonne correspondantes. Cette technique est à la fois efficace et peu coûteuse, mais elle ne permet pas de corriger des erreurs multiples.

3. Systèmes répartis

Nous développons dans ce paragraphe les techniques propres aux systèmes répartis. La notion de système réparti considérée est tout système composé de plusieurs *processus* communicant par *messages* au moyen de *canaux*. Nous supposons que les défaillances des processus sont indépendantes (par exemple, parce qu'ils s'exécutent sur des processeurs distincts).

L'abstraction de processus communicants offre un cadre intéressant pour aborder la tolérance aux fautes — celui de l'algorithmique répartie — et de la concrétiser par une mise en œuvre par logiciel. Cependant, il faut nuancer ce point du vue par l'observation que la distribution de l'état du système et les incertitudes induites par une communication interprocessus non fiable et non instantanée peuvent nuire à la sûreté de fonctionnement. En pratique, pour être utilisable, un système réparti doit quasi obligatoirement incorporer des mécanismes de tolérance aux fautes.

3.1. Modèles et hypothèses

Il existe une vaste littérature sur la théorie de l'algorithmique réparti (voir le chapitre ££de M. Raynal££). La correction des algorithmes est

argumentée dans le cadre d'un modèle logique qui représente les hypothèses prises en compte. Un tel modèle logique est en fait constitué d'au moins deux sous-modèles : un *modèle temporel* décrivant une connaissance a priori sur les aspects temporels et un *modèle de faute* définissant les types de fautes pris en compte au niveau de processus et des canaux.

3.1.1. Modèles temporels

Le modèle temporel le plus simple pour définir et raisonner sur la correction d'algorithmes répartis est le modèle synchrone ou à temps borné : le temps maximum nécessaire pour interroger et recevoir la réponse d'un processus non défaillant est strictement borné. Il s'agit d'un modèle très puissant car il permet d'utiliser des temporisations pour détecter de façon non ambiguë l'arrêt ou le retard de réaction d'un processus distant. Ce modèle est approprié pour des applications critiques qui doivent garantir des propriétés temps réel de terminaison, même en présence de fautes. Par contre, il impose un ordonnancement temps réel strict de la communication et du traitement, et repose sur l'hypothèse que la borne sur le temps de communication est *toujours* respectée.

À l'extrême opposé, le modèle asynchrone ou « sans temps » fait l'abstraction totale de la notion de temps et donc ne suppose aucune borne temporelle. Dans ce modèle, on garantit qu'un message transmis sur un canal sera inéluctablement délivré au processus destinataire (la littérature parle de canal « fiable » ou « équitable »), mais à un instant non défini (puisque le modèle ne comporte aucune notion de temps). Les algorithmes conçus selon ce modèle sont attractifs car leur comportement logique est totalement indépendant des performances des réseaux et des systèmes mis en jeu. Malheureusement, il a été démontré que certains problèmes fondamentaux de tolérance aux fautes (par exemple, le consensus) n'admettent pas de solution déterministe avec ce modèle.

On doit donc faire face au dilemme que le modèle asynchrone ne permet pas de définir des algorithmes tolérants aux fautes et que le modèle synchrone repose sur des hypothèses fortes qui ne sont pas faciles à garantir. Des recherches récentes se sont donc focalisées sur la définition de modèles temporels intermédiaires. Les plus pertinents sont le modèle asynchrone temporisé (Cristian & Fetzer 1998) et le modèle asynchrone augmenté de « détecteurs de défaillances » (Chandra & Toueg 1996). Dans les deux cas, il est possible de garantir le respect d'invariants de sûreté, mais toute garantie de progrès (par exemple, pour assurer un consensus) est conditionnée par le fait qu'au moins une majorité de processus se comporte de façon synchrone pendant « suffisamment » de temps.

En pratique, beaucoup d'applications réparties reposent implicitement sur le modèle synchrone, en faisant appel à des temporisations pour détecter l'arrêt de processus distants, malgré le fait que les hypothèses du modèle synchrone ne puissent pas être garanties (même en surdimensionnant très largement les temporisations). Une telle approche ne peut se justifier que dans le cas d'applications peu critiques, pour lesquelles une incohérence logique occasionnelle n'a pas de conséquences graves.

3.1.2. Modèles de faute

Dans les systèmes répartis, un modèle de faute se définit en termes de défaillances de processus et de canaux de communication entre processus. Au niveau des processus, les défaillances les plus communément admises sont, par ordre croissant de généralité : les arrêts, les omissions, les défaillances temporelles et les défaillances arbitraires, incluant les défaillances en valeur et les défaillances incohérentes ou byzantines. Dans ce dernier cas, aucune hypothèse restrictive n'est faite — un processus défaillant peut même envoyer des messages contradictoires à des destinataires différents. Enfin, notons que : (a) la notion de défaillances temporelles n'a pas de sens dans le cas d'un modèle temporel totalement asynchrone ; et (b) l'hypothèse de défaillances par arrêt est parfois augmentée par une hypothèse de rétablissement, par exemple, que l'état antérieur d'un processus survit à l'arrêt car stocké sur mémoire stable.

La même classification de défaillances peut s'appliquer aux canaux de communication, bien que l'utilisation de codes détecteurs d'erreur autorise le plus souvent à négliger les défaillances en valeur. Notons, cependant, que les modèles temporels de base (synchrone et asynchrone) n'admettent pas de défaillances de la communication.

3.1.3. Partitionnement

Un ensemble de processus est partitionné s'il se divise en sous-ensembles qui ne peuvent pas communiquer. Le partitionnement peut avoir lieu en fonctionnement normal, comme dans les systèmes mobiles, ou en raison de défaillances des processus ou des canaux, par exemple, dues à des situations de surcharge.

Certaines techniques de tolérance aux fautes visent à permettre aux composants d'une partition d'assurer un fonctionnement « déconnecté », éventuellement dégradé, jusqu'à ce que les composants puissent fusionner de nouveau, mettant ainsi fin au partitionnement.

Il est à noter que le partitionnement est exclu par principe du modèle synchrone, tandis que le modèle asynchrone suppose que tout partitionnement prendra inéluctablement fin.

3.2. Cohérence

La programmation des systèmes répartis est notoirement difficile, même en l'absence de fautes. Cela est dû au fait que l'état global du système est réparti parmi les processus et les canaux de communication. Puisque la communication ne peut pas être instantanée, aucun processus ne peut avoir une vue globale de cet état. Nous présentons ici des techniques de cohérence qui peuvent grandement faciliter la tâche du programmeur, malgré cette absence de vue globale.

3.2.1. Temps global

Le fait que les processus d'un système réparti n'aient pas accès à une horloge commune complexifie la coordination entre processus et la détermination d'un ordre entre événements. Par conséquent, une notion de temps global constitue une abstraction simplificatrice importante. Deux types de temps globaux peuvent être définis : le temps physique et le temps logique.

Un temps physique global peut être approché en synchronisant des horloges physiques réparties. La synchronisation peut être effectuée de façon pair-à-pair (synchronisation interne) ou par rapport à une référence de temps tierce (synchronisation externe). Pour effectuer une synchronisation interne, chaque horloge lit périodiquement les valeurs des horloges distantes, génère une fonction de correction (par exemple, une moyenne tolérante aux fautes) et l'applique localement. La synchronisation externe peut s'effectuer en interrogeant périodiquement un serveur de temps, mis en œuvre, par exemple, à l'aide d'un récepteur GPS ('Global Positioning System'), ou au moyen d'un ensemble tolérant aux fautes d'horloges synchronisées de façon interne.

La précision de la synchronisation des horloges dépend essentiellement de l'incertitude sur le temps nécessaire pour lire une horloge distante. Il faut adopter le modèle de temporisation synchrone si l'on veut définir une borne déterministe sur les décalages entre des horloges correctes. Le modèle asynchrone temporisé autorise une synchronisation fine, mais probabiliste. Par exemple, le protocole NTP ('Network Time Protocol') d'Internet assure une synchronisation avec une précision (non déterministe) de quelques dizaines de millisecondes.

Un temps physique global ne permet pas d'ordonner des événements qui sont séparés par un intervalle de durée inférieure à la précision (qui est forcément non nulle). Par contre, un temps logique global permet d'ordonner des événements selon une relation de causalité potentielle. Un système d'horloges logiques peut être mis en œuvre au moyen de compteurs associés à chaque processus. Ceux-ci sont mis à jour lors de l'occurrence d'événements locaux, et en particulier lors de la réception de messages de processus distants, qui portent les valeurs des horloges logiques correspondantes.

3.2.2. Consensus

Le consensus constitue un problème fondamental de la tolérance aux fautes répartie. Dans sa forme de base, tous les processus d'un ensemble doivent prendre une décision binaire. Chaque processus possède initialement sa propre « valeur » ou avis sur la décision à prendre. La définition du problème requiert que tous les processus corrects prennent la même décision finale. Si tous les processus ont la même valeur initiale (et les deux valeurs sont possibles), c'est cette valeur qui doit être choisie comme décision finale. Un problème équivalent d'*accord* peut être défini pour décider d'une valeur parmi un ensemble de cardinalité supérieure à deux. Le problème d'accord en présence de fautes arbitraires des processus est appelé l'*accord byzantin*. L'accord sur un vecteur de valeurs initiales est appelé la *cohérence interactive*.

Une solution au consensus est nécessaire si les processus exempts de fautes doivent prendre des décisions cohérentes. Malheureusement, il a été démontré qu'il n'est pas possible de résoudre le problème de consensus de façon déterministe si des messages peuvent être perdus (canaux non fiables) ou retardés indéfiniment (modèle asynchrone) et si l'un des processus peut s'arrêter. Il est résoluble cependant avec le modèle synchrone, ou avec le modèle asynchrone temporisé ou augmenté de détecteurs de défaillance. Il existe aussi des solutions non déterministes avec le modèle totalement asynchrone.

Deux autres résultats théoriques importants précisent des conditions nécessaires pour assurer le consensus malgré f processus défaillants : l'obligation d'au moins $f+1$ tours d'échange d'information et la participation d'au moins $3f+1$ processus si ceux-ci peuvent défaillir de façon arbitraire.

