Avant propos *1*

Table des Matières *3*

Chapitre I Introduction Générale *9*

Chapitre II Avionique et Tolérance aux Fautes : Etat de l'Art *13*

Introduction **14**

II.1 Applications avioniques : état de l'art **15**
II.1.1 Standards avioniques pour la sécurité-innocuité 15
II.1.1.1 Sécurité des systèmes avioniques : ARP 4754 16
Evaluation des risques fonctionnels (FHA) 18
Evaluation préliminaire de la sécurité des systèmes (PSSA) 18
Evaluation de la sécurité des systèmes (SSA) 18
II.1.1.2 Sécurité des entités logicielles en avionique DO-178B 19
II.1.1.3 Considérations pour la diversification des logiciels selon la DO-178B 20
II.1.2 Architectures IMA 21
II.1.2.1 Systèmes avioniques classiques 21
II.1.2.2 Caractéristiques des architectures IMA 22
II.1.3 Sécurité-immunité dans les systèmes avioniques 24
II.1.3.1 L'approche sécurité-immunité dans l'avionique 24
II.1.3.2 Contraintes de sécurité-immunité dans l'avion 25
II.1.3.3 Communications bord-sol et contraintes de sécurité-immunité 28

II.2 Sûreté de fonctionnement : concepts et terminologie **31**
II.2.1 Définitions 31
II.2.1.1 La sûreté de fonctionnement : 31
II.2.1.2 Entraves à la sûreté de fonctionnement : 32
II.2.1.3 Moyens de la sûreté de fonctionnement : 33
II.2.2 La tolérance aux fautes 34
II.2.2.1 Tolérance aux fautes accidentelles 34
Mécanismes de détection: 34
Mécanismes de recouvrement: 35
II.2.2.2 Tolérance aux fautes malveillantes 36
Mécanismes de détection 37
Recouvrement avec la fragmentation – redondance – dissémination: 37
II.2.2.3 Tolérance aux fautes et redondance 39

II.3 Conclusion **40**

Chapitre III Considérations Immunité - Innocuité *41*

Introduction **42**

III.1 Intégrité et Sécurité-innocuité **43**
III.1.1 Retour sur les normes de développement des systèmes critiques 43
III.1.1.1 Les critères SQUALE 43
III.1.1.2 La norme IEC 61508 44
III.1.2 Considérations multiniveaux 45
III.1.2.1 Criticité et confiance 46
III.1.2.2 Paramètres de confiance 47

III.2 Intégrité dans les systèmes à criticités multiples **52**
III.2.1 Modèle Biba 52
III.2.1.1 Définition du treillis 52
III.2.1.2 Concepts et notations de la politique d'intégrité 52
III.2.1.3 Spécification de la politique 53
III.2.2 Modèle de non interférence 54
III.2.3 Modèle de dépendances causales 54
III.2.4 Modèle Clark & Wilson 56
III.2.5 Modèle Totel 57
III.2.5.1 Définitions et notation 57

III.2.5.2 Flux de données entre les SLO 58

III.3 Conclusion **60**

Chapitre IV La virtualisation pour la tolérance aux fautes ***61***

Introduction **62**

IV.1 Retour sur la tolérance aux fautes **63**
IV.1.1 La redondance et les systèmes tolérants aux fautes 63
IV.1.2 La redondance logicielle 65

IV.2 La virtualisation **67**
IV.2.1 Origines de la virtualisation 67
IV.2.2 Définition simple de la virtualisation 67
IV.2.3 Interfaces d'abstraction logicielles 68
IV.2.3.1 Interface binaire d'application (ABI) 69
IV.2.3.2 Interfaces de Programme d'Application 70
IV.2.4 Virtualisation et Interfaces Logicielles 70
IV.2.4.1 Machine virtuelle de processus 71
La multiprogrammation 71
L'émulation 72
Les langages de haut niveau 72
IV.2.4.2 Machine virtuelle de système 72

IV.3 Implémentation et utilisation de la virtualisation **75**
IV.3.1 Implémentations de machines virtuelles de système 75
IV.3.1.1 Isolateur 75
IV.3.1.2 Virtualisation complète 76
IV.3.1.3 Virtualisation assistée par le matériel 78
IV.3.1.4 Paravirtualisation 79
IV.3.1.5 Combinaisons des implémentations de machines virtuelles 80
IV.3.2 Exemples d'utilisation des techniques de virtualisation système 81
IV.3.2.1 Isolation 82
IV.3.2.2 Observation d'activités de systèmes virtualisés 83
Traçage d'activité d'un système 83
Détection d'intrusion 84
IV.3.2.3 Contrôle de systèmes virtualisés 85
Rajeunissement 85
Sauvegarde et reprise des activités d'un système 85
Equilibrage de charge 86

IV.4 Conclusion **87**

Chapitre V Architectures de Sécurités pour les applications critiques embarquées ***89***

Introduction **90**

V.1 Hypothèses de fautes et de fonctionnement **92**
V.1.1 Hypothèses de fautes de l'architecture déployée 92
V.1.1.1 Fautes au niveau du matériel 93
V.1.1.2 Fautes au niveau du moniteur de machines virtuelles 93
V.1.1.3 Fautes au niveau des systèmes invités 94
V.1.1.4 Fautes au niveau de l'objet de validation et son environnement d'exécution 95
V.1.1.5 Fautes au niveau des applications 95
V.1.1.6 Synthèse des hypothèses de fautes 95
V.1.2 Déterminisme des versions et validation 96
V.1.2.1 Sources d'indéterminisme 97
V.1.2.2 Renforcement du déterminisme 98

V.2 Formalisation du modele totel **100**
V.2.1 Dépendances causales et flux d'informations 101
V.2.1.1 Dépendance causale directe et indirecte 101
V.2.1.2 Flux et causalité 102

V.2.2 Contrôle d'intégrité : utilisation du modèle Totel avec les dépendances causales 103
V.2.2.1 Rappel des règles de contrôle d'intégrité 104
V.2.2.2 Conventions de notations dans le modèle Totel 105
V.2.2.3 Objets de validation par vraisemblance 106
V.2.2.4 Objets de validation par réplication 107
V.2.3 Validité des résultats de l'objet de validation 108
V.2.3.1 Niveau de confiance et validation des flux 109
V.2.3.2 Observations réelles et validation des flux 111

V.3 Analyse fonctionnelle et conception 113
V.3.1 Niveaux d'identification et de validation des flux 113
V.3.1.1 Niveaux d'identification et de capture 113
Au niveau de l'objet 113
Au niveau d'un intergiciel 114
Au niveau du système virtualisé 114
Au niveau de l'hyperviseur 114
V.3.1.2 Niveaux de validation 115
Implantation au niveau du MMV 115
Implantation au niveau d'un binaire 115
Implantation au niveau d'une machine virtuelle sûre 115
V.3.2 La TCB : entre complexité, validation et environnement distribué 116
V.3.2.1 Décomposition de la TCB 116
V.3.2.2 Adaptation au contexte distribué 117

V.4 Conclusion 119

Chapitre VI Cas d'etude et mise en oeuvre 121

Introduction 122

VI.1 1er cas d'etude : l'ordinateur de maintenance 123
VI.1.1 Cadre d'utilisation 123
VI.1.1.1 Tâches de maintenance classique 123
VI.1.1.2 Opérations de maintenance avec support électronique 124
VI.1.2 Analyse des flux de données 126
VI.1.3 Application du modèle Totel à l'ordinateur de maintenance 127

VI.2 2ème cas d'étude : l'ordinateur du pilote 130
VI.2.1 Cadre d'utilisation 130
VI.2.1.1 Calcul classique du profil de décollage 130
VI.2.1.2 Nouvelle utilisation de l'ordinateur du pilote 131
VI.2.2 Analyse des flux de données 132
VI.2.3 Application du modèle Totel à l'ordinateur du pilote 133

VI.3 Considérations de mise en œuvre 135
VI.3.1 Capture des flux ascendants des applications redondantes en vue de leur validation 135
VI.3.1.1 Implémentation de la diversification par virtualisation 135
VI.3.1.2 Capture au niveau du matériel 137
VI.3.1.3 Capture au niveau de l'hyperviseur 138
VI.3.1.4 Capture au niveau des appels des machines virtuelles Java 138
VI.3.1.5 Capture au niveau des entrées graphiques des bibliothèques java 139
VI.3.2 Interactions avec les applications diversifiées 139
VI.3.2.1 Gestion de l'interaction avec l'opérateur 139
VI.3.2.2 Cas des applications existantes 140

VI.4 Mise en œuvre ArSec 142
VI.4.1 Interception des appels *Java Swing* et *Awt* 142
VI.4.1.1 Description d'une machine virtuelle non sûre 142
VI.4.1.2 Description d'une machine virtuelle sûre 143
VI.4.1.3 Capture des sorties et exécution 144
VI.4.2 Contrôle au niveau d'une machine virtuelle sûre 145
VI.4.3 Déterminisme dans ArSec 145
VI.4.4 Description du démonstrateur ArSec 146

VI.5 Conclusion **148**
VI.6 Recommandations pour les applications futures **149**
Chapitre VII Conclusion Générale ***151***
VII.1 Bilan **152**
VII.2 Perspectives **154**
Références bibliographiques ***155***
Annexes techniques ***163***
Introduction **164**
Annexe A : Exécution du prototype **165**
Annexe 2 : Exemple de capture d'une fonction de connexion **170**
Annexe 3 : Evaluation par simulation d'attaques **171**

Chapitre I

INTRODUCTION GENERALE

Au courant du mois d'avril 2008, un rapport de l'agence fédérale de l'aviation (*FAA :Federal Aviation Agency*) a déclenché une vague d'articles et d'analyses essayant de corréler le retard de livraison du *Dreamliner 787* de l'avionneur américain avec des vulnérabilités supposées, mentionnées dans ce rapport. Les vulnérabilités en question seraient liées à une possible communication entre le réseau mis à disposition des passagers et les réseaux avioniques en charge de commander l'avion. Cet événement médiatique met en évidence l'ouverture des systèmes avioniques sur les systèmes informatiques et les réseaux plus classiques, généralement appelés *systèmes ouverts*, et souligne de nouvelles problématiques relatives à la sécurité (au sens *security* ou sécurité-immunité) qui viennent s'ajouter aux contraintes de *safety* ou sécurité-innocuité déjà connues dans le domaine avionique.

Cette importance croissante de la composante sécurité-immunité, c'est-à-dire la prise en compte d'éventuelles malveillances lors de la réalisation et de l'exploitation des systèmes embarqués critiques, est due à plusieurs facteurs :

- *L'utilisation accrue de composants sur étagère* (COTS), matériels et logiciels : ces composants offrent à faible coût des fonctionnalités très larges, qui permettent de mettre en œuvre facilement des applications efficaces et performantes. Ils présentent des avantages importants : ils sont utilisés en grand nombre dans des contextes très différents, ce qui augmente leur maturité, beaucoup de fautes de conception ont déjà été détectées et corrigées, et leur fabrication est bien maîtrisée. Cependant, de tels composants n'ont pas été conçus pour des applications critiques, ce qui rend difficile leur certification. De plus, ces composants sur étagère sont souvent plus complexes qu'il ne serait nécessaire pour les seules applications embarquées critiques, ce qui peut les rendre plus vulnérables à des attaques. D'autre part, leur large diffusion fait qu'il est facile pour des attaquants éventuels d'en obtenir une connaissance détaillée, y compris sur leurs vulnérabilités.

- *L'ouverture des réseaux et des applications* : dans le transport aérien, les communications numériques bord-sol se multiplient (contrôle de trafic aérien, compagnie, passagers), voire directement d'avion à avion ; on voit aussi cette tendance se généraliser dans les transports routiers et ferroviaires. Cela procure davantage d'opportunités pour un attaquant de s'introduire dans un système embarqué.

