TOLÉRANCE AUX FAUTES, SÉCURITÉ ET PROTECTION DANS LES SYSTÈMES RÉPARTIS

Ce rapport présente les méthodes et techniques de la sûreté de fonctionnement ainsi que leur application aux systèmes répartis. Dans la première section, les notions de base de la sûreté de fonctionnement et la terminologie associée sont introduites. La deuxième section décrit les techniques de la tolérance aux fautes et montre le profit que l'on peut tirer de la répartition des systèmes pour tolérer les fautes. Enfin, la troisième section est consacrée aux aspects liés à la sécurité, qui ont une importance particulière dans les systèmes répartis.

1. NOTIONS DE SÛRETÉ DE FONCTIONNEMENT[1]

1.1. Concepts de base et terminologie

La *sûreté de fonctionnement* d'un système informatique est la propriété qui permet à ses utilisateurs de placer une *confiance justifiée dans le service* qu'il leur délivre.

En fonction des applications du système informatique, les différentes facettes de la sûreté de fonctionnement se verront accorder une importance plus ou moins grande, c'est-à-dire que la sûreté de fonctionnement peut être considérée selon des points de vue différents mais complémentaires, ce qui permet de définir les attributs de la sûreté de fonctionnement :

- par rapport à la capacité du système à *être prêt à délivrer le service*, la sûreté de fonctionnement est perçue comme la *disponibilité* (en anglais : "*availability*"),
- par rapport à la *continuité de service*, la sûreté de fonctionnement est perçue comme la *fiabilité* (en anglais : "*reliability*"),
- par rapport à l'évitement de *conséquences catastrophiques sur l'environnement*, la sûreté de fonctionnement est perçue comme la *sécurité-innocuité* (en anglais : "*safety*"),
- par rapport à la *préservation de la confidentialité et de l'intégrité* des informations, la sûreté de fonctionnement est perçue comme la *sécurité* (en anglais : "*security*").

---

[1] Le contenu de cette section est directement adapté de [Laprie 89] et [Laprie 92]. Les concepts et la terminologie présentés ici sont le résultat actuel des réflexions et discussions menées depuis une quinzaine d'années dans un certain nombre de groupes de travail (en particulier de l'IFIP et de l'IEEE) et dans le groupe de recherche *Tolérance aux fautes et sûreté de fonctionnement informatique* du LAAS-CNRS, dirigé par Jean-Claude Laprie qui a été le principal animateur de ces réflexions.

1.1.1. Entraves à la sûreté de fonctionnement

Unrrecte. t lorsque le service délivré dévie du service spécifié, la *spécification* étant une description agréée (par exemple entre le fournisseur et l'utilisateur) du service attendu.

La défaillance survient parce que le système a un comportement erroné : une *erreur* est la partie de l'état du système (par rapport au processus de traitement) qui est susceptible d'entraîner une défaillance. La cause adjugée ou supposée de l'erreur est une *faute*. Une erreur est donc la manifestation d'une faute *dans le système*, et une défaillance est donc l'effet d'une erreur *sur le service*.

Une faute est *active* lorsqu'elle produit une erreur. Une faute active est soit une faute interne qui était précédemment *dormante* (c'est-à-dire qu'elle ne produisait pas d'erreur) et qui a été activée par le processus de traitement, soit une faute externe. Une faute interne peut passer, de manière cyclique, de l'état dormant à l'état actif.

Une erreur est, par nature, temporaire. Elle peut être latente ou détectée : une erreur est *latente* tant qu'elle n'a pas été reconnue en tant que telle ; elle est détectée soit par des mécanismes de détection d'erreur qui analysent l'état du système, soit par l'effet de l'erreur sur le service (défaillance). Généralement, une erreur propage d'autres erreurs, nouvelles, dans d'autres parties du système.

Une défaillance survient lorsqu'une erreur traverse l'interface système-utilisateur et affecte le service délivré par le système. Si un système peut être considéré comme un ensemble de composants, la conséquence de la défaillance d'un composant est une faute interne pour le système qui le contient, et aussi une faute externe pour le ou les composants qui interagissent avec lui.

Ceci conduit à la chaîne fondamentale suivante :

... ➙ défaillance ➙ faute ➙ erreur ➙ défaillance ➙ faute ➙ ...

Quelques exemples permettent d'illustrer cette terminologie :

- Un programmeur qui se trompe a une *défaillance* suite à une *erreur* de raisonnement ; la conséquence en est une *faute dormante* dans le logiciel écrit. Lorsque cette faute sera sensibilisée, elle deviendra *active*. La faute active produit une ou des *erreurs* dans les données traitées. Lorsque ces erreurs affectent le service délivré, une *défaillance* survient.
- Un court-circuit qui se produit dans un circuit intégré est une défaillance du circuit. La conséquence est une faute (connexion collée à une valeur booléenne, modification de la fonction du circuit, etc.) qui restera dormante tant qu'elle ne sera pas activée. La suite du processus est identique à l'exemple précédent.
- Une perturbation électromagnétique d'énergie suffisante est une *faute externe*. Cette faute peut soit créer directement une *erreur* par interférence électromagnétique avec les charges électriques circulant dans les connexions, soit créer une autre faute (interne).

- Une interaction homme-machine inappropriée, effectuée par un opérateur durant la vie opérationnelle du système, est une *faute*. L'altération des données qui en résulte est une *erreur*, etc.
- L'erreur d'un rédacteur de manuel de maintenance ou d'utilisation peut résulter en une faute dans le manuel correspondant, sous la forme d'une ou plusieurs directives fautives. Cette faute restera dormante tant que la ou les directives ne seront pas appliquées pour faire face à une situation donnée, etc.

Ces exemples montrent clairement que la dormance de faute peut varier considérablement en fonction de la faute considérée, de l'utilisation du système, etc.

Les fautes d'origine humaine peuvent être accidentelles ou intentionnelles. Ainsi, l'exemple précédent relatif à une erreur de programmation et à ses conséquences peut être ré-écrit de la façon suivante : une bombe logique (faute de conception volontaire) est créée par un programmeur malintentionné ; elle restera dormante jusqu'à son activation (par exemple à une date prédéterminée) ; elle produira alors une erreur qui peut se manifester par un débordement mémoire ou par le ralentissement de l'exécution des programmes ; le service délivré sera alors affecté de ce qu'on appelle un "déni de service", qui est un type de défaillance particulier (cf. §3.1).

1.1.2. Moyens de la sûreté de fonctionnement

Les moyens de la sûreté de fonctionnement sont les méthodes, outils et solutions qui permettent de fournir au système l'aptitude à délivrer un service conforme au service spécifié (*obtention de la sûreté de fonctionnement*), et de donner confiance dans cette aptitude (*validation de la sûreté de fonctionnement*).

La conception et la réalisation d'un système informatique sûr de fonctionnement passent par l'utilisation *combinée* d'un ensemble de méthodes qui peuvent être classées de la manière suivante :

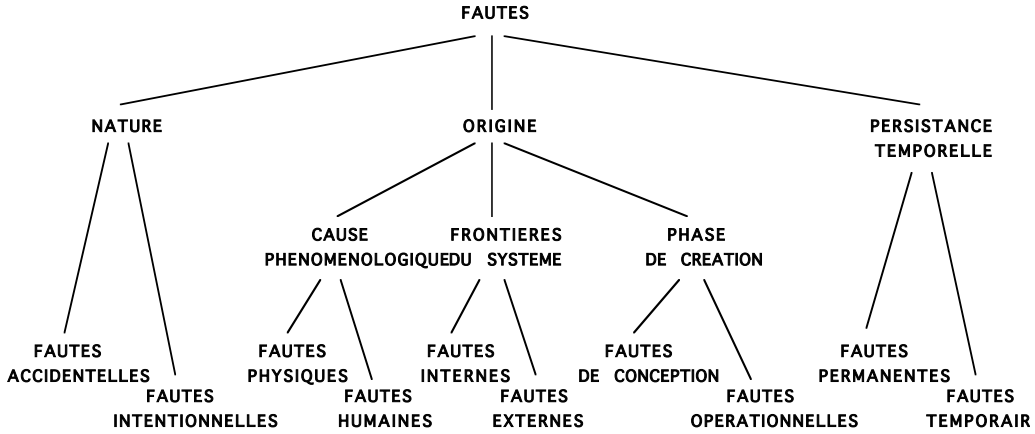
- *prévention des fautes* : comment empêcher, *par construction*, l'occurrence ou l'introduction de fautes,
- *tolérance aux fautes* : comment fournir, *par redondance*, un service conforme à la spécification, en dépit des fautes,
- *élimination des fautes* : comment réduire, *par vérification*, la présence de fautes (leur nombre et leur gravité),
- *prévision des fautes* : comment estimer, *par évaluation*, la présence et la création des fautes et leurs conséquences.

Ces méthodes peuvent être groupées de différentes façons. Ainsi, prévention des fautes et élimination des fautes peuvent être vues comme constituant l'*évitement des fautes* : comment tendre vers un système exempt de fautes. De même, prévention des fautes et tolérance aux fautes peuvent être vues comme constituant l'*obtention* de la sûreté de fonctionnement ; élimination des fautes et prévision des fautes peuvent être vues comme constituant la *validation* de la sûreté de fonctionnement.

1.2. Classifications des fautes, des erreurs et des défaillances

1.2.1. Des fautes

Les fautes et leurs sources sont extrêmement diverses. Les trois points de vue principaux selon lesquels elles peuvent être classées sont leur nature, leur origine et leur persistance. Ceci conduit à classer les fautes selon l'arbre suivant :



Certaines combinaisons issues de cet arbre n'existent pas ; par exemple, il est difficile d'imaginer que des fautes intentionnelles ne soient pas aussi des fautes humaines. On peut dès lors regrouper les combinaisons possibles et leur affecter leur appellation usuelle, comme dans le tableau suivant :

Nature		**Origine**						**Persistance temporelle**		**Appelation usuelle**
		Cause		**Extension**		**Phase**				
acci-der	inten-tion	physi-que	humai-n	inter-ne	exter-n	concep-tion	opéra-tion	perma-nente	tempo-raire	
✧		✧		✧			✧	✧		**Faute physique**
✧		✧		✧			✧		✧	**Intermit-tente**
✧		✧			✧		✧		✧	**Faute transitoir**
✧			✧	✧		✧		✧		**Faute de conception**
✧			✧	✧		✧			✧	**Intermit-tente**
✧			✧		✧		✧		✧	**d'Interac-tion**
	✧		✧	✧		✧		✧		**Logique malicieuse**
	✧		✧		✧		✧		✧	**Intrusion**

1.2.2. Des erreurs

Une erreur a été définie comme étant susceptible de provoquer une défaillance. Qu'une erreur conduise ou non à défaillance dépend de trois facteurs principaux :

- La composition du système, et particulièrement la nature de la redondance existante :
  - redondance intentionnelle (introduite pour tolérer les fautes), qui est explicitement destinée à éviter qu'une erreur ne conduise à défaillance,
  - redondance non intentionnelle (il est, en pratique, difficile sinon impossible de construire un système sans aucune forme de redondance), qui peut avoir le même effet (mais inattendu) que la redondance intentionnelle.
- L'activité du système : une erreur peut être corrigée par réécriture avant qu'elle ne crée de dégât.
- La définition d'une défaillance du point de vue de l'utilisateur : ce qui est une défaillance pour un utilisateur donné peut n'être qu'une nuisance supportable pour un autre utilisateur. Peuvent être, par exemple, prises en compte :
  - la granularité temporelle de l'utilisateur : selon cette granularité, une erreur qui traverse l'interface système-utilisateur peut ou non être considérée comme une défaillance ;
  - la notion de "taux d'erreur acceptable" (implicitement, avant de considérer qu'une défaillance est survenue), qui est classique en transmission de données.

1.2.3. Des défaillances

Un système ne défaille généralement pas toujours de la même façon, ce qui conduit à caractériser les modes de défaillances selon trois points de vue : leurs domaines, leur perception par les utilisateurs, et leurs conséquences sur l'environnement.

- Le *domaine de défaillance* conduit à distinguer :
  - les *défaillances de valeurs* : les valeurs numériques du service délivré ne sont pas conformes à la spécification ;
  - les *défaillances temporelles* : les instants de délivrance du service ne sont pas conformes à la spécification ; selon que le service est délivré trop tôt ou trop tard, la notion de défaillance temporelle peut être affinée en défaillance temporelle *en avance*, ou défaillance temporelle *en retard*.

  Un cas particulier est celui des *défaillances par omission* : aucun service n'est délivré [Cristian 85] [Ezhilchelvan 86]. Une telle défaillance peut être vue comme un cas limite soit d'une défaillance de valeur (valeur absente) ou de défaillance temporelle (défaillance de retard infini). Un système dont les défaillances ne peuvent être que (ou généralement ne sont considérées que comme) des

défaillances par omission, est un système *à silence sur défaillance* ("*fail-silent*") [Powell 88].

- Quand un système a plusieurs utilisateurs, il faut distinguer, selon leur *perception des défaillances* :
  - les *défaillances cohérentes* : tous les utilisateurs ont la même perception des défaillances ;
  - les *défaillances incohérentes* : les différents utilisateurs peuvent avoir différentes perceptions d'une défaillance donnée ; une défaillance incohérente est généralement dénommée *défaillance byzantine* [Lamport 82].

2. TECHNIQUES DE TOLÉRANCE AUX FAUTES ACCIDENTELLES

2.1. Traitement d'erreur et traitement de faute

La tolérance aux fautes a pour but de permettre au système de fournir un service conforme aux spécifications en dépit de la présence ou de l'occurrence de fautes. Elle est mise en œuvre par le traitement d'erreur et le traitement de faute [Laprie 89]. Le *traitement d'erreur* est destiné à éliminer les erreurs, si possible avant qu'une défaillance ne survienne. Le *traitement de faute* est destiné à éviter qu'une ou des fautes ne soient activées à nouveau.

2.1.1. Traitement d'erreur

Le traitement d'erreur peut revêtir deux formes :

- *recouvrement d'erreur*, où un état exempt d'erreur est substitué à l'état erroné, soit par reprise, soit par poursuite (cf. §2.2) ;
- *compensation d'erreur*, où l'état erroné comporte suffisamment de redondance pour permettre la délivrance d'un service approprié (cf. §2.3).

Lorsque le recouvrement d'erreur est utilisé, il est nécessaire que l'état erroné soit identifié comme tel avant de pouvoir le transformer ; c'est le but de la *détection d'erreur*, d'où la locution de *détection et recouvrement d'erreur* qui est couramment employée. Dans le cas de la compensation d'erreur, le traitement d'erreur est le plus souvent appliqué systématiquement, même en absence d'erreur ; c'est le *masquage d'erreur*. Mais la compensation peut aussi n'être déclenchée qu'en cas de détection d'erreur. Dans ce cas, la frontière entre "détection et recouvrement" et "détection et compensation" peut paraître assez floue et dépend, en fait, de la mise en œuvre fine des mécanismes de traitement d'erreur employés (cette distinction sera développée au §2.3).

Les principaux paramètres du traitement d'erreur sont la latence et la couverture. La *latence d'erreur* est définie comme la durée qui sépare l'apparition de l'erreur (c'est-à-dire l'activation de la faute) et le traitement de l'erreur (ou la défaillance du système). La latence est directement liée à la *couverture* des mécanismes de traitement

d'erreur, c'est-à-dire à leur probabilité de traiter correctement une erreur sachant qu'une faute a été activée. Plus la latence est longue ou plus la couverture est faible, plus grand est le risque de propagation et de défaillance. Cette notion de propagation d'erreur est particulièrement importante dans les systèmes répartis où il faut coordonner l'activité de composants multiples, et où, pourtant, il est souhaitable que la défaillance d'un composant n'affecte pas les opérations des autres composants. Cet aspect devient primordial quand un composant donné doit communiquer à d'autres composants une information qui lui est spécifique. Des exemples typiques de telles *informations de source unique* sont les données d'un capteur local, la valeur d'une horloge locale, la vue locale de l'état des autres composants, etc. Il est généralement nécessaire que les composants non-défaillants atteignent un consensus sur ces informations de source unique pour pouvoir maintenir une cohérence mutuelle sur leurs traitements ultérieurs ; ce consensus est bien sûr plus difficile à obtenir s'il n'est pas fait d'hypothèse sur les modes de défaillance, c'est-à-dire si est prise en compte la possibilité de défaillance incohérente ou byzantine (cf. § 1.2.3). Des algorithmes ont été développés pour certains problèmes spécifiques comme la synchronisation des horloges [Lamport 85] ou les protocoles d'appartenance [Cristian 88]. Le coût de ces algorithmes est souvent beaucoup plus important pour des systèmes répartis faiblement couplés que pour les systèmes fortement couplés pour lesquels des développements matériels spécifiques sont envisageables (par exemple, les "*interstages*" de [Smith 86] et [Lala 86]).

2.1.2. Traitement de faute

La première étape du traitement de faute est le *diagnostic* de faute, qui consiste à déterminer les causes des erreurs, en termes de localisation et de nature. Puis viennent les actions destinées à remplir l'objectif principal du traitement de faute : empêcher une nouvelle activation des fautes, c'est-à-dire la *passivation*. Cela est réalisé en retirant les composants considérés comme fautifs du processus d'exécution ultérieur. Si le système ne peut plus délivrer le même service qu'auparavant, il faut effectuer une *reconfiguration*.

S'il est estimé que le traitement d'erreur a pu directement éliminer la faute, ou si sa probabilité de récurrence est suffisamment faible, l'étape de passivation n'est pas nécessaire. Tant que la passivation des fautes n'est pas entreprise, une faute est considérée comme une *faute douce* ; entreprendre la passivation implique que la faute est considérée comme une *faute dure* ou *solide*. A première vue, les notions de fautes douce et dure peut sembler synonyme des notions précédemment introduites de fautes temporaires et permanentes. Effectivement, la tolérance des fautes temporaires ne devrait pas nécessiter de traitement de faute puisque le traitement d'erreur devrait dans ce cas éliminer directement les effets de la faute, qui elle-même a disparu, pourvu qu'une faute permanente n'ait pas été créée dans le processus de propagation. En fait, les notions de faute douce et dure sont utiles pour les raisons suivantes :

- distinguer une faute temporaire d'une faute permanente est une tâche difficile et complexe puisque d'une part une faute temporaire disparaît après un certain

temps, généralement avant qu'un diagnostic ne soit entrepris, et que d'autre part des fautes de classes différentes peuvent conduire à des erreurs très similaires ; en fait, la notion de faute douce ou dure incorpore la subjectivité associée à ces difficultés, y compris le fait qu'une faute peut être déclarée comme douce simplement parce que le diagnostic a échoué ("*Heisenbugs*" [Gray 86]) ;

- ces notions peuvent prendre en compte des subtilités sur les modes d'action de certaines fautes transitoires ; par exemple, faut-il appeler *faute temporaire* une faute interne dormante qui résulte de l'action de particules alpha (due à l'ionisation résiduelle des boîtiers de circuits), ou d'ions lourds dans l'espace, sur des éléments de mémoire (au sens large du terme, incluant les bascules) ? Quoiqu'il en soit, une telle faute est une *faute douce*.