3.2.3. Communication de groupe

Les services de communication de groupe visent à faciliter la communication avec ou parmi des ensembles de processus. Ils sont donc utiles pour la mise en place de services tolérants aux fautes basés sur la réplication. Trois aspects doivent être pris en compte : comment définir les destinataires des messages ; comment acheminer les messages vers ces

destinataires ; et, comment fournir des garanties sur l'acceptation et l'ordre d'acceptation des messages.

Un protocole autorisant l'envoi de messages vers tous les processus présents est appelé un protocole de *diffusion générale* ('broadcast'), tandis qu'un protocole qui désigne un sous-ensemble des processus présents est appelé un protocole de *diffusion sélective* ('multicast'). La connaissance mutuelle des destinataires des messages requiert en plus la mise en place d'un service d'*appartenance* ('membership'). Typiquement, un service d'appartenance permet aux processus de joindre ou quitter dynamiquement un groupe, soit volontairement, ou en raison de partitionnements ou de défaillances.

L'acheminement de messages vers des destinations multiples est trivial lorsque les processus peuvent communiquer par un réseau à diffusion (de type Ethernet, par exemple). Lorsque la communication doit s'effectuer au travers d'un réseau maillé (Internet, par exemple), les protocoles doivent faire appel à des listes de diffusion, des arbres d'acheminement, voire à l'inondation.

Un protocole qui garantit que toutes les destinations correctes délivrent les mêmes messages à la couche supérieure est un protocole de *diffusion fiable*. Un protocole de diffusion fiable qui garantit en plus que toutes les destinations acceptent les messages dans le même ordre est un protocole de *diffusion atomique*. Certains protocoles assurent d'autres propriétés d'ordre, tels que l'ordre « premier entré-premier sorti » ou l'ordre causal. On démontre que, comme pour consensus, la diffusion atomique n'a pas de solution déterministe avec le modèle totalement asynchrone, alors que la diffusion fiable admet des solutions à la seule condition que les canaux de communication soient équitables.

3.3. Techniques de tolérance

Nous abordons maintenant les techniques de tolérance visant à garantir la disponibilité de services de traitement dans un contexte réparti. Sauf indication contraire, nous supposerons que les processus défaillent seulement en s'arrêtant.

Dans sa forme la plus simple, la tolérance peut se limiter au rétablissement d'un processeur ou nœud de traitement après une défaillance temporaire. Cependant, la continuité de service malgré la défaillance ou l'inaccessibilité de nœuds de traitement nécessite la réplication de processus ou de données sur des nœuds multiples. Nous abordons successivement la reprise locale (d'un seul processus), la reprise répartie (impliquant plusieurs processus), la réplication de processus et la réplication de données.

3.3.1. Reprise locale

La forme minimale de tolérance aux fautes dans un système réparti vise à minimiser l'impact négatif de la défaillance temporaire d'un processus serveur en facilitant un redémarrage rapide. Cela peut être facilité si le serveur est conçu de façon à ne conserver aucun état entre deux requêtes et que toutes ses opérations soient idempotentes. L'absence d'état interne élimine la nécessité même de le restaurer lors d'une reprise. Cela implique aussi qu'aucune information n'est conservée en ce qui concerne les clients — ni client ni serveur ne sont contraints par le fait que l'autre effectue une reprise. Par ailleurs, l'idempotence des opérations du serveur permet à chaque client de pouvoir répéter une requête tant qu'elle n'est pas satisfaite. C'est cette stratégie qui a guidé, par exemple, la conception du système de fichier réseau de Sun (NFS).

Si au contraire un processus serveur comporte un état interne, il faut que celui-ci soit sauvegardé sur une mémoire stable en tant que point de reprise (cf. § **2.2.1**). Un processus peut effectuer une reprise purement locale : (a) soit s'il n'a pas interagi avec d'autres processus depuis l'établissement du point de reprise, soit si les interactions éventuelles peuvent être rejouées (par exemple, à partir d'un journal en mémoire stable). Si cela n'est pas le cas, une reprise répartie de plusieurs processus est nécessaire.

3.3.2. Reprise répartie

Une reprise répartie correspond au cas où les dépendances induites par la communication interprocessus ont pour effet que la reprise d'un processus conduit à reprendre aussi d'autres processus ayant interagi avec le premier. Les processus doivent reprendre leur exécution depuis un ensemble de points de reprise qui constituent un état global cohérent. Un effet domino peut survenir lorsque le seul état global cohérent antérieur est l'état initial du système.

La définition d'un état global cohérent dépend du positionnement du protocole de reprise par rapport au protocole assurant la fiabilité des communications (figure **3.1**) (Elnozahy *et al.* 2002). Lorsque le protocole de reprise est situé au-dessus d'un protocole de communication fiable (figure **3.1-a**), un ensemble de points de reprise ne constituent un état global cohérent que si la « ligne de reprise » qui représente cet ensemble n'est traversée pas aucun message (A et B sur la figure **3.1-b**). Dans le cas contraire (figure **3.1-c**), une ligne de reprise traversée éventuellement par des messages de gauche à droite (C sur la figure **3.1-b**) représente aussi un état global cohérent, car ces messages « émis » mais pas encore reçus, seront ré-émis plus tard par le protocole de communication fiable

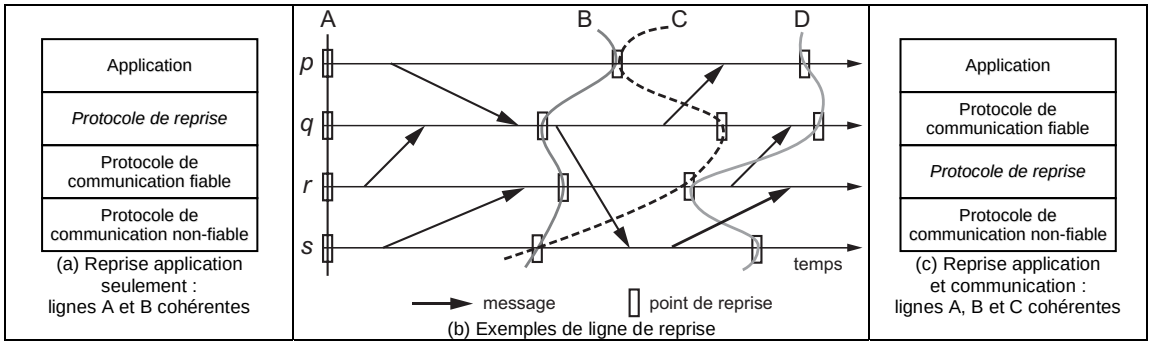


Figure 3.1 - Processus communicants et lignes de reprise

(dont l'état fait parti de l'état sauvegardé par les points de reprise). La ligne D de la figure **3.1-b** ne correspond à aucun état global cohérent, car elle est traversée par un message de droite à gauche, ce qui correspondrait à un message reçu mais pas encore émis.

On distingue généralement trois approches pour la reprise répartie :

- la *création indépendante des points de reprise* autorise chaque processus à créer des points de reprise au moment opportun pour lui (par exemple, lorsque l'état à sauvegarder contient peu d'informations). Cette approche induit un surcoût potentiellement important lors d'une reprise éventuelle, car il faut que les processus recherchent dynamiquement une ligne de reprise cohérente (s'il en existe une, car l'absence de coordination peut donner lieu à l'effet domino) ;
- la *création coordonnée des points de reprise* permet d'assurer que seules des lignes de reprise cohérentes sont créées. On améliore ainsi les performances lors d'une reprise au prix d'un surcoût en fonctionnement normal (c'est-à-dire, en l'absence de fautes). Chaque processus maintient un ou deux points de reprise : un permanent, et un autre temporaire. La transformation de points de reprise temporaires en points permanents s'effectue au moyen d'un protocole à deux phases gérées par le processus qui a déclenché la création coordonnée d'une ligne de reprise ;
- la *création de points de reprise induite par la communication* est un compromis entre les deux approches précédentes. Chaque processus peut créer indépendamment des points de reprise de base qui sont complétés par des points de reprise forcés lors de la réception de certains messages, identifiés comme induisant des dépendances susceptible d'entraîner un début d'« effet domino ».

Les techniques de reprise considérées dans cette section et dans la précédente peuvent être à même de tolérer des fautes permanentes si les points de reprise sont enregistrés sur une mémoire stable accessible depuis le réseau de communication. En effet, cela autorise la création de nouveaux processus sur des processeurs non défaillants qui peuvent être initialisés à partir de ces points de reprise. On peut même créer, en avance, des processus de secours prêts à se substituer à tout moment aux processus « primaires », conduisant ainsi à la notion de processus dupliqués.

3.3.3. Processus répliqués

Un service tolérant aux fautes peut être mis en œuvre en coordonnant un groupe de processus répliqués sur deux ou plusieurs processeurs différents. Le groupe de processus doit être géré de façon à donner l'illusion d'un seul processus logique qui continue à fournir un service correct, malgré la défaillance d'un sous-ensemble des membres du groupe. On distingue généralement trois stratégies : la réplication passive, active et semi-active.

Avec la *réplication passive*, une seule copie (la copie primaire) traite tous les messages reçus, met à jour son état interne et effectue l'envoi de messages de sortie. La copie primaire met à jour régulièrement une copie de son état interne, qui constitue ainsi un point de reprise . Celui-ci est stocké soit, sur une mémoire stable accessible par les copies de secours, qui ne font rien tant que la copie primaire fonctionne (technique appelée « réplication passive froide » dans FT-CORBA (OMG 2004)), soit, par les copies de secours elles-mêmes, qui maintiennent ainsi à jour leur état interne (« réplication passive tiède »). Lorsque la copie primaire défaille, une des copies de secours est élue pour prendre sa place.

La *réplication active* traite toutes les copies sur un pied d'égalité : chacune traite tous les messages reçus, met à jour son état interne de façon autonome, et génère les messages de sortie. Les messages de sortie effectifs sont choisis au moyen d'une fonction de décision qui dépend de l'hypothèse sur les défaillances des processus. Pour des arrêts simples, la fonction de

décision peut être de choisir le premier message disponible. Mais la réplication active permet aussi de s'affranchir des défaillances arbitraires, au moyen d'une fonction de décision à vote majoritaire .