- *La complexité croissante* des systèmes embarqués multiplie le risque d'introduire des vulnérabilités susceptibles d'être exploitées par des attaquants, d'autant plus que les méthodes de développement des applications critiques ne prennent pas forcément en compte le risque de malveillance.

- Enfin, les *menaces sont croissantes*, en particulier avec le développement du terrorisme international, mais aussi du *hacking*. Pour le transport aérien ou ferroviaire, il faut prendre en compte les malveillances éventuelles des passagers ou du personnel des compagnies, voire du personnel des constructeurs ou des équipementiers. Dans l'automobile, il faut même prendre en compte la volonté de certains propriétaires de customiser le système embarqué dans leur véhicule pour en améliorer les performances.

Ce constat a déjà conduit la communauté aéronautique internationale à s'organiser afin de mettre à jour les normes existantes ou développer de nouvelles normes pour prendre en compte les problèmes de sécurité vis-à-vis des malveillances.

Pour nos travaux, nous nous plaçons dans un contexte avionique, où des applications ayant différents niveaux de criticité doivent interagir pour fournir de nouveaux services que les avionneurs cherchent à offrir pour répondre aux demandes de leurs clients. Une telle interaction ne peut se faire sans soulever les problématiques de sécurité-immunité et de sécurité-innocuité et de leurs interactions. En effet, un composant peu critique est généralement conçu et développé sans exigences spécifiques relatives à l'immunité ou l'innocuité. Un tel composant est donc a priori peu robuste vis-à-vis d'attaques malveillantes qui visent à le corrompre. Permettre une communication directe entre un tel composant peu critique et un composant critique résulterait en une contamination potentielle de ce dernier. Or cette contamination pourrait conduire à une défaillance du composant critique en question, ce qui aurait des conséquences catastrophiques sur le système. Il convient ainsi de noter le lien qui existe entre les notions de sécurité-immunité et de sécurité-innocuité : à travers une attaque, un système peut être corrompu (sécurité-immunité mise en défaut) et mettre en danger la vie de ses utilisateurs (sécurité-innocuité mise en défaut également).

Pour se prémunir contre ces risques de contamination, l'approche classique dans le monde avionique est d'isoler les systèmes critiques des autres systèmes, afin d'éviter toute interaction. Une autre approche plus récente consiste à permettre uniquement des communications unidirectionnelles depuis les composants critiques vers les composant moins critiques, dans un schéma de communication qui rappelle le fonctionnement des diodes dans les circuits électroniques.

Remarquons que ces deux approches restent peu flexibles, et ne permettent pas une réelle interaction fonctionnelle entre les systèmes critiques et les systèmes moins critiques. Ainsi, comme nous l'avons précédemment mentionné, nous proposons, dans le cadre de nos travaux, d'étendre ces communications inter domaines de criticité en développant des architectures qui prennent en compte aussi bien les aspects immunités qu'innocuité. Cette flexibilité est obtenue à travers l'utilisation de mécanismes de tolérance aux fautes qui permettent d'augmenter la confiance que l'on peut avoir dans le résultat fourni par un composant peu critique. De plus, nous proposons d'exploiter la technologie de virtualisation afin d'implémenter ces mécanismes de tolérance aux fautes.

Ce manuscrit est organisé comme suit : le chapitre II présente un état de l'art sur les systèmes avioniques, avec notamment les contraintes liées à la sécurité-innocuité et à la sécurité-immunité, et également une description synthétique des principales techniques de tolérance aux fautes qui s'inscrivent dans le domaine de sûreté de fonctionnement. Nous présentons ensuite, dans le chapitre III, une vision cohérente entre les notions de criticité et de confiance, en nous basant sur les standards existants pour les systèmes critiques. Cette vision permet de dégager une classification en niveaux de criticité des composants du système, et nous présentons les différents modèles qui ont traité des interactions entre composants à niveaux distincts d'intégrité. Le chapitre IV présente la technologie de virtualisation et son lien avec les techniques de tolérance aux fautes. La virtualisation est utilisée pour répondre à certaines hypothèses de fautes que nous établissons dans le cadre de ces travaux pour développer une architecture sûre de fonctionnement qui est présentée au chapitre V. Nous proposons également dans ce chapitre une formalisation d'un modèle de contrôle d'intégrité (le modèle Totel) pour en proposer une extension et une adaptation au contexte réparti. Le chapitre VI décrit la mise en œuvre de cette architecture pour deux cas d'étude que nous avons identifiés en concertation avec Airbus, et donne une vue globale du démonstrateur développé dans le cadre de nos travaux. Le chapitre VII conclut le manuscrit, avec un bilan des travaux réalisés et une perspective pour les travaux futurs. Une annexe complète ce document en détaillant l'implémentation du démonstrateur.

Chapitre II

AVIONIQUE ET TOLERANCE AUX FAUTES : ETAT DE L'ART

INTRODUCTION

Le domaine du transport aérien est historiquement lié à la notion de sécurité des passagers. Cette contrainte de sécurité a conduit à des exigences de conception et de construction des appareils de manière à ce que ce moyen de transport soit au moins aussi sûr que les autres moyens de transport en commun classiques (par exemple, le ferroviaire)

Les moyens de transport font partie d'une famille de systèmes considérés comme des systèmes critiques (au même titre que les centrales nucléaires, par exemple). La notion de criticité est liée à la gravité des conséquences d'une défaillance affectant le système. Ainsi, un train est considéré comme système critique parce qu'en cas de défaillance, les conséquences peuvent entraîner des pertes de vies humaines, ce qui correspond à un événement catastrophique. Il en est clairement de même pour un avion.

Un système avionique est donc un système critique qui doit répondre aux exigences de sécurité requise par de tels domaines d'application. Par sécurité, nous désignons ici la sécurité des passagers (donc sécurité au sens *safety*, ou encore sécurité-innocuité). Cependant, depuis quelques années, une nouvelle contrainte de sécurité s'est imposée dans le monde avionique : la sécurité au sens immunité (connue par le terme *security*). En effet, depuis l'apparition des premiers systèmes de commande de vol électrique (CDVE) dans les années 1980, les entités logicielles à bord sont de plus en plus présentes. L'intégration de composants logiciels de différent niveaux de criticité et de plus en plus complexes (y compris des composants sur étagères ou COTS) et l'ouverture des systèmes bord vers d'autres systèmes (par exemple, communications numériques) induisent des risques liés non seulement aux fautes accidentelles (*bugs*) pouvant affecter ces logiciels, mais aussi à l'exploitation de vulnérabilités potentielles par des actes malveillants.

Dans ce chapitre, nous proposons de détailler dans un premier temps quelques normes avioniques ainsi que l'évolution des applications avioniques embarquées liée aux avancées importantes dans le monde du logiciel durant les deux dernières décennies. Les systèmes critiques devant être sûrs de fonctionnement, nous proposons alors de présenter la terminologie et définitions du domaine de sûreté de fonctionnement. Nous mettrons enfin l'accent sur la composante sécurité-immunité qui a guidé notre réflexion tout au long de ce manuscrit pour concevoir une architecture avionique sûre de fonctionnement.

II.1 APPLICATIONS AVIONIQUES : ETAT DE L'ART

Le processus d'évaluation de la sécurité d'un appareil a pour but de vérifier la conformité de la conception avec les exigences de navigabilité spécifiées par les autorités[2]. Chaque constructeur apporte ses propres directives allant souvent au-delà de ces exigences de sécurité (exemple ADB d'Airbus Directives chez le constructeur Airbus). Des recommandations ou des normes internationales ont été émises pour rendre standards ces exigences. Nous proposons dans cette section de présenter les standards avioniques liés à la sécurité-innocuité (ARP 4754, DO-178B) et ensuite les nouvelles architectures modulaires émergentes dans l'avionique (IMA, ARINC 653). Nous présenterons ensuite les contraintes de sécurité-immunité (ARINC 811) liées aussi bien au déploiement de telles architectures mais également à l'interfaçage des applications avioniques avec le monde ouvert.

II.1.1 Standards avioniques pour la sécurité-innocuité

La principale norme qui a guidé le développement des systèmes avioniques sûrs (au sens *safety*) est la norme ARP 4754. Elle concerne la notion de sécurité dans les systèmes avioniques complexes, et de cette norme ont découlé diverses normes et recommandations pour le développement avionique (cf. figure II.1).

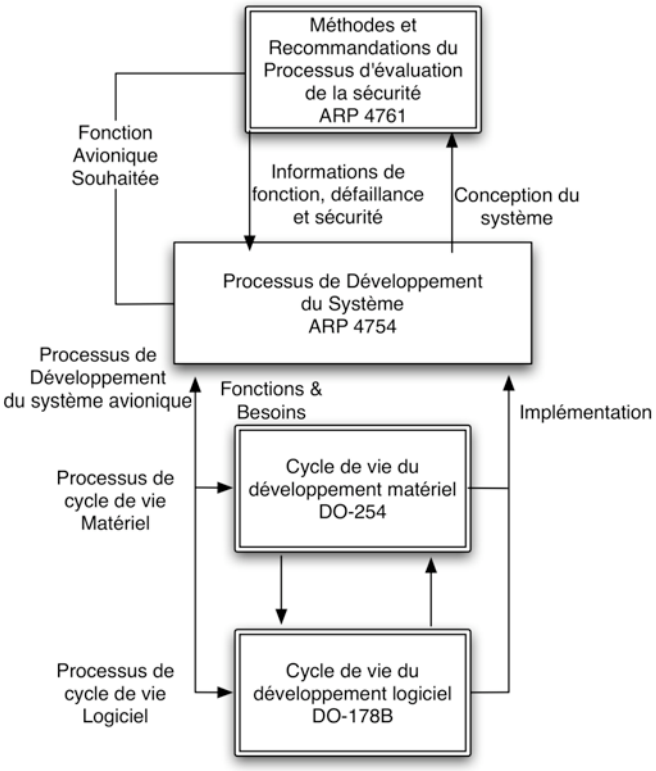


Figure II.1. La norme ARP 4754 et liens avec les autres normes de sécurité avionique

[2] Ces autorités sont spécifiques aux pays du constructeur. Ainsi en Europe, il s'agit de l'agence européenne de la sécurité aérienne EASA (*European Aviation Safety Agency*), pour les Etats Unis, il s'agit de l'administration fédérale d'aviation (*Federal Aviation Administration*). Un agrément doit être accordé par les autres autorités aéronautiques dans le monde.

Comme le montre la Figure II.1, la norme ARP 4754 concerne tout le processus de développement des systèmes avioniques complexes. Par système, cette norme désigne un ensemble de fonctions avioniques, basées en grande partie sur des entités logicielles, et nécessitant une forte interaction les unes avec les autres. Pour le développement de tels systèmes, on distingue entre le processus de développement d'applications logicielles et du matériel. Ainsi, la DO-178B [DO178-B 1992] propose des recommandations pour le développement des logiciels, et la DO-254 [DO-254 2000] est lié au matériel. Afin d'évaluer le processus de développement d'un point de vue sécurité-innocuité, des recommandations ont été émises dans la norme ARP 4761 [ARP 4761].

Nous proposons de détailler les aspects liés à l'évaluation de la sécurité-innocuité dans les systèmes avioniques (ARP 4754), puis lors du processus de développement logiciel (DO-178B).

II.1.1.1 Sécurité des systèmes avioniques : ARP 4754

Le processus d'évaluation de la sécurité tel qu'il est décrit dans cette norme comprend la génération, la mise à jour et la vérification d'exigences en parallèle avec des activités de développement de l'avion. Il fournit une méthodologie pour évaluer les fonctions de l'avion et la conception des systèmes qui remplissent ces fonctions, afin de déterminer si les risques associés ont bien été pris en compte. Le processus d'évaluation de la sécurité est qualitatif et quantitatif.