Les définitions précédentes s'appliquent aux fautes physiques aussi bien qu'aux fautes de conception : les classes de fautes qui peuvent être tolérées dépendent des hypothèses de fautes qui sont prises en compte lors de la conception, et donc reposent sur l'*indépendance* des redondances vis-à-vis du processus de création et d'activation des fautes. Considérons par exemple la tolérance aux fautes physiques et la tolérance aux fautes de conception. Une méthode largement utilisée pour obtenir de la tolérance aux fautes consiste à faire exécuter des traitements multiples par des unités multiples. Quand c'est la tolérance aux fautes physiques qui est recherchée, les unités peuvent être identiques, en se basant sur l'hypothèse que les composants matériels défaillent indépendamment les uns des autres ; une telle approche ne convient pas pour la tolérance aux fautes de conception où les unités doivent fournir des services identiques tout en étant *conçues et implémentées séparément*, par exemple par *conception diversifiée* [Avizienis 84].

2.2. Détection et recouvrement

La technique de détection et recouvrement des erreurs consiste, comme son nom l'indique, d'abord à détecter les erreurs, puis, lorsqu'une erreur est détectée, à corriger l'état du système en substituant un état exempt d'erreur à l'état erroné.

2.2.1. Moyens de détection d'erreur

Pour détecter des erreurs, il faut soit appliquer des contrôles de vraisemblance sur l'état interne du système ou sur les interactions entre composants, soit comparer les sorties de plusieurs composants qui effectuent les mêmes traitements.

- Les *contrôles de vraisemblance* présentent l'avantage de ne nécessiter généralement qu'un surcoût peu important par rapport aux éléments fonctionnels du système. Ils peuvent être multiples, ce qui permet de détecter des erreurs dues à de larges classes de fautes. Mais leur couverture est généralement relativement faible[2]**.** Les contrôles de vraisemblance peuvent être mis en œuvre par :

[2] En fait, le compromis surcoût-couverture peut varier considérablement selon les mises en œuvre. Un exemple extrême est celui du processeur SACEM développé pour des applications de transport ferro-

- du matériel spécifique, pour détecter des valeurs erronées (violation de code détecteur d'erreur, adresse mémoire inexistante, instruction inexistante), des instants ou des durées erronés (par chien-de-garde) ou des violations de la protection (cf. §3.2) ; il convient de noter ici que les mécanismes de protection sont effectivement des contrôles de vraisemblance qui permettent de détecter des erreurs dues à des fautes d'interaction volontaires (intrusions), mais aussi des erreurs dues à des fautes de conception (accidentelles ou intentionnelles) ou à des fautes physiques ;
- du logiciel spécifique pour vérifier les valeurs et le format des résultats d'étapes de traitement ou vérifier le séquencement ou les instants de certains événements ; ces contrôles de vraisemblance par logiciel peuvent être intégrés dans le logiciel système, c'est-à-dire mis en œuvre quel que soit le programme d'application (par exemple, contrôle de type, contrôle des valeurs d'indice de tableaux, etc.), ou spécifiques du logiciel d'application (par exemple, fourchettes de valeurs possibles, écart maximal par rapport au résultat précédent dans un contrôle de processus continu, etc.) ; les *contraintes d'intégrité* (classiques en gestion de transactions) ou le *test d'acceptation* ("*acceptance test*") utilisé dans la technique des *blocs de recouvrement* ("*recovery blocks*") [Horning 74] sont des exemples typiques de contrôle de vraisemblance par le logiciel d'application ;
- des programmes de test périodiques ou apériodiques, qui permettent de vérifier (avec une certaine couverture) que le matériel fonctionne correctement [Abadir 82].

• La *comparaison entre plusieurs exemplaires* exige généralement une redondance plus forte, mais offre l'avantage d'une couverture plus grande et d'une latence moindre. Cependant, ainsi que cela a été établi plus haut, il faut que les unités redondantes soient indépendantes vis-à-vis du processus de création et d'activation des fautes que l'on considère : il faut s'assurer que soit les fautes sont créées ou activées indépendamment dans les différents exemplaires, soit si une même faute provoque des erreurs dans plusieurs exemplaires, ces erreurs sont différentes. Ainsi, si seules les fautes physiques internes sont considérées, il est possible d'utiliser des exemplaires identiques dans la mesure où il semble raisonnable d'admettre que les défaillances des différentes unités sont suffisamment indépendantes. En revanche, s'il faut prendre en compte les fautes physiques externes, il est souhaitable que les exemplaires soient similaires mais pas strictement identiques, de façon à ce qu'ils ne soient pas soumis aux mêmes fautes, ou que les erreurs provoquées soient différentes ; par exemple, on pourra utiliser des unités identiques (matériel et logiciel), mais isolées géographiquement (on tire alors profit de la répartition du système pour se prémunir des fautes de mode commun),

viaire où les aspects de sécurité (au sens "*safety*") sont primordiaux ; pour ce processeur la détection d'erreur est basée sur des techniques de codage et de signature associée à chaque donnée ; la couverture obtenue est très bonne, mais au prix d'un surcoût logiciel colossal : pour N lignes de code fonctionnel, il faut compter de l'ordre de 100 x N lignes de code exécutable.

ou ayant des traitements décalés dans le temps [Fabre 82]. S'il n'est pas possible de faire ces hypothèses simplificatrices sur les fautes, c'est-à-dire s'il faut prendre en compte les fautes d'interaction humaines (fautes des opérateurs et intrusions) ou les fautes de conception, intentionnelles ou accidentelles, du matériel ou du logiciel, il faut que les exemplaires soient diversifiés vis-à-vis de ces hypothèses de fautes : développements indépendants de plusieurs variantes de logiciel pour les fautes de conception du logiciel, utilisation de matériels différents pour les fautes de conception du matériel, interfaces différents ou confirmation par des opérateurs différents pour les fautes d'interaction. Un exemple où la diversification matérielle et logicielle est particulièrement poussée est donné par l'architecture du système avionique de l'Airbus A320 [Traverse 89].

Bien sûr, le type de comparaison dépend du type de redondance appliquée. S'il s'agit de deux exemplaires identiques, une comparaison bit-à-bit synchrone est possible. Si les exemplaires sont désynchronisés ou géographiquement dispersés, il faut tenir compte d'un certain asynchronisme dans la fourniture des résultats à comparer. En revanche, si les exemplaires sont diversifiés, une simple comparaison ne suffit pas pour décider si les exemplaires sont ou non équivalents : il faut pour cela mettre en œuvre une véritable fonction de décision qui peut être complexe, et dont la couverture peut ne pas être parfaite. En fait, même si les matériels et logiciels sont identiques, une comparaison bit-à-bit n'est pas toujours applicable. En effet, le comportement des deux exemplaires peut ne pas être totalement déterministe, même en absence de faute, en raison soit de vues locales différentes (par exemple, des adresses locales différentes pour les mêmes données peuvent donner des références différentes), soit de synchronismes différents (horloges locales différentes, séquencement différent des interactions avec d'autres traitements, etc.). Dans ce cas, il faut soit forcer le déterminisme pour conserver la possibilité d'une comparaison bit-à-bit, soit mettre en œuvre une fonction de décision analogue à celle utilisée dans le cas de la diversification.

L'association dans un même composant de capacités de traitement fonctionnelles avec des mécanismes de détection d'erreur conduit à la notion de *composant auto-testable* (matériel ou logiciel), que les mécanismes de détection reposent sur des contrôles de vraisemblance ou sur une comparaison de deux exemplaires. S'il est possible de considérer que la couverture de détection est totale (et la latence nulle), il est facile à partir d'un composant auto-testable de réaliser un composant à *silence sur défaillance* ("*fail-silent*", cf. § 1.2.3) : il suffit pour cela d'empêcher toute interaction avec les autres composants en cas de détection d'erreur, par exemple par l'arrêt du processeur s'il s'agit d'un module de traitement[3]. L'un des principaux avantages de

[3] Cette notion de composant à silence sur défaillance recouvre celle de *logique à défaillance rapide* ("*fail-fast logic*") utilisée dans [Bartlett 87] et correspond à la propriété d'*arrêt sur défaillance* ("*halt-on-failure property*") des processeurs "*fail-stop*" de Schneider [Schneider 84] ; les autres propriétés exigées par Schneider pour ces processeurs, à savoir la propriété d'état de défaillance ("*failure status property*") et la propriété de mémoire stable ("*stable storage property*"), vont au-delà de ce qui est nécessaire pour un composant à silence sur défaillance.

cette approche est de permettre de définir clairement les *zones de confinement d'erreur*, puisqu'il ne peut pas y avoir de propagation d'erreur à l'extérieur d'un composant à silence sur défaillance. Mais pour que cette approche soit valable, elle doit reposer sur une conception spécifique des composants de façon à obtenir une couverture suffisante des mécanismes de détection et d'isolation[4].

2.2.2. Recouvrement des erreurs

Le recouvrement d'erreur consiste à substituer un état exempt d'erreur à l'état erroné. Cette substitution peut elle-même prendre deux formes :

- *reprise*, où le système est ramené dans un état survenu avant l'occurrence d'erreur ; ceci suppose qu'on ait sauvegardé auparavant cet état dans un point de reprise ;
- *poursuite*, où la transformation de l'état erroné consiste à trouver un nouvel état à partir duquel le système peut fonctionner (généralement dans un mode dégradé).

La reprise et la poursuite ne sont pas exclusives : la reprise peut être tentée d'abord ; si elle échoue (par exemple, parce que le point de reprise est lui-même erroné), il convient d'essayer la poursuite.

Pour la reprise, les deux points fondamentaux sont la génération des points de reprise et la restauration de l'état.

- Ce qui est sauvegardé lors de la génération d'un point de reprise n'est généralement pas un cliché de l'état de l'ensemble du système, mais seulement l'état d'une partie du système, généralement un processus ; la sauvegarde de l'état du système est donc en fait constituée d'un ensemble de sauvegardes non-cohérentes entre elles, puisque non-synchronisées. L'établissement de points de reprise et la sauvegarde des informations peut être facilitée par une structuration adaptée de l'application (par exemple par blocs de recouvrement), mais aussi par des mécanismes matériels ou logiciels permettant de sauvegarder automatiquement les données modifiées entre deux points de reprise : antémémoire récursive ("*recursive cache*") [Horning 74], pile de reprise [Deswarte 75], mémoire stable [Banâtre 86] [Banâtre 87].
- La restauration d'un état exempt d'erreur consiste à remettre dans l'état de leur dernier point de reprise au moins l'ensemble des processus qui ont pu être directement contaminés par l'erreur (par exemple, ceux qui s'exécutaient sur l'unité sur laquelle l'erreur a été détectée). En fait, cela ne suffit pas pour obtenir un état cohérent du système. En effet, ces processus ont pu interagir avec d'autres depuis leur dernier point de reprise. Ces autres processus doivent donc eux-mêmes être repris, mais ils ont pu interagir avec d'autres, etc. C'est l'effet

[4] Cette hypothèse de silence sur défaillance est souvent implicite dans beaucoup d'architectures qui prétendent tolérer les fautes, sans pour autant que la conception des composants ne justifie de supposer que la couverture soit suffisante. David Powell [Powell 89] a pourtant montré que ce paramètre de couverture avait plus d'influence sur la fiabilité et la disponibilité du système global que la redondance.

"domino" [Randell 75]. Des algorithmes ont été développés pour rechercher le point de reprise cohérent le plus proche (par exemple, les "*chase protocols*" de [Merlin 78]); dans ce cas, il est recherché un état cohérent que le système aurait pu occuper, plutôt qu'un état que le système a réellement occupé auparavant. Mais la solution la plus simple consiste en général à restreindre les interactions possibles entre processus pour limiter le nombre de processus à reprendre. Les *conversations* [Randell 75] représentent l'une des méthodes pour limiter les interactions.

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, une reprise ne pourra être efficace que s'il est possible de restituer un état exempt d'erreur pour les processus concernés, c'est-à-dire s'il existe plusieurs points de reprise successifs pour chaque processus ou si la structure de l'application permet de conserver des points de reprise *emboîtés* ("*nested*"), comme dans le cas des blocs de recouvrement [Horning 74].

Notons que l'approche transactionnelle donne un support élégant à la fois pour la génération des points de reprise et pour la restauration de l'état du système, en particulier dans le cas de "transactions emboîtées" ("*nested transactions*").

Si, au lieu de l'approche par reprise, c'est la technique de la poursuite qui est adoptée, il faut reconstruire un état acceptable pour le système. En fonction de l'application, cela pourra se faire par une réinitialisation du système et l'acquisition d'un nouveau contexte d'exécution auprès de l'environnement (par exemple, relecture des capteurs dans un système de contrôle-commande). Une autre approche est celle des *traitements d'exceptions* [Cristian 85] : 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 en un traitement d'exception (généralement dégradé).

2.2.3. Avantages et inconvénients des techniques à détection et recouvrement d'erreur

Parmi les avantages des techniques à détection et recouvrement, il faut compter la faible redondance structurelle nécessaire, en particulier, si la détection repose sur des contrôles de vraisemblance. De plus, si de multiples contrôles de vraisemblance sont mis en œuvre, de larges classes de fautes peuvent être tolérées : le contrôle de vraisemblance porte sur l'état du système, indépendamment de l'origine des erreurs ; par exemple, comme cela a déjà été mentionné, les mécanismes de protection détectent des erreurs dues à des intrusions, mais aussi des erreurs dues à des fautes de conception ou à des fautes physiques.

Dans le cas de la technique de reprise, le surcoût temporel (ou redondance temporelle) nécessaire pour l'établissement des points de reprise peut être très important s'il faut sauvegarder des états volumineux (des fichiers, par exemple). Ce coût existe en permanence, même s'il n'y a pas d'erreur.

A cela s'ajoute, aussi bien pour la poursuite que pour la reprise, un surcoût spécifique pour le recouvrement d'erreur. Dans le cas de la poursuite, ce sera le temps de la réinitialisation et de l'acquisition du contexte ou de la commutation vers le traitement d'exception. Dans le cas de la reprise, ce sera le temps de restauration de l'état du système depuis le ou les points de reprise suivi du temps nécessaire pour ré-exécuter les traitements qui avaient déjà été effectués entre le point de reprise et la détection d'erreur.

Au titre des inconvénients de ces techniques, il faut ajouter qu'il est en général nécessaire de structurer l'application pour que le recouvrement soit possible : établissement des points de reprise, contrôle de vraisemblance et traitement d'exceptions doivent généralement être pris en compte dans le développement de l'application, avec un support spécifique du système d'exploitation, ce qui interdit l'usage de systèmes opératoires généraux tels qu'Unix et de progiciels qui n'auraient pas été développés spécialement pour l'architecture considérée. Mais, en général, cette structuration permet aussi d'améliorer la qualité (et donc la fiabilité) des logiciels correspondants.

Exemple : Tandem Non-Stop [Bartlett 87]

Les systèmes Tandem Non-Stop sont conçus pour tolérer une faute matérielle unique. Leur architecture matérielle est présentée par la figure 9.1. Elle est constituée de composants "*fail-fast*", c'est-à-dire que 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. Pour réduire les coûts, ces mécanismes de détection d'erreur sont basés sur des contrôles de vraisemblance (contrôle de parité, codage, tests de vraisemblance par logiciel et micro-logiciel), mais aussi dans certains cas par des circuits autotestables. Ces 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, même en cas de défaillance d'un processeur, d'un bus ou d'un contrôleur.

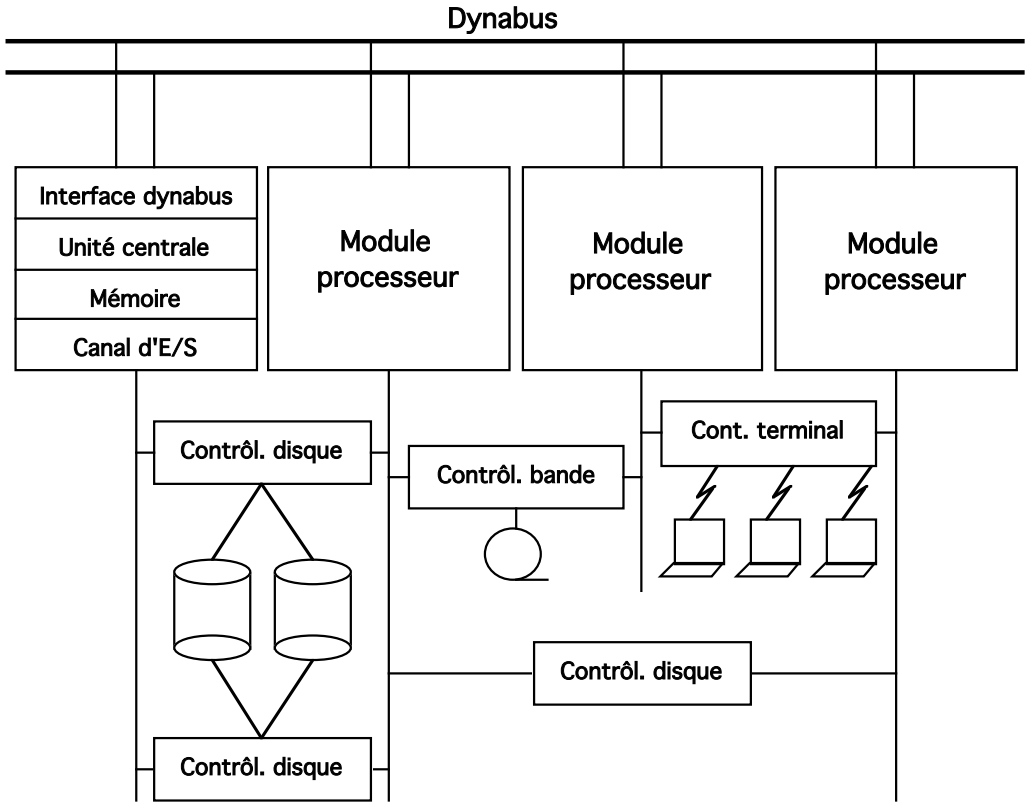


Figure 9.1 : Tandem Non-stop

Les disques sont organisés en *disques miroirs*, c'est-à-dire qu'à chaque disque correspond une copie identique ; chaque écriture est envoyée aux deux disques, la lecture n'étant faite que sur un seul disque, de façon à optimiser les temps d'accès. En cas d'erreur de lecture sur un disque, l'ordre de lecture sera 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 autre unité capable d'effectuer la même opération en secours.

Cette même notion est appliquée au logiciel qui est organisé sous forme de *paire de processus* : à chaque processus 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 *deltas* 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 qu'il reçoit ; si le processeur sur lequel s'exécute 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 en fonctionnement correct) ; le système d'exploitation du processeur sur lequel s'exécute le processus de secours active alors ce processus qui prend la main sur le dernier point-de-reprise reçu.

Cette organisation sous forme de paires de processus impose une conception spécifique du système d'exploitation et des logiciels d'application, en particulier pour la génération des points-de-reprise. La réalisation des applications est facilitée par des bibliothèques de fonctions élémentaires, mais l'incompatibilité avec les produits standard provoque une augmentation significative des coûts.

Un autre inconvénient de cette architecture est lié au fait que la couverture des mécanismes de détection n'est pas totale, et qu'il est donc possible qu'une erreur soit propagée avant le blocage de l'unité défaillante. Cependant, d'après les informations publiées par Tandem, il apparaît que les défaillances globales de leurs systèmes soient en forte majorité dues à des fautes logicielles ou à des fautes d'exploitation, la plupart étant des *Heisenbugs* [Gray 86], c'est-à-dire non diagnostiquées, car difficiles à reproduire.