La *réplication semi-active* est similaire à la réplication active dans le sens que toutes les copies reçoivent les messages d'entrée et peuvent ainsi les traiter. Cependant, comme pour la réplication passive, le traitement est asymétrique car une copie privilégiée (la meneuse) assume la responsabilité de certaines décisions (par exemple, sur l'acceptation des messages, ou sur la préemption du traitement en cours). La meneuse peut imposer ses décisions aux autres copies (les suiveuses) sans recourir à un vote. De façon optionnelle, la meneuse peut assumer seule la responsabilité d'envoyer les messages de sortie. Bien que visant principalement à tolérer les défaillances par arrêt, cette technique peut être étendue au cas des défaillances arbitraires.

Le rétablissement ou la création d'une copie de processus, et son insertion dans un groupe à réplication active ou semi-active, nécessitent l'initialisation de son état interne à partir de celui des copies existantes. Cette opération est équivalente à la création et le transfert d'un point de reprise dans un groupe à réplication passive tiède.

3.3.4. Données répliquées

Du point de vue des données, la réplication sert à améliorer à la fois la disponibilité de données et la performance des opérations de lecture. D'une part, une donnée dupliquée peut être accédée même si certaines copies résident sur des nœuds défaillants ou inaccessibles. D'autre part, il est habituellement plus rapide de lire une copie proche qu'une copie distante. Cependant, les opérations d'écriture sur des données dupliquées peuvent être ralenties, car elles impliquent potentiellement toutes les copies.

Les protocoles de gestion de données dupliquées sont qualifiés d'*optimistes* ou de *pessimistes* selon s'ils négligent ou non la probabilité d'accès conflictuels en cas de partitionnement des copies.

Les protocoles pessimistes assurent l'équivalence à une copie (tout utilisateur perçoit la donnée comme si elle n'existait qu'en un seul exemplaire) en forçant l'exclusion mutuelle entre les opérations d'écriture et de lecture. Le protocole le plus simple s'appelle « lire une, écrire toutes » : un processus utilisateur peut lire n'importe quelle copie, mais doit effectuer toute écriture sur l'ensemble des copies. Cette technique fournit une performance excellente en lecture, au prix d'une piètre performance en écriture — les écritures sont même bloquées si une seule copie de la donnée n'est pas accessible. Les protocoles à quorum généralisent cette approche et permettent d'améliorer la performance en écriture au prix de l'implication de plusieurs copies dans toute opération de lecture. D'autres protocoles pessimistes existent, par exemple, les protocoles à copie principale ou à partition virtuelle.

Les protocoles optimistes sacrifient la cohérence afin d'améliorer la disponibilité. Ces protocoles autorisent des opérations d'écriture sur les copies présentes dans des composants distincts d'une partition du système. Par exemple, le protocole à copies disponibles est une variante optimiste du protocole « lire une, écrire toutes » par laquelle seules les copies accessibles sont modifiées lors d'une opération d'écriture. Dès que le partitionnement cesse, tout conflit résultant d'écritures effectuées dans des composants différents doit être détecté et résolu. La résolution de conflit dépend de la sémantique des données et ne peut donc être automatique que dans certains cas spécifiques.

4. Fautes de développement

Bien que la problématique ait été identifiée de longue date, les fautes de conception du logiciel posent toujours un défi pour l'informatique tolérante aux fautes. Les problèmes concernent les logiciels applicatifs, les logiciels exécutifs et aussi les logiciels destinés à mettre en œuvre la tolérance aux fautes.

Dans le premier cas, la difficulté a trait à l'accroissement de la complexité des fonctionnalités et à l'inflation du logiciel qui en résulte, y compris dans les cas des systèmes embarqués. Par exemple, pour l'avionique, le nombre de lignes de code a subi une croissance fulgurante (douze millions de lignes pour l'A320, plus de vingt pour l'A340 et, soixante-cinq pour l'A380 !).

Dans le cas des logiciels exécutifs, les principales difficultés proviennent des pilotes de périphériques qui sont souvent développés par des tiers n'ayant qu'une maîtrise partielle des spécificités du système d'exploitation auxquels ils sont associés. Les pilotes constituent une proportion très large et croissante du code des systèmes d'exploitation (cette part est de 70% dans le cas de Linux).

Par ailleurs, les logiciels de tolérance aux fautes — même restreinte au cas des fautes physiques — peut constituer jusqu'à 50% ou plus du code relatif à la mise en œuvre d'un système tolérant aux fautes. Les sérieux problèmes rencontrés en 2004 par les opérateurs téléphoniques français attestent bien de la difficulté associée à la maîtrise de ces types de logiciels.

Toutefois, le problème des fautes de conception n'est pas l'exclusivité du logiciel ; il affecte aussi les

développements matériels. Le problème rencontré il y a une dizaine d'années par la première version du microprocesseur Pentium (faute de conception affectant l'opération de division) est un exemple bien connu. Il est clair que le développement des processeurs modernes est une tâche tout aussi délicate que celui des logiciels complexes : le chiffre de 170 millions de transistors est affiché pour la dernière version (le Pentium 4), alors que la version de 1993 n'en comportait que 3,3 millions ! Le recensement et l'analyse des statistiques publiées par Intel sur ses familles de processeurs révèle l'acuité du problème lié aux fautes de conception (Avižienis & He 1999).

Bien que la tolérance aux fautes de conception (du matériel ou du logiciel) ait reçu une attention moindre que les fautes physiques, il faut toutefois noter des résultats significatifs. En plus du cas des fautes furtives pour lesquelles s'applique la technique de reprise (cf. § **1.6**), pour le cas des fautes solides, deux grands types de techniques peuvent être distingués en fonction des objectifs visés afin de maîtriser les fautes de conception en opération : soit, éviter que la défaillance d'une tâche ne provoque la défaillance du système global, soit, assurer la continuité du service.

Dans le premier cas, le but est de détecter rapidement une tâche erronée et de l'éliminer afin d'éviter que l'erreur ne propage ; aussi, cette approche est souvent qualifiée de « défaillance au plus tôt » ('fail-fast'). La détection d'erreur est obtenue au moyen d'une « programmation défensive » (utilisation d'assertions exécutables) et le rétablissement s'appuie généralement sur le traitement d'exception. Comme les erreurs dues au logiciel résultent le plus souvent de conditions de fautes subtiles, difficiles à reproduire, l'expérience montre que l'association d'une simple approche de ce type avec des techniques de recouvrement destinées aux fautes matérielles pouvait constituer un moyen très efficace pour tolérer les fautes du logiciel.

Le second cas suppose que l'on dispose d'au moins un autre composant capable de réaliser la même tâche, et que ce composant a été obtenu en appliquant le principe de *diversification fonctionnelle* (conception et réalisation indépendantes à partir de la même spécification). Trois approches de base peuvent être identifiées (Laprie *et al.* 1990) : les blocs de recouvrement, la programmation N-versions et la programmation N-autotestable. Ces approches correspondent à la transposition au cas du logiciel de techniques classiques de réplication du matériel.

Ces résultats sont largement exploités dans les systèmes réels, qu'il s'agisse des serveurs commerciaux, ou bien des systèmes embarqués de l'avionique ou du ferroviaire (voir par exemple les architectures décrites à la section **7**). De la même façon, la diversification est aussi explicitement utilisée pour tolérer des fautes de conception du matériel : par exemple, les trois voies redondantes du calculateur de commande de vol du Boeing 777 mettent chacune en œuvre un microprocesseur d'origine distincte (Intel, Motorola et AMD) (Yeh 1998).

5. Fautes d'interaction homme-machine

L'utilisation des principes et techniques de tolérance aux fautes vis-à-vis de fautes du matériel et du logiciel est maintenant habituelle au sein des systèmes critiques. Cette évolution fait que les fautes d'interaction accidentelles entre homme et machine — opérateur(s) et système(s) technique(s) informatisé(s) — revêtent une importance croissante sur la sûreté de fonctionnement des systèmes critiques mettant en jeu des opérateurs.

De plus, les progrès techniques ont induit des modifications importantes dans les formes d'interaction : l'opérateur est de moins en moins impliqué dans des tâches manuelles au profit d'activités mentales plus délicates. En conséquence, il en résulte qu'un grand nombre d'accidents est souvent considéré comme causé par une erreur humaine.

Les statistiques relatives aux causes d'accidents affectant les vols commerciaux illustrent clairement l'impact des fautes humaines : bien que le nombre d'accidents soit en régression, les fautes humaines sont de loin la première cause des accidents. En particulier les statistiques portant sur la période 1994-2003[2], indiquent que les fautes humaines contribuent en tant que facteur principal à 70% des accidents dont les causes sont connues : pilotes 62%, maintenance 4%, contrôle aérien 4%.

Même si une proportion importante des fautes d'interaction peut être associée à des fautes de conception (déficiences au niveau de la conception de l'interface homme-machine, insuffisance de l'aide procurée par le système aux opérateurs, etc.), les actions des opérateurs humains constituent une menace potentielle indéniable. Il est donc primordial de prendre en compte le rôle et les caractéristiques de l'opérateur lors du développement d'un système sociotechnique.

Cette observation a motivé la conduite d'un certain nombre d'études mettant l'accent sur les problèmes liés à la fiabilité humaine au sein d'un système sociotechnique complexe. La plupart des travaux

[2] *Statistical Summary of Commercial Jet Aircraft Accidents – Worldwide Operation 1994-2003*, Boeing Commercial Aircraft Group, Seattle, USA, 2004 (http://www.boeing.com/news/techissues).

cherchent à éliminer les conditions susceptibles de favoriser la commission des fautes humaines. L'harmonisation de l'allocation des fonctions entre l'homme et la machine et la définition d'interfaces mettant en avant des critères liés à « l'utilisabilité » sont quelques exemples de méthodes susceptibles de réduire les risques de fautes humaines. Ces méthodes sont importantes en vue d'améliorer la sûreté de fonctionnement d'un système sociotechnique, mais l'éradication des fautes des opérateurs humains n'est pas un objectif réaliste. En effet, les opérateurs sont fréquemment confrontés à des situations délicates et urgentes nécessitant un savoir-faire et/ou un niveau de connaissance complexes. En cas de stress, il n'est pas raisonnable d'attendre (Amalberti & Malaterre 2001) que l'opérateur agisse sans commettre d'erreur. Il est donc essentiel de prévoir des moyens pour tolérer les fautes d'interaction homme-machine, au même titre que les autres classes de fautes.