Le processus doit être planifié et supervisé pour fournir l'assurance nécessaire que toutes les conditions de défaillances (*Failure Condition : FC*) aient été identifiées et que toutes les combinaisons significatives de ces conditions aient été considérées.

Ce processus d'évaluation de la sécurité pour des systèmes intégrés doit tenir compte de toute complexité ou interdépendance complémentaire qui surgit en raison de l'intégration. Dans tous les cas impliquant des systèmes intégrés, le processus d'évaluation de la sécurité a une importance fondamentale pour établir les objectifs de sécurité appropriés pour le système et déterminer si la mise en oeuvre les satisfait.

L'évaluation de la sécurité se décompose en quatre étapes :

- évaluation des risques fonctionnels (*Functional Hazard Assessment : FHA*) : cette évaluation concerne les fonctions avion et système afin de déterminer les défaillances fonctionnelles possibles et classifier les risques associés à des conditions de défaillances spécifiques.
- évaluation préliminaire de la sécurité des systèmes (*Preliminary System Safety Assessment* : *PSSA*) : cette étape établit les exigences de sécurité spécifiques aux systèmes et aux composants des systèmes et fournit une première évaluation de la conformité de l'architecture avec les exigences de sécurité à satisfaire. La PSSA est mise à jour tout au long du processus de développement de chaque système.
- évaluation de la sécurité des systèmes (*System Safety Assessment : SSA*) : la SSA vérifie que le système implémenté satisfait aux exigences formulées dans la FHA et la PSSA.
- analyse des causes communes (*Common Cause Analysis : CCA*) : complément des FHA, PSSA, SSA, la CCA permet de minimiser les risques liés à des causes communes au système ou à plusieurs systèmes ; elles concernent entre autres les aspects : installation des systèmes, facteurs humains…

Les résultats de la FHA sont les données initiales de la PSSA, et de même pour les résultats de la PSSA vis-à-vis de la SSA. La Figure II.2 montre les principales relations entre les quatre étapes précisées ci-dessus et le processus de développement des systèmes.

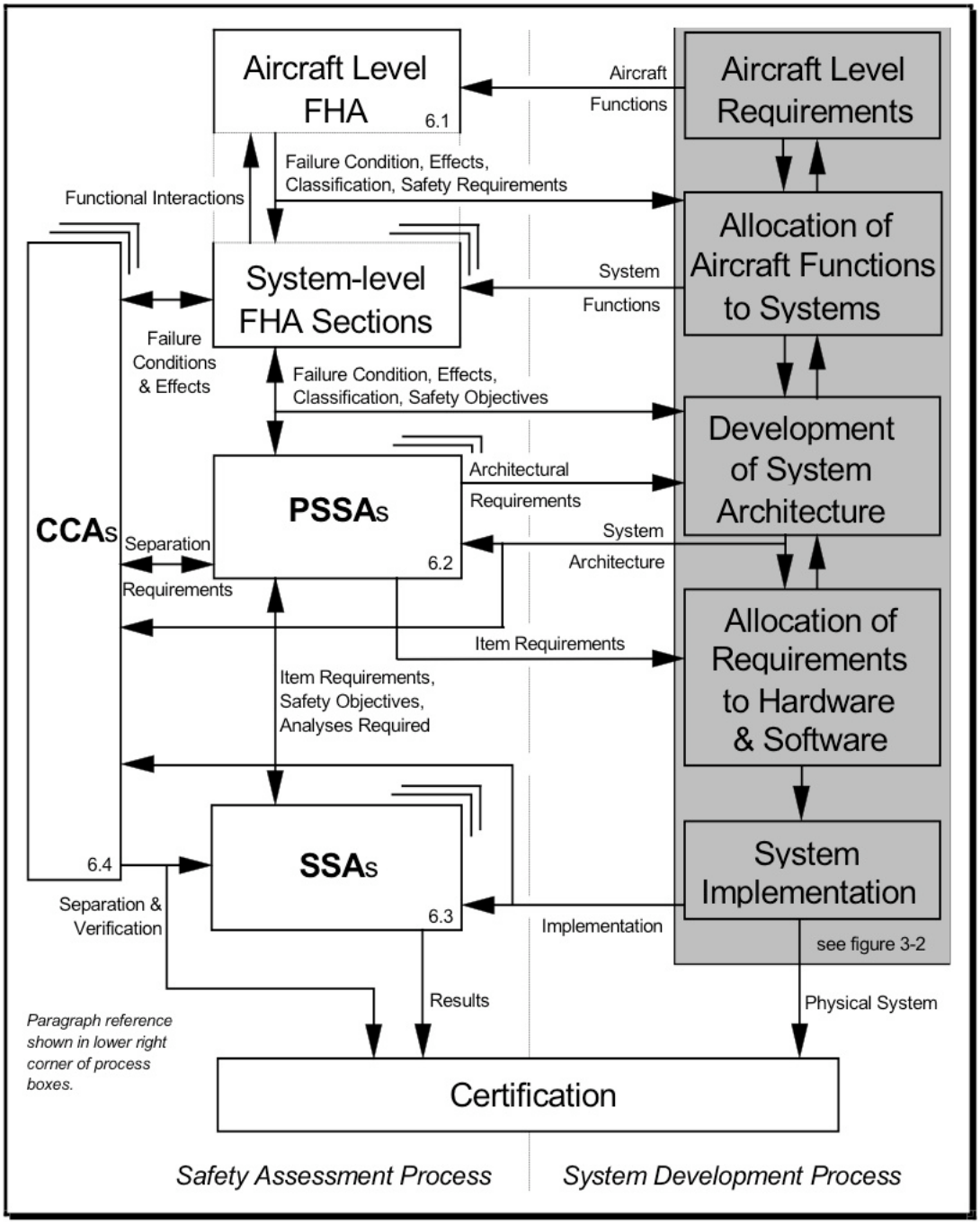


Figure II.2. Modèle du processus d'évaluation de la sécurité innocuité (extrait de [ARP 4754 1996])

Nous proposons de détailler dans ce qui suit les trois premiers types d'évaluations introduits dans la norme ARP 4754, à savoir la FHA, PSSA et PSA, le quatrième étant liée aux points communs existant dans les trois premières.

Evaluation des risques fonctionnels (FHA)

Une FHA correspond à un examen complet systématique des fonctions afin d'identifier et de classifier les défaillances de ces fonctions selon leur sévérité. Il existe deux types de FHA : la FHA avion (*Aircraft Level FHA*) et la FHA système (*System Level FHA*).

La FHA avion est une étude de haut niveau, une évaluation qualitative des fonctions principales de l'avion définies au début du développement de l'appareil. Une FHA avion identifie et classifie les conditions de défaillances (*FC*) associées aux fonctions de niveau avion. La classification de toutes ces FC définit les exigences de sécurité que doit satisfaire l'avion.

La FHA système est également une évaluation qualitative. Elle est conduite de façon itérative de manière à ce que cette évaluation soit de plus en plus précise au fur et à mesure que les fonctions de l'avion évoluent. La FHA système est liée à une défaillance ou une combinaison de défaillances qui affectent les fonctions du système puis de l'avion. Lorsque les fonctions de l'appareil ont été affectées à des systèmes par le processus de conception, chaque système fait l'objet d'une FHA système.

Evaluation préliminaire de la sécurité des systèmes (PSSA)

Une PSSA sert à démontrer comment le système est susceptible de satisfaire aux exigences, qualitatives et quantitatives, relatives aux différents risques identifiés. La PSSA sert également à compléter la liste de FC et les exigences de sécurité correspondantes. La PSSA identifie les moyens utilisés permettant d'atteindre les objectifs de sécurité. Cette analyse identifie et recense toutes les exigences de sécurité système dérivées (par ex : l'autotest, les tâches à accomplir,…).

La PSSA constitue une analyse itérative dans la continuité des FHA. Elle permet aussi d'allouer au niveau équipement les exigences sécurité du niveau système. Pratiquement, il s'agit d'identifier les fautes qui contribuent aux FC recensées dans la FHA système en utilisant des arbres de fautes ou des diagrammes de dépendance.

Le calcul de probabilité d'une FC diffère selon que les défaillances sont ou non cachées. Par exemple, le temps durant lequel un équipement de secours peut défaillir ou être défaillant avant d'être détecté et réparé doit être considéré. Dans de nombreux cas, les défaillances sont décelées par l'équipage : soit par observation directe, soit par des tests. Dans certains cas, la latence de détection est associée à l'intervalle de temps entre tests de l'équipement en atelier ou entre tâches de maintenance avion. Ces tâches et intervalles de temps sont identifiés durant la PSSA et vérifiés durant la SSA, en utilisant des arbres de fautes ou des diagrammes de dépendance.

Evaluation de la sécurité des systèmes (SSA)

Une SSA est une analyse itérative complète des fonctions systèmes implémentées destinée à montrer que les exigences de sécurité sont satisfaites. Le processus est similaire à celui de la PSSA du point de vue des activités, mais différent dans l'intention : une PSSA évalue des architectures proposées et en déduit des exigences de sécurité, alors qu'une SSA vérifie que l'architecture sélectionnée satisfait aux exigences qualitatives et quantitatives définies dans la FHA et la PSSA.

Une SSA combine les résultats des différentes analyses menées pour vérifier la sécurité du système complet et couvrir toutes les considérations spécifiques de sécurité identifiées dans la PSSA.

Comme l'avons mentionné, la norme ARP 4754 (cf. Figure II.1) est liée au développement des systèmes avioniques. Dans le cadre de nos travaux, nous nous intéressons à la composante logicielle des systèmes avioniques. Les recommandations de développement d'applications logicielles sûres de fonctionnement dans le domaine avionique sont décrites dans la norme DO-178B que nous proposons de présenter dans la section suivante.

II.1.1.2 Sécurité des entités logicielles en avionique DO-178B

Dans l'avionique, la certification des systèmes embarqués met principalement l'accent, d'un point de vue de la sûreté de fonctionnement, sur la sécurité-innocuité. La prise en compte explicite des aspects liés à la sécurité-immunité fait l'objet de travaux en cours pour faire évoluer la norme (DO-178C).

La démarche générale repose sur un processus d'analyse de la sécurité-innocuité (intégrant des études de risques préliminaires, fonctionnelles et architecturales) qui consiste à analyser les fonctions et l'architecture des systèmes avioniques et à identifier leurs modes de défaillance. Ceux-ci sont classés en fonction d'une échelle de sévérité (vis-à-vis des conséquences de ces défaillances sur la sécurité des vols et des passagers) qui comprend cinq niveaux: catastrophique, dangereux, majeur, mineur et sans effet. La criticité d'un système est alors déterminée par la plus forte sévérité de ses modes de défaillance. On définit alors cinq niveaux d'assurance de développement (*Development Assurance Level DAL*) notés DAL-A, DAL-B, DAL-C, DAL-D et DAL-E par ordre décroissant. Ainsi, les logiciels critiques sont développés au niveau DAL-A alors que les logiciels qui n'ont aucun impact sur la sécurité de l'appareil sont développés au niveau DAL-E (correspondant généralement au niveau des applications existantes dans le monde ouvert). Cette classification détermine ainsi le niveau d'assurance que le système doit satisfaire. Pour chacun de ces niveaux, un ensemble de critères et de preuves est requis.

Comme nous l'avons précédemment indiqué, la norme DO-178B est consacrée au processus de développement des logiciels embarqués (cf. Figure II.1). Elle est basée sur une approche orientée processus. On distingue trois types de processus dans le cycle de vie du logiciel :

- le processus de développement, qui comprend la spécification, la conception, le codage et l'intégration ;
- les processus intégraux, qui comprennent la vérification, la gestion de configuration, l'assurance qualité et la coordination pour la certification ;
- le processus de planification qui coordonne le processus de développement et les processus intégraux.