2.3. Compensation d'erreur

La compensation d'erreur est un traitement particulier effectué sur l'état du système pour fournir un service conforme en dépit des erreurs qui pourraient affecter cet état. Ainsi qu'il a été indiqué plus haut, la compensation d'erreur peut être soit systématique (masquage), soit consécutive à une détection d'erreur (détection et compensation).

Dans les deux cas, il est utile de signaler l'erreur de façon à déclencher le traitement de faute (cf. §2.1.2). En effet, s'il n'est pas effectué de traitement de faute, la redondance peut être dégradée à l'insu des utilisateurs et conduire à défaillance lorsqu'une autre faute est activée[5]. C'est pourquoi, la plupart des mises en œuvre du masquage d'erreur comportent également des moyens de détection d'erreur ; mais, dans ce cas, la détection peut être menée *après* la transformation d'état.

2.3.1. Masquage d'erreur

Un exemple typique de masquage d'erreur est celui du vote majoritaire : les traitements sont exécutés par trois (ou plus) composants identiques dont les sorties sont votées ; les résultats majoritaires sont transmis, les résultats minoritaires (supposés erronés) sont éliminés (cf. figure 9.2). Le vote étant 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 d'erreur de la technique de détection et compensation.

[5] Un exemple notable est celui des mémoires à semi-conducteurs des premières machines IBM 370. En raison de la fiabilité relativement faible (à l'époque) de ces mémoires, elles avaient été munies d'un code correcteur d'erreur. Mais comme aucun programme de test n'était capable de contourner ce mécanisme de correction d'erreur, les équipes de maintenance ne pouvaient pas connaître directement les positions mémoires défaillantes (de façon à remplacer préventivement les modules mémoires correspondants). Le seul moyen de "détecter" ces erreurs, était de mesurer le temps d'exécution des programmes de test mémoire, l'algorithme de correction d'erreur ralentissant significativement l'accès en lecture.

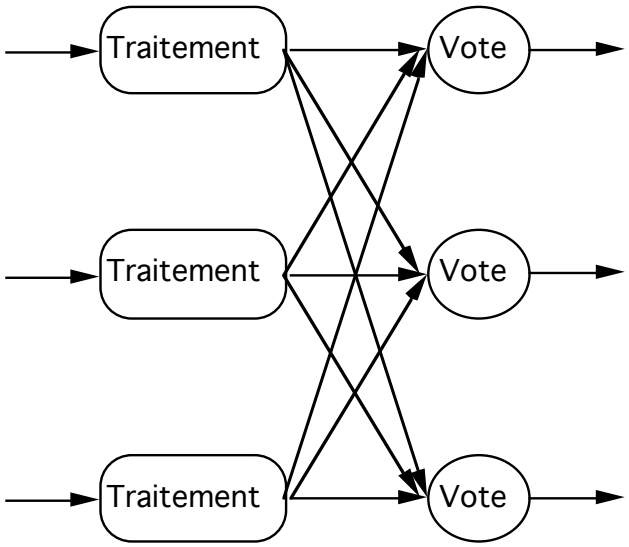


Figure 9.2 : vote majoritaire

Il peut être fait la même remarque sur le vote que sur la détection par comparaison d'exemplaires (cf. §2.2.1) : 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 [Avizienis 85].

Exemple : Tandem Integrity S2 [Jewett, 1991]

Annoncé en 1989, le système Tandem Integrity S2 vise, comme son prédécesseur le système Non-Stop, à tolérer une faute matérielle unique, mais avec comme exigence supplémentaire de pouvoir utiliser des logiciels non spécifiques, et, en particulier, d'avoir un système d'exploitation compatible avec Unix. Ceci a conduit à concevoir l'architecture présentée à la figure 9.3.

Cette architecture est caractérisée par une structure triplée pour les processeurs et leurs mémoires locales et par une structure duplex pour les voteurs (autotestables), les mémoires globales et les processeurs et bus d'entrée-sortie. Les mémoires locales contiennent chacune une copie du noyau Unix (dans une zone protégée) et des zones de programmes et de données d'application. Les mémoires globales contiennent également des zones d'application et des zones de contrôle et de tampon pour les entrées-sorties. En fonctionnement normal, les 3 mémoires locales ont un contenu identique ; de même les 2 mémoires globales ont le même contenu. Chacun des processeurs a sa propre horloge ; leur traitement est resynchronisé par les accès à la mémoire globale, ces accès donnant lieu à un vote majoritaire.

En cas d'inégalité, une erreur est signalée et le traitement se poursuit sans interruption sur les processeurs majoritaires. Un programme d'autotest est alors lancé sur le processeur minoritaire pour déterminer si l'erreur a été produite par une faute douce non-reproductible, auquel cas le processeur peut être réinséré ; dans le cas contraire, le processeur doit être remplacé.

Les entrées-sorties sont basées sur les mêmes techniques que les Tandem Non-Stop ou les Stratus : bus doublés, processeurs d'entrée-sortie autotestables IOP, disques miroirs. Il faut néanmoins noter une particularité : les modules d'interface bus BIM servent à interfacer les bus doublés spécifiques avec des bus VME standard, ce qui permet d'utiliser des périphériques et des contrôleurs d'autres fournisseurs. En cas de défaillance d'un IOP ou du bus associé, le BIM commute le contrôle du bus VME vers l'autre IOP.

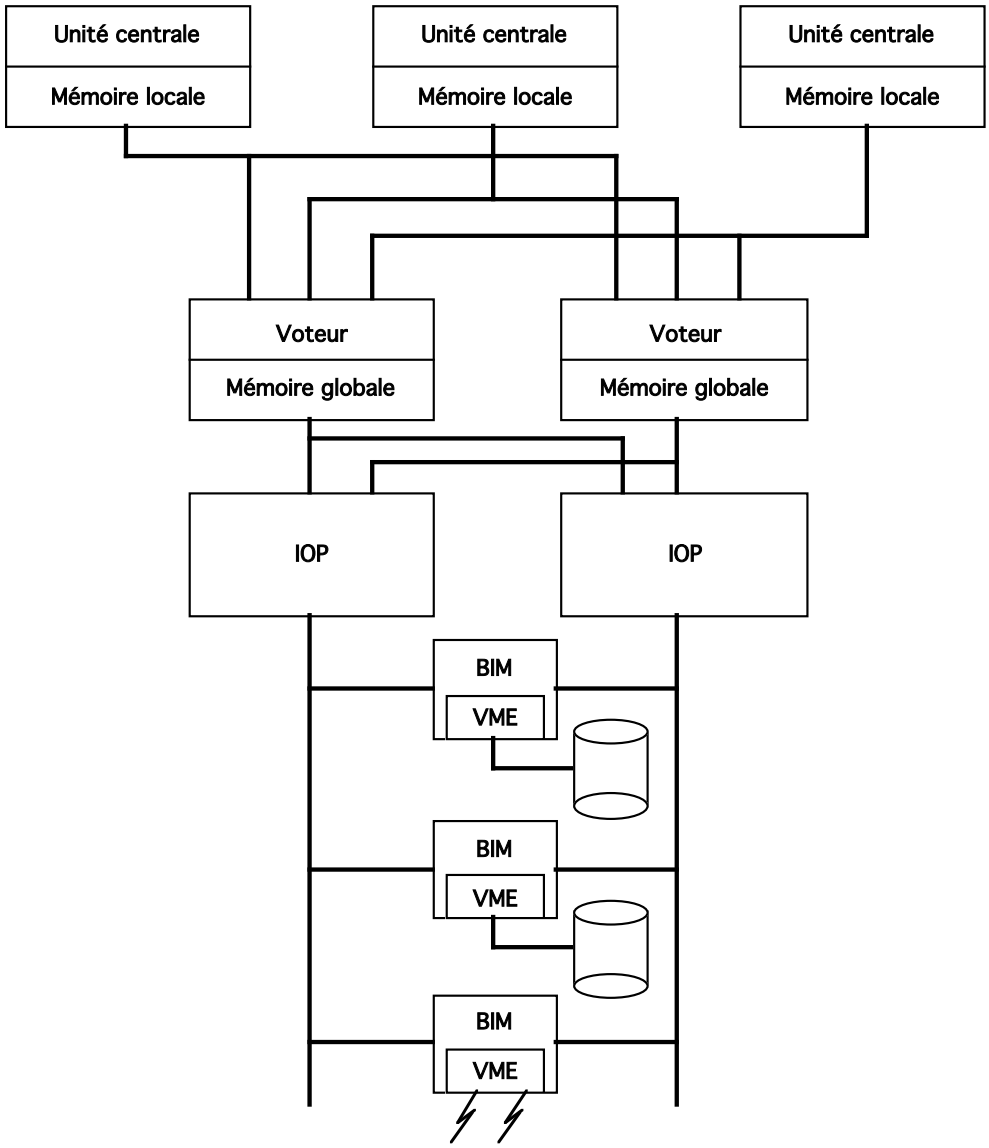


Figure 9.3 : Tandem Integrity S2

2.3.2. Détection et compensation d'erreur

La compensation peut être déclenchée sur détection d'erreur. Les mécanismes de détection peuvent être identiques à ceux de la détection et recouvrement (cf. §2.2.1). Ce qui distingue la compensation d'erreur du recouvrement tel qu'il a été présenté au §2.2.2, c'est que l'état du système est suffisamment redondant pour qu'il ne soit pas nécessaire de ré-exécuter une partie de l'application (reprise) ou d'exécuter un traitement d'application spécifique (poursuite) pour permettre de continuer le traitement fonctionnel du système : il suffit de transformer l'état courant.

Un exemple typique est donné par la plupart des implémentations des codes correcteurs d'erreur : la validité de la valeur codée est vérifiée en permanence et en cas de détection d'erreur, un traitement de correction de cette valeur est lancé.

Une autre mise en œuvre classique de cette technique est d'utiliser des composants auto-testables (cf. § 2.2.1) 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 la commutation des composants.

Exemple : Stratus S/32 [Webber, 1991] ou IBM System/88 [Harrison 87]

Les systèmes Stratus S/32 (également commercialisés par IBM comme System/88) sont des multi-processeurs qui, comme les Tandem Non-Stop, sont conçus pour tolérer une faute matérielle unique. Leur architecture matérielle est présentée par la figure 9.4. Elle est constituée de cartes autotestables, les entrées de chaque carte étant envoyées sur des circuits doublés synchrones dont les sorties sont comparées ; en cas d'inégalité, la carte est isolée électriquement. Ainsi, une carte processeur comporte deux Motorola 68020 dont les horloges et les entrées sont communes et dont les sorties sont comparées. Les contrôleurs de mémoire, de disque et de réseau sont constitués de manière analogue.

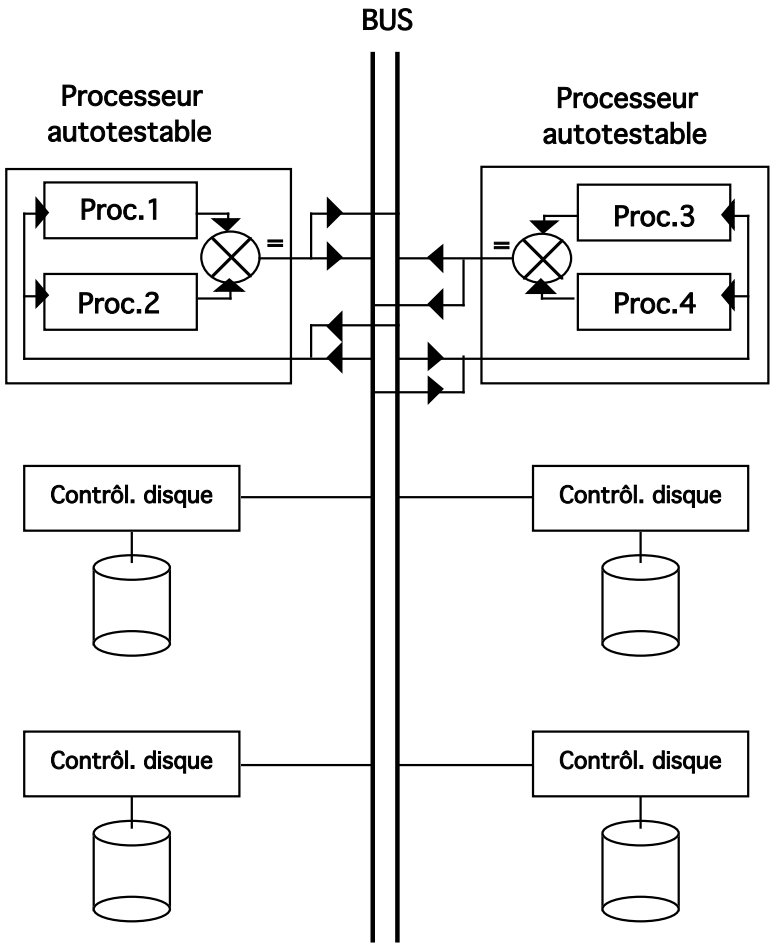


Figure 9.4 : Stratus S/32 ou IBM System/88

Pour poursuivre le traitement sans interruption en cas de défaillance d'un composant, chaque processeur, contrôleur de mémoire ou bus est doublé pour constituer une structure *duplex*, et les disques sont gérés en disques miroir (comme sur le Tandem Non-Stop). Toutes les unités sont actives en même temps ("*hot standby*") ; ainsi, le même traitement s'exécute en même temps sur deux cartes processeurs, et sur deux 68020 sur chaque carte : il y a donc 4 processeurs en redondance active pour exécuter les mêmes traitements qu'exécuterait un processeur unique dans un système non tolérant aux fautes.

Le System/88 peut comporter jusqu'à 8 cartes processeurs, soit donc au total 16 microprocesseurs 68020 pour réaliser une structure fonctionnellement équivalente à un quadri-processeur.

La tolérance aux fautes est transparente au logiciel, ce qui permet d'utiliser des progiciels standard. Le système d'exploitation ne comporte pas de particularité si ce n'est des programmes de maintenance activés en cas de détection d'erreur pour déterminer si la faute est transitoire, auquel cas il suffit de réinitialiser la carte ; dans le cas contraire, il faut remplacer la carte avant de la réinitialiser. Ces opérations se font sans interruption ni ralentissement du traitement, tant qu'il n'y a qu'une faute matérielle.

2.3.3. Avantages et inconvénients des techniques à compensation d'erreur

L'inconvénient majeur des techniques à compensation par rapport aux techniques à recouvrement en est la redondance nécessairement plus élevée : de l'ordre de trois fois plus de matériel en cas de vote majoritaire simple, de l'ordre de quatre fois dans le cas de composants auto-testables en redondance active.

Le premier avantage de ces techniques est que la durée du traitement d'erreur est plus faible que pour le recouvrement d'erreur. Dans le cas du masquage, cette durée est même constante qu'il y ait ou non erreur. Ceci peut être très utile pour les systèmes temps-réels dont il faut pouvoir montrer la capacité à tenir les échéances, même en présence de fautes.

Le second avantage est la transparence des mécanismes de traitement d'erreur vis-à-vis de l'application : il n'est pas nécessaire de structurer l'application en vue d'un éventuel traitement d'erreur. Ceci permet d'utiliser des systèmes d'exploitation et des progiciels standard.

3. SÉCURITÉ ET PROTECTION

Cette section est consacrée à la sécurité vis-à-vis de la confidentialité et de l'intégrité des informations. Il faut pour cela s'intéresser principalement aux fautes intentionnelles, mais pas exclusivement : la confidentialité et l'intégrité peuvent également être mises en danger par des fautes accidentelles.

La *confidentialité* peut être définie comme la capacité du système informatique à empêcher la divulgation d'informations, c'est-à-dire à faire en sorte que les informations soient inaccessibles (ou incompréhensibles) pour les utilisateurs non désignés comme autorisés à y accéder. Le terme "information" doit être pris dans son sens le plus large : il recouvre non seulement les données et les programmes, mais aussi les flux d'information et la connaissance de l'existence de données, de programmes ou de communications.

L'*intégrité* peut être définie comme la capacité du système informatique à empêcher la corruption des informations par des fautes accidentelles ou intentionnelles. Ainsi définie, l'intégrité est une condition nécessaire pour la sûreté de fonctionnement. Mais, dans le cas des fautes intentionnelles, les attaques contre l'intégrité visent soit à

introduire de fausses informations soit à modifier ou à détruire des informations (c'est-à-dire à provoquer des erreurs) pour que le service (inapproprié) délivré par le système produise un bénéfice pour l'attaquant, au détriment des utilisateurs autorisés. C'est typiquement le cas des fraudes informatiques. Bien sûr, l'attaquant essayera en général de faire en sorte que les erreurs qu'il introduit ne soient pas détectables et que la défaillance qui en résulte ne soit pas visible.

Dans d'autres cas de fautes intentionnelles, ce qui est visé par l'attaquant est simplement d'empêcher que le système délivre un service approprié. Il s'agit alors d'une attaque contre la *disponibilité* qui est appelée *déni de service*.

Enfin, un autre type de fautes intentionnelles est constitué par les utilisations (non autorisées) du système à des fins personnelles. Même si ces fautes n'ont pas de conséquence sur le service délivré aux utilisateurs autorisés, elles provoquent une *défaillance* du système dans le sens où le service n'est pas approprié s'il est délivré à des utilisateurs non autorisés.

Les motivations des attaquants peuvent être multiples. Certains considèrent qu'une tentative d'intrusion est un défi intellectuel ou que l'attaque des systèmes informatiques est un sport somme toute honorable ; c'est ce genre d'individus qui fait qu'aujourd'hui le terme "*hacker*" a un sens péjoratif, équivalent de pirate, dans les media américains ; ils sont à l'origine de la plupart des virus et des vers, qui généralement semblent avoir été conçus pour ne pas créer de dégâts, mais qui souvent ont eu des effets désastreux par suite de fautes de conception, d'insuffisance de test et d'incapacité à prendre en compte les différents environnements où ils se sont propagés, en un mot à cause de l'incompétence de leurs créateurs. D'autres s'adonnent à ce passe-temps uniquement par vandalisme, c'est-à-dire pour le plaisir de détruire l'œuvre des autres. D'autres encore ont des motifs politiques ou idéologiques : c'est le cas des membres du Chaos Computer Club allemand qui s'exercent au piratage pour, prétendent-ils, lutter contre les dangers que représente l'informatique pour les libertés individuelles. Ce genre de raisonnement peut également conduire certains au terrorisme informatique et à l'espionnage au profit de gouvernements étrangers. Mais l'espionnage peut aussi avoir des raisons plus triviales, comme le profit financier, qu'il s'agisse d'espionnage industriel ou international. Toujours pour s'enrichir malhonnêtement, l'informatique est aujourd'hui l'un des moyens modernes employés pour commettre crimes et délits, comme le vol, la fraude, le chantage ou l'extorsion de fonds. Dans le même ordre d'idée, il est possible que certains virus ou chevaux de Troie aient été créés, ou seront créés, par des entreprises pour lutter contre la concurrence en provoquant la méfiance vis-à-vis de produits d'autres entreprises ou de ces logiciels plus ou moins gratuits, disponibles sur les réseaux ou sur des serveurs. Enfin, il n'est pas impossible que certaines attaques aient été délibérément commises par des agences gouvernementales pour faire admettre par des utilisateurs négligents de nouvelles réglementations ou restrictions aux communications.

3.1. Classification des attaques contre la sécurité

3.1.1. Ecoute passive

L'écoute passive est un type d'attaque contre la confidentialité qui consiste à accéder sans modification aux informations qui sont générées, transmises, stockées ou affichées dans des composants vulnérables du système informatique :

- voies de communications : directement par accès physique au médium de communication ou indirectement par analyse du rayonnement émis,
- mémoires ou disques : par des accès en lecture,
- périphériques tels que les écrans de terminaux, les claviers ou les imprimantes : par analyse du rayonnement.