Les méthodes actuelles de tolérance aux fautes des opérateurs sont essentiellement fondées sur la contribution de l'homme en tant que support pour la tolérance, soit par l'opérateur lui-même ou bien par le biais d'une équipe d'opérateurs (tant pour le masquage de commandes erronées que pour l'analyse de situations délicates). Cependant, certains travaux récents explorent les possibilités offertes par l'utilisation de la machine pour contribuer à la tolérance aux fautes de l'opérateur. Pour les systèmes disposant de redondances pour tolérer les fautes physiques ou de conception, il peut être intéressant d'essayer d'analyser comment cette redondance pourrait être mise à profit pour assurer la tolérance aux fautes d'interaction homme-machine.

6. Malveillances

Ces dernières années ont marqué un tournant dans le développement de systèmes critiques : les menaces terroristes ont conduit à prendre conscience que les malveillances représentent un risque prépondérant pour la sécurité-innocuité de ces systèmes, qu'il s'agisse de transport de passagers, d'installations potentiellement dangereuses comme les centrales nucléaires ou les usines chimiques, ou des infrastructures critiques de transport d'énergie, de télécommunications, etc. L'imbrication de l'informatique dans ces systèmes critiques fait qu'une défaillance du système informatique peut avoir des conséquences catastrophiques, alors que pour quelqu'un de malveillant, il est souvent plus facile et moins dangereux d'attaquer le système informatique que le processus ou le système physique contrôlé par l'informatique.

Mais les systèmes critiques ne sont pas les seuls visés par les malveillances. Avec le développement d'Internet, celles-ci sont même devenues le principal souci des administrateurs de systèmes informatiques. En effet, le nombre d'attaquants potentiels sur Internet est très important, même si globalement ils ne forment qu'une très petite proportion de la population totale des utilisateurs. Les motivations des attaquants sont variées. Ils peuvent agir par jeu ou curiosité, par vanité, par plaisir de détruire ou de provoquer des dégâts, par esprit de vengeance, par appât du gain, voire pour des motifs politiques, stratégiques ou terroristes. Les attaquants sont dès lors très divers, tant en ténacité et en compétence qu'en moyens financiers ou en puissance de calcul qu'ils peuvent mettre en œuvre.

Les techniques utilisées pour les attaques sont, elles aussi, nombreuses et variées. Elles peuvent exploiter les vulnérabilités des réseaux et de leurs protocoles : écoute passive, destruction, insertion, modification ou rejeu de messages, falsification d'adresse, injection de faux messages de contrôle du réseau (de routage, par exemple), déni de service par « éblouissement » de réseau, etc. Les attaques peuvent aussi exploiter les multiples failles des systèmes d'exploitation et des logiciels d'application, en particulier les possibilités de débordement de tampon ou de pile.

Nous examinons successivement les principes des parades appropriées et décrivons quelques exemples de leur mise en œuvre.

6.1. Principes

Face à ces attaques, de plus en plus nombreuses, les méthodes traditionnelles de la sécurité informatique sont peu efficaces sur Internet : il n'est pas possible d'utiliser des moyens forts de contrôle d'accès (c'est-à-dire d'authentification et d'autorisation), lorsque le nombre d'utilisateurs potentiels est trop grand ou que ces utilisateurs ne sont pas connus personnellement. D'ailleurs, l'arsenal classique de ces méthodes est également peu efficace pour une autre raison : la plupart des attaques sur Internet exploitent les vulnérabilités des réseaux ou des logiciels, et contournent ainsi les mécanismes d'authentification et d'autorisation. Les fournisseurs de logiciels ont beau développer des « rustines » ('patches') le plus tôt possible après la découverte des vulnérabilités, ces rustines ne sont pas toujours appliquées, soit par manque de temps ou de compétence des responsables de l'informatique, soit parce que leur application invaliderait des fonctionnalités nécessaires pour le bon fonctionnement des applications existantes.

Les méthodes préventives, qui permettraient d'éviter d'introduire des failles ou des vulnérabilités dans la conception, la réalisation ou l'exploitation des systèmes, sont malheureusement trop souvent réservées aux systèmes les plus critiques. Les systèmes informatiques commerciaux sont, eux, généralement trop complexes pour être totalement maîtrisables, et leur sécurité n'est pas toujours un objectif primordial pour les développeurs. Ainsi un constructeur pourra considérer plus rentable de mettre rapidement sur le marché un système imparfait plutôt que de lui faire subir les méthodes de développement et de validation nécessaires pour obtenir un bon niveau de sécurité-immunité.

Il est donc justifié de tenter d'appliquer aux malveillances les méthodes de la tolérance aux fautes (Deswarte & Powell 2004). Deux principales classes de fautes sont dès lors à considérer : les logiques malignes, et les intrusions. Les logiques malignes sont des parties du système conçues pour provoquer des dégâts (bombes logiques) ou pour faciliter des intrusions futures (vulnérabilités créées volontairement, par exemple les portes dérobées). Les logiques malignes peuvent être introduites dès la création du système (par un concepteur malveillant), ou en phase opérationnelle, par l'installation d'un logiciel contenant un cheval de Troie ou par une intrusion. Les intrusions sont le résultat d'attaques (fautes externes créées avec l'intention de nuire, y compris les attaques lancées par des outils automatiques : vers, virus, zombies) ayant réussi à exploiter des vulnérabilités (fautes de conception ou d'exploitation, créées accidentellement ou volontairement, avec ou sans l'intention de nuire). Une intrusion est donc une faute interne, résultant de la combinaison d'une faute externe (l'attaque) et d'une autre faute interne (la vulnérabilité). Tolérer les malveillances revient donc à détecter les erreurs créées par les logiques malignes ou par les intrusions, et à les éliminer avant qu'elles ne produisent une défaillance du système.

Pour détecter les erreurs dues aux malveillances, on peut utiliser les mêmes techniques que pour les autres erreurs (cf. § **2.1**). En effet, une même erreur peut être créée par différents types de fautes, et la détection ne repose pas sur l'origine de l'erreur (c'est-à-dire, la faute), mais sur l'analyse de l'état du système. Il existe aussi des mécanismes de détection dédiés aux intrusions, couramment appelés *systèmes de détection d'intrusions*[3]. Ces systèmes de détection d'intrusions utilisent deux types de contrôles de vraisemblance :

- la *détection d'anomalies* ('anomaly detection') consiste à comparer le comportement observé du système avec une référence de comportement normal, basée sur des profils statistiques d'utilisation, ou créée par apprentissage, ou encore issue de spécifications fonctionnelles ou de la politique de sécurité du système ;
- la *détection d'attaques* ('misuse detection') analyse le comportement observé du système pour essayer d'y retrouver les symptômes d'attaques connues, généralement stockés dans une base de signatures caractéristiques de ces attaques. Les signatures doivent être suffisamment générales pour reconnaître les attaques connues, mais aussi de légères variantes de ces attaques, sans pour autant déclencher trop de fausses alarmes.

Pour rétablir le système dans un état exempt d'erreurs dues à des malveillances, on peut appliquer les trois types de techniques citées au paragraphe **1.5** :

- *reprise* : cela peut aller du simple redémarrage de l'ordinateur (interrompant ainsi les connexions en cours, et invalidant les codes malveillants actifs en mémoire), jusqu'à la réinstallation du système d'exploitation et des logiciels et la restauration des données depuis un point de reprise ;
- *poursuite* : cela consiste en général à commuter vers un mode de fonctionnement restreint, où seules les tâches les plus critiques sont maintenues, réduisant ainsi les possibilités d'intrusion ;
- *compensation* : dans ce cas, il faut que les traitements soient exécutés en redondance sur des ordinateurs indépendants (du point de vue des attaques), de sorte que les traitements continuent de produire des résultats corrects malgré la défaillance ou l'intrusion de certains de ces ordinateurs.

Mais l'application de ces techniques au cas des malveillances pose de sérieux problèmes :

- pour que la reprise ou la poursuite soient efficaces, il faut avoir une grande probabilité de détecter les erreurs rapidement, alors que les attaquants cherchent, eux, à ne pas être détectés. En particulier, les techniques actuelles de détection d'intrusion sont très imparfaites, avec un taux important de fausses alarmes (qui risquent de déclencher de nombreuses reprises, ou l'abandon de tâches non critiques, contribuant ainsi à des dénis de service), et un taux important de non détection (en particulier, la détection d'attaques est inefficace sur les attaques encore inconnues) ;
- pour la compensation, il faut que si une attaque a réussi (au moins partiellement) sur une partie du système, il ne soit pas trop facile de réussir la même attaque sur une autre partie. Cela signifie que chaque

[3] Il vaudrait mieux parler de « détection d'erreurs dues à des intrusions ».

« partie » soit suffisamment sécurisée et donc, de préférence, qu'elles soient diversifiées pour réduire les risques de vulnérabilités et de bombes communes. Ceci est d'autant plus important que, contrairement aux fautes accidentelles, les attaques ne sont généralement pas indépendantes entre elles : si une attaque a réussi sur une partie du système, l'attaquant ciblera sans doute rapidement d'autres parties ;

- il ne faut pas qu'une seule intrusion dans une partie du système fournisse à l'attaquant des informations sensibles (confidentialité). Ceci est d'autant plus important que la redondance, nécessaire à la tolérance aux fautes, peut fournir plus d'occasions d'attaques aux pirates éventuels.