Pour chaque processus et chaque niveau d'assurance, les objectifs sont définis et une description des données du cycle de vie permettant de démontrer que les objectifs sont satisfaits est également fournie. La norme n'impose aucune méthode pour atteindre les objectifs fixés. C'est à la charge du postulant à la certification (donc les constructeurs avioniques) d'apporter les preuves que les méthodes employées permettent de répondre aux objectifs. De même, bien que la norme ne l'impose pas, elle recommande d'utiliser des stratégies de tolérance aux fautes qui peuvent être employées pour la détection et le recouvrement d'erreurs. Le tableau II.1 résume le lien entre les différents niveaux de DAL présents dans la DO-178B et les objectifs en termes de probabilité et de sévérité de défaillance.

Dérivé du FHA	Objectifs de Sécurité-innocuité

Classification des FC	Niveau DAL	Fail-Safe	Besoins Quantitatifs
Catastrophique	A	Exigé	$P<10^{-9}$
Dangereux	B	Peut être requis	$P<10^{-7}$
Majeur	C	Peut être requis	$P<10^{-5}$
Mineur	D	Non	Aucun
Sans effet sur la sécurité-innocuité	E	Non	Aucun

Tableau 1 **Sévérité de défaillance et lien avec les objectifs de probabilité et de niveaux DAL**

Afin d'atteindre l'objectif de probabilité d'erreur inférieure à 10^{-9} (comme le montre le Tableau II.1), les systèmes avioniques se basent habituellement sur la redondance pour implanter des systèmes tolérant les fautes, comme nous allons le détailler dans la Section II.2. Il convient en particulier de souligner que la notion de redondance logicielle diversifiée a été identifiée et détaillée dans la norme DO-178B. Nous proposons de présenter les considérations liées à une telle diversification logicielle, sachant que ces considérations guideront les analyses de fautes menées dans le cadre de l'étude présentée dans ce manuscrit.

II.1.1.3 Considérations pour la diversification des logiciels selon la DO-178B

L'utilisation d'une diversification (cf. Section II.2.2.3 pour la terminologie utilisée) permet d'augmenter le niveau de confiance que l'ont peut avoir dans le résultat fourni par l'ensemble des logiciels diversifiés. Cependant, l'ajout de mécanismes de diversification doit s'accompagner d'un processus de vérification qui doit en particulier prouver qu'un tel ajout n'a pas d'effet de bord sur les propriétés de sécurité-innocuité du système, telles qu'elles ont été définies par les différentes évaluations présentées dans la Section II.1.1.1. La diversification logicielle est ainsi assurée en combinant les règles de diversification suivantes :

- Le code source est implanté en utilisant deux ou plusieurs langages de programmation différents.
- Le code exécutable est généré en utilisant un ou plusieurs compilateurs différents.
- Chaque version de code exécutable est exécutée sur un processeur différent. Dans le cas où un seul processeur serait utilisé, il faut s'assurer du bon fonctionnement des mécanismes d'isolation entre les versions exécutées.
- Les besoins logiciels, la phase de conception et d'implantation sont réalisées sur plus de deux environnements de développement. De plus, il est possible que chaque version mise en œuvre soit vérifiée sur des environnements de test différents.
- Le code exécutable est lié (*linked*) puis chargé (*loaded*) en utilisant deux ou plusieurs éditeurs de liens et chargeurs.
- Les besoins logiciels, la conception et l'implémentation de chaque version doit être réalisée en utilisant respectivement des standards de besoins logiciels (*Software Requirements Standards*), standards de conception et standards de code source différents.

Ainsi ces différentes règles peuvent être combinées pour implanter des applications logicielles diversifiées. Dans ce cas, le processus de vérification dépend de la nature de la règle (ou des règles) choisie pour la diversification, en particulier en fonction de la diversification introduite au niveau matériel et/ou logiciel. Cependant, il faut également montrer que chaque version est correcte, et que les versions sont également compatibles entre elles, vis-à-vis des spécifications (et ce pour un fonctionnement normal ou défectueux du système). De plus, il faut montrer que la solution diversifiée est au moins aussi performante qu'un système non diversifié par rapport à la détection d'erreurs.

Selon la norme ARP 4754, l'utilisation d'une diversification (logicielle et/ou matérielle) permet de réduire les efforts de validation pour réaliser une tâche d'un niveau de criticité élevé. Cette vision est différente de celle requise dans la DO-178B dans laquelle on demande qu'au moins l'une des versions soit développée au même niveau que celui de la tâche la plus critique. Nous présenterons plus en détails cet aspect lié à la diversification dans le Chapitre III.

Remarquons que pour la norme ARP 4754 présentée, les fonctions avioniques possèdent des niveaux de DAL qui leur sont propres. Ces fonctions sont développées généralement sur un composant matériel dédié à l'exécution des applications implantant ces fonctionnalités. Cependant, depuis quelques années, une nouvelle génération de calculateurs permet d'intégrer sur un même module physique, plusieurs applications à criticité disjointes. Il s'agit des architectures IMA (*Integrated Modular Avionics*). Cette approche est fort intéressante puisqu'elle rejoint la problématique traitée dans ce manuscrit, à savoir la communication entre applications à multiples niveaux de criticité. Nous proposons dans ce qui suit de présenter plus en détail les architectures IMA.

II.1.2 Architectures IMA

Les architectures IMA sont des architectures avioniques qui permettent de répondre à des besoins que les architectures avioniques classiques ne possédaient pas. Ces besoins sont essentiellement d'ordre économique (coûts de développement, de possession, de maintenance, d'immobilisation, etc.) Nous proposons de présenter dans un premier temps ces architectures classiques, puis détailler les caractéristiques d'une architecture IMA.

II.1.2.1 Systèmes avioniques classiques

Les premières fonctions logicielles utilisées en avionique sont des fonctions dédiées à une tâche bien spécifique dans l'appareil. Chaque fonction est associée à un ensemble d'entités matérielles en charge d'exécuter cette fonction. De tels systèmes sont appelés systèmes fédérés (*federated systems*) puisque la partie logicielle d'un tel système est fortement liée au matériel sous-jacent. Les normes avioniques classiques (celles présentées dans la section précédente) utilisent cette vision de systèmes avioniques lors des processus d'évaluation de la sécurité-innocuité de tels systèmes. Par exemple, pour calculer le pire temps d'exécution (*WCET : Worst Case Execution Time*) d'une fonction avionique, il faut prendre en considération aussi bien les aspects applicatifs propres à la fonction que les interactions qui existent avec le matériel d'exécution.

De tels systèmes ont été largement utilisés aussi bien dans le monde avionique que dans le domaine spatial. Ainsi, dans la station spatiale internationale (*International Space Station : ISS*), un grand nombre d'entités logicielles fédérées a été utilisé. Ces entités sont appelées unités orbitales remplaçables (*Orbital Replacable Units : ORU*) et correspondent aux unités de lignes remplaçables du monde avionique (Line Replaceable Units : LRU). Ces unités sont

conçues pour des fonctions bien spécifiques (pouvant intégrer quelques modes de configuration), et sont facilement remplaçables en cas de défaillance.

Cependant, cette vision classique de systèmes fédérés présente des inconvénients (liées à cette forte dépendance entre le logiciel et le matériel) que nous résumons dans les points suivants :

1. La durée de vie d'un avion est de l'ordre de plusieurs dizaines d'années (25-30 ans, voire plus). Cette durée de vie est longue relativement à la durée de vie d'un matériel. En effet, vu les avancées technologiques actuelles, un matériel devient obsolète au bout seulement de quelques années. Dans de nombreux cas, le matériel planifié durant la conception peut même devenir obsolète au moment de la construction de l'appareil. De plus, même si ce matériel est mis à jour, il faut bien s'assurer de la portabilité du logiciel qui n'a pas été originellement développé pour ce nouveau matériel. Cette tâche s'avère particulièrement coûteuse vu le lien étroit qui existe entre le logiciel et le matériel dans un système fédéré.
2. L'hétérogénéité des matériels utilisés dans les applications avioniques. En effet, chaque application possède ses propres contraintes fonctionnelles, et est exécutée sur un matériel qui lui a été conçu. Assurer la maintenance matérielle dans ces conditions est une tâche fort coûteuse, d'un point de vue économique.
3. La communication entre systèmes dépend des applications et également du matériel d'exécution. Toute mise à jour d'une entité logicielle implique une modification importante dans le mécanisme de communication, entraînant souvent la modification des autres entités logicielles pour permettre une interopérabilité.

L'expérience dans le monde avionique montre que même si les systèmes fédérés sont robustes vis-à-vis des défaillances (compte tenu de la forte indépendance entre les supports d'exécution matériels des fonctions avioniques), ils sont cependant lents et difficiles à développer et à mettre à jour [Alena et al. 2007].

II.1.2.2 Caractéristiques des architectures IMA

Les architectures IMA ont été proposées pour répondre à ce besoin de dissocier les composants logiciels des composants matériels dans les systèmes fédérés. Cette architecture présente une plateforme unique pour toutes les applications logicielles avioniques. La plateforme consiste en une architecture temps-réel distribué, permettant à différentes applications de s'exécuter en parallèle. Ces applications partagent alors les ressources fournies par la plateforme IMA (ressource de calcul, de mémoire,…). Il est donc clair qu'une telle architecture doit assurer un partage de ressource sûr entre les applications. Dans ce cas, il est question d'une isolation forte entre les modules logiciels, ou encore connue dans la littérature par partitionnement. Ce partitionnement est classé en partitionnement spatial (par exemple, la gestion de la mémoire) et temporel (la gestion des ressources temporelles par la CPU) [J. Rushby 2000]. Cependant, nous trouvons que cette classification ne présente aucune limite claire entre la notion de spatial et temporel. Dans le cadre de ce manuscrit, nous désignerons par partitionnement (ou isolation) tous les mécanismes mis en place pour assurer une gestion des ressources qui ne mettent pas en défaut le fonctionnement des applications utilisant ces ressources.

Ainsi, avec cette vision du partitionnement, une architecture IMA n'est autre qu'un gestionnaire de ressources matérielles pour des applications avioniques embarquées. Ces applications sont implantées dans des modules pouvant avoir des niveaux de criticité hétérogènes. L'architecture IMA se doit d'être sûre afin de permettre un tel partitionnement entre applications à criticités multiples. Ainsi, dans [ARINC 653 1997], le partitionnement

doit assurer une isolation totale entre des applications ayant des niveaux de DAL distincts. L'isolation est réalisée en introduisant des partitions qui sont des espaces d'exécution logiques attribués à chaque fonction avionique. L'accès aux ressources (CPU, mémoire,…) est ensuite contrôlé pour qu'il n'y ait aucune altération du fonctionnement des entités les plus critiques (des entités du niveau DAL-A ou DAL-B, par exemple).

Remarquons qu'une telle architecture est similaire à l'architecture d'un système d'exploitation classique. En effet, un système d'exploitation offre aux différentes applications qui y sont installées de s'exécuter en parallèle, indépendamment de la couche matérielle. Une architecture IMA, tout comme un système d'exploitation, offre des services (par exemple service de communication) aux applications qui y sont installées. Ces services sont accessibles via une interface de programmation des applications (*Application Programming Interface* : *API,* cf. Chapitre IV) unique pour toutes les fonctions avioniques. Cependant, les architectures IMA se distinguent des systèmes d'exploitation classiques par les fortes contraintes auxquelles elles sont soumises lors de leur développement, ainsi que de l'aspect distribué des différentes partitions dans l'appareil.

Ainsi, une plateforme IMA est constituée d'un certain nombre de modules distribués dans l'avion. Ces modules sont connectés entre eux et également avec des périphériques de l'avion (typiquement des périphériques d'entrée/sortie comme des capteurs ou des interfaces homme machine). La Figure II.3 présente l'aspect en couche d'un module IMA.