En fait, tout matériel informatique émet un rayonnement électromagnétique qui est transmis soit par ondes hertziennes, soit par induction dans des matériaux conducteurs (alimentation électrique, circuits de refroidissement, lignes téléphoniques, etc.). Pour se prémunir contre les écoutes par analyse de rayonnement, il faut soit isoler le système informatique (par exemple dans une cage de Faraday), soit utiliser des équipements conçus pour limiter le rayonnement (matériels "*TEMPEST*").

3.1.2. Interception

L'interception est un accès avec modification des informations transmises sur des voies de communication. Il s'agit donc d'une attaque contre l'intégrité. Les trois types d'interception sont :

- la destruction de messages
- la modification de messages
- l'insertion de messages (l'attaquant peut par exemple *rejouer* des séquences de messages qu'il aura "écoutés" précédemment)

3.1.3. Répudiation

La répudiation consiste, pour un utilisateur, à refuser de reconnaître une opération qu'il a effectuée. Ceci a en particulier été défini pour les communications [ISO 88], pour caractériser la répudiation d'émission (l'émetteur d'un message refuse de reconnaître qu'il l'a émis) et la répudiation de réception (le récepteur d'un message refuse de reconnaître qu'il l'a reçu). Cet aspect est très important pour les applications bancaires comme le transfert électronique de fonds.

3.1.4. Cryptanalyse

Quand des informations sensibles doivent être stockées ou transmises par des moyens considérés comme vulnérables à une écoute passive ou à une interception, des techniques de chiffrement sont généralement utilisées. Le chiffrement consiste à transformer des informations en clair (appelées *texte clair*) en un texte chiffré (appelé *cryptogramme*) à l'aide d'une clé de chiffrement maintenue secrète. L'opération inverse est le déchiffrement. La cryptanalyse consiste, pour un attaquant, à obtenir des informations secrètes (texte clair, clés, algorithme de chiffrement), à partir d'informations non-secrètes (cryptogramme).

3.1.5. Déduction par inférence

La déduction par inférence consiste, pour un intrus, à recouper des informations auxquelles il a légitimement accès pour en déduire des informations confidentielles auxquelles il ne devrait pas avoir accès. Ce type d'attaque vise principalement les bases de données pour lesquelles, par exemple, certains utilisateurs n'ont accès qu'à des requêtes statistiques et non pas à des informations individuelles. Cette technique s'apparente à celle du *furetage* qui consiste à parcourir l'ensemble des informations auxquelles il est possible d'accéder pour découvrir des informations confidentielles.

3.1.6. Déguisement

Le déguisement (ou mascarade) consiste à tromper les mécanismes d'authentification pour se faire passer pour un utilisateur autorisé (personne ou service) et ainsi obtenir des droits d'accès anormaux pour compromettre la confidentialité, l'intégrité ou la disponibilité.

3.1.7. Utilisation de canaux couverts

Les *canaux couverts* sont utilisés pour transmettre, en contournant les contrôles d'accès, des informations entre un utilisateur autorisé à accéder à ces informations et un autre utilisateur non-autorisé, en particulier dans les systèmes de sécurité par mandats (cf. § 3.2.3.1). Il s'agit donc d'attaques contre la confidentialité. Il faut distinguer les canaux de mémoire (réutilisation de tampons ou de fichiers temporaires) et les canaux temporels (modulation de l'utilisation de ressources communes : unité centrale, disque, imprimante, etc.). Plutôt que d'éliminer complètement ces canaux couverts, ce qui est pratiquement impossible si des ressources sont partagées, le système de sécurité aura pour objectif d'en réduire la bande passante à des niveaux inutilisables.

3.1.8. Porte dérobée

Une porte dérobée ("*trap-door*" en anglais) est un moyen de contourner les contrôles d'accès. Il s'agit d'une faille du système de sécurité due à une faute de conception accidentelle ou intentionnelle.

Exemple de porte dérobée : "Le nid du coucou" ***[Stoll 88] [Stoll 89]***

En août 1986, une erreur de quelques cents est apparue dans la comptabilité d'un des ordinateurs du centre de calcul du Lawrence Berkeley Laboratory. Plus pour apprendre comment fonctionnait ce système de comptabilité qu'à cause du préjudice, Clifford Stoll s'est penché sur ce problème et a rapidement découvert que quelqu'un s'était introduit dans le système en utilisant le compte d'un utilisateur autorisé (*déguisement*), puis avait acquis les privilèges système (super-utilisateur) en exploitant une faille connue de Gnu-emacs sur Unix-BSD (*porte dérobée*).

Plutôt que de corriger le système pour empêcher l'intrus de renouveler ce type d'attaque, il a été décidé (compte-tenu de l'absence de donnée classifiée sur ce système) d'observer les intrusions pour tenter d'identifier l'individu qui en était responsable et de surveiller les dégâts qu'il pourrait commettre.

En dix mois, il a tenté de pénétrer, par l'intermédiaire du calculateur du LBL, dans 450 machines, principalement des ordinateurs militaires connectés sur le réseau MILNET. Il a réussi à s'introduire dans 30 de ces machines (par déguisement ou par des portes dérobées, le plus souvent grâce à la négligence des utilisateurs). Il essayait alors d'obtenir par *furetage* des informations sensibles sur le nucléaire, sur l'Initiative de Défense Stratégique (SDI ou "guerre des étoiles"), etc.

Grâce à une coopération internationale, il a été possible de localiser et d'identifier l'intrus : il s'agissait d'un allemand d'Hanovre, lié au Chaos Computer Club, et travaillant pour le KGB. Il a été condamné le 15 février 1990 à 20 mois de prison avec sursis et 10.000 DM d'amende.

3.1.9. Bombe logique

Une bombe logique est une fonction (dévastatrice) déclenchée à retardement (à une date et heure prédéterminées) ou déclenchée par certaines conditions spécifiques comme la présence de certains utilisateurs, de certains logiciels ou matériels, ou après un nombre donné d'activations, etc. Une bombe logique est donc une faute intentionnelle de conception qui s'apparente à un sabotage.

Même si elle n'est mise en œuvre que par logiciel, une bombe logique peut provoquer des dégâts importants :

- destruction d'informations stockées : données, programmes, informations de contrôle et de sécurité, etc. ; ces informations peuvent être détruites, par exemple, en réinitialisant les disques ;
- diffusion d'informations de diagnostic fausses pour entraîner le remplacement par la maintenance de composants sains ou pour mettre en défaut les mécanismes de tolérance aux fautes ;
- dégâts matériels : il est facile de provoquer, par logiciel, une usure anormale de périphériques mécaniques ou d'activer certains matériels dans des configurations dangereuses ; ainsi sur les premiers IBM-PC, il était possible de détruire l'écran monochrome par logiciel en bloquant le balayage électronique qui était généré par le coupleur ; on cite également le cas de destruction d'imprimantes à chaîne dont l'avance papier était bloquée par un programme qui en même temps faisait imprimer en permanence certaines séquences de caractères pour provoquer le déchirement du papier, mais aussi un échauffement anormal pouvant aller jusqu'à un incendie.

3.1.10. Cheval de Troie

Un cheval de Troie est un programme effectuant une fonction illicite tout en donnant l'apparence d'effectuer une fonction légitime. La fonction illicite peut être de divulguer ou de modifier des informations (attaque contre la confidentialité ou l'intégrité), ou peut être une bombe logique. Un cheval de Troie peut être vu comme un cas particulier de déguisement, dans le sens où la fonction illicite s'exécute avec les droits qui ont été accordés au programme légitime.

Exemple de cheval de Troie : la disquette "AIDS" ***[Solomon 89]***

Le 11 décembre 1989, environ 10.000 pochettes publicitaires ont été adressées par la société "*PC-Cyborg*" à 7.000 abonnés du magazine britannique "*PC Business World*" et à 3.000 participants européens d'une conférence de l'Organisation Mondiale de la Santé consacrée au SIDA en octobre 1988 à Stockholm.

Chaque pochette contenait une disquette, le texte d'une "licence", et un mode d'emploi pour l'installation sur le disque dur d'un PC d'un logiciel contenu dans la disquette. L'objet prétendu de ce logiciel était d'évaluer le risque d'être atteint par le SIDA en fonction de réponses à un questionnaire.

En fait, au quatre-vingt-dixième redémarrage du micro-ordinateur après l'installation du logiciel, une *bombe logique* était lancée, provoquant le chiffrement des noms de fichiers sur le disque, ce qui rend le système inutilisable, ainsi que l'édition d'un bon de commande pour payer la licence du logiciel à PC-Cyborg au Panama. Il s'agit donc d'une extorsion de fonds.

Cette tentative n'a sans doute pas été fructueuse pour ses auteurs, puisque dès le 13 décembre, grâce à des groupes d'utilisateurs de PC, de nombreuses informations de mise en garde ont été publiées dans les journaux et la télévision ainsi que dans les Unix-news.

Le premier jour de février 1990, le Docteur Joseph Lewis Popp, ancien employé de l'OMS, a été arrêté par le FBI dans l'Ohio, suite à une demande d'extradition britannique pour "chantage".

3.1.11. Virus

Un virus est un programme qui, lorsqu'il s'exécute, se propage et se reproduit pour s'adjoindre à un autre programme (système ou d'application), qui devient ainsi un cheval de Troie. Un virus peut être également porteur d'une bombe logique. Les virus se propagent d'un système à un autre par échange de support de fichiers (disquettes) ou par réseau.

Il faut noter que, pour s'exécuter, les virus doivent s'attacher à des programmes d'application ou à des programmes du système (dont le "*boot*"). Ils ne s'attachent pas, en principe aux fichiers de données. Il peut néanmoins exister un certain nombre d'exceptions :

- Sur Macintosh, les fichiers de données peuvent posséder des sections exécutables dans le "*resource fork*", pour l'initialisation des fenêtres, le lancement d'applications, etc. Un virus peut donc être exécuté s'il s'attache à ces sections exécutables. C'est la cas du virus WDEF qui infecte le fichier de données systèmes "*Desktop*".

- Dans certaines applications, des commandes peuvent être enregistrées comme des données : c'est le cas des formules de calcul dans certains tableurs ou bases de données. Ce que l'utilisateur pense être un fichier de données peut donc contenir des parties exécutables, susceptibles d'être contaminées par des virus (spécifiques).

Jusqu'à présent, la plupart des virus ont été conçus pour des micro-ordinateurs. Ceci tient en particulier à l'absence de mécanismes de protection sur ce type de matériel, ainsi qu'à la négligence de la plupart de leurs utilisateurs et à l'immoralité de quelques uns d'entre eux. Cependant, Fred Cohen [Cohen 87] a montré que de tels virus sont faciles à réaliser pour des systèmes informatiques plus importants et bien protégés, et qu'ils peuvent provoquer des dégâts beaucoup plus importants que pour des micro-ordinateurs. Le danger est d'autant plus grand que le développement actuel des systèmes répartis facilite la contamination par les réseaux de communication, pour les virus comme pour les vers.

Exemples de virus

Les premières expériences connues de production de virus informatique semblent dater de décembre 1981 où quelques étudiants d'une Université du Texas, fanatiques de l'Apple II, ont développé plusieurs versions d'un virus qui se propageait de disquette en disquette par modification du DOS 3.3. Ce virus était conçu pour être invisible, il était donc bénin (pas de bombe logique, pas d'effet de bord autre que la propagation).

De novembre 1983 à août 1984, alors qu'il préparait son PhD [Cohen 86], Fred Cohen a conçu et expérimenté des virus sur divers types de machines pour démontrer l'inefficacité des mécanismes de protection face à ce type d'attaques, même dans le cas de sécurité multi-niveaux (cf. § 3.2.3.1.) [Cohen 87].

En 1986, une compagnie pakistanaise de distribution de logiciels pour PC a eu l'idée d'utiliser un virus, sinon comme moyen de protection, au moins comme un moyen de suivre le piratage qui était fait de leurs logiciels ; c'est le virus "*Brain*" (du nom de la compagnie), qui se propage par modification du secteur de "*boot*" des disquettes systèmes. Selon de récentes estimations, ce virus aurait contaminé entre 100.000 et 500.000 IBM-PC et compatibles, sur un parc estimé à 30.000.000 d'unités. Il est généralement bénin.

En décembre 1987, un nouveau virus pour PC a été découvert en Israël. Ce virus s'attachait à tout programme exécutable (en augmentant sa taille). Par suite d'une faute de conception, ce virus pouvait réinfecter indéfiniment les mêmes programmes, dont la taille augmentait jusqu'à saturation de l'espace allouable, ce qui a permis sa détection. En fait, le virus était porteur d'une bombe logique : chaque vendredi 13, il détruisait tous les fichiers exécutables. Comme le vendredi 13 suivant devait être le 13 mai 1988, veille du quarantième anniversaire de la République d'Israël, certains ont émis l'hypothèse d'un acte de terrorisme. Selon les mêmes estimations que pour le virus "*Brain*", ce virus aurait lui aussi infecté entre 100.000 et 500.000 IBM-PC et compatibles.

On dénombre actuellement plusieurs centaines de virus différents pour PC. Les autres types de micro-ordinateurs ont eux aussi leurs virus : on connaît plusieurs dizaines de virus différents pour Macintosh, autant pour Atari, Amstrad, Apple II, etc.

3.1.12. Ver

Un ver ("*worm*" en anglais) est un programme autonome qui se propage sur les réseaux et se reproduit à l'insu des utilisateurs normaux. Il peut lui aussi être porteur d'une bombe logique.

Exemples de vers

Les premiers *vers*[6] ont été conçus par des chercheurs de Xerox à la fin des années 70 [Shoch 82]. Il s'agissait de programmes *utiles* composés de segments qui se copiaient sur les machines inactives du réseau et se détruisaient lorsqu'elles redevenaient actives. Ces programmes pouvaient servir à tester les machines, à mesurer les performances ou à des applications parallèles comme le traitement d'image. Il est arrivé que ces programmes provoquent accidentellement des dénis de service [Cohen 87].

Un autre exemple de ver est celui qui a provoqué la saturation du réseau VNET d'IBM en décembre 1987 [Kurzban 90]. Un étudiant allemand a envoyé à l'un de ses collègues un message sur le réseau EARN contenant un fichier exécutable (CHRISTMA EXEC). Lorsque le destinataire lisait ce message dans sa boîte-aux-lettres, le programme était exécuté : il affichait sur l'écran une carte de vœux, mais en même temps une copie de ce ficher était envoyée à tous les correspondants habituels de celui qui exécutait le programme : la propagation était donc exponentielle. Le message s'est rapidement propagé de EARN à BITNET et à VNET, qui plus rapide que BITNET, a vite été saturé.

Mais l'exemple de ver le plus notable est le ver de Robert T. Morris Jr[7] [Eichin 89] [Spafford 88]. Cet étudiant de Cornell a lancé le 2 novembre 1988 vers 17h. un programme qui au cours de la nuit a contaminé et saturé environ 6.000 des 60.000 ordinateurs connectés sur Internet aux Etats-Unis, ce qui a rapidement bloqué l'ensemble du réseau. En fait, il semble qu'une faute de conception soit à l'origine de ce blocage : le ver se reproduisait beaucoup plus rapidement que prévu. Il est probable que si la vitesse de reproduction du ver avait été beaucoup plus faible, il n'aurait probablement pas été détecté rapidement, puisqu'il ne comportait aucune bombe logique. Le ver utilisait trois failles connues d'Unix-BSD (fingerd, sendmail et attaque par dictionnaire des mots-de-passe), tous les ordinateurs contaminés étaient des Sun3 avec SunOS (variante d'Unix-BSD) ou des Vax avec Unix-BSD. Robert T. Morris a été jugé coupable et condamné le 4 mai 1990 à 10.000 dollars d'amende et 400 heures de travaux d'intérêt général. Une conséquence bénéfique de cette attaque a été la création le 13 décembre 1988 du Computer Emergency Response Team (CERT) qui coordonne les actions pour renforcer la sécurité des grands réseaux américains.

En octobre 1989, un autre ver, le "*WANK*", s'est propagé sur le réseau DECNET en contaminant le système VMS.

3.2. Contrôles d'accès et protection

La sécurité des systèmes informatiques repose principalement sur les contrôles d'accès qui ont pour premier objectif d'empêcher les intrusions : c'est donc un moyen de *prévention des fautes*. Les *intrusions* sont des fautes d'interaction intentionnelles qui peuvent être des attaques externes, c'est-à-dire venant d'individus qui ne sont pas des

[6] Ils ont été appelés "*worms*" d'après un roman de science-fiction de John Brunner [Brunner 82], où un programme "immortel" était lancé sur un réseau d'ordinateurs pour libérer le monde de l'emprise de l'informatique.

[7] Robert T. Morris Jr. est le fils de Robert H. Morris, "*Chief Scientist*" du "*National Computer Security Center*", bien connu entre autres pour ses travaux sur la sécurité d'Unix [Morris 79] [Grampp 84].

utilisateurs enregistrés du système informatique ; mais le plus souvent, les intrusions sont le fait d'utilisateurs malintentionnés quoique régulièrement enregistrés par le système. Le terme d'intrusion est donc à prendre dans un sens large : utilisation anormale délibérée du système informatique.

Pour empêcher les intrusions, les contrôles d'accès visent à limiter les possibilités de s'introduire dans le système informatique et à réduire les possibilités d'opération à ce qui est indispensable ; c'est le *principe du moindre privilège*, qui peut se formuler de la façon suivante : "un utilisateur ne doit pouvoir accéder à un instant donné qu'aux informations et services strictement nécessaires pour l'accomplissement du travail qui lui a été confié". Dans le cas des systèmes où la confidentialité est l'objectif principal, comme certains systèmes militaires ou gouvernementaux, un principe équivalent prévaut : le *besoin d'en connaître* ("*need-to-know*", en anglais) : un individu ne pourra lire que les informations qu'il a besoin de connaître pour accomplir correctement sa tâche.

Les contrôles d'accès se classent en deux catégories :

- les contrôles d'accès administratifs et physiques, externes au système informatique,
- les contrôles d'accès logiques, c'est-à-dire réalisés par le système informatique, qui eux-mêmes se décomposent en *authentification* et *autorisation*.

3.2.1. Contrôles d'accès administratifs et physiques

Les contrôles d'accès administratifs et physiques sont un ensemble de contrôles externes au système informatique. Ils sont l'objet de ce qu'on appelle généralement la sécurité du personnel, la sécurité physique et la sécurité procédurale.

- La *sécurité du personnel* consiste à déterminer les personnes qui sont dignes de confiance pour certaines opérations, et donc à qui pourra être confiée l'autorisation d'exécuter ces opérations. Des autorisations différentes seront accordées selon les personnes, soit en fonction de degrés de confiance différents, soit en fonction de tâches différentes.
- La *sécurité physique* est l'ensemble des moyens qui limitent l'accès physique au système informatique : verrouillage des portes, gardiens, etc. Dans le cas des systèmes répartis, la sécurité physique est de moins en moins efficace, en raison du grand nombre de points d'accès possible (stations et réseau) et en raison des interconnexions avec des réseaux à large échelle.
- La *sécurité procédurale* est l'ensemble de règles administratives concernant l'accès physique aux machines, la manipulation physiques des entrées-sorties (sorties d'imprimantes, sauvegardes, etc.), l'installation de nouveaux logiciels, le raccordement de nouveaux terminaux, la maintenance, etc. Ces règles, contrôlées par les mécanismes de sécurité physique ou par le personnel mais non par le système informatique lui-même, servent à compléter les contrôles d'accès logiques, voire à en suppléer les insuffisances.