6.2. Exemples de mise en œuvre

Pour préserver la confidentialité, malgré la redondance nécessaire, une technique de compensation particulière a été développée : la *fragmentation-redondance-dissémination* (FRD) (Fabre *et al.* 1996, Deswarte & Powell 2004). La fragmentation consiste à découper les informations sensibles en fragments de telle sorte qu'un fragment isolé ne contienne pas d'information significative (confidentialité). On ajoute de la redondance à ces fragments de façon à ce que la modification ou la destruction de fragments ou de copie de fragments n'empêche pas la reconstruction de l'information (intégrité et disponibilité). Enfin, la dissémination vise à ce qu'une intrusion ne donne accès qu'à des fragments isolés. La dissémination peut être topologique, en utilisant des sites de stockage différents, ou en transmettant les fragments sur des canaux de communications indépendants. Elle peut être temporelle, en transmettant des fragments dans un ordre aléatoire et en y ajoutant éventuellement des faux fragments de bourrage. La dissémination peut aussi porter sur les privilèges, en exigeant la coopération de plusieurs personnes ayant des privilèges différents pour accomplir une opération (séparation des pouvoirs). Cette technique a été appliquée à un système tolérant les fautes, réparti sur un réseau local, dans le cadre du projet européen Delta-4 (Powell 1991).

Plus récemment, le projet européen MAFTIA (Deswarte & Powell 2004) a visé directement à faciliter les développements d'applications Internet tolérant les intrusions. Pour cela, des protocoles et des intergiciels ('middleware') ont été développés pour gérer plus facilement les communications de groupe tolérant les fautes (y compris les fautes byzantines), avec ou non des contraintes de temps réel, et avec ou non des garanties de confidentialité et d'intégrité. Ces protocoles et intergiciels ont en particulier permis le

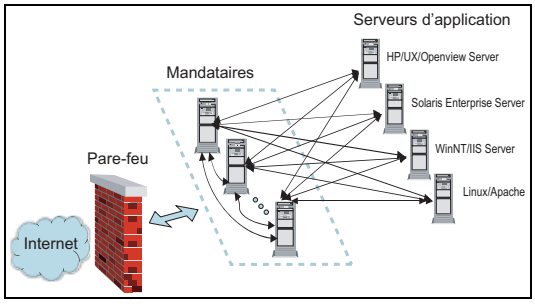


Figure 6.1 - Architecture DIT

développement de tierces parties de confiance (par exemple, une autorité de certification) qui tolèrent les intrusions (y compris de certains des administrateurs). Des méthodes de détection d'intrusions répartie sur Internet ont été étudiées avec une attention particulière, puisque la détection d'intrusions contribue à la tolérance aux intrusions, mais c'est aussi l'une des cibles privilégiées des attaquants. Il faut donc faire en sorte que ces mécanismes de détection tolèrent eux-mêmes les intrusions. Enfin des schémas d'autorisation ont été développés pour des applications mettant en jeu des organisations mutuellement méfiantes. Dans ce cadre, un serveur d'autorisation, tierce partie de confiance tolérant les intrusions, vérifie que chaque transaction (mettant en jeu plusieurs parties) est autorisée, et dans ce cas génère les preuves d'autorisation nécessaires pour l'exécution de chaque élément de la transaction (invocation de méthodes sur des objets élémentaires). Sur chacune des machines participant à ces schémas, un moniteur de référence, implémenté par une carte à puce Java, vérifie que chaque invocation de méthode est accompagnée d'une preuve d'autorisation valide.

Un autre exemple récent est l'architecture DIT ('Dependable Intrusion Tolerance') (Deswarte & Powell 2004) dont l'objectif est de fournir des serveurs Web capables de continuer à fournir un service correct en présence d'attaques. Pour ce type d'application, la confidentialité n'est pas essentielle, en revanche l'intégrité et la disponibilité doivent être assurées, même en cas d'attaque par des adversaires compétents (par exemple, des groupe terroristes ou des services gouvernementaux étrangers).

L'architecture (figure **6.1)** se compose d'un banc de serveurs Web ordinaires, mais aussi diversifiés que possible en ce qui concerne la plateforme matérielle (processeurs Sparc, Pentium, PowerPC, etc.), les logiciels d'exploitation (Solaris, Microsoft Windows, Linux, MacOS, etc.) et les logiciels d'application Web (Apache, IIS, Enterprise Server, Openview Server, etc.). Seul le contenu des pages Web est identique sur

chaque serveur. Les serveurs d'application sont en nombre suffisant pour assurer un temps de réponse satisfaisant sur un taux de requêtes nominal dans un régime de redondance donné (voir ci-après). Les serveurs sont isolés de l'Internet par des « mandataires », eux-mêmes composés d'ordinateurs diversifiés pour ce qui concerne le matériel, mais animés par un logiciel développé spécifiquement. Les requêtes venant de l'Internet, filtrées par un pare-feu, sont prises en compte par l'un des mandataires qui joue le rôle de leader. Le leader répartit les requêtes venant d'Internet vers les différents serveurs et vérifie leurs réponses avant de les envoyer à l'émetteur de la requête. Les mandataires de secours surveillent le fonctionnement du leader en observant les réseaux pare-feu/mandataires et mandataires/serveurs, et en cas de défaillance du leader, élisent entre eux un nouveau leader. Les mandataires traitent également les alarmes émises par les capteurs de détection d'intrusions installés sur les serveurs Web et sur les deux réseaux. Selon le niveau d'alerte, le leader envoie chaque requête à un serveur (régime simplex), deux (régime duplex), trois (régime triplex) ou la totalité des serveurs. Le régime de redondance évolue vers un régime plus sévère (redondance plus élevée) dès que des alarmes sont reçues, mais revient à un régime moins sévère (redondance moindre) lorsque les éléments défaillants ont été diagnostiqués et réparés, et lorsque la fréquence des alarmes diminue.

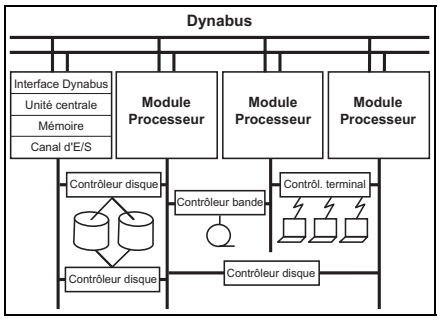
7. Exemples de systèmes opérationnels tolérant les fautes accidentelles

Cette section décrit des architectures illustrant trois techniques de tolérance aux fautes : détection d'erreur et reprise ; détection d'erreur et compensation ; et masquage.

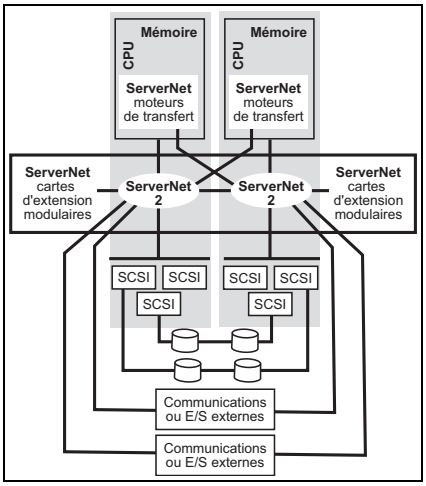
7.1. Détection d'erreur et reprise : les serveurs HP (Tandem) NonStop

Depuis les années 1960, plusieurs systèmes commerciaux mettent en exergue les propriétés de disponibilité ou de tolérance aux fautes. On peut notamment citer la famille S/360 d'IBM (et son évolution vers la famille zSeries) et les systèmes NonStop de Tandem (maintenant commercialisés sous la marque HP). À titre d'exemple, nous mettons l'accent sur ces derniers systèmes qui occupent toujours une très large part du marché des systèmes transactionnels critiques (Bartlett & Spainhower 2004).

Les premiers serveurs Tandem NonStop ont été commercialisés en 1976. Leur architecture



a) architecture initiale: HP *K-Series*



b) HP S-Series (deux processeurs)

Figure 7.1 - Serveurs HP (*Tandem*) *NonStop*

(figure **7.1-a**) est constituée de composants *à défaillance au plus tôt* : les unités centrales et les contrôleurs d'entrée-sortie intègrent des mécanismes de détection d'erreur qui, lorsqu'ils sont activés, bloquent l'unité où est détectée l'erreur (cf. § **4**). La détection d'erreur est essentiellement basée sur le contrôle de parité, le codage et les tests de vraisemblance par logiciel et micrologiciel. Les unités sont également conçues pour limiter la propagation d'erreur : par exemple, les contrôleurs de *Dynabus* et les contrôleurs d'entrée-sortie sont construits de telle sorte qu'aucune faute matérielle unique ne puisse bloquer les deux bus auxquels ils sont connectés. Il existe ainsi toujours un chemin pour accéder à un périphérique à double accès.

Les disques sont organisés en « disques miroirs ». Chaque écriture est envoyée aux deux disques, la lecture n'étant faite que sur un seul disque de façon à optimiser le temps d'accès. Si cette lecture déclenche une détection d'erreur, l'ordre de lecture est répercuté sur l'autre disque, qui sert alors d'unité de secours. Cette notion d'unité de secours est généralisée : en cas d'échec d'une opération sur un processeur, un bus ou

un contrôleur, il existe une unité de secours capable d'effectuer la même opération.

Ce même principe est appliqué au logiciel, qui est organisé sous forme de paire de processus en duplication passive (cf. § **3.3.3**) : à chaque processus actif correspond un processus de secours s'exécutant sur un autre processeur ; le processus actif envoie régulièrement des points de reprise au processus de secours. Ces points de reprise sont soit des copies de l'état du processus actif, soit des écarts par rapport à l'état précédent, soit encore une fonction de transformation de l'état. En fonctionnement normal, le processus de secours ne fait que mettre à jour son état en fonction des points de reprise reçus. Si le processeur hébergeant le processus actif défaille, les autres processeurs le détectent par l'absence de message « je suis vivant » (diffusé toutes les deux secondes par tout processeur opérationnel). Le système d'exploitation du processeur sur lequel s'exécute le processus de secours active alors ce processus qui prend la main au niveau du dernier point de reprise reçu.