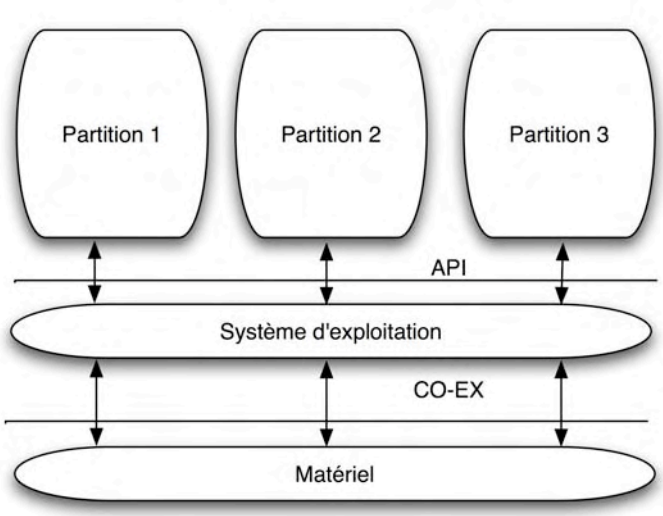


Figure II.3. Module IMA selon l'ARINC 653

Comme le montre la Figure II.3, le module est constitué d'un système d'exploitation en charge d'exécuter plusieurs partitions. La communication entre les partitions et le système d'exploitation du module se fait via l'API. Le système du module communique avec la matériel via un noyau exécutif (*Core Executive* : CO-EX). Ce noyau offre une transparence du support matériel pour le système d'exploitation du module. Notons également que la seule couche visible pour une application dans une partition donnée est la couche API.

Cette abstraction est intéressante est présente les avantages suivants :

- Compte tenu de la transparence du matériel par rapport aux fonctions avioniques, il est possible d'utiliser du matériel différent pour assurer une diversification matérielle (cf. section suivante pour la tolérance aux fautes), sans que cela n'impacte le fonctionnement de l'application. Cette diversification serait donc gérée au niveau du

système d'exploitation qui se charge de traduire les instructions applicatives en instructions matérielles.

- La robustesse du partitionnement fait qu'une application peut être développée et testée d'une façon incrémentale, sans que cela n'altère le fonctionnement des autres partitions.
- Grâce à l'interopérabilité des modules, en cas de défaillance d'un module, il est possible de reconfigurer une application logicielle pour qu'elle s'exécute sur un second module, ce qui présente une propriété fort intéressante en terme de disponibilité de l'appareil.

Les architectures IMA sont déjà utilisées pour des fonctionnalités spécifiques sur certains appareils. Par exemple, le système d'exploitation DEOS (*Digital Engine Operating System*) [Libin Dong et al. 1999] qui implémente l'IMA a été utilisé dans des avions commerciaux certifiés. Cependant, la certification des systèmes avioniques utilise toujours la vision des systèmes fédérés. La DO-178B ne présente le partitionnement que dans le cas d'une séparation logique entre sections d'une application unique. Ainsi, certifier un système IMA en se basant sur cette vision fédérée implique un effort de validation considérable, ce qui rend l'utilisation de tels systèmes coûteuse (d'un point de vue économique) [Conmy et McDermid 2001].

Comme nous avons pu le constater, la sécurité, au sens innocuité, est l'une des premières exigences qui a guidé le développement des fonctions avioniques (aussi bien au niveau matériel que logiciel). Cependant, depuis quelques années (surtout après les événements du 11 septembre 2001), la composante sécurité-immunité est devenue de plus en plus présentes dans le monde avionique. Nous proposons de détailler quelques pistes de réflexions sur l'aspect immunité. Ces pistes sont encore en cours d'analyse et d'investigation dans le monde avionique.

II.1.3 Sécurité-immunité dans les systèmes avioniques

Depuis plusieurs années, le développement des architectures avioniques tend vers une intégration de plus en plus importante de composants issus des technologies de l'information grand public (typiquement des composants sur étagères ou aussi communément identifiés par le vocable COTS : *Commercial-Off-The-Shelf*). De plus, l'augmentation du nombre des modules IMA dans les appareils durant les dernières années prouve que la tendance d'utiliser des fonctionnalités étendues s'impose sur le terrain des constructeurs avioniques.

L'utilisation croissante des composants COTS est accompagnée d'un besoin de communication bidirectionnel entre les modules avioniques et le monde ouvert. En effet, il est fort souhaitable d'utiliser toutes les fonctionnalités étendues qu'offrent les applications COTS dans des modules avioniques, et ces fonctionnalités nécessitent dans la majorité des cas une forte interaction avec le monde ouvert. Nous présentons dans un premier temps l'approche sécurité-immunité en avionique. Nous présentons ensuite les considérations de sécurité-immunité déjà existantes au bord des appareils. Nous détaillons enfin les communications bord-sol et les aspects de sécurité qui y sont liés.

II.1.3.1 L'approche sécurité-immunité dans l'avionique

L'approche de la sécurité-immunité dans l'avionique consiste à traiter les menaces qui pourraient avoir un impact sur la sécurité-innocuité des fonctions avioniques. Ainsi, une menace est appréhendée en fonction de son influence sur l'altération des propriétés de *safety*

de l'appareil. Ces menaces sont classées alors selon différents niveaux, comme le montre le tableau II.2.

Symbole	Effet	Définition
V	**Pas d'effet**	N'affecte pas le fonctionnement de l'avion et n'ajoute pas de charge de travail à l'équipage
IV	**Mineur**	Légère réduction des marges de sécurité-innocuité ou des fonctions avionique ou légère augmentation de la charge de travail de l'équipage ou légère diminution du confort des passagers ou équipage.
III	**Majeur**	Réduction significative des marges de sécurité-innocuité ou des fonctions de l'avion ou augmentation significative de la charge de travail de l'équipage ou manque de confort ou stress physique des passagers ou équipage, avec possibilité de blessures.
II	**Dangereux**	Réduction importante des marges de sécurité-innocuité ou des fonctions de l'avion ou augmentation importante de la charge de travail de l'équipage (qui ne peut plus remplir correctement ses fonctions) ou blessures importantes voire fatales d'un des passagers.
I	**Catastrophique**	Immobilité ou blessures fatales d'un membre de l'équipage, avec perte de l'appareil.

Tableau 2 **Classification des menaces selon l'impact sur la sécurité-innocuité**

Remarquons que cette classification est liée à celle proposée par la DO-178B. Les menaces classifiées dans le tableau présentent des conséquences sur les paramètres de sécurité-innocuité du vol [ED-127 2009]. Les paramètres principaux sont les suivants :

- Réduction des marges de sécurité-innocuité de l'appareil, ou des capacités fonctionnelles de l'appareil.
- Augmentation de la charge de travail de l'équipage ou altération des conditions de travail de l'équipage, réduisant ainsi sa capacité à remplir correctement ses fonctions.
- Détresse ou blessure des occupants de l'appareil.

Les menaces considérées sont essentiellement d'origine humaine (utilisation inappropriée, attaque délibérée, faute accidentelle, résultant d'un accès non autorisé, etc). La menace humaine peut être couplée avec d'autres facteurs (environnementaux, par exemple) pour conduire à l'aboutissement de l'attaque. Pour pallier ces menaces, les systèmes internes à l'avion sont développés de façon à ce qu'il y ait des contraintes de communication entre les différents composants avioniques, afin de réduire les risques de contamination en cas d'attaque réussie, comme nous allons le montrer dans la section suivante.

II.1.3.2 Contraintes de sécurité-immunité dans l'avion

Les systèmes avioniques ont été développés en respectant la classification de la DO-178B en affectant à chaque composant logiciel un niveau de criticité. Cette même vision de niveaux de criticité est utilisée dans [ARINC 811 2005] pour distinguer différents domaines de sécurité au sein de l'avion [Olive et al. 2006]. Comme le montre la Figure II.4, quatre domaines sont définis en fonction de différents critères : sécurité (existence de contrôle ou pas), responsabilité (constructeur, compagnie aérienne, passager), opération aérienne (secrète, privée, publique) et rôle (commande de l'avion, exploitation avionique, information et

divertissement des passagers). Nous présentons dans ce qui suit les différents domaines identifiés dans la Figure II.4 et détaillés dans le rapport ARINC 811, sans pour autant spécifier les spécifications internes de chaque domaine :

- Le domaine de commande de l'appareil (*Aircraft Control Domain*) est le domaine le plus critique dans le monde avionique. Il contient les applications de contrôle et de commande de l'appareil. Ces applications sont généralement installées sur des calculateurs qui récupèrent les commandes du pilote, les transforment (en fonctions des lois de pilotage et des données environnementales par exemple) en données numériques, et les communiquent (via un réseau dédié) aux différents actionneurs de l'appareil.
- Le domaine de services d'information de la compagnie (*Airline Information Services Domain*) contient les différents supports relatifs au vol, la cabine, la maintenance (cf. Figure II.4). Ce domaine est moins critique que le précédent, mais il reste tout de même d'un niveau assez élevé de criticité. En effet, les informations contenues dans ce domaine sont critiques pour l'exploitaton de l'appareil (surtout par rapport à la maintenance) par les compagnies aériennes.
- Le domaine de services d'information et de divertissement des passagers (*Passenger Information and Entertainment Services Domain*) est en charge de la communication avec les passagers. Ainsi la gestion des écrans de divertissement (*In Flight Entertainment : IFE*) ainsi que l'interface de connexion (à l'Internet par exemple) des périphériques du passager. Ce domaine est fortement lié à l'image de marque de la compagnie aérienne auprès des passagers. Aussi, les compagnies accordent une grande importance aux équipements de divertissement des passagers, certaines allant même jusqu'à refuser l'autorisation de décollage d'un appareil si un terminal de divertissement (*IFE*) est en panne.
- Le domaine des équipements des passagers (*Passenger-owned Devices*) est réservé à tous les équipements électroniques des passagers (ordinateurs portables, PDA, téléphones, consoles de jeu,…). Dans certains avions récents, ces équipements sont connectables, via des interfaces spécifiques (présentes dans le domaine précédent), à un réseau que la compagnie aérienne peut configurer en fonction de sa stratégie commerciale (par exemple proposer une connexion Internet aux passagers).