3.2.2. Authentification

L'authentification est le premier des contrôles d'accès logiques. Elle comprend l'*identification* des utilisateurs[8] et la *vérification* de cette identification.

L'identification est la présentation par l'utilisateur de son *identité* qui est une information non secrète, différente pour chaque utilisateur, et qui est associée à l'utilisateur lors de son enregistrement sur le système informatique ; cela peut être un nom, un numéro, ou toute autre information connue (au moins) par l'utilisateur et par le système.

La vérification consiste à s'assurer que l'utilisateur est bien celui dont il présente l'identité, au moyen de :

- quelque chose *que connaît* l'utilisateur (par exemple un mot-de-passe),
- quelque chose *qu'il possède* (badge, carte magnétique, carte à puce, etc.),
- quelque chose *qui lui est propre* (empreinte digitale, voix, etc.),
- quelque chose *qu'il sait faire* (par exemple une signature).

La qualité des dispositifs de vérification de l'identité se juge par le taux d'acceptation à tort (le système accepte de reconnaître l'utilisateur, alors qu'il s'agit d'un autre individu), mais aussi par le taux de rejet à tort (refus de reconnaître l'utilisateur quand c'est bien lui qui se présente). Les systèmes de reconnaissance biométrique (analyse d'empreinte digitale, reconnaissance vocale, analyse du fond de l'œil, reconnaissance dynamique de signature) sont généralement les plus efficaces parce qu'ils ne sont pas aussi facilement transmissibles qu'un badge ni aussi facilement copiables qu'un mot-de-passe ; ils ont cependant l'inconvénient d'être coûteux et parfois mal acceptés par les utilisateurs : par exemple, le fichage des empreintes digitales est souvent considéré comme une atteinte à la liberté individuelle et l'analyse du fond de l'œil peut faire peur ("qu'est-ce-que je risque si la machine tombe en panne ?"). Pour augmenter l'efficacité, il est souhaitable d'utiliser une combinaison de plusieurs dispositifs : par exemple, carte à puce et numéro d'identification personnel ("*P.I.N. : personal identification number*").

La vérification de l'identité est basée :

- sur un secret partagé par l'authentificateur[9] et l'utilisateur (par exemple, un mot-de-passe),
- sur un secret maintenu par l'authentificateur, caractéristique de l'utilisateur (par exemple, les informations condensées de reconnaissance vocale),

---

[8] Le terme *utilisateur* est à prendre dans un sens large : il peut s'agir d'une personne physique ou d'un service (d'archivage, d'impression, de calcul, etc.) qui peut être considéré comme l'équivalent d'une *personne morale*.

[9] Nous utilisons le terme "authentificateur" pour désigner le processus, le processeur ou l'extrémité de communication qui réalise la vérification de l'identité, et le terme "authentifieur" pour désigner l'information qui permet de prouver l'identité.

- sur un secret maintenu par l'utilisateur et sur des informations publiques : protocoles d'authentification *sans apport de connaissance*.

La technique des mots-de-passe est encore la plus courante aujourd'hui, même si elle présente de multiples inconvénients, au nombre desquels il faut compter la facilité qu'a un utilisateur de transmettre volontairement son mot-de-passe à quelqu'un d'autre qui pourra ainsi se faire authentifier à tort sans difficulté. Un problème majeur est la mémorisation des mots-de-passe par l'authentificateur : si les mots-de-passe sont enregistrés en clair dans un fichier, la confidentialité de ce fichier est particulièrement sensible puisqu'un intrus qui réussirait à lire ce fichier pourrait se faire passer pour n'importe quel utilisateur. Dans Unix, les mots-de-passe sont enregistrés dans le fichier "*/etc/passwd"* sous forme chiffrée par une fonction à sens unique (c'est-à-dire non-inversible) [Grampp 84]. Au "*login*", le mot-de-passe de l'utilisateur est chiffré par la même fonction à sens unique et il est comparé sous cette forme chiffrée au mot-de-passe chiffré enregistré dans "*/etc/passwd"*. Ceux qui ont conçu cette fonction à sens unique ont tellement confiance dans son efficacité que, dans les Unix classiques, le fichier "*/etc/passwd"* peut être lu par tout utilisateur ! Mais cette technique est vulnérable aux attaques par dictionnaire : un intrus peut chiffrer par la fonction à sens unique un dictionnaire des mots-de-passe usuels et les comparer avec le fichier "*/etc/passwd"*. Selon certaines statistiques un dictionnaire de 3000 mots permet de deviner de 8 à 30% des mots-de-passe choisis par des utilisateurs négligents ; cette technique d'attaque a été utilisée dans le cas du "Nid de coucou" (cf. §3.1.8) et dans le ver de R. Morris (cf. §3.1.12). Dans les Unix sécurisés, les mots-de-passe chiffrés sont mémorisés dans un fichier non accessible par les utilisateurs normaux.

Un autre inconvénient majeur de l'authentification par mot-de-passe, en particulier dans les systèmes répartis, est le risque d'interception du mot-de-passe lorsqu'il est transmis en clair soit sur des lignes asynchrones d'un terminal à un ordinateur, soit sur un réseau. Dans ce cas, il faut mettre en œuvre des protocoles d'authentification utilisant des techniques de chiffrement. Ces protocoles sont souvent basés sur ceux proposés par Needham et Schroeder [Needham 78], utilisant soit des algorithmes de chiffrement symétriques soit des algorithmes de chiffrement à clé publique.

3.2.2.1. Authentification de Needham-Schroeder par chiffre symétrique

Nous utiliserons la notation suivante : si M est un message en clair, $C = \{M\}_K$ est le cryptogramme correspondant au message M chiffré par la clé K, et $M = [C]_{K'}$ est le même message M obtenu après déchiffrement par la clé K'.

Un chiffre symétrique est un algorithme de chiffrement qui utilise la même clé pour le chiffrement et le déchiffrement : $M = [\{M\}_K]_K$. Le DES ("*Data Encryption Standard*") est un exemple de chiffre symétrique dont l'algorithme a été publié et normalisé.

Soient deux entités A et B d'un système réparti, A voulant se faire identifier par B. Soit SA un serveur d'authentification connu de A et B. Pour chaque entité E du système réparti, SA conserve une clé secrète K_e connue uniquement de E et de SA.

Pour empêcher qu'un intrus en écoute passive puisse rejouer une *transaction* (c'est-à-dire un ensemble de messages échangés selon un protocole donné), pour chaque transaction *t* on génère un identificateur unique I_t (I_t peut être généré aléatoirement).

Le protocole d'établissement d'une session entre A et B comprend 5 étapes :

① A envoie à SA son identité *a*, l'identité *b* de B, et l'identificateur de transaction I_1

② SA génère (aléatoirement) une clé de session K_{ab} et envoie à A un message M_1 : $M_1 = \{I_1, b, K_{ab}, T_{ab}\}_{K_a}$ où $T_{ab} = \{K_{ab}, a\}_{K_b}$

③ A déchiffre M_1 à l'aide de sa clé secrète K_a, mémorise K_{ab} et envoie à B le message $M_2 = T_{ab}$

④ B déchiffre M_2 à l'aide de sa clé secrète K_b et mémorise K_{ab} ; pour s'assurer que M_2 n'est pas le rejeu d'une ancienne transaction, il envoie à A un identificateur I_2 chiffré par K_{ab} : $M_3 = \{I_2\}_{K_{ab}}$

⑤ A déchiffre M_3 et renvoie à B le message $M_4 = \{I_2\text{-}1\}_{K_{ab}}$; cette étape correspond à la réponse au défi envoyé par B à l'étape précédente : A prouve ainsi qu'il connaît K_{ab}

A l'issue de ces 5 étapes, A et B possèdent en commun la clé de session K_{ab} et s'ils font confiance à SA, ils sont sûrs de l'identité de l'autre partenaire.

Exemple : Kerberos [Steiner 88]

Kerberos est le service d'authentification d'Athena, le système réparti développé par le MIT pour interconnecter les milliers de stations de travail du campus. Compte tenu de cet environnement, il n'est pas possible de faire confiance aux mécanismes d'authentification inclus dans les stations de travail, et pourtant, il n'est pas possible d'utiliser des matériels spécifiques (pour des raisons de coût) et il faut rester compatible avec les normes de fait utilisées dans ce système : procédure de "*login*" compatible avec celle d'Unix, protocole TCP/IP, NFS, etc.

Ce qu'il faut sécuriser, c'est la relation client-serveur ; ceci est réalisé au moyen d'une clé de session qui sera utilisée pour chiffrer (par DES) tous les messages entre le client et le serveur, cette clé ne devant être connue que du client et du serveur, et valide uniquement pendant la durée de la session. Le client C est une station de travail quelconque sur laquelle se connecte un utilisateur U. Le serveur S ne fait pas confiance à la station de travail, mais connaît d'autres serveurs dans lesquels il a confiance.

Les protocoles utilisent deux types de certificats : les tickets $T_{c,s}$ et les authentifieurs $A_{c,s}(t)$.

Un ticket $T_{c,s}$ est caractéristique d'une session entre C et S, et il est valable pendant toute la durée de la session. $T_{c,s}$ contient l'identité *s* du serveur, l'identité *c* du client, l'adresse-IP *adr* du client, l'instant t_d de début de la session, la durée *life* maximale pour la session et la clé de session $K_{c,s}$, le tout chiffré par la clé permanente K_s du serveur :

$$T_{c,s} = \{s, c, adr, t_d, life, K_{c,s}\}_{K_s}$$

L'authentifieur $A_{c,s}(t)$ est un certificat généré à l'instant t par le client C pour se faire reconnaître par le serveur S. $A_{c,s}(t)$ contient l'identité *c* du client, *adr* son adresse-IP et l'instant *t*, le tout chiffré par la clé de session $K_{c,s}$:

$$A_{c,s}(t) = \{c,adr,t\}_{K_{c,s}}$$

L'établissement d'une session entre le client C où l'utilisateur U se connecte et le serveur S nécessite 5 étapes :

① Le client C transmet au serveur Kerberos K l'identité *u* de l'utilisateur U qui vient de lancer son "*login*" sur C. K recherche dans sa base de données le mot-de-passe *mdp* (en clair) enregistré pour *u* ; *mdp* servira de clé pour chiffrer le message renvoyé par K à C à l'étape ②. K génère alors une valeur aléatoire $K_{c,st}$ qui servira de clé de session entre C et le serveur de tickets ST. Comme K connaît la clé permanente K_{st} de ST, K peut générer le ticket $T_{c,st} = \{st,c,adr,t_d,life,K_{c,st}\}_{K_{st}}$.

② K envoie à C un message M contenant la clé de session $K_{c,st}$ et le ticket $T_{c,st}$, le tout chiffré par *mdp* : $M = \{K_{c,st},T_{c,st}\}_{mdp}$. C déchiffre M à l'aide du mot-de-passe fourni en clair par l'utilisateur U au moment du "*login*", et mémorise $K_{c,st}$ et $T_{c,st}$.

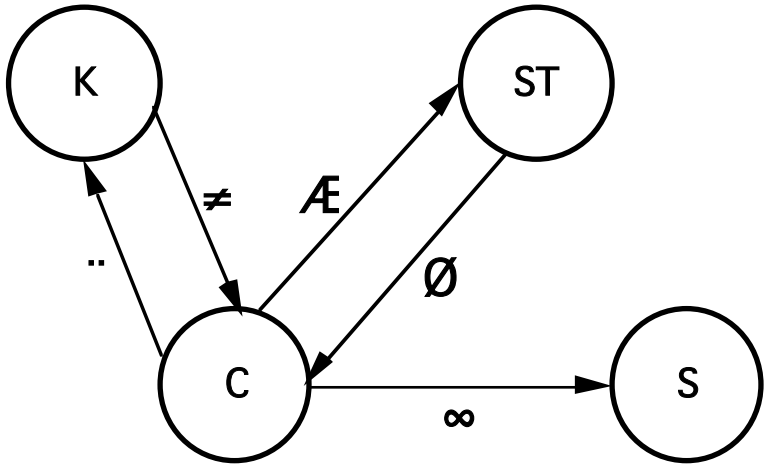


Figure 9.5 : Etablissement d'une session client-serveur

③ Lorsque C veut établir une session avec S, C envoie à ST un message contenant l'identité *s* du serveur S auquel il veut accéder, le ticket $T_{c,st}$ et un authentifieur $A_{c,st}(t)$. ST déchiffre $T_{c,st}$ à l'aide de sa clé permanente K_{st}, vérifie la validité de s,c,adr,t_d et *life* et utilise $K_{c,st}$ pour déchiffrer $A_{c,st}(t)$ dont il vérifie la validité. Si tous ces paramètres sont valides, il génère une valeur aléatoire $K_{c,s}$ qui servira de clé de session entre C et S, et génère un ticket $T_{c,s}$ (ST connaît la clé permanente K_s de S).

④ ST envoie à C un message M' contenant la clé de session $K_{c,s}$ et le ticket $T_{c,s}$, le tout chiffré par $K_{c,st}$: $M' = \{K_{c,s},T_{c,s}\}_{K_{c,st}}$. C déchiffre M' à l'aide de $K_{c,st}$ et mémorise la clé de session $K_{c,s}$ et le ticket $T_{c,s}$.

⑤ C envoie à S sa requête avec le ticket $T_{c,s}$ et un authentifieur $A_{c,s}(t)$. S déchiffre $T_{c,s}$ à l'aide de sa clé permanente K_s, vérifie la validité de s,c,adr,t_d et *life* et utilise $K_{c,s}$ pour déchiffrer $A_{c,s}(t)$ dont il vérifie la validité. Si tous ces paramètres sont valides, il accepte la requête de C.

Ce protocole ne nécessite pas de matériel spécifique, mais une version câblée du DES permet de l'accélérer. Il ne nécessite pas de confiance dans la station de travail (sauf pendant la phase de "*login*" où il faut que le mot-de-passe en clair ne soit pas mémorisé par un cheval de Troie).

En revanche, il nécessite que les serveurs S fassent confiance au serveur de tickets ST (partage d'une clé permanente K_S) et que ST fasse confiance à K (partage d'une clé permanente K_{St}). Il faut également faire confiance à l'administrateur et aux opérateurs de K, qui est un site

particulièrement sensible pour la sécurité du système, puisque K possède dans une base de données tous les mots-de-passe en clair des utilisateurs !

3.2.2.2. Authentification de Needham-Schroeder par chiffre à clé publique

Un chiffre à clé publique est un algorithme de chiffrement qui utilise des clés différentes pour le chiffrement et le déchiffrement : $M = [\{M\}_{KP}]_{KS} = [\{M\}_{KS}]_{KP}$; KP est publiée et KS est gardée secrète. L'algorithme RSA [Rivest 78] est le plus connu des chiffres à clé publique[10].

Chaque entité E connaît KP_s la clé publique du serveur d'authentification SA et conserve secrète KS_e ; SA connaît toutes les KP_e.

[10] De plus, dans le cas du RSA, les algorithmes de chiffrement et de déchiffrement sont identiques : $M = \{\{M\}_{KP}\}_{KS} = \{\{M\}_{KS}\}_{KP}$

Le protocole d'établissement d'une session entre A et B comprend 7 étapes :

① A envoie à SA son identité *a* et l'identité *b* de B

② SA envoie à A la clé publique de B dans un message $M_1 = \{KP_b,b\}_{KS_s}$. Le fait de chiffrer avec KS_s ne sert pas à la confidentialité (tout composant connaît KP_s), mais sert à prouver l'identité de SA

③ A déchiffre M_1 à l'aide de la clé publique KP_s de SA et mémorise KP_b ; il envoie à B le message $M_2=\{I_1,a\}_{KP_b}$

④ B déchiffre M_2 à l'aide de sa clé secrète KS_b ; il envoie alors à SA son identité *b* et l'identité *a* de A pour obtenir la clé publique de A

⑤ SA renvoie à B la clé publique de A dans un message $M_3 = \{KP_a,a\}_{KS_s}$

⑥ B envoie à A un message M_4 prouvant qu'il est bien B : $M_4 = \{I_1, I_2\}_{KP_a}$

⑦ A renvoie à B un message M_5 prouvant qu'il est bien A : $M_5 = \{I_2\}_{KP_b}$

Ce protocole exige donc 7 étapes, mais si A et B conservent dans un cache les clés publiques des entités avec lesquelles elles ont communiqué récemment, les étapes ①, ②, ④ et ⑤ peuvent être évitées.

Ce protocole à clé publique est dit *sans apport de connaissance* ("*zero-knowledge protocol*" en anglais), puisque l'authentifié ne fournit aucune information secrète à l'authentificateur, contrairement aux techniques à clé secrète ou à mot-de-passe qui nécessitent le partage d'un secret entre l'authentificateur et l'authentifié. Il existe d'autres protocoles sans apport de connaissance qui ne sont pas basés sur des chiffrements à clé publique, mais sur des couples valeur secrète – valeur publique, pour lesquels il n'est pas possible de calculer la valeur secrète en un temps polynomial en connaissant la valeur publique. L'authentification consiste alors pour l'authentifié à prouver qu'il connaît la valeur secrète, alors que l'authentificateur ne connaît que la valeur publique.

Exemple : Strongbox [Yee 88]

Soit n le produit de deux grands nombres premiers ; n est connu de tous les composants du système, mais sa factorisation est inconnue. Chaque composant C du système génère une valeur aléatoire r_c satisfaisant la relation $1 \le r_c \le n-1$; il conserve secrète la valeur r_c et calcule $x_c = r_c^2 \bmod n$, qu'il publie. Il n'est pas possible de calculer r_c connaissant x_c en un temps polynomial, car Rabin a prouvé que s'il était possible d'extraire la racine carrée modulo n en un temps polynomial, il serait possible de factoriser n en un temps polynomial, ce qui est considéré comme faux actuellement.

Supposons que B veuille authentifier A avec une confiance de $1-2^{-t}$, t pouvant être choisi aussi grand qu'on veut. A génère t valeurs aléatoires v_i, i allant de 1 à t, avec $1 \le v_i \le n-1$. A calcule les carrés v_i^2 modulo n des v_i et les transmets à B. B génère alors une chaîne aléatoire de t bits b_i qu'il transmet à A. Pour chaque i, A transmet à B $z_i = v_i$ si $b_i = 0$ et $z_i = r_a.v_i \bmod n$ si $b_i = 1$. B peut facilement calculer z_i^2 et vérifier que $z_i^2 = v_i^2 \bmod n$ si $b_i = 0$ et que $z_i^2 = x_a.v_i^2 \bmod n$ si $b_i = 1$. Le protocole est sans apport de connaissance puisque B ne peut calculer r_a quelles que soient les valeurs b_i qu'il génère. Un intrus X n'a qu'une chance sur 2^t de réussir à se faire passer pour A s'il ne connaît pas r_a, puisqu'il n'a qu'une chance sur 2^t de deviner à l'avance les b_i pour générer des valeurs v_i^2 qui satisferont les demandes de B.