Cette architecture a fortement évolué depuis son introduction. Une des principales modifications a consisté en l'amélioration de l'efficacité des mécanismes de détection d'erreur « de bout en bout » : en particulier, adjonction de signatures aux messages et de numéros de séquence aux paquets élémentaires. Un autre changement majeur, introduit en 1997 avec les systèmes HP S-series (figure **7.1-b**), concerne le dispositif d'interconnexion : il s'appuie sur une approche plus « orientée réseau », mieux à même de faciliter l'adaptation des configurations. Le module ServerNet est au cœur de ce dispositif et assure le trafic entre les processeurs et les E/S. Ce module met en œuvre un routeur 6 voies avec une bande passante cumulée de 300Mb/sec. Il utilise un protocole de routage de type « trou de ver » ('wormhole') : l'envoi d'un paquet de messages est réalisé dès que suffisamment d'information de l'entête est disponible pour déterminer le chemin de destination du paquet (ceci permet de limiter la latence de diffusion sur le réseau et le besoin de ressources tampon au niveau du routeur). Chaque paquet est protégé par un code CRC qui est recalculé et vérifié par chaque routeur.

Au niveau logiciel, des facilités de programmation ont été introduites (en particulier, avec le moniteur de traitement de transaction PATHWAY), afin de soulager le programmeur d'application de la prise en compte des aspects liés à la répartition et à la tolérance aux fautes. La conception du gestionnaire de la base de données garantit les propriétés transactionnelles d'atomicité, cohérence, isolation et durabilité (ACID) et facilite la réplication de la base sur des sites distants.

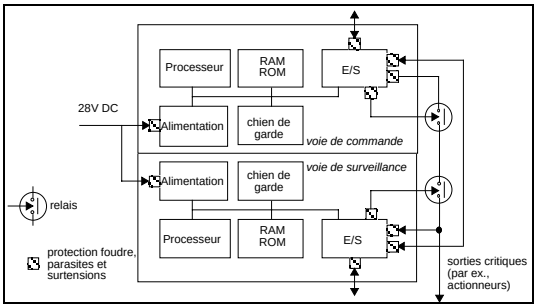


Figure 7.2 - Calculateur autotestable

Plusieurs aides matérielles et logicielles (infrastructure de recueil de traces d'exécution et de préanalyse d'incidents de rapports d'erreurs) ont été développées afin d'assister les administrateurs. Ces dispositifs doivent être continuellement mis à jour pour s'adapter aux nouvelles formes d'erreur observées en exploitation.

7.2. Détection d'erreur et compensation : le système de commande de vol de l'A320

Les avions civils récents, tels que ceux de la famille Airbus 320/330/340 et le Boeing 777, utilisent des calculateurs numériques dans la boucle principale de commande de vol de l'avion. L'objectif est de renforcer encore plus la sûreté globale de l'avion (via une augmentation de la stabilité, la surveillance de l'enveloppe de vol, une protection contre le cisaillement du vent, etc.), ainsi que réduire la fatigue du pilote. Les effets escomptés ne doivent bien sûr pas être anéantis par de nouveaux risques introduits par la technologie même des calculateurs numériques. C'est pour cela que les systèmes de gestion de commandes de vol de ces avions sont conçus de manière à être tolérants aux fautes.

La tolérance aux fautes du système de gestion de commandes de vol de la famille Airbus 320/330/340 est basée sur la technique de détection et de compensation d'erreur (Traverse *et al.* 2004). Chaque calculateur de commande de vol est conçu de manière autotestable vis-à-vis des fautes physiques et de conception, formant ainsi un sous-système sûr en présence de défaillance. Chaque calculateur est composé de deux voies qui sont fonctionnellement équivalentes, mais exécutent des programmes qui ont été conçus de manière diversifiée (figure **7.2**). Les deux voies reçoivent les mêmes entrées, déterminent les sorties correspondantes et vérifient que celles-ci sont en accord avec l'autre voie, une seule voie pilotant les sorties physiques. Toute divergence entre les résultats produits par chaque voie provoque l'inhibition des sorties physiques.

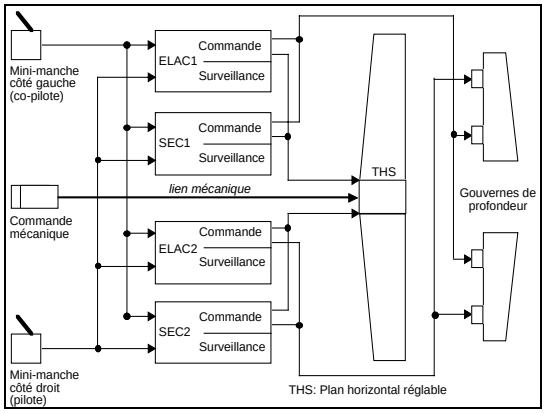


Figure 7.3 - Contrôle du tangage de l'Airbus 320

Chaque axe de commande du vol de l'avion est géré par de tels calculateurs autotestables. Les calculateurs associés à un axe traitent les données des capteurs en exécutant les fonctions de la boucle de commande. Toutefois, à un instant donné, un seul calculateur faisant partie de l'ensemble (calculateur primaire) commande effectivement les actionneurs.

Ce calculateur envoie périodiquement des messages « je suis vivant » aux autres calculateurs leur permettant ainsi de se rendre compte de sa défaillance. Lorsque le calculateur primaire défaille, il le fera dans un mode sûr, c'est-à-dire sans envoyer d'ordre erroné aux actionneurs, ceci grâce à sa conception autotestable. Suivant un ordre prédéterminé, un autre calculateur de l'ensemble devient le nouveau primaire et assure le traitement de la boucle de commande sans aucune secousse notable sur la surface contrôlée.

Le principe de conception diversifiée est appliqué au niveau système. Quatre calculateurs autotestables commandent l'axe de tangage (figure **7.3)** : deux calculateurs (ELAC) associés aux gouvernes de profondeur et aux ailerons, et deux calculateurs (SEC) associés aux aérofreins et aux gouvernes de profondeur. Ces calculateurs sont basés sur des processeurs différents et réalisés par des constructeurs différents. Étant donné que chaque calculateur supporte deux programmes différents, cela fait au total quatre programmes différents pour la commande de tangage.

Par ailleurs, il existe une importante redondance fonctionnelle entre les différentes surfaces de l'avion de sorte qu'il est possible de faire face à la perte de tous les calculateurs associés à certaines surfaces, pour peu que les calculateurs défaillent de manière contrôlée (sûre). De plus, si tous les calculateurs venaient à défaillir, il reste encore un secours manuel permettant de commander l'avion, de façon certes plus limitée.

7.3. Masquage : le Computer Based Interlocking system

Les applications ferroviaires font elles également de plus en plus appel à des architectures informatiques. C'est en particulier le cas des systèmes embarqués de régulation de vitesse ou de contrôle-commande du trafic. Il est clair que les exigences de sécurité-innocuité ont conduit à développer des architectures tolérantes aux fautes mettant en œuvre des redondances matérielles et aussi bien souvent logicielles.

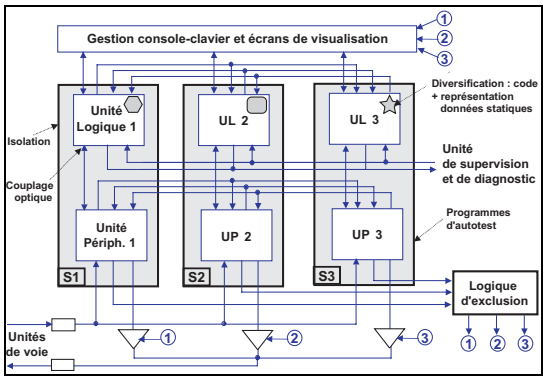
À titre d'exemple, nous considérons le cas du système de contrôle-commande du trafic ferroviaire développé par Ansaldo Segnalamento Ferroviario (Mongardi 1993) et qui gère la gare Termini de Rome.

Comme la plupart des systèmes de ce type, le CBI met en œuvre une architecture hiérarchique incluant :

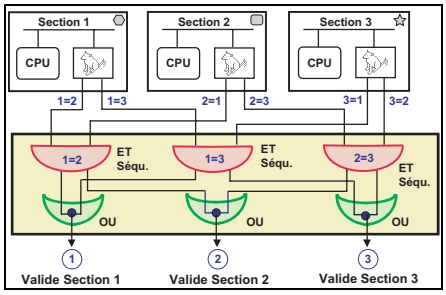
- niveau 1 : un centre de commande équipé de stations de travail (ST) pour le pilotage de la gare et la supervision des opérations de diagnostic et de maintenance ;
- niveau 2 : un calculateur central, ou *noyau de sécurité* (NS) du CBI, gérant globalement pour la gare concernée les opérations sécuritaires ;
- niveau 3 : un ensemble de calculateurs déportés, appelées unités *de voie* (UV), destinés à l'acquisition des informations et à l'application des commandes issues du noyau de sécurité (repérage des trains, signalisation, aiguilles).

Différentes formes de redondances sont utilisées à chacun de ces niveaux : (1) ST redondance sélective active, (2) NS en redondance modulaire triple (TMR), (3) UV autotestables (paire de calculateurs). Nous nous focalisons ici sur le calculateur NS (figure **7.4-a**) qui constitue l'épine dorsale du système CBI.

Les trois sections redondantes du NS permettent de réaliser une décision logique de type vote majoritaire et s'appuient sur un dispositif matériel autotestable (*logique d'exclusion* : LE) assurant l'exclusion de la section déclarée défaillante. Chaque module TMR est isolé des deux autres et possède une alimentation propre. Il est constitué d'une unité logique (UL) et d'une unité périphérique (UP). Le NS est ainsi constitué de six unités de traitement interconnectées par le biais de lignes séries dédiées, avec couplage optique. L'architecture logicielle de chaque module est similaire ; le système d'exploitation, les applications et les données sont rangées en mémoire morte. Toutefois, le principe de diversification est utilisé pour le logiciel applicatif : équipes de programmeurs distinctes, langages distincts, différents codages des données, stockage du code et des données à des adresses distinctes.



a) Le noyau de sécurité



b) La logique d'exclusion (principe)

Figure 7.4 - Le système CBI

Compte tenu de sa conception, le NS combine la propriété de sécurité-innocuité d'une « vote 2-sur-2 » et la disponibilité procurée par la structure TMR. L'élimination d'une unité défaillante par la logique d'exclusion évite tout risque de défaillance de mode commun pouvant résulter de la défaillance d'une des deux unités opérationnelles. L'unité isolée peut alors être accédée à fin de diagnostic et de maintenance puis être réinsérée dans l'architecture opérationnelle du NS.