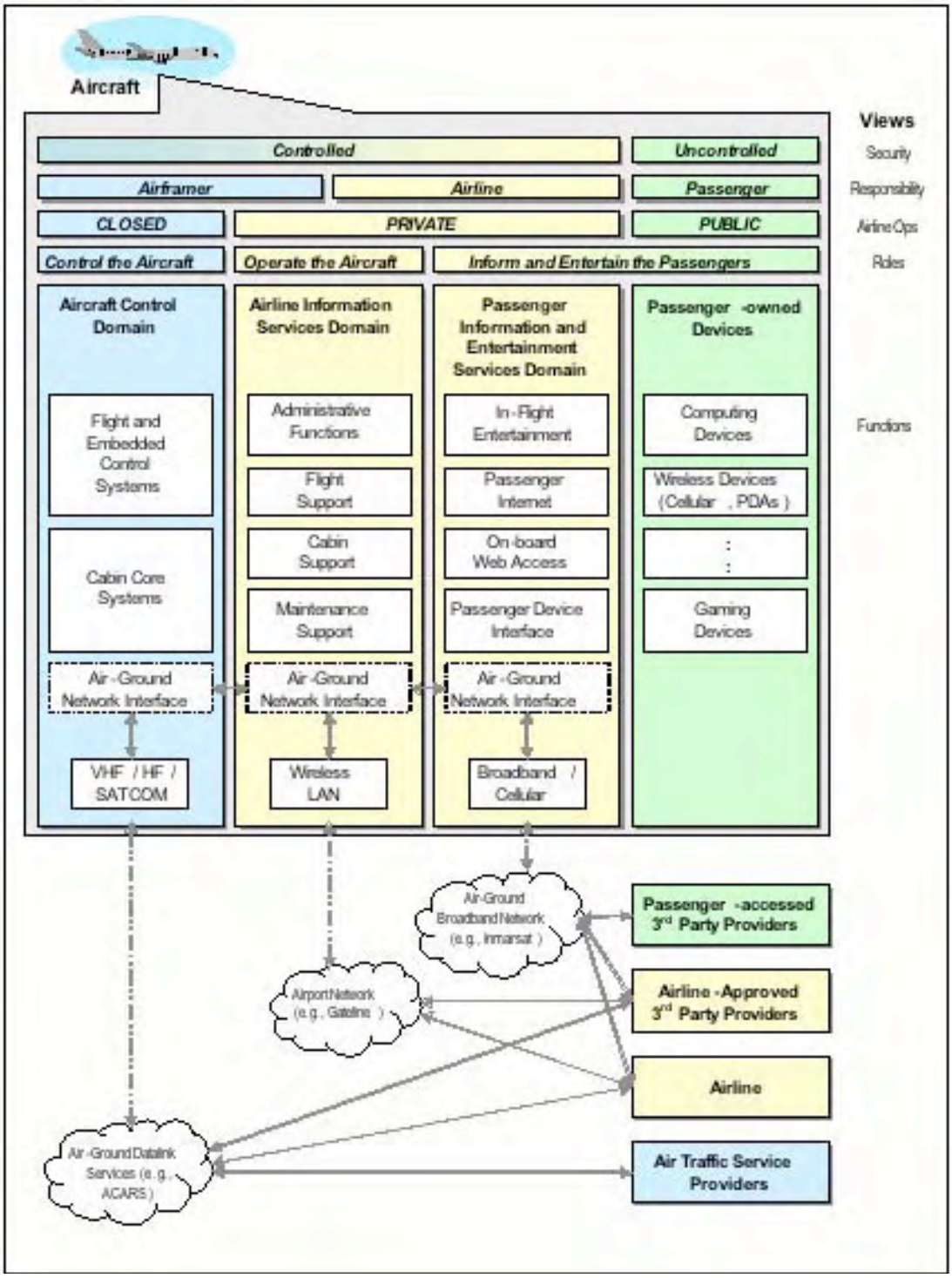


Figure II.4. Différents domaines avioniques et leurs connexions (ARINC 811)

La communication inter-domaine est complètement contrôlée et ne permet pas de remontée de flux d'un niveau peu critique (par exemple le domaine des équipements des passagers) à un niveau critique (par exemple le domaine de commande). Une telle isolation assure ainsi la sécurité-immunité des applications critiques vis-à-vis des attaques, en supposant que toute information arrivant à ces applications critiques est de confiance.

Remarquons que chaque domaine a également la possibilité de se connecter à des réseaux externes bien identifiés. Ces communications sont appelées communications bord-sol et sont détaillées dans la section suivante.

II.1.3.3 Communications bord-sol et contraintes de sécurité-immunité

Certaines communications bord-sol sont considérées comme historiques, et remontent à l'utilisation des ondes radio pour communiquer des informations (par exemple des informations relatives à la position de l'avion) aux instruments de bord. De nos jours, ces communications sont plus diversifiées, aussi bien par rapport à leur utilisation que par rapport au moyen de communication utilisé (HF, VHF, satellite, WiFi…,) (Figure II.5).

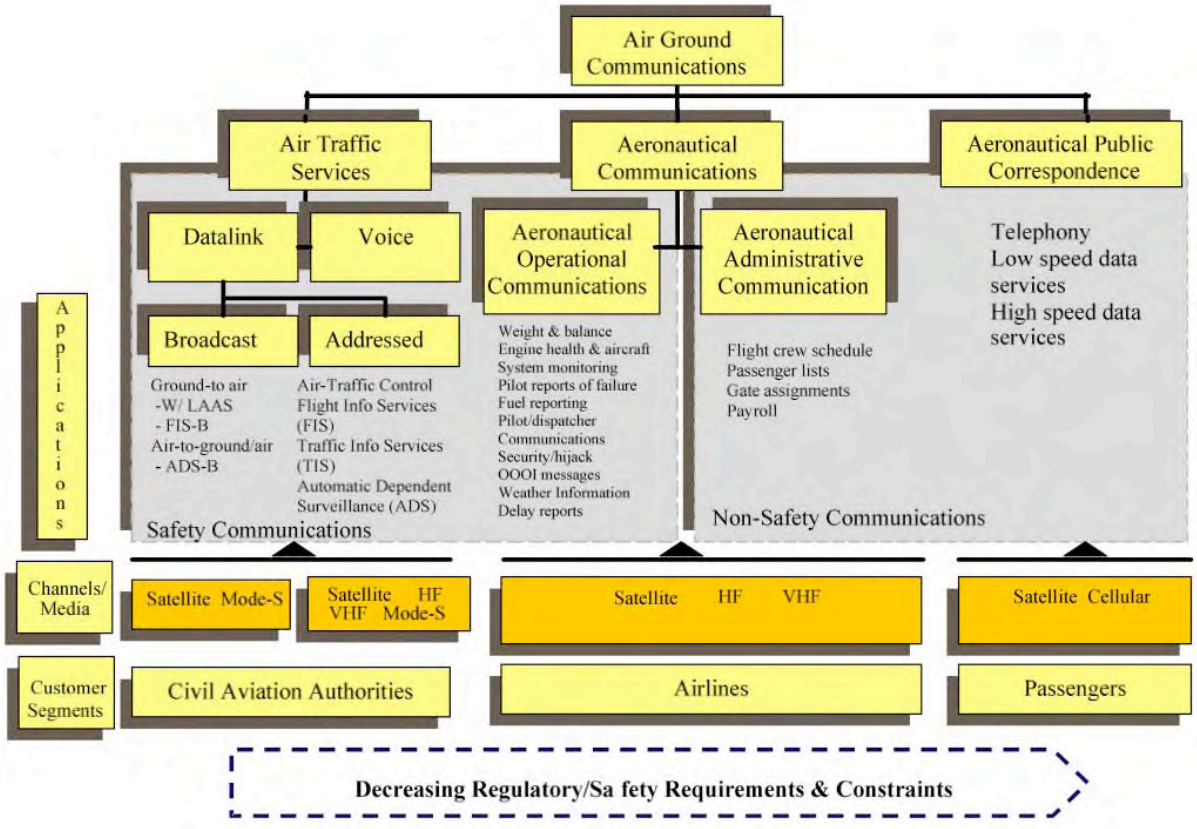


Figure II.5. Différentes communications bord sol (ARINC 811)

Comme l'illustre la Figure II.5, l'ARINC 811 distingue 4 catégories de communications bord-sol :

- L'*Air Traffic Service (ATS*) est en charge des informations de trafic aérien.
- Les *Aeronautical Communications* (AC) sont utilisées par les compagnies aériennes. Ces communications sont décomposées en communications opérationnelles et administratives :
    - Les *Aeronautical Operational Communications* (AOC) permettent de récolter des informations relatives à l'état opérationnel de l'appareil (par exemple, poids total, équilibrage de la masse,…). Elles sont donc essentielles pour le bon déroulement du vol de l'appareil aussi bien en terme de sécurité-innocuité qu'en terme de disponibilité.

- Les *Aeronautical Administrative Communications* (AAC) concernent l'aspect gestion administrative du vol (par exemple le point de stationnement, la liste des passagers, l'emploi du temps de l'équipage,…).

- L'*Aeronautical Public Correspondence* concerne essentiellement les communications liées aux passagers (par exemple, les communications téléphoniques ou informatiques qu'ils peuvent réaliser alors qu'ils sont dans l'appareil).

Comme on le voit sur la Figure II.5, ces communications n'ont pas toutes les mêmes exigences en terme de sécurité-innocuité et ne sont donc pas réglementées de la même manière. Ainsi, les ATS sont les plus critiques et obéissent à des réglementations bien spécifiques qui dépendent des autorités de l' Aviation Civile. Les APC, quant à elles, ne sont pas réglementées et n'ont pas un impact sur la sécurité-innocuité de l'appareil. La Figure II.6 présente une vue globale des domaines avioniques ainsi que des différents types de communication utilisés.

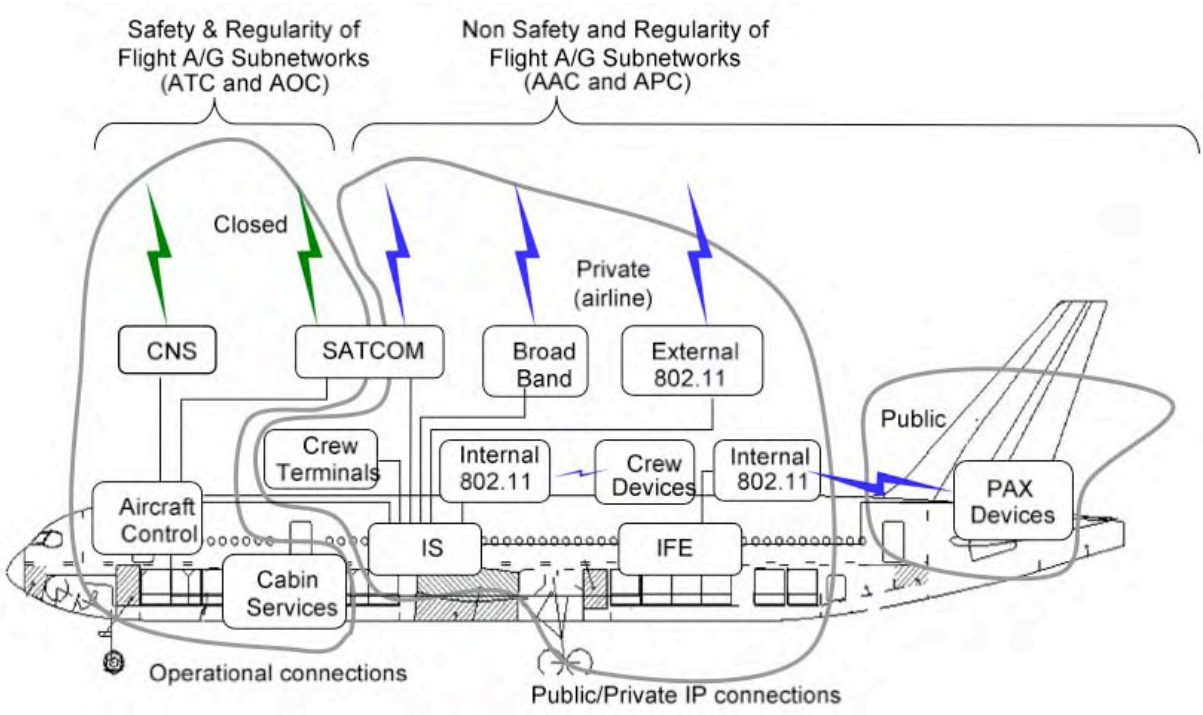


Figure II.6. Domaines avioniques et types de communications (ARINC 811)

Comme nous l'avons précédemment mentionné, cette approche ségrégative des communications dans le monde avionique permet d'assurer une immunité des systèmes avioniques vis-à-vis des malveillances. Cependant, cette approche est trop restrictive puisqu'elle ne permet pas une interaction entre entités appartenant à des domaines avioniques distincts.

Dans le cadre de ce manuscrit, nous nous intéressons aux interactions entre applications avioniques ayant des niveaux de criticités hétérogènes. Ainsi, ces applications appartiennent à des domaines avioniques disjoints, et possèdent des niveaux de DAL distincts également.