3.2.3. Autorisation

L'autorisation est le second des contrôles d'accès logiques. Son rôle est de gérer et de vérifier les droits d'accès. L'autorisation est mise en œuvre par les mécanismes de *protection*. Les droits d'accès se définissent de la manière suivante :

Définitions :

- Un *sujet* a un *droit d'accès* sur un *objet* si et seulement si le sujet est autorisé à exécuter la fonction d'accès correspondante sur cet objet. Le sujet est un processus qui s'exécute pour le compte d'un utilisateur. L'objet est n'importe quelle entité du système informatique qui est définie par un nom, un état et des fonctions d'accès (ou opérations, ou méthodes).
- La *protection* consiste à associer des droits aux relations processus-objets et ne permettre que les accès autorisés par ces droits.
- Les règles qui définissent les droits d'accès des sujets sur les objets constituent la *politique d'autorisation* (parfois improprement appelée "politique de sécurité").

La politique d'autorisation doit être basée, dans la mesure du possible, sur un modèle formel, pour permettre de vérifier que la politique est complète et cohérente et que sa mise en œuvre est conforme [ITSEC 91]. Il existe de nombreux modèles formels, dont les plus connus sont :

- le modèle HRU [Harrison 76] qui représente les droits d'accès sous forme d'une matrice et permet une description formelle de leur évolution ;
- le modèle Take-Grant [Jones 76] permet de d'énumérer les modifications possibles de droits et de vérifier s'il est possible d'atteindre des états non-sûrs ;
- le modèle de Bell-LaPadula [Bell 74] pour des politiques multi-niveaux axées sur la confidentialité ;
- le modèle de Biba [Biba 75] qui est équivalent au modèle de Bell-LaPadula pour l'intégrité.

Pour la suite, nous adopterons une notation proche de celle du modèle HRU. Dans cette notation, la configuration (instantanée) de protection du système informatique est définie par un triplet (S,O,A) constitué de trois ensembles : l'ensemble S des sujets courants, l'ensemble O des objets courants et l'ensemble A des droits d'accès courants des sujets sur les objets. Les sujets étant eux-mêmes des objets (processus), on a $S \subseteq O$. L'ensemble A est une matrice de droits d'accès avec une ligne par sujet s_i, et une colonne par objet o_j : $A(s_i, o_j) = A_{ij} = d_{ij}$, où d_{ij} est un sous-ensemble de D l'ensemble de tous les droits possibles. Nous écrirons que la relation (s_i, o_j, d_k) est vraie si et seulement si le sujet s_i a le droit d_k sur l'objet o_j. D'où $d_{ij} = \{d_k \in D \mid (s_i,o_j,d_k)\}$.

3.2.3.1. Politiques d'autorisation

On distingue deux types de politiques d'autorisation : les politiques discrétionnaires et les politiques par mandats.

a) Politique discrétionnaire

Dans le cas d'une politique discrétionnaire, les droits d'accès à chaque information sont manipulés librement par le responsable de l'information (généralement le propriétaire), à sa *discrétion*. Les droits peuvent être accordés à chaque utilisateur individuellement ou à des groupes d'utilisateurs, ou les deux. Ceci peut amener le système dans un état non sûr (c'est-à-dire contraire à la politique d'autorisation qui a été choisie).

Prenons un exemple simple, basé sur les mécanismes de protection d'Unix. Dans un tel système, les droits d'accès à un fichier sont définis et modifiables librement par l'utilisateur propriétaire du fichier (et donc par les *sujets* qui s'exécutent pour son compte). Supposons que la politique d'autorisation se définisse (informellement) de la manière suivante : un utilisateur peut créer des fichiers dont il devient alors propriétaire ; le propriétaire d'un fichier peut décider quels utilisateurs sont autorisés à lire ses fichiers ; les utilisateurs qui n'ont pas le droit de lire un fichier ne doivent pas pouvoir en connaître le contenu. Une telle politique n'est pas réalisable par des mécanismes d'autorisation discrétionnaire :

- si s_1 est un sujet s'exécutant pour le compte de l'utilisateur u_1 propriétaire du fichier f_1, il peut donner au sujet s_2 (s'exécutant pour u_2) le droit de lecture sur f_1 :

  $(s_1, f_1, \text{propriétaire}) \rightarrow (s_2, f_1, \text{lire})$

- s_2 peut créer un fichier f_2 (dans lequel il peut donc écrire) sur lequel il peut donner le droit de lecture à s_3 (s'exécutant pour u_3) :

  $(s_2, f_2, \text{créer}) \rightarrow (s_2, f_2, \text{écrire}) \wedge (s_3, f_2, \text{lire})$

- s_2 peut donc recopier f_1 dans f_2 pour transmettre les informations de f_1 à s_3 à l'insu du propriétaire s_1 :

  $(s_2, f_1, \text{lire}) \wedge (s_2, f_2, \text{écrire}) \wedge (s_3, f_2, \text{lire}) \rightarrow (s_3, k(f_1), \text{lire})$

Une politique discrétionnaire n'est donc applicable que dans la mesure où il est possible de faire totalement confiance aux utilisateurs et aux sujets qui s'exécutent pour leur compte ; une telle politique est donc vulnérable aux *abus de pouvoir* provoqués par maladresse ou par malveillance. Ainsi, s'il est possible à un utilisateur d'accéder à certains objets ou d'en modifier les droits d'accès, il est possible qu'un cheval de Troie s'exécutant pour le compte de cet utilisateur (à son insu) en fasse de même ; de plus, si un utilisateur a le droit de lire une information, il a (en général) le droit de la transmettre à n'importe qui.

b) Politique par mandats

Dans le cadre d'une politique par mandats, des règles incontournables sont imposées (en plus des règles discrétionnaires). L'une des façons de spécifier ces règles est d'imposer une structure hiérarchique pour les sujets et pour les objets ; c'est le cas des politiques *multi-niveaux* basées sur le modèle de la sécurité appliquée par les militaires pour la confidentialité des informations.

Dans une telle politique, les informations et les utilisateurs appartiennent à des *classes* de sécurité prédéfinies et ordonnées : non-classifié, à diffusion restreinte, confidentiel, secret, très secret. A chaque utilisateur, correspond un niveau de sécurité appelé *habilitation*, à chaque information correspond un niveau de sécurité appelé *classification*. Des règles sont imposées qui déterminent quelles habilitations permettent quels accès à des informations de quelles classifications. Ces règles sont incontournables, même par les propriétaires des informations.

Le modèle de Bell-LaPadula [Bell 74] fournit un exemple de telles règles :

- à chaque sujet s_i correspond une habilitation $h(s_i)$
- à chaque objet o_j correspond une classification $c(o_j)$
- *Propriété simple* : $(s_i, o_j, \text{lire}) \Rightarrow h(s_i) \geq c(o_j)$
- *Propriété étoile* : $(s_i, o_j, \text{lire}) \wedge (s_i, o_k, \text{écrire}) \Rightarrow c(o_k) \geq c(o_j)$

Dans ce modèle, la propriété simple interdit de lire des informations d'un niveau de classification supérieur au niveau d'habilitation, et la propriété étoile empêche les flux d'information d'un niveau de classification donné vers un niveau de classification inférieur : on peut vérifier facilement que si $h(s_n) < c(o_i)$, il n'existe pas de suites {i,j,...,k} et {l,m,...,n} telles que :

$$(s_l, o_i, \text{lire}) \wedge (s_l, o_j, \text{écrire}) \wedge (s_m, o_j, \text{lire}) \wedge ... \wedge (s_x, o_k, \text{écrire}) \wedge (s_n, o_k, \text{lire})$$

en effet, ceci conduirait (par la propriété étoile et la propriété simple) à :

$$c(o_i) \leq c(o_j) \leq h(s_m) \leq ... \leq c(o_k) \leq h(s_n) \Rightarrow c(o_i) \leq h(s_n)$$

ce qui est contraire à l'hypothèse de départ.

Il existe d'autres politiques par mandats qui visent à préserver des niveaux d'intégrité plutôt que des niveaux de confidentialité [Biba 75] [Clark 87]. Les politiques par mandats visent donc à empêcher les fuites d'informations (ou les fraudes) *à l'insu* de leur propriétaire par exemple au moyen d'un cheval de Troie ou par accident, mais aussi à empêcher les fuites ou les fraudes provoquées par un utilisateur *malintentionné*. De telles politiques peuvent être mises en œuvre par des mécanismes d'autorisation intégrés au système informatique, mais doivent être également respectées par les mécanismes d'authentification : si on veut empêcher les fuites, il ne faut pas qu'un utilisateur puisse transmettre à un autre des informations d'authentification lui permettant de se faire authentifier à tort ; les authentifications par mot-de-passe sont donc à proscrire.

3.2.3.2. Moniteur de référence et de sous-système de sécurité

Ainsi qu'elle a été définie plus haut, la protection consiste à associer des droits aux relations processus-objets et ne permettre que les accès autorisés par ces droits. Il faut remarquer que, dans ces conditions, la protection empêche les accès malveillants, mais permet aussi de détecter des erreurs dues à des fautes accidentelles matérielles ou de programmation et d'empêcher leur propagation. Les droits d'accès peuvent être vérifiés de façon statique, c'est-à-dire lors de la préparation de programme

(compilation, édition de liens statique, ...) et de façon dynamique, c'est-à-dire à l'exécution (éventuellement lors d'une édition de liens dynamique).

La vérification statique est généralement insuffisante. L'exemple du Burroughs B6700 (ci-dessous) montre qu'il peut exister des portes dérobées permettant de contourner à l'exécution les protections vérifiées lors de la préparation de programme.

Exemple de porte dérobée dans le Burroughs B6700

La protection de ce système reposait sur des compilateurs certifiés, qui étaient responsables de tous les contrôles d'accès et qui, seuls, pouvaient générer des programmes exécutables. Les programmes exécutables étaient des fichiers spéciaux étiquetés de telle sorte qu'ils ne puissent être modifiés. Dans ces conditions, il ne semblait pas nécessaire de renforcer la protection à l'exécution par du matériel spécifique.

Cependant, il était possible, par programme, d'écrire sur bande magnétique un fichier quelconque, y compris un programme exécutable. Il était alors possible, toujours par programme, de ré-écrire sur la bande l'étiquette du fichier, de façon à tromper le contrôle de type de fichier, puis de modifier le fichier sur la bande. On pouvait alors recharger le fichier comme un programme valide.

(Cet exemple a été cité par Carl Landwehr lors d'une conférence à Toulouse en mai 1990)

La protection repose donc généralement sur des vérifications dynamiques réalisées par du matériel et/ou du logiciel spécifique. Pour des raisons de simplicité et de facilité de validation, il est souhaitable de regrouper ces matériel et logiciel réalisant la vérification dynamique des droits d'accès dans un *moniteur de références* [TCSEC 85], qui soit à la fois inviolable, incontournable et prouvé correct. Cette dernière exigence implique que ces mécanismes de protection soient suffisamment *petits* pour qu'il soit possible de les analyser et de les tester exhaustivement.

Le système informatique comporte d'autres fonctions liées à la sécurité, comme l'enregistrement des utilisateurs, l'authentification, la journalisation ("*audit*") des événements de sécurité, etc. Ces fonctions doivent être particulièrement protégées ; c'est pourquoi on les regroupe dans le *sous-système de sécurité* ou TCB ("*Trusted Computing Base*", [TCSEC 85]).

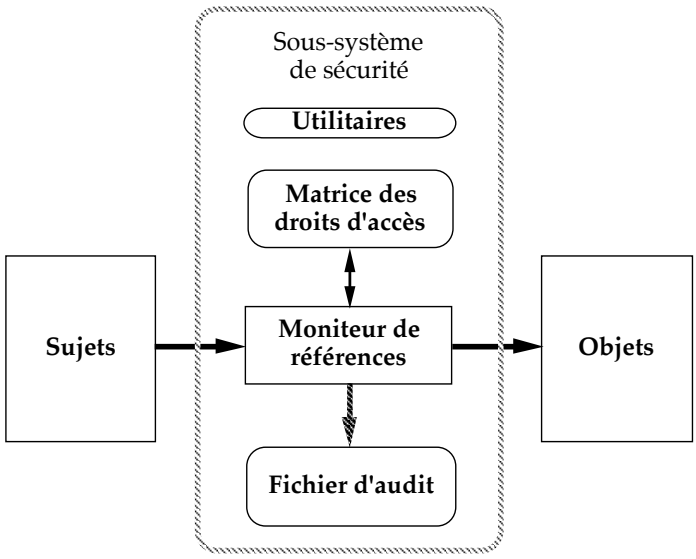


Figure 9.6 : Sous-système de sécurité

3.2.3.3. Gestion de la matrice de droits d'accès

Rappelons que la matrice A des droits d'accès représente la configuration courante de la protection du système. Chaque ligne correspondant à un sujet s_i (c'est-à-dire un processus actif) et chaque colonne correspondant à un objet o_j, la case A_{ij} contient la liste des droits courants du sujet s_i sur l'objet o_j, c'est-à-dire la liste des opérations que le sujet s_i peut exécuter sur l'objet o_j. Cette liste de droits est un sous-ensemble de D, l'ensemble des droits *vérifiables* par le moniteur de références. Ceci définit également la granularité des objets référencés dans la matrice. Par exemple, dans le cas d'Unix, ce niveau de granularité correspond aux fichiers (et aux périphériques), aux répertoires et aux processus puisque seuls sont vérifiés les droits d'accès aux fichiers et aux *signaux* transmis aux processus. Voici un exemple de matrice de droits d'accès :

	Fichier 1	Fichier 2	Imprimante	Processus 1	Processus 2	Processus 3
Processus 1	ouvrir (l,e,x)	lire,écrire		se terminer	tuer	tuer
Processus 2	exécuter		écrire		se terminer	
Processus 3		ouvrir (l,e)				se terminer

Cette matrice évolue dans le temps, en fonction de l'exécution des processus actifs : création d'un nouveau processus, terminaison ou destruction d'un processus, création ou destruction d'un objet, opérations exécutées sur les objets (par exemple, ouverture d'un fichier). A la création d'un processus, la ligne de la matrice est initialisée par le sous-système de sécurité en fonction des droits de l'utilisateur pour le compte duquel

va s'exécuter le nouveau processus. De même, à la création d'un objet, la colonne correspondante de la matrice est initialisée par le sous-système de sécurité avec des droits qui peuvent être définis implicitement par le créateur (par exemple, par la commande "*umask*" d'Unix). A la destruction du processus ou de l'objet, la ligne ou la colonne correspondante est détruite. Les modifications de la matrice en fonction des opérations exécutées peuvent être complexes : elles peuvent affecter soit les droits correspondant à l'opération en cours par le sujet sur l'objet (par exemple, après ouverture en lecture d'un fichier, le processus aura le droit d'effectuer des lectures sur ce fichier), soit les droits des autres sujets sur le même objet (gestion des exclusions, par exemple), soit encore les droits du même sujet sur d'autres objets (un cas particulier de ce type de modification est imposé par le modèle du *Mur Chinois* [Brewer 89], où le fait qu'un utilisateur a pris connaissance de certaines informations lui interdit l'accès à d'autres informations). D'autre part, la matrice de droits est elle-même un objet auquel seul le sous-système de sécurité (moniteur de références et utilitaires) peut avoir accès. Si un sujet a le droit de modifier les droits d'accès à certains objets (par exemple, le propriétaire d'un fichier dans Unix), il doit pour cela transmettre une requête à un utilitaire du sous-système de sécurité ; il ne peut pas accéder directement à la matrice de droits, même dans le cas d'une politique discrétionnaire. Dans tous les cas, les modifications de la matrice de droits d'accès sont faites par le sous-système de sécurité, en respectant la politique d'autorisation.

En général, les droits d'accès ne sont pas gérés directement sous forme d'une matrice complète de droits d'accès, car une telle matrice serait trop importante (nombre de cases = nombre de processus actifs x nombre d'objets présents dans le système), et changerait trop souvent.

Une première solution pour réduire la taille de la matrice consiste à regrouper dans des *domaines de sujets* l'ensemble des sujets qui ont les mêmes droits sur les mêmes objets, ce qui conduit à réduire le nombre de lignes de la matrice puisqu'il suffit alors d'une ligne par domaine. Ces domaines de sujets sont eux-mêmes des objets, sur lesquels les sujets d'un domaine peuvent exécuter des opérations telles que changer les droits d'un domaine ou appeler un domaine (pour changer de privilèges). Typiquement, cela peut correspondre à des modes de fonctionnement maître-esclave ou superviseur-utilisateur. Cela peut donner par exemple :

	Fichier 1	Fichier 2	Imprimante	Domaine 1	Domaine 2
Domaine 1 (superviseur)	ouvrir (l,e,x)	ouvrir (l,e)			changer les droits
Domaine 2 (utilisateur)	ouvrir (x)	ouvrir (l,e)	ouvrir (e)	appel	

La même technique de domaine de sujet est appliquée par les systèmes pour lesquels la vérification des droits se fait uniquement sur les utilisateurs et non sur les processus, ce qui revient à regrouper tous les sujets d'un même utilisateur dans un même domaine. C'est le cas d'Unix, par exemple. Ceci conduit à une gestion moins fine de la protection, et en particulier ne permet pas d'appliquer de façon optimale le principe du moindre privilège.

Une autre méthode pour réduire la taille de la matrice de droits d'accès consiste à regrouper dans des *domaines d'objets* (ou *domaines de protection*) les objets qui présentent les mêmes droits d'accès pour les mêmes sujets. La matrice ne contient plus alors qu'une colonne par domaine de protection. Ces domaines sont généralement exclusifs : quand un sujet entre dans un domaine, il perd tous les droits qu'il avait auparavant. Cette technique permet d'implémenter de façon simple les niveaux de classification des politiques par mandats multi-niveaux.

Plutôt que de gérer la matrice de droits d'accès comme une structure de données centralisée, on préfère généralement l'organiser de manière distribuée par lignes ou par colonnes :

- par lignes, cela consiste à associer à chaque sujet s_i la liste des objets o_j appartenant au contexte de s_i, et pour chaque o_j la liste des droits d_{ij} correspondants ; cette liste $\{o_j, d_{ij}\}$ est appelée *liste de droits* ou *liste de capacités* ("*capabilities*" en anglais), ou encore *c-liste* [Fabry 74] ; notons qu'il convient de protéger spécifiquement ces c-listes qui ne doivent être manipulables que par le sous-système de sécurité, et qu'en particulier, il n'est pas souhaitable qu'un sujet puisse directement créer ou copier une capacité qu'il a sur un objet pour transmettre des droits sur cet objet à un autre sujet, car cela pourrait être en contradiction avec une politique par mandats [Kain 87] ; c'est pourtant la technique qui a été choisie pour Amœba [Tanenbaum 86] qui utilise des capacités comme des tickets d'accès ;
- par colonnes, cela consiste à associer à chaque objet o_j la liste des sujets s_i qui ont des droits d'accès à o_j, et pour chaque s_i la liste des droits d_{ij} correspondants ; cette liste $\{s_i, d_{ij}\}$ est appelée *liste de contrôle d'accès* ou plus simplement *liste d'accès* (*ACL* pour "*access control list*", en anglais) ; ce type de représentation est en particulier souvent utilisée pour les fichiers ; c'est par exemple le cas des *permissions* d'Unix.

Notons que ces deux organisations en c-listes et en ACL ne sont pas incompatibles, mais sont souvent utilisées conjointement pour représenter les droits plus ou moins dynamiques : généralement, les listes de contrôles d'accès sont relativement statiques car elles changent peu durant la vie d'un objet (dans ce cas, elles contiennent les identités des utilisateurs plutôt que celles des sujets actifs) ; en revanche, les capacités sont associées aux processus et évoluent durant l'exécution des processus, mais ont une durée de vie limitée à celle du processus. Dans Multics, par exemple, il y a une liste de contrôle d'accès associée à chaque segment et à chaque catalogue ainsi qu'un *descriptif* associé à chaque processus ; ce descriptif contient en

fait les capacités que possède le processus sur chaque segment lié (c'est-à-dire pour lequel le processus a effectué une édition de liens dynamique).