La logique d'exclusion (figure **7.4-b**) est constituée de trois modules identiques qui traitent les commandes analogiques produites par les modules du NS, après qu'une opération de vote soit effectuée par logiciel. L'information concernant l'accord d'un module (signal actif) (ou désaccord (signal passif)) avec les deux autres est communiquée à la LE en même temps que les commandes. Chaque module de la LE vérifie qu'il y a accord (fonction ET séquentielle) entre les deux signaux correspondants qui lui sont fournis et active le signal de validation de l'alimentation des pilotes de transmission des modules du NS (communication avec les ST du poste de commande et les UV). En cas de désaccord avec les deux autres, l'alimentation des organes de transmission associés à la section défaillante est coupée, ce qui l'isole des deux autres et des UV.

8. Tendances et défis

La plupart des efforts en matière de tolérance aux fautes en informatique ont porté sur les fautes physiques internes ou externes. Des solutions éprouvées existent et cette catégorie de fautes est généralement bien maîtrisée. Ceci est confirmé par les analyses de données d'exploitation concernant les systèmes tolérants aux fautes du matériel (par exemple (Gray 1990)) qui montrent qu'en pratique la tolérance aux fautes se traduit par une amélioration sensible du temps moyen jusqu'à défaillance : de plusieurs semaines à plusieurs années. Par référence à la durée de vie de tels équipements, une interprétation pratique est que, en moyenne, un système tolérant aux fautes ne sera pas soumis à défaillance imputable à des fautes physiques avant de devenir obsolète. Les analyses ont aussi montré que cet état de fait a une influence significative sur la modification du classement des causes des défaillances : les fautes de conception (en particulier, du logiciel) constituent la principale cause, suivie par les fautes d'interaction (qu'il s'agisse de méprises ou de malveillances).

Une douzaine d'années plus tard ce constat est toujours d'actualité : l'analyse de trois sites Internet majeurs rapportée dans (Oppenheimer & Patterson 2002) identifie les erreurs d'opérateurs et les fautes logicielles comme les principales sources de défaillance. Aussi, c'est tout naturellement vers le traitement des fautes logicielles, y compris au niveau des mécanismes logiciels de tolérance aux fautes, et des fautes d'interaction, tant accidentelles que délibérées, que doivent s'orienter les efforts de recherche.

Il faut souligner à nouveau que le déploiement réussi d'un système sûr de fonctionnement nécessite, en plus de l'utilisation de techniques de tolérance aux fautes, la mise en place d'une stratégie volontariste de validation, couvrant à la fois l'élimination de fautes et la prévision de fautes, notamment à des fins :

- de vérification de l'adéquation des mécanismes et algorithmes de tolérance aux fautes vis-à-vis des classes de fautes visées ;
- d'estimation de l'efficacité (facteur de couverture) de la tolérance aux fautes, au moyen d'expériences contrôlées par injection de fautes ;
- d'évaluation des mesures de sûreté de fonctionnement par les processus stochastiques, en tenant compte de l'influence du facteur de couverture.

Les progrès faits par la combinaison de ces approches se concrétisent par l'émergence de propositions d'« étalons » ('benchmarks') visant à caractériser objectivement la sûreté de fonctionnement des systèmes (voir par exemple (Kalakech *et al.* 2004)).

Comme le montrent les exemples de la section **7**, les techniques de tolérance aux fautes mettant en œuvre des redondances architecturales ont été largement déployées dans la plupart des domaines industriels, tout d'abord dans des domaines spécifiques, tels que l'espace et les télécommunications, puis, suivant en cela la tendance forte vers l'informatisation, dans tous les secteurs de l'industrie et des services : transport, production et fourniture d'énergie, finance, etc.

Dans ce contexte, les besoins en sûreté de fonctionnement et les soucis économiques sont étroitement mêlés. Aussi, les solutions « massives » en particulier, celles basées sur des matériels propriétaires sont devenues difficilement acceptables. Dès lors, il est nécessaire de trouver des compromis qui s'appuient, d'une part sur l'adoption de redondances « à faible coût » (contrôle de flux et de vraisemblance, empaquetage, cf. § **2**) en privilégiant une mise en œuvre logicielle, et d'autre part sur l'utilisation de composants préexistants « sur étagère », soit d'origine commerciale (COTS) (Arlat *et al.* 2000), soit en « source libre » (David *et al.* 2003). Cette tendance touche à la fois les composants matériels et les composants logiciels.

En ce qui concerne le matériel, si l'utilisation de composants commerciaux est maintenant largement répandue, il n'en demeure pas moins que les évolutions technologiques ne sont pas sans poser problème. La complexification des architectures (multiplication des registres internes, parallélisme des opérations) se traduit en pratique par des comportements temporels non déterministes et rend l'estimation de bornes sur les temps d'exécution (voir le chapitre ££de C. Rochange et P. Sainrat££) de plus en plus délicate, voire impossible. Par ailleurs, la réduction des dimensions et des énergies mises en jeu augmente significativement la susceptibilité vis-à-vis des perturbations et des radiations. Comme c'est déjà le cas pour la grande majorité des erreurs rencontrées en exploitation, ces nouvelles causes peuvent être considérées comme des fautes furtives. Aussi, il y a lieu de tenir compte a priori du caractère furtif de la faute avant d'entreprendre (inutilement) une action de passivation. Par exemple, l'utilisation de compteurs à seuil (comptabilisant le nombre d'erreurs successives) peut grandement aider pour guider le choix d'une action entre traitement d'erreur et traitement de faute.

Deux principaux problèmes se posent à un intégrateur désirant incorporer des composants logiciels COTS dans des systèmes destinés à des applications critiques : d'une part, le peu d'information objective généralement disponible quant à leur comportement en présence de fautes, et d'autre part, la difficulté, voire l'impossibilité, d'accéder au code source, ce qui peut constituer un blocage vis-à-vis du processus de certification. En réponse au premier point, parmi les méthodes et solutions architecturales proposées dans (Arlat *et al.* 2000), la combinaison des techniques de caractérisation expérimentale (étalonnage) et de programmation défensive (empaquetage) peuvent permettre de proposer des moyens de confinement, voire de tolérance. Même si ce n'est pas encore entré dans les habitudes des entreprises, le recours à des composants en « source libre » permet de lever la limitation posée par le second point et facilite la mise en œuvre des solutions d'empaquetage.

Au-delà de ces tendances, plusieurs défis majeurs se profilent. Quelques exemples concernent :

- la maîtrise du passage à l'échelle, c'est-à-dire le maintien des propriétés de tolérance aux fautes lors des évolutions des configurations ;
- l'ouverture des systèmes, déjà largement entreprise autour des nombreux services déployés sur Internet, et l'augmentation de leur interconnexion de par la généralisation des communications numériques (par ex., contrôle du trafic aérien par voie satellitaire) ;
- l'accroissement de la complexité et l'hétérogénéité des systèmes matériels et logiciels mettant en œuvre, gérant et utilisant les infrastructures informationnelles ;
- l'évolution vers un plus grande autonomie et vers une mobilité accrue des entités impliquées (nœuds de traitement, code et données), et le développement des architectures pair-à-pair ;
- les évolutions attendues dans les technologies de base pour le traitement l'information, l'émergence des microsystèmes (par exemple, leur exploitation dans le cadre d'applications médicales) ;
- la conception d'interfaces homme-machine assurant un bon équilibre entre l'augmentation de l'autonomie des systèmes informatiques et le besoin de convivialité et d'interopérabilité sémantique des opérateurs en exploitation ;
- la recherche de compromis acceptables entre les soucis de sécurité vis-à-vis des malveillances et de protection de la vie privée.

Deux facteurs importants sont à prendre en compte pour relever ces défis :

- la tolérance aux fautes est une activité d'ingénierie qui doit suivre des règles précises et s'appuyer sur un cadre rigoureux pour la capture des besoins, l'identification des risques et des solutions potentielles dès les phases amont du développement ;
- une bonne gestion de la redondance est essentielle pour assurer le succès d'un système tolérant aux

fautes ; cette réussite dépend fortement des hypothèses de fautes considérées, ainsi que de l'analyse explicite de la couverture de ces hypothèses et de l'influence de cette couverture sur les mesures de sûreté de fonctionnement.

9. Conclusion

Ce chapitre a brossé les grands principes d'obtention de la sûreté de fonctionnement des systèmes informatiques, en s'appuyant sur la notion de tolérance aux fautes. Les principales méthodes ont été détaillées. Des points de focalisation ont été fournis, d'une part sur les techniques propres au cas des systèmes répartis et d'autre part sur la prise en compte spécifique des fautes du logiciel, des erreurs d'interaction homme-machine et des malveillances. Quelques exemples de systèmes réels ont illustré la mise en pratique de différentes techniques de tolérance aux fautes. Enfin, les principales tendances et les défis posés à l'informatique en ce qui concerne la tolérance aux fautes ont été brièvement exposés. Le lecteur est encouragé à approfondir l'ensemble de ces aspects à partir de la bibliographie.

Remerciements. Les auteurs tiennent à associer l'ensemble des membres du groupe « Tolérance aux fautes et Sûreté de Fonctionnement informatique » du LAAS-CNRS, qui ont largement contribué, depuis plus de trente ans, à la « culture » de la tolérance aux fautes.

Bibliographie

(Amalberti & Malaterre 2001) R. Amalberti, G. Malaterre, "De l'erreur humaine au risque : évolution des concepts en psycho-ergonomie", dans *Risques erreurs et défaillances - Approche interdisciplinaire* (Actes de la première séance du séminaire « Le risque de défaillance, son contrôle par les individus et les organisations dans les activités à hauts risques »), (Ed. R. Amalberti. C. Fuchs, C. Gilbert), pp. 71-106, Publications de la MSH-Alpes, 2001.