L'attribution d'un niveau de DAL à une fonction avionique est réalisée après une analyse de risques du système, visant à rendre l'appareil sûr de fonctionnement. Nous proposons de

présenter dans la section suivante la terminologie liée à la sûreté de fonctionnement, et plus particulièrement aux techniques de tolérance aux fautes.

II.2 SURETE DE FONCTIONNEMENT : CONCEPTS ET TERMINOLOGIE

Nous avons vu dans la section précédente que le monde avionique est fortement lié aux différentes notions de sécurité (au sens innocuité ou immunité). Ces notions s'inscrivent pleinement dans le domaine de la sûreté de fonctionnement que nous présentons dans cette section. Pour ce faire, il convient d'introduire une partie du vocabulaire, en considérant les concepts et la terminologie introduits dans le « Guide de la Sûreté de Fonctionnement » [Jean-claude Laprie et al. 1996] et mis à jour dans [A Avizienis et al. 2004]. Plus précisément, nous commençons par définir les notions de sûreté de fonctionnement, faute, erreur et défaillance pour ainsi aborder avec plus d'aisance la définition de la tolérance aux fautes accidentelles et aux malveillances, ce qui nous permet de situer nos travaux par rapport au thème et aux moyens classiques de la sûreté de fonctionnement en général et de la sécurité informatique, en particulier.

II.2.1 Définitions

II.2.1.1 La sûreté de fonctionnement :

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. Le service délivré par un système est son comportement tel qu'il est perçu par ses utilisateurs. Ces derniers peuvent être humains ou physiques. La sûreté de fonctionnement informatique s'articule autour de trois principaux axes : les attributs qui la caractérisent, les entraves qui empêchent sa réalisation et les moyens de l'atteindre (cf. Figure II.7).

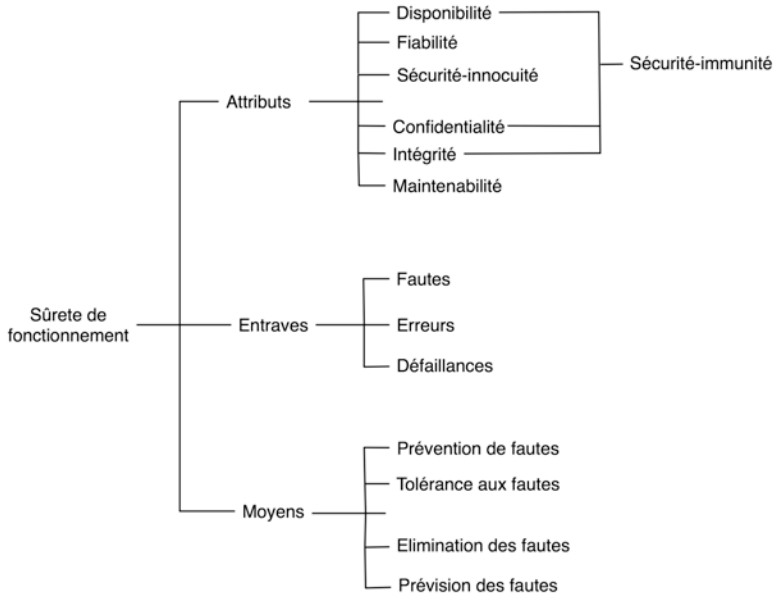


Figure II.7. Arbre de la sûreté de fonctionnement

La sûreté de fonctionnement d'un système peut être perçue selon différentes propriétés. Ces propriétés sont appelées les *attributs* de la sûreté de fonctionnement. Les attributs à considérer

dépendent des applications auxquelles le systèmes est destiné. Les six attributs sont les suivants :

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

Les attributs précédents peuvent être mis à mal par des entraves, présentés dans la section suivante.

II.2.1.2 Entraves à la sûreté de fonctionnement :

Une entrave à la sûreté de fonctionnement est une circonstance indésirable mais non inattendue. Elle est la cause ou le résultat de la non-sûreté de fonctionnement. Nous en distinguons trois sortes : les défaillances, les erreurs et les fautes.

Une **défaillance** survient lorsque le service délivré par le système dévie de l'accomplissement de la fonction du système. Toutefois, un système ne défaille pas toujours de la même façon, ce qui conduit à définir la notion *de mode de défaillance* qui peut être caractérisée selon trois points de vues: domaine de défaillance, perception des défaillances par les utilisateurs du système, conséquences des défaillances sur l'environnement du système.

Le domaine de défaillance est divisé en deux catégories :

- Les défaillances en valeur : la valeur du service délivré ne permet plus l'accomplissement de la fonction du système;
- Les défaillances temporelles : les conditions temporelles de délivrance du service ne permettent pas l'accomplissement de la fonction du système.

La perception des défaillances conduit à distinguer, lorsqu'un système a plusieurs utilisateurs :

- Les défaillances cohérentes : tous les utilisateurs du système ont la même perception des défaillances ;
- Les défaillances incohérentes : les utilisateurs du système peuvent avoir des perceptions différentes des défaillances ; les défaillances incohérentes sont souvent qualifiées de défaillances byzantines.

Les conséquences des défaillances conduisent à distinguer :

- Les défaillances bénignes, dont les conséquences sont du même ordre de grandeur que le bénéfice procuré par le service délivré en l'absence de défaillance ;
- Les défaillances catastrophiques, dont les conséquences sont incommensurablement différentes du bénéfice procuré par le service délivré en l'absence de défaillance.

Comme nous l'avons précédemment vu, la sévérité de la défaillance d'un composant définit son niveau de criticité, et par conséquent fixe les exigences de développement qui sont requises. Dans le Chapitre III, nous présenterons le lien entre ces différentes notions de criticité, sévérité, niveaux de validation, etc.

Les **erreurs** sont le deuxième type d'entraves à la sûreté de fonctionnement. Une erreur est la partie de l'état du système qui est susceptible d'entraîner une défaillance : une erreur affectant le service est une indication qu'une défaillance survient ou est survenue.

Une **faute** est la cause adjugée ou supposée d'une erreur. Les fautes sont de nature extrêmement diverses et peuvent être classées selon cinq points de vue : leur cause phénoménologique (fautes physiques ou fautes dues à l'homme), leur nature (fautes accidentelles ou fautes délibérées), leur phase de création ou d'occurrence (fautes de développement ou fautes opérationnelles), leur situation par rapport aux frontières du système (fautes internes ou externes) et leur persistance (fautes permanentes ou fautes temporaires). Une faute est **active** lorsqu'elle produit une erreur. Une faute active est soit une faute interne qui était préalablement *dormante* et qui a été activée par le processus de traitement, soit une faute externe. Une faute interne peut cycler entre ses états dormant et actif. Une erreur peut être latente ou détectée ; une erreur est *latente* tant qu'elle n'a pas été reconnue en tant que telle ; une erreur est *détectée* par un algorithme ou un mécanisme de détection. Une erreur peut disparaître sans être détectée. Par propagation, une erreur crée de nouvelles erreurs. Une défaillance survient lorsque, par propagation, une erreur affecte le service délivré par le système. Cette défaillance peut alors apparaître comme une faute du point de vue d'un autre composant. On obtient ainsi la chaîne fondamentale suivante :

…→ défaillance → faute → erreur → défaillance → faute → …

Les flèches dans cette chaîne expriment la relation de causalité entre fautes, erreurs et défaillances. Elles ne doivent pas être interprétées au sens strict : par propagation plusieurs erreurs peuvent être créées avant qu'une défaillance ne survienne.

Pour minimiser l'impact de ces entraves sur les attributs retenus d'un système, la sûreté de fonctionnement dispose de moyens.

II.2.1.3 Moyens de la sûreté de fonctionnement :

Ces moyens sont les méthodes et techniques qui permettent de conforter les utilisateurs quant au bon accomplissement de la fonction du système. Ces moyens peuvent être utilisés simultanément lors de la phase de conception et développement d'un système sûr de fonctionnement. Ils sont classés en quatre moyens suivant l'objectif visé:

- La prévention : empêcher l'occurrence ou l'introduction de fautes.
- La tolérance : fournir un service qui remplit la fonction du système en dépit des fautes.
- L'élimination : réduire la présence (nombre, sévérité) des fautes.
- La prévision : estimer la présence, la création et les conséquences des fautes.

Ces moyens sont en fait complémentaires et dépendants et doivent être utilisés de façon combinée. En effet, en dépit de la prévention des fautes grâce à des méthodes de conception et à des règles de construction rigoureuses, des erreurs surviennent, résultant en des fautes. D'où le rôle de l'élimination des fautes : lorsqu'une erreur est générée durant la vérification, un diagnostic est entrepris afin de déterminer la, ou les fautes causes de l'erreur, en vue de les éliminer. Cette élimination étant elle-même imparfaite, il est également nécessaire d'effectuer de la prévision de fautes. Et, bien entendu, la dépendance croissante dans les systèmes informatiques conduit à mettre en œuvre de la tolérance aux fautes.

Dans le cadre de nos travaux, nous nous intéressons essentiellement à la tolérance aux fautes. Nous faisons donc l'hypothèse que des fautes peuvent toujours exister, et faisons en sorte que les architectures que nous étudions soient robustes vis-à-vis de ces fautes. Nous proposons de

détailler dans ce qui suit la tolérance aux fautes, et de préciser son utilisation dans le cadre de notre travail.

II.2.2 La tolérance aux fautes

Comme nous l'avons précédemment indiqué, la tolérance aux fautes a pour but d'empêcher les défaillances malgré la présence ou l'occurrence de fautes. La tolérance aux fautes est mise en œuvre par le traitement des erreurs et par le traitement des fautes [Lee et Anderson 1990]. Le traitement d'erreur est destiné à éliminer les erreurs, de préférence avant qu'une défaillance ne survienne. Le traitement de faute est destiné à éviter qu'une ou des fautes ne soient activées de nouveau. Dans le cadre de nos travaux, nous nous intéressons au traitement d'erreurs. Ce traitement consiste à :

- Détecter les erreurs : à travers l'identification d'un état erroné suite à une activation de faute.
- Recouvrir les erreurs : en substituant un état exempt d'erreur à un état erroné.

Rappelons que les fautes peuvent être classifiées en fautes accidentelles (physiques ou humaines) et fautes délibérées (bienveillante et malveillante) [Yves Deswarte et al. 1998]. Les fautes délibérées bienveillantes sont toujours déclarées parce qu'elles font partie d'une décision d'implémentation justifiée par diverses contraintes[3].

Nous proposons de détailler dans les sections suivantes les techniques de détection et de recouvrement pour les fautes accidentelles et malveillantes, résumées dans le Tableau II.3.

II.2.2.1 Tolérance aux fautes accidentelles

Les fautes accidentelles peuvent être d'origine physique (par exemple phénomène de rayonnement cosmique) ou humaine (fautes affectant la conception du système, ou son utilisation). Dans ce qui suit, nous présentons les principales techniques de détection et recouvrement pour les fautes physiques et de conception.