3.2.4. Application aux systèmes répartis

La notion de sous-système de sécurité ou TCB, en particulier avec le moniteur de références comme médiateur incontournable de toutes les interactions sujets-objets, semble imposer une structure centralisée pour gérer la sécurité. Cette structure centralisée a été effectivement adoptée par certains projets de systèmes répartis, dont le serveur de fichiers sécurisé de la Newcastle Connection [Rushby 83]. Dans ce projet, les utilisateurs, depuis leurs terminaux, ne peuvent avoir d'interaction qu'avec un service de fichiers et seulement par l'intermédiaire d'un *gestionnaire de fichiers sécurisé* centralisé qui gère et vérifie tous les droits d'accès.

Une autre approche est celle proposée par le "*Livre Rouge*" [TNI 87], où chaque site possède sa propre TCB chargée du contrôle d'accès aux objets locaux, mais aussi des accès distants. Comme leur nom anglais l'indique, les TCB doivent être dignes de confiance, car leur efficacité repose sur leur cohérence mutuelle. En particulier, la gestion de la matrice de droits d'accès est répartie et doit rester cohérente.

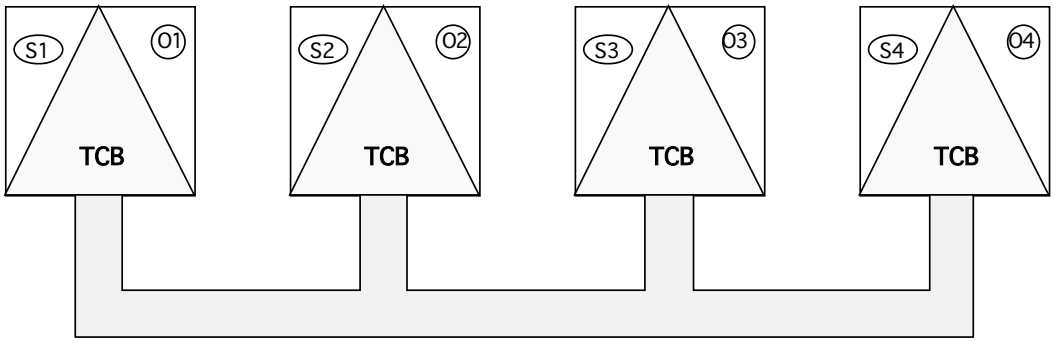


Figure 9.7 : L'approche du "Livre Rouge"

Dans cette approche, si un sujet s_1 accède à un objet distant o_2, la requête est transmise par la TCB du site 1 à la TCB du site 2 ; celle-ci fait confiance à l'authentification de l'utilisateur par la TCB du site 1, et consulte les listes de contrôle d'accès de l'objet o_2 pour autoriser ou refuser l'accès. A l'inverse, la TCB du site 1 fait confiance à la TCB du site 2 pour la vérification des droits, et en particulier pour empêcher des fuites par violation de la politique de sécurité globale. Ceci conduit assez naturellement à une gestion décentralisée de la sécurité où chaque TCB est responsable de l'authentification des utilisateurs locaux et de la gestion des listes de contrôle d'accès des objets locaux, en plus de la liaison avec les autres TCB.

Mais cette approche où chaque site doit faire confiance aux autres, n'est pas compatible avec les systèmes ouverts actuels où la gestion de la sécurité est hétérogène, ce qui rend difficile une gestion cohérente de la sécurité. D'autre part, l'approche du Livre Rouge impose qu'on ait confiance dans les personnes responsables de l'administration

de la sécurité de *chacun* des sites, puisque la violation de la TCB dans un des sites met en danger la sécurité de l'ensemble du système réparti ; ceci est incompatible avec les stations de travail actuelles où les utilisateurs peuvent facilement obtenir le contrôle total de leur machine.

Une autre solution consiste à distribuer les fonctions de sécurité uniquement sur des sites de confiance. C'est l'approche adoptée par exemple par Kerberos [Steiner 88] ou par AFS (Andrew File System) [Satyanarayanan 89], où l'authentification est gérée par un serveur d'authentification et où l'autorisation est de la responsabilité de chaque serveur. Dans ce cas, il n'est pas nécessaire de faire confiance aux sites utilisateurs, mais il faut protéger particulièrement les serveurs (en particulier le serveur d'authentification qui possède *en clair* les mots-de-passe des utilisateurs). Cette technique n'est pas compatible avec des politiques d'autorisation par mandats, car il n'est pas possible d'empêcher qu'il puisse y avoir des interactions entre utilisateurs qui ne passent pas par des serveurs sécurisés.

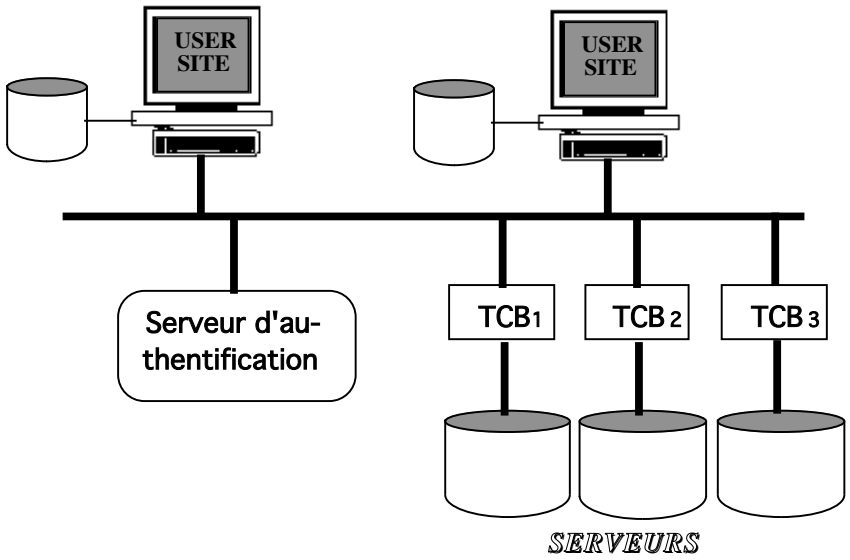


Figure 9.8 : L'approche de Kerberos

L'approche choisie dans le projet Saturne (et reprise dans le projet ESPRIT Delta-4) consiste à ne faire confiance à aucun site particulier, mais seulement à un quorum de sites. Ainsi, un utilisateur ne sera pas authentifié seulement par un site, mais au contraire devra être authentifié par une majorité de sites dits de sécurité, pour pouvoir par la suite obtenir des accès à des serveurs distants. De la même façon, les sites de sécurité sont responsables de l'autorisation, c'est-à-dire qu'ils gèrent les listes de contrôle d'accès, ce qui décharge d'autant les serveurs, dans lesquels il n'est pas nécessaire d'avoir confiance. Cette technique permet de tolérer des intrusions dans un nombre minoritaire de sites des sécurité, et permet même de tolérer la malveillance d'un petit nombre d'administrateurs des sites de sécurité [Deswarte 91].

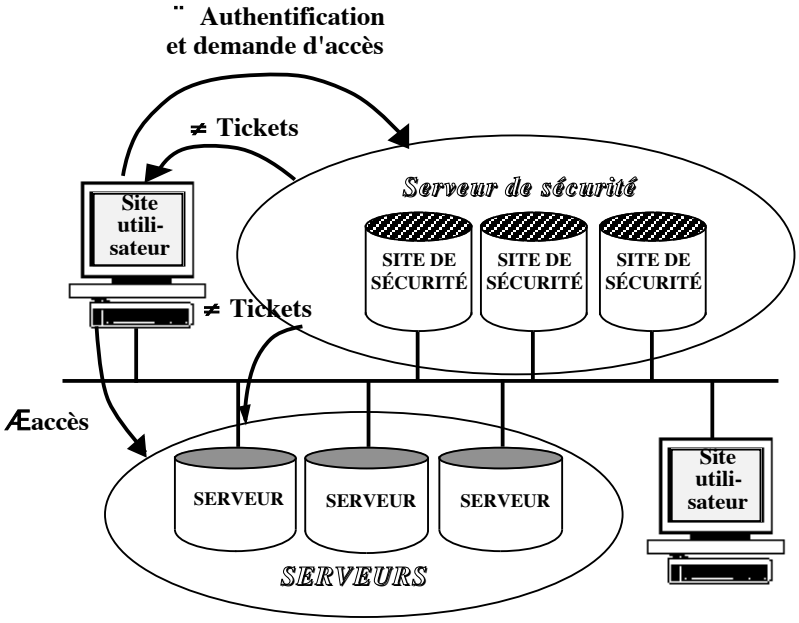


Figure 9.9 : L'approche de Saturne

3.3. Autres aspects de la sécurité

Il est possible d'utiliser la même classification pour les techniques de la sécurité que pour celles de la sûreté de fonctionnement au sens général, en s'intéressant principalement aux fautes intentionnelles. Ceci conduit à distinguer :

- *la prévention des fautes*, qui consiste à empêcher l'introduction de fautes de conception intentionnelles ou d'intrusions,
- *la tolérance aux fautes*, qui vise à permettre de délivrer un service conforme malgré la présence de fautes de conception intentionnelles ou d'intrusions,
- *la suppression de fautes*, c'est-à-dire la minimisation, par vérification, de la présence de fautes (de conception intentionnelles),
- *la prévision des fautes*, c'est-à-dire l'évaluation des conséquences des fautes éventuelles.

3.3.1 Prévention des fautes intentionnelles

La première méthode de prévention est la dissuasion : faire en sorte que l'attaquant sache qu'il court un risque sérieux s'il commet délibérément des fautes (qu'il s'agisse d'intrusions ou de fautes de conception). La dissuasion peut reposer sur des règlements intérieurs de l'entreprise pour ce qui concerne les attaques les plus courantes qui proviennent de membres de l'entreprise, ou sur un arsenal législatif de plus en plus complet et efficace, tant en France qu'à l'étranger. Pour la France, on peut citer en particulier :

- la loi "Informatique et libertés",
- la convention européenne sur la défense des libertés individuelles,

- les lois sur les droits d'auteurs et la protection des logiciels,
- la loi "Godfrain" sur le piratage informatique.

Pour ce qui concerne les fautes de conception délibérées, les techniques de génie logiciel, destinées principalement à prévenir les fautes involontaires, peuvent être efficaces : lecture croisée des sources, programmation orientée-objets, génération des tests à partir des spécifications, etc. Mais elles peuvent être complétées par des techniques administratives : habilitation et contrôle des personnels de développement et des fournisseurs, procédures de mise en place de nouveaux logiciels, procédures de maintenance, cloisonnement entre systèmes de développement et systèmes opérationnels.

La prévention des intrusions repose principalement sur l'authentification des utilisateurs. La prévention des canaux couverts s'obtient par le contrôle des flux d'information : pour les canaux de mémoire, il faut contrôler la réutilisation des ressources (tampons mémoire, fichiers temporaires, ...) ; pour les canaux temporels, difficiles à éliminer complètement, il faut chercher à en réduire la bande passante, par exemple en s'assurant que les ressources sont allouées statiquement, ou au moins pour des durées fixes. La prévention des déductions par inférences sur les bases de données nécessite la restriction des requêtes statistiques autorisées, mais comme pour les canaux temporels, il est difficile de les éliminer complètement.

3.3.2. Tolérance aux fautes intentionnelles

Pour tolérer les fautes de conception délibérées, la seule voie, encore peu explorée, semble être la diversification de la conception, c'est-à-dire faire produire par des équipes différentes des versions diversifiées, avec vote sur les résultats [Joseph 88].

En revanche, il existe de nombreuses techniques pour tolérer les intrusions :

- le *chiffrement* permet de se protéger contre les écoutes passives ; il existe aussi d'autres techniques cryptographiques pour détecter des modifications non-autorisées (signatures numériques, fonctions de compression, etc.) ;
- la *réplication* permet de se protéger contre les modifications et les destructions non-autorisées, puisqu'un intrus devra modifier ou détruire la majorité des exemplaires pour réussir son attaque ; mais la réplication est néfaste pour la confidentialité : il suffit de lire l'un des exemplaires pour en obtenir le contenu ;
- la *détection-recouvrement des intrusions* consiste à détecter, ralentir et intervenir :
  - détection des intrusions, par les mécanismes de contrôle d'accès, complétés par des dispositifs permettant de discriminer entre le comportement normal des utilisateurs autorisés et un comportement anormal qui peut être le signe d'une intrusion ; ces dispositifs de discrimination peuvent être basés sur des systèmes experts [Denning 86] ;
  - ralentissement des intrusions, soit en combattant l'intrus par la résiliation des droits qu'il a pu obtenir (mais alors il se saura détecté), soit en le trompant ou

en le gavant d'informations inutiles (c'est la solution qu'avait adopté Clifford Stoll dans le cas du "Nid de Coucou", cf. §3.1.8.) ;

- intervention, par la localisation de l'intrus et les poursuites judiciaires à son encontre, ainsi que par la restauration éventuelle des informations détruites et la correction des failles de sécurité ;

- le *brouillage* consiste à ajouter des informations superflues (par exemple, messages de *bourrage* sur les voies de communication), ou à augmenter l'incertitude dans les réponses aux requêtes statistiques dans les bases de données pour tolérer les attaques de déduction par inférences ;
- la *fragmentation-dissémination* consiste à découper l'information en fragments de telle sorte que des fragments isolés ne puissent fournir d'information significative, puis à séparer les fragments les uns des autres par dissémination de sorte que l'intrusion d'une partie du système ne fournisse que des fragments isolés ; la dissémination peut être géographique (utiliser des sites ou des voies de communication différents), temporelle (transmettre les fragments dans un ordre aléatoire ou mélangés avec d'autres sources de fragments), fréquentielle (utiliser des fréquences différentes dans des communications à large bande) ; cette technique est utilisée dans le projet Saturne pour réaliser un service d'archivage de fichiers sûr ainsi que pour gérer la sécurité dans les systèmes répartis, et son application est envisagée pour le traitement fiable de données confidentielles [Fabre 89].

3.3.3. Élimination des fautes : Vérification de la sécurité

L'élimination des fautes dans un système traitant des informations sensibles vise aussi bien les fautes d'origine intentionnelle que celles d'origine accidentelle :

- les techniques de vérification de la sûreté de fonctionnement "classique" s'appliquent (relecture du code, preuve formelle, etc.)
- une conception structurée facilite grandement cette vérification ; deux approches sont couramment utilisées :
  - concevoir le sous-système de sécurité comme un *noyau de sécurité*, c'est-à-dire sous forme d'un sous-ensemble compact du logiciel système et du matériel, uniquement responsable de la sécurité ;
  - décomposition en *niveaux d'abstraction* pour faciliter la mise en œuvre d'une preuve hiérarchique.

Le test des mécanismes de prévention et de tolérance aux intrusions peut se faire par une *analyse de pénétration* ("*Tiger Team*"), mais il faut se rappeler qu'un test permet de prouver qu'il y a des fautes, mais ne permet en aucun cas d'assurer qu'il n'y en a plus.

3.3.4. Prévision des fautes : Évaluation de la sécurité

Il existe des méthodes globales d'évaluation de la sécurité d'une installation informatique. En France, les plus utilisées sont MARION [Lamère 85], développée et soutenue par l'APSAIRD, l'Assemblée Plénière des Sociétés d'Assurances contre l'Incendie et les Risques Divers, et MELISA développée pour le compte de la Délégation Générale pour l'Armement.

Pour ces méthodes, les entraves à la sécurité des systèmes informatiques sont généralement classées en menaces, vulnérabilités et risques :

- Les *menaces* sont caractérisées par les possibilités et les probabilités d'attaque contre la sécurité. Elles sont définies par la source (interne au système ou externe), par la motivation des attaquants, par le processus d'attaque, par la cible et par le résultat (conséquences de la réussite d'une attaque).
- Les *vulnérabilités* sont les fautes de conception, intentionnelles ou accidentelles, qui favorisent la réalisation d'une menace ou la réussite d'une attaque. Les fautes de conception intentionnelles peuvent être malicieuses, dans le but de mettre en défaut la sécurité du système, mais elles peuvent aussi être le résultat délibéré d'un compromis acceptable entre fonctionnalité, sécurité et coût[11].
- Les *risques* sont le résultat de la combinaison des menaces et des vulnérabilités. Les risques doivent être évalués, soit pour obtenir le meilleur compromis possible entre sécurité et coût pour un système donné, soit simplement pour calculer le montant des primes d'assurance pour couvrir ces risques.

Mais, pour comparer la capacité de divers systèmes informatiques à faire face à des fautes intentionnelles, on utilise généralement les critères d'évaluation qualitative définis par le DoD dans ce qui est couramment appelé le *Livre Orange* ou TCSEC ("*Trusted Computer System Evaluation Criteria*") [TCSEC 85], ou dans les livres de diverses couleurs qui l'accompagnent, comme le *Livre Rouge* ou TNI ("*Trusted Network Interpretation of the TCSEC*"). Ces critères, basés à la fois sur des listes de fonctions de sécurité à remplir et sur les techniques employées pour la vérification, conduisent à classer les systèmes en 7 catégories (dans un ordre décroissant de sécurité : A1, B3, B2, B1, C2, C1, D). Plus récemment, des critères *harmonisés* européens sont parus, qui séparent explicitement les aspects fonctionnels des aspects de vérification [ITSEC 91].

L'évaluation quantitative de la sécurité n'a encore reçu que peu d'attention de la part des spécialistes, à l'exception de l'évaluation de la bande passante des canaux couverts et de l'application de la théorie de l'information aux mécanismes cryptographiques.

---

[11] Un parallèle peut être fait avec l'absence ou l'insuffisance d'un blindage contre les radiations qui peut, par exemple, être le résultat de contraintes sur la masse d'un système embarqué dans un véhicule spatial.

3.3.5. Sécurité des communications

Ainsi qu'il a été montré dans l'ensemble de ce rapport, la sécurité des systèmes répartis peut être vue d'abord en considérant un système réparti comme une machine abstraite sur laquelle s'exécutent des applications réparties. Mais, complémentairement à cette vue, il convient de prendre en compte également la sécurité des communications, qui servent de support au système réparti. La sécurité des communications est l'objet d'études depuis très longtemps, bien avant le développement de l'informatique : c'est pour les communications que les premiers algorithmes de chiffrement ont été utilisés (Suétone, dans "De vita Cæsarum", décrit un chiffre utilisé par Jules César pour les messages qu'il envoyait à ses intimes).

Aujourd'hui, la sécurité des communications a atteint un niveau de maturité suffisant pour que des normes soient publiées, dont en particulier la Norme Internationale IS-7498-2 de l'ISO [ISO 88]. Cette norme définit des fonctions de sécurité (pour les communications) et des mécanismes pour réaliser ces fonctions. Les fonctions de sécurité sont :

- l'authentification des entités communicantes,
- l'autorisation de communiquer,
- la confidentialité des communications (données et flux),
- l'intégrité des communications,
- la non-répudiation.

Les mécanismes définis dans cette norme sont :

- le chiffrement,
- les signatures numériques,
- la "*notarisation*" (qui consiste à insérer un troisième partenaire dans une communication bi-point pour garantir l'authenticité des échanges),
- le bourrage,
- des techniques de routage spécifiques, etc.