(Arlat *et al.* 1990) J. Arlat, M. Aguera, L. Amat, Y. Crouzet, J.-C. Fabre, J.-C. Laprie, E. Martins, D. Powell, "Fault Injection for Dependability Validation — A Methodology and Some Applications", *IEEE Trans. on Software Engineering*, 16 (2), pp.166-182, février 1990.

(Arlat *et al.* 1999) J. Arlat, Y. Crouzet, P. David, J.-L. Dega, Y. Deswarte, J.-C. Laprie, D. Powell, C. Rabéjac, H. Schindler, J.-F. Soucailles, "Fault Tolerant Computing," dans *Encyclopedia of Electrical and Electronics Engineering*, (Ed. J. G. Webster), Ch. 7, pp. 285-313, J. Wiley & Sons, 1999.

(Arlat *et al.* 2000) J. Arlat, J.-P. Blanquart, T. Boyer, Y. Crouzet, M.-H. Durand, J.-C. Fabre, M. Founau, M. Kaâniche, K. Kanoun, P. Le Meur, C. Mazet, D. Powell, P. Thévenod-Fosse, F. Scheerens, H. Waeselynck, *Composants logiciels sûrs de fonctionnement - intégration de COTS*, 158 p., Hermes Science Publications, Paris, 2000.

(Arlat *et al.* 2002) J. Arlat, J.-C. Fabre, M. Rodríguez, F. Salles, "Dependability of COTS Microkernel-based Systems". *IEEE Trans. on Computers*, 51 (2), pp.138-163, 2002.

(Avižienis & Kelly 1984) A. Avižienis, J. P. J. Kelly, "Fault Tolerance by Design Diversity: Concepts and Experiments", *IEEE Computer*, 17 (8), pp.67-80, août 1984.

(Avižienis & He 1999) A. Avižienis, Y. He, "Microprocessor Entomology: The Taxonomy of Design Faults in COTS Microprocessors", dans *Dependable Computing for Critical Applications (Proc. 7th IFIP Working Conf. on Dependable Computing for Critical Applications: DCCA-7, San Jose, CA, USA, January 1999*, San Jose, CA, USA), (Eds. C. B. Weinstock, J. Rushby), pp.3-23, IEEE CS Press, 1999.

(Avižienis *et al.* 2004) A. Avižienis, Jean-Claude Laprie, B. Randell, C. Landwehr, "Basic Concepts and Taxonomy of Dependable and Secure Computing", *IEEE Trans. on Dependable and Secure Computing*, 1 (1), pp.11-33, janv.-mars 2004.

(Bartlett & Spainhower 2004) W. Bartlett, L. Spainhower, "Commercial Fault Tolerance: A Tale of Two Systems", *IEEE Trans. on Dependable and Secure Computing*, 1 (1), pp.87-96, janv.-mars 2004.

(Chandra & Toueg 1996) R. D. Chandra, S. Toueg, "Unreliable Failure Detectors for Reliable Distributed Systems", *Journal of the ACM*, 43 (2), pp.225-67, 1996.

(Cristian & Fetzer 1998) F. Cristian, C. Fetzer, "The Timed Asynchronous System Model", *Proc. 28th IEEE Int. Symp. Fault-Tolerant Computing (FTCS-28)*, (Munich, Allemagne), pp.140-49, 1998.

(David *et al.* 2003) P. David, H. Waeselynck, B. Bérard, P. Coupoux, Y. Crouzet, Y. Garnier, S. Goiffon, G. Mariano, V. Nicomette, L. Planche, I. Puaut, J.-M. Tanneau, *Logiciel libre et sûreté de fonctionnement*, 234 p., Hermes Science Publications, 2003.

(Deswarte & Powell 2004) Y. Deswarte, D. Powell, "Intrusion Tolerance for Internet Applications", dans *Building the Information Society – Proc. 18h IFIP World Computer Congress, Toulouse*, (R. Jacquart, Ed.), pp. 241-256, Kluwer Academic Publishers, Norwell, MA, USA, 2004.

(Elnozahy *et al.* 2002) E. N. Elnozahy, L. Alvisi, Y.-M. Wang, D. B. Johnson, "A Survey of Rollback-Recovery Protocols in Message-Passing Systems", *ACM Computing Surveys*, 34 (3), pp.375-408, 2002.

(Fabre *et al.* 1996) J.-C. Fabre, Y. Deswarte, L. Blain, "Tolérance aux fautes et sécurité par fragmentation-redondance-dissémination", *Technique et Science Informatiques (TSI)*, 15 (4), pp.405-427, 1996.

(Gray 1986) J. Gray, "Why Do Computers Stop and What Can Be Done About It?" *Proc. 5th IEEE Int. Symp. on Reliability in Distributed Software and Database Systems (SRDSDS-5)*, (Los Angeles, CA, USA), pp.3-12, IEEE CS Press, 1986.

(Gray 1990) J. Gray, "A Census of Tandem System Availability Between 1985 and 1990", *IEEE Trans. on Reliability*, 39 (4), pp.409-432, octobre 1990.

(Huang *et al.* 1995) Y. Huang, C. Kintala, N. Kolettis, N. Fulton, "Software Rejuvenation: Analysis, Module and Applications", *Proc. 25th IEEE Int. Symp. Fault-Tolerant Computing (FTCS-25)*, (Pasadena, CA, USA), pp.381-390, IEEE CS Press, 1995.

(Kalakech *et al.* 2004) A. Kalakech, K. Kanoun, Y. Crouzet, J. Arlat, "Benchmarking the Dependability of Windows NT, 2000 and XP", *Proc. IEEE/IFIP Int. Conf. on Dependable Systems and Networks (DSN 2004)*, (Florence, Italie), pp.681-686, IEEE CS Press, 2004.

(Laprie *et al.* 1990) J.-C. Laprie, J. Arlat, C. Béounes, K. Kanoun, "Definition and Analysis of Hardware-and-Software Fault-Tolerant Architectures", *IEEE Computer*, 23 (7), pp.39-51, 1990.

(Laprie *et al.* 1996) J.-C. Laprie, J. Arlat, J.-P. Blanquart, A. Costes, Y. Crouzet, Y. Deswarte, J.-C. Fabre, H. Guillermain, M. Kaâniche, K. Kanoun, C. Mazet, D. Powell, C. Rabéjac, P. Thévenod, *Guide de la sûreté de fonctionnement, 2ème édition*, 369 p., Cépaduès-Éditions, Toulouse, 1996.

(Laprie 2004) J.-C. Laprie, "Sûreté de fonctionnement des systèmes : concepts de base et terminologie", *Revue de l'Électricité et de l'Électronique (REE)*, (11), pp.95-105, novembre 2004.

(Maes1987) P. Maes, "Concepts and Experiments in Computational Reflection", *Proc. ACM Conf. on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'87)*, (Orlando, FL, USA), pp.147-155, 1987.

(Mongardi 1993) G. Mongardi, "Dependable Computing for Railway Control Systems", dans *Dependable Computing for Critical Applications (Proc. 3rd IFIP Working Conf. on Dependable Computing for Critical Applications: DCCA-3, Palermo, Italy, September 1992, Vienna, Austria*) (Eds. C. E. Landwehr, B. Randell, L. Simoncini), pp.255-277, Springer-Verlag, 1993.

(Muller *et al.* 1996) G. Muller, M. Banâtre, N. Peyrouse, B. Rochat, "Lessons from FTM: An Experiment in the Design and Implementation of a Low Cost Fault Tolerant System" *IEEE Transactions on Reliability*, 45 (2), pp.332-340, juin 1996.

(OMG 2004) OMG, "Fault Tolerant CORBA", dans *Common Object Request Broker Architecture (CORBA), v3.0.3*, pp.23.1-23.106, Object Management Group, 2004.

(Oppenheimer & Patterson 2002) D. Oppenheimer, D. A. Patterson, "Architecture and Dependability of Large-Scale Internet Services", *IEEE Internet Computing*, 6 (5), pp.41-49, sept.-oct. 2002.

(Powell 1991) D. Powell (Ed.), *Delta-4: A Generic Architecture for Dependable Distributed Computing*, 484 p., Springer-Verlag, ISBN 3-540-54985-4, 1991.

(Powell 1992) D. Powell, "Failure Mode Assumptions and Assumption Coverage", *Proc. 22nd IEEE Int. Symp. Fault-Tolerant Computing (FTCS-22)*, (Boston, MA, USA), pp.386-395, IEEE CS Press, 1992.

(Rodríguez *et al.* 2002) M. Rodríguez, J.-C. Fabre, J. Arlat, "Empaquetâches de tolérance aux fautes pour les systèmes temps-réel", *Technique et Science Informatiques (TSI)*, 23 (8), pp.179-514, 2002.

(Rushby 1992) J. Rushby, "Formal Specification and Verification of a Fault-Masking and Transient-Recovery Model for Digital Flight Control Systems", *Proc. 2nd Int. Symp. on Formal Techniques in Real Time and Fault-Tolerant Systems*, LNCS 571, pp.237-258, Springer-Verlag, Nijmegen, Pay-Bas, 1992.

(Siewiorek & Swarz 1992) D. P. Siewiorek, R. S. Swarz, *Reliable Computer Systems — Design and Evaluation*, 908 p., Digital Press, Burlington, MA, USA, 1992.

(Traverse *et al.* 2004) P. Traverse, I. Lacaze, J. Souyris, "Airbus Fly-by-Wire: A Total Approach to Dependability", dans *Building the Information Society – Proc. 18h IFIP World Computer Congress, Toulouse*, (Ed., R. Jacquart), pp.191-212, Kluwer Academic Publishers, Norwell, MA, USA, 2004.

(Wakerly 1978) J. Wakerly, *Error Detecting Codes, Self-Checking Circuits and Applications*, 231 p., Elsevier North-Holland, New York, NY, USA, 1978.

(Yeh 1998) Y. C. Yeh, "Dependability of the 777 Primary Flight Control System", dans *Dependable Computing for Critical Applications (Proc. DCCA-5, Urbana-Champaign, IL, USA, September 1995)* (Eds. R. K. Iyer, M. Morganti, W. K. Fuchs, V. Gligor), pp.1-17, IEEE CS Press, 1998.

Index

A

C

D

E

F

I

M

N

P

R

S

T

V

Z