Il faut noter que la disponibilité est absente de la liste des fonctions de sécurité préconisée par l'ISO, et que les mécanismes proposés sont bien adaptés aux communications bi-point, mais pas à la diffusion.

4. CONCLUSION

Si ce rapport a montré que la répartition des systèmes était favorable à la tolérance aux fautes, cela ne surprend personne : bien plus, il est difficile aujourd'hui de concevoir un système tolérant aux fautes qui ne soit pas réparti. Il ne faut pas en conclure pour autant qu'un système réparti tolère de facto les fautes : un soin particulier doit être accordé à la conception des mécanismes de détection et de recouvrement des erreurs, ou de compensation des erreurs, pour obtenir une couverture suffisante et ainsi

éviter une propagation d'erreurs, qui serait favorisée par les interactions inhérentes aux systèmes répartis.

En revanche, les systèmes répartis ont toujours été considérés comme mal adaptés au traitement de données sensibles parce que leur dispersion géographique et leur ouverture vers d'autres systèmes rendent difficiles sinon impossibles les contrôles d'accès physiques, et parce que leur complexité, due en particulier à l'asynchronisme des traitements, interdit encore aujourd'hui d'en vérifier exhaustivement la sécurité. De plus, les principes sur lesquels ont été bâtis les systèmes informatiques centralisés les plus sûrs, comme les notions de moniteur de références et de noyau de sécurité, ne sont pas applicables directement aux systèmes répartis. De nouvelles voies sont donc encore à explorer, comme celles de la tolérance aux intrusions pour laquelle il a été montré que des techniques comme la fragmentation-dissémination pouvaient tirer profit de la répartition des systèmes pour en améliorer la sécurité.

5. NOTES BIBLIOGRAPHIQUES

Il existe de nombreux ouvrages généraux sur la sûreté de fonctionnement ; parmi les plus intéressants, citons [Lee 90], [Siewiorek 82] et [Laprie 89b], les deux premiers pour les nombreux exemples de systèmes tolérant les fautes, le troisième pour son approche très structurée et pour l'importance donnée à la validation et aux méthodes de conception. Les principaux résultats de la recherche en tolérance aux fautes sont publiés à la conférence annuelle "International Symposium on Faut-Tolerant Computing (FTCS)" de l'IEEE.

Pour ce qui concerne la sécurité informatique, une approche didactique est présentée dans [Gasser 89] et dans [Russell 91], alors que [Denning 83] est un ouvrage de référence fort complet. On pourra consulter aussi [Garfinkel 91] pour la sécurité d'Unix. La principale conférence sur la recherche dans ce domaine est le symposium annuel d'Oakland "IEEE Computer Society Symposium on Research in Security and Privacy".

Références bibliographiques

[Abadir 82]
J. Abadir, Y. Deswarte, "Programme d'auto-test des processeurs du système Sargos", *Proceedings of the 3rd International Conference on Reliability and Maintainability*, CNES,ESA, CNET and SEE, Toulouse, France, October 1982, pp. 639-648.

[Avizienis 84]
A. Avizienis, J.P.J. Kelly, "Fault tolerance by design diversity: concepts and experiments", *IEEE Computer*, vol.17, n°8 (August 1984), pp. 67-80.

[Avizienis 85]
A. Avizienis, P. Gunningberg, J.P.J. Kelly, P.J. Traverse, K.S. Tso, U. Voges, "The UCLA DEDIX system: a distributed testbed for multiple-version software", *Proceedings of the 15th International Symposium on Faut-Tolerant Computing (FTCS-15),* Ann Arbor (Mi.), USA, June 1985, IEEE, pp. 126-134.

[Banâtre 86]
J.P. Banâtre, M. Banâtre, G. Lapalme, F. Ployette, "The design and building of ENCHERE, a distributed electronic marketing system", *Communications of the ACM*, vol.29, n°1 (January 1986), pp. 19-29.

[Banâtre 87]
J.P. Banâtre, M. Banâtre, G. Muller, F. Ployette, "Quelques aspects du système GOTHIC", *Technique et Science Informatique (T.S.I.)*, vol.6, n°2 (1987), pp. 170-174.

[Bartlett 87]
J. Bartlett, J. Gray, B. Horst, "Fault tolerance in Tandem computer systems", in *Dependable Computing and Fautl-Tolerant Systems, Vol.1: The Evolution of Fault-Tolerant Computing*, Ed. A. Avizienis, H. Kopetz, J.C. Laprie, Springer-Verlag, ISBN 0-387-81941-X (1987), pp. 55-76.

[Bell 74]
D.E. Bell, L.J. LaPadula, *Secure computer systems: unified exposition and Multics interpretation*, Tech. Rept. MTR-2997 (AD/A-020-445), The MITRE Corporation, April 1974.

[Biba 75]
K.J. Biba, *Integrity consideration for secure computer systems*, Tech. Rept. MTR-3153 (ESD-TR-76-372), The MITRE Corporation, June 1975.

[Brewer 89]
D.F.C. Brewer, M.J. Nash, "The Chinese Wall Security Policy", *Proceedings of the 1989 IEEE Computer Society Symposium on Security and Privacy,* IEEE, Oakland (Ca.), USA, May 1989, pp. 206-214.

[Brunner 82]
J. Brunner, *Sur l'onde de choc*, J'ai lu, n°1368, ISBN 2-277-21368-3 (1982), Titre original: The shockwave rider.

[Clark 87]
D.D. Clark, D.R. Wilson, "A comparison of commercial and military computer security policies", *Proceedings of the 1987 IEEE Computer Society Symposium on Security and Privacy,* IEEE, Oakland (Ca.), USA, May 1987, pp. 184-194.

[Cohen 86]
F. Cohen, *Computer Viruses*, Ph.D. dissertation, University of Southern California, January 1986.

[Cohen 87]
F. Cohen, "Computer Viruses: Theory and Experiments", *Computers and Security (North-Holland)*, vol.6, n°1 (February 1987), pp. 22-35.

[Cristian 85]
F. Cristian, "Exceptions, défaillances et erreurs", *Technique et Science Informatique (T.S.I.)*, vol.4, n°4 (1985), pp. 385-390.

[Cristian 88]
F. Cristian, "Agreeing on who is present and who is absent in a synchronous distributed system", *Proceedings of the 18th International Symposium on Fault Tolerant Computing (FTCS-18),* Tokyo, Japan, June 1988, IEEE, pp. 206-211.

[Denning 83]
D.E. Denning, *Cryptography and Data Security*, Addison-Wesley, ISBN 0-201-10150-5 (1983)

[Denning 86]
D.E. Denning, "An intrusion-detection model", *Proceedings of the 1986 IEEE Computer Society Symposium on Security and Privacy,* IEEE, Oakland (Ca.), USA, May 1986, pp. 118-132.

[Deswarte 75]
Y. Deswarte, J. Lavictoire, "MARIGNAN : an intermittent failure correction method", *Proceedings of the 5th International Symposium on Fault Tolerant Computing (FTCS-5),* Paris, France, June 1975, IEEE, pp. 191-195.

[Deswarte 91]
Y. Deswarte, L. Blain, J.C. Fabre, "Intrusion Tolerance in Distributed Systems", *Proceedings of the 1991 IEEE Symposium on Research in Security and Privacy*, Oakland (Ca.), mai 1991, pp.110-121.

[Eichin 89]
M.W. Eichin, J.A. Rochlis, "With microscope and tweezers: an analysis of the Internet virus of November 1988", *Proceedings of the 1989 IEEE Computer Society Symposium on Security and Privacy,* Oakland (Ca.), USA, May 1989, pp. 326-343.

[Ezhilchelvan 86]
P.D. Ezhilchelvan, S.K. Shrivastava, "A characterisation of faults in systems", *Proceedings of the 5th Symposium on Reliability in Distributed Software and Database Systems,* IEEE, Los Angeles (Ca.), USA, January 1986, pp. 215-222.

[Fabre 82]
J.C. Fabre, *Un mécanisme pour la tolérance aux pannes dans l'architecture répartie Chorus*, Thèse de 3ème cycle, n°2695, Université Paul Sabatier, Toulouse, octobre 1982.

[Fabre 89]
J.C. Fabre, M. Aguera, M. Allia, Y. Deswarte, J.M. Fray, J.C. Laprie, J.M. Pons, D. Powell, P.G. Ranéa, "Saturne: un système réparti tolérant les fautes et les intrusions", *Actes du Séminaire Franco-Brésilien sur les Systèmes Informatiques Répartis,* LCMI-UFSCet LAAS-CNRS, Florianopolis-SC, Brésil, septembre 1989, pp. 291-298.

[Fabry 74]
R.S. Fabry, "Capability-based addressing", *Communications of the ACM*, vol.17, n°7 (July 1974), pp. 40-412.

[Garfinkel 91]
S. Garfinkel, G. Spafford, *Practical Unix Security*, O'Reilly & Associates, ISBN 0-937175-72-2 (1991).

[Gasser 89]
M. Gasser, *Building a Secure Computer System*, Van Nostrand Reinhold Company (1989)

[Grampp 84]
F.T. Grampp, R.H. Morris, "Unix operating system security", *AT&T Bell Laboratories Technical Journal*, vol.63, n°8 (October 1984), pp. 1649-1672.

[Gray 86]
J. Gray, "Why do computers stop and what can be done about it?", *Proceedings of the 5th Symposium on Reliability in Distributed Software and Database Systems,* IEEE, Los Angeles (Ca.), USA, January 1986, pp. 3-12.

[Harrison 76]
M.A. Harrison, W.L. Ruzzo, J.D. Ullman, "Protection in operating systems", *Communications of the ACM*, vol.19, n°8 (August 1976), pp. 461-471.

[Harrison 87]
E.S. Harrison, E.J. Schmitt, "The structure of System/88, a fault-tolerant computer", *IBM Systems Journal*, vol.26, n°3 (1987), pp. 293-318.

[Horning 74]
J.J. Horning, H.C. Lauer, P.M. Melliar-Smith, B. Randell, “A program structure for error detection and recovery”, *Proceedings of the Conference on Operating Systems : Theoretical and Practical Aspects,* IRIA, Rocquencourt, France, April 1974, Springer-Verlag, Lecture Notes in Computer Science Vol.16, pp. 177-193.

[ISO 88]
ISO, *International Standard 7498-2: Information processing systems - OSI Reference model - Part 2: Security Architecture*, n°2890, ISO/IEC JTC1/SC21, July 1988.

[ITSEC 91]
ITSEC, *Information Technology Security Evaluation Criteria*, Provisional Harmonised Criteria, Version 1.2, June 1991, ISBN 92-826-3004-8, Office for Publications of the European Communities, L-2985 Luxembourg. Existe également en français : *Critères d’évaluation de la sécurité des systèmes informatiques*, Critères harmonisés provisoires, Version 1.2, 28 juin 1991.

[Jewett 91]
D.Jewett, "Integrity S2: A Fault-Tolerant Unix Platform", in *Proc. 21th Int. Symp. on Fault-Tolerant Computing Systems (FTCS-21)*, (Montréal, Canada), pp.512-519, IEEE Computer Society Press, 1991.

[Jones 76]
A.K. Jones, R.J. Lipton, L. Snyder, “A linear time algorithm for deciding security”, *Proceedings of the 17th Annual Symposium on Foundation of Computer Science,* 1976.

[Joseph 88]
M.K. Joseph, A. Avizienis, “A fault-tolerance approach to computer viruses”, *Proceedings of the 1988 IEEE Computer Society Symposium on Security and Privacy,* IEEE, Oakland (Ca.), USA, May 1988, pp. 52-58.

[Kain 87]
R.Y. Kain, C.E. Landwehr, “On access checking in capability-based systems”, *IEEE Transactions on Software Engineering*, vol.SE-13, n°2 (February 1987), pp. 202-207.

[Kurzban 90]
S.A. Kurzban, “Defending against Viruses and Worms”, *Cipher: Newsletter of the Technical Committee on Security and Privacy, IEEE Computer Society* (Winter 1990), pp. 23-40.

[Lala 86]
J.H. Lala, “A byzantine resilient fault tolerant computer for nuclear power plant applications”, *Proceedings of the 16th International Symposium on Fault Tolerant Computing (FTCS-16),* Vienne, Autriche, July 1986, IEEE, pp. 338-343.

[Lamère 85]
J.M. Lamère, *La sécurité informatique: approche méthodologique*, Dunod Informatique, ISBN 2-04-016503-7 (1985).

[Lamport 82]
L. Lamport, R. Shostak, M. Pease, “The Byzantine generals problem”, *ACM Transactions on Programming Languages and Systems*, vol.4, n°3 (July 1982), pp. 382-401.

[Lamport 85]
L. Lamport, P.M. Melliar-Smith, “Synchronizing clocks in the presence of faults”, *Journal of the ACM*, vol.32, n°1 (January 1985), pp. 52-78.

[Laprie 89]
J.C. Laprie, “Sûreté de fonctionnement des systèmes informatiques et tolérance aux fautes”, *Les Techniques de l'Ingénieur*, n° H.4.450 (septembre 1989), pp. 1-12.

[Laprie 89b]
J.C. Laprie, B. Courtois, M.C. Gaudel, D. Powell, *Sûreté de fonctionnement des systèmes informatiques*, Dunod Informatique, ISBN 2-04-016942-3 (1989).

[Laprie 92]
J.C. Laprie, *Dependability: Basic Concepts and Terminology (in English, French, German, Italian and Japanese)*, Série : Dependable Computing and Fault-Tolerant Systems, (A. Avizienis, H. Kopetz, J.C. Laprie Eds.), Vol.5, Springer-Verlag, 265 p., ISBN 3-211-82296-8 and 0-387-82296-8, 1992

[Lee 90]
P.A. Lee, T. Anderson, *Fault Tolerance : Principles and Practice*, (2nd revised edition), Springer-Verlag, Collection "Dependable Computing and Fault-Tolerant Systems", vol. 3 (Editors : A. Avizienis, H. Kopetz and J.C. Laprie), ISBN 3-211-82077-9 (1990).

[Merlin 78]
P.M. Merlin, B. Randell, "State restoration in distributed systems", *Proceedings of the 8th International Symposium on Fault Tolerant Computing (FTCS-8),* Toulouse, France, June 1978, IEEE, pp. 129-134.

[Morris 79]
R.H. Morris, K. Thompson, "Unix password security", *Communications of the ACM*, vol.22, n°11 (November 1979), pp. 594-597.

[Needham 78]
R.M. Needham, M.D. Schroeder, "Using encryption for authentication in large networks of computers", *Communications of the ACM*, vol.21, n°12 (December 1978), pp. 993-999.

[Powell 88]
D. Powell, G. Bonn, D. Seaton, P. Verissimo, F. Waeselynck, "The Delta-4 approach to dependability in open distributed computing systems", *Proceedings of the 18th International Symposium on Fault Tolerant Computing (FTCS-18),* Tokyo, Japan, June 1988, IEEE, pp. 246-251.

[Powell 89]
D. Powell, "La tolérance aux fautes dans les systèmes répartis : Les hypothèses d'erreur et leur importance", *Actes du Séminaire Franco-Brésilien sur les Systèmes Informatiques Répartis,* LCMI-UFSCet LAAS-CNRS, Florianopolis-SC, Brésil, septembre 1989, pp. 36-43.

[Randell 75]
B. Randell, "System structure for software faut tolerance", *IEEE Transactions on Software Engineering*, vol.SE-1, n°2 (June 1975), pp. 220-232.

[Rivest 78]
R. Rivest, A. Shamir, L. Adleman, "A method for obtaining digital signatures and public-key cryptosystems", *Communications of the ACM*, vol.21, n°2 (February 1978), pp. 120-126.

[Rushby 83]
J.M. Rushby, B. Randell, "A distributed secure system", *IEEE Computer*, vol.16, n°7 (July 1983), pp. 55-67.

[Russell 91]
D. Russell, G.T. Gangemi Sr, *Computer Security Basics*, O'Reilly & Associates, ISBN 0-937175-71-4 (1991).

[Satyanarayanan 89]
M. Satyanarayanan, "Integrating Security in a Large Distributed System", *ACM Transactions on Computing Systems*, vol.7, n°3 (August 1989), pp. 247-280.

[Schneider 84]
F.B. Schneider, "Byzantine generals in action: implementing fail-stop processors", *ACM Transactions on Computing Systems*, vol.2, n°2 (May 1984), pp. 145-154.

[Shoch 82]
J.F. Shoch, J.A. Hupp, "The Worm Programs — Early Experience with a Distributed Computation", *Communications of the ACM*, vol.25, n°3 (March 1982), pp. 172-180.

[Siewiorek 82]
D.P. Siewiorek, R.S. Swartz, *The Theory and Practice of Reliable System Design*, Digital Press, ISBN 0-932376-13-4 (1982)

[Smith 86]
T.B. Smith, "High performance fault tolerant real time computer architecture", *Proceedings of the 16th International Symposium on Fault Tolerant Computing (FTCS-16),* Vienne, Autriche, July 1986, IEEE, pp. 14-19.

[Solomon 89]
A. Solomon, B. Nielson, S. Meldrum, *AIDS Trojan*, Tech. Rept. Water Meadow, Chesham, Bucks, HP5 1LP England, December 1989.

[Spafford 88]
E.H. Spafford, *The Internet Worm Program: An analysis*, Tech. Rept. CSD-TR-823, Purdue University, Dept of Computer Sciences, 40 p., November 1988.

[Steiner 88]
J.G. Steiner, C. Neumann, J.I. Schiller, "Kerberos: an authentication service for open network systems", *Proceedings of the USENIX Winter Conference,* Dallas (Tx.), USA, February 9-12, 1988, pp. 191-202.

[Stoll 88]
C. Stoll, "Stalking the wily hacker", *Communications of the ACM*, vol.31, n°5 (May 1988), pp. 484-497.

[Stoll 89]
C. Stoll, *Le nid du coucou*, Albin-Michel, ISBN 2-226-03781-0 (1989), Titre original: The cuckoo's egg.

[Tanenbaum 86]
A.S. Tanenbaum, S.J. Mullender, R. van Renesse, "Using sparse capabilities in a distributed operating system", *Proceedings of the 6th International Conference on Distributed Computing Systems (ICDCS-6),* IEEE, Cambridge (Ma.), USA, May 1986, pp. 558-563.

[TCSEC 85]
TCSEC, *Trusted Computer System Evaluation Criteria*, DoD 5200.28-STD, Department of Defense, USA, December 1985.

[TNI 87]
TNI, *Trusted Network Interpretation of the Trusted Computer System Evaluation Criteria*, NCSC-TG-005, National Computer Security Center, 31 July 1987.

[Traverse 89]
P. Traverse, "Dependability of digital computers on board airplanes", *Proceedings of the First International Working Conference on Dependable Computing for Critical Applications,* IFIP WG 10.4, Santa Barbara (Ca.), USA, August 1989.

[Webber 91]
S.Webber and J.Beirne, "The Stratus Architecture", in *Proc. 21th Int. Symp. on Fault-Tolerant Computing Systems (FTCS-21)*, (Montréal, Canada), pp.79-85, IEEE Computer Society Press, 1991.

[Yee 88]
B.S. Yee, J.D. Tygar, A.Z. Spector, *Strongbox: a self-securing protection system for distributed programs*, Technical Report CMU-CS-87-184, Dept. of Computer Science, Carnegie-Mellon University, January 1988 1988.