Mécanismes de détection:

Les mécanismes de détections (1-4) sont relatifs aux fautes physiques et les mécanismes (5-6) sont relatifs aux fautes de conception :

1. La détection via ces **codes détecteurs d'erreurs** [Peterson et Weldon 1972] consiste à transformer les données (les coder) en ajoutant de la redondance de manière à ce qu'une erreur à la réception (décodage) soit directement détectée. La capacité de détection d'erreurs est liée à la complexité du code. Le codage, le décodage et la détection provoquent un surcoût en termes de temps de calcul qui peut devenir non négligeable pour certaines applications.
2. **La réplication** consiste à faire accomplir la même tâche par plusieurs unités qui doivent être indépendantes par rapport au processus d'activation des fautes : si une même faute est susceptible d'engendrer une erreur dans les différentes unités, les erreurs doivent être différentes — succeptibles d'être discriminées au moins relativement. Il est ainsi souhaitable de placer les unités dans des endroits géographiquement distants ou de décaler les exécutions dans le temps afin d'éviter que l'activation d'une faute externe ne provoque les mêmes erreurs. En revanche, pour

[3] Un exemple typique d'une telle faute est l'envoi d'un satellite dans l'espace sans une protection totale contre les rayons cosmiques. Il s'agit bien d'une faute délibérée (parce qu'elle est intentionnelle et réfléchie) et bienveillante (pour des raisons économiques, par exemple).

tolérer des fautes physiques internes, on peut utiliser des unités identiques et synchronisées, et il suffit de comparer bit à bit les résultats obtenus pour détecter une erreur. Cette comparaison sous-entend un synchronisme entre les unités considérées et un déterminisme dans le comportement des répliques. À défaut, l'utilisation de fonctions de décision plus complexes permet de détecter la présence d'erreurs.

3. Les **chiens de garde** (*watchdogs*) [Namjoo 1983] permettent de contrôler le temps d'exécution d'une tâche ou le temps de réponse d'un périphérique donné. Il s'assure que ce temps ne dépasse pas une valeur maximale précédemment définie (*time-out*). Cette technique a pour avantage de détecter certaines erreurs en ligne et est caractérisée par un faible coût.
4. **Les contrôles de vraisemblance** consistent à vérifier si les données sont conformes à certaines règles générales ou définies en fonction de l'application. Ces contrôles peuvent être effectués au niveau matériel (division par zéro, accès à une zone mémoire interdite ou inexistante,...) ou au niveau logiciel en les intégrant au niveau du système d'exploitation ou même au niveau de l'application (*cf.* programmation défensive).
5. L'utilisation de **la programmation défensive** [Rabéjac 1995] consiste à ajouter des contrôles sur les entrées et les sorties des modules sous la forme d'assertions, servant à détecter s'il y a erreur et éventuellement déclencher un traitement d'exceptions. Le traitement d'exceptions constitue alors une manière d'éviter qu'une défaillance d'une tâche n'entraîne la défaillance de tout le système.
6. Le ***Run-time model-checking*** consiste à vérifier, durant l'exécution d'une application, que toutes les étapes du modèle théorique de l'application sont exécutées. Cette vérification est assurée en exécutant un programme qui parcourt le graphe d'état traduisant le modèle en même temps que l'exécution de l'application. Chaque résultat intermédiaire correspond à un état dans le graphe du modèle et l'on vérifie ainsi la cohérence entre l'exécution du programme et le modèle.

Mécanismes de recouvrement:

Parmi les mécanismes de recouvrement nous distinguerons les mécanismes relatifs aux fautes physiques (1-4) et ceux relatifs aux fautes de conception (5-7) :

1. **La reprise** consiste à relancer l'exécution à partir d'un état sain préalablement sauvegardé. Il est clair que la sauvegarde nécessite un espace de stockage protégé et qui peut être important et que le rétablissement du système dans un état antérieur et la ré-exécution à partir de cet état induit des retards dans l'exécution globale des tâches.
2. **La poursuite** -repose sur le même principe que la reprise dans le sens où l'on essaie de mettre le système dans un état correct, mais en acceptant cette fois de perdre certaines données, voire certaines étapes de traitement, quitte à dégrader temporairement les fonctionnalités du système. Ainsi, la poursuite consiste souvent à réinitialiser le système en essayant d'acquérir de nouvelles données depuis l'environnement externe d'exécution.
3. **La détection-compensation** permet, lors de la détection d'une erreur, de reconstituer un état exempt d'erreur, afin d'être en mesure de poursuivre le traitement, ce qui nécessite généralement une redondance forte. Un exemple est celui de l'utilisation en parallèle de deux composants autotestables : un composant est considéré comme unité principale le second est une copie à laquelle on fait appel en cas de détection d'erreurs. La compensation consiste alors à commuter d'une unité à une autre.

4. **Le masquage**, contrairement à la compensation qui fait suite à la détection d'une erreur, est exécuté de façon continue, qu'il y ait ou non erreur. L'utilisation d'un algorithme de vote (parmi les différentes options possibles) portant sur les résultats produits par un nombre suffisant de répliques constitue la principale technique pour implémenter le masquage (*TMP*, *Formalized Majority Voter*, *Generalized Median Voter*, *Formalized Plurality Voter*). Certains algorithmes de votes utilisent une connaissance préalable des sources des données fournies en entrées des voteurs, afin de les pondérer en fonction de leur intégrité. Ainsi, une source considérée fiable aura plus de poids lors du vote qu'une source qui est supposée moins fiable. (*Weighted Averaging Technique*) [Yves Deswarte et al. 1991].
5. **Les blocs de recouvrement** [B. Randell 1975] utilisent plusieurs variantes exécutées séquentiellement jusqu'à obtenir un résultat acceptable. Le décideur alors fait appel à la variante suivante s'il considère que le résultat fourni par une première variante est invalide. Dans le cas où aucune variante ne fournit de résultat valide, une alerte est levée pour traiter l'erreur à un niveau englobant.
6. **La programmation en N-versions** [A. Avizienis 1985] se base sur le même principe que les blocs de recouvrement, sauf que les variantes ne sont pas exécutées séquentiellement mais parallèlement. Dans ce cas, les résultats des variantes peuvent être différents, même en l'absence d'erreur, en raison de la diversification. Il faut donc traiter les résultats des variantes pour produire un résultat valide cohérent entre les variantes. Cette technique offre une bonne disponibilité du système, même en cas d'activation de fautes, puisque le surcoût en temps peut être négligeable ; cependant elle crée un surcoût de traitement permanent et régulier (N exécutions au lieu d'une). Les blocs de recouvrement minimise cette surcharge d'exécution, mais en cas d'activation de faute, une surcharge temporelle doit être prise en compte.
7. **La programmation N-autotestable** [Jean-Claude Laprie et al. 1990] met en œuvre des composants autotestables exécutés en parallèle. Chaque composant autotestable peut être un bloc de recouvrement ou même un composant N-versions. Comme pour la programmation en N-versions, les composants corrects ne fournissent pas nécessairement un résultat identique. Il faut donc traiter les résultats des variantes pour produire un résultat valide cohérent entre les variantes. Cependant, ce traitement des résultats est simplifié par la propriété d'autotestabilité des composants.

II.2.2.2 Tolérance aux fautes malveillantes

Les fautes malveillantes se déclinent en deux classes principales : les logiques malignes et les intrusions. Les logiques malignes sont des parties du système conçues pour provoquer des dégâts (bombes logiques) ou pour faciliter des intrusions futures (vulnérabilités créées volontairement). Elles peuvent être introduite dès la conception du système (par un concepteur malveillant), ou en phase opérationnelle (par l'installation d'un logiciel contenant un cheval de Troie ou par une intrusion). La définition d'une intrusion est étroitement liée aux notions d'attaque et de vulnérabilité :

- **Une attaque** est une faute d'interaction malveillante visant à violer une ou plusieurs propriétés de sécurité. Il s'agit d'une faute externe créée avec l'intention de nuire, y compris les attaques lancées par des outils automatiques (vers, virus, etc).
- **Une vulnérabilité** est une faute accidentelle ou intentionnelle (avec ou sans volonté de nuire) dans la spécification des besoins, la spécification fonctionnelle, la conception ou la configuration du système ou dans la façon selon laquelle il est utilisé. La vulnérabilité peut être exploitée pour créer une intrusion.

- **Une intrusion** est une faute malveillante interne d'origine externe, résultant d'une attaque qui a réussi à exploiter une vulnérabilité qui peut produire des erreurs pouvant provoquer une défaillance vis-à-vis de la sécurité, c'est-à-dire une violation de la politique de sécurité du système.

Ces fautes malveillantes, tout comme les autres types de fautes peuvent être tolérées en utilisant des mécanismes de détection et de recouvrement que nous présentons ci-dessous.

Mécanismes de détection

La détection de fautes malveillantes se fait soit à travers la détection d'intrusion, soit à travers l'authentification des utilisateurs qui ont le droit d'accéder au système.

La **détection d'intrusions** [Y. Deswarte et D. Powell 2006] est basée sur la comparaison entre le comportement observé du système et soit une référence de comportement normal (on parle alors de détection d'anomalie quand le comportement observé s'éloigne de la référence), soit une référence de comportements anormaux, basés sur des scénarios d'attaques connues (on parle alors de détection d'attaque lorsque le comportement observé correspond à une attaque connue).

L**'authentification** (des utilisateurs) et l'autorisation (vérification des droits d'accès) sont les moyens de contrôle d'accès classiques en informatique. Ils constituent un moyen pour détecter les malveillances (en plus d'empêcher les intrusions).

Recouvrement avec la fragmentation – redondance – dissémination:

Le mécanisme de fragmentation – redondance – dissémination est une technique qui permet de tolérer les intrusions tout en protégeant la confidentialité des informations sensibles. Elle consiste à découper l'information en fragments qui ne peuvent pas fournir, à eux seuls, des informations significatives. Ces fragments sont ensuite isolés par dissémination de manière à ce qu'une intrusion dans une partie du système ne fournisse que des fragments isolés. De la redondance est ajoutée aux fragments (par réplication ou par codage) pour permettre de détecter la modification ou la destruction de fragments, et de reconstituer l'information même si des fragments ont été modifiés ou détruits. [Yves Deswarte, Blain, et Fabre 1991].

Le Tableau 3 résume toutes les techniques de tolérance aux fautes présentées dans cette section. Nous remarquons qu'un nombre important de ces mécanismes utilisent plusieurs versions pour implémenter la détection et le recouvrement. Nous parlons donc de redondance. En fait, pour effectuer un traitement de fautes ou d'erreurs en vue de tolérer des fautes, il est obligatoire de disposer d'une redondance dans le système. Nous proposons de détailler cet aspect qui nous permettra de fixer la terminologie utilisée tout au long de ce manuscrit.

<table>
<tr><th></th><th>Tolérance aux fautes physiques</th><th>Tolérance aux fautes de conception</th><th>Tolérance aux malveillances</th></tr>
<tr><td rowspan="4">Détection</td><td>Codes détecteurs d'erreurs</td><td rowspan="2">Programmation défensive</td><td>Détection d'intrusions</td></tr>
<tr><td>Réplication / Comparaison</td><td>Détection d'anomalies
Détection d'attaques</td></tr>
<tr><td>Contrôle temporel d'exécution</td><td rowspan="2">Run-time model-checking</td><td rowspan="2">Authentification-Autorisation</td></tr>
<tr><td>Contrôle de vraisemblance</td></tr>
<tr><td rowspan="3">Recouvrement</td><td>Reprise</td><td>Blocs de recouvrement</td><td rowspan="3">Fragmentation-
Redondance-
Dissémination</td></tr>
<tr><td>Poursuite</td><td>N-versions</td></tr>
<tr><td>Compensation
Détection / compensation
Masquage</td><td>N-versions autotestables</td></tr>
</table>

Tableau 3 **Détection et recouvrement des fautes accidentelles et malveillantes.**

II.2.2.3 Tolérance aux fautes et redondance

Comme nous l'avons indiqué ci-dessus, il ne peut y avoir de tolérance aux fautes sans présence de redondance. En effet, pour décider de la validité d'un état ou pas (et par conséquent détecter une erreur), il faut disposer d'un critère permettant d'effectuer ce processus de décision. Le critère peut prendre la forme d'un autre état représentant la même information, ou la forme d'une valeur de référence à comparer avec la valeur observer. Cependant, quelle que soit la nature de ce critère, ce dernier doit contenir une information permettant de valider l'état en question, et cette information est toujours une forme de redondance de l'état contrôlé. Ainsi, une programmation N-versions utilise, comme son nom l'indique, plusieurs versions que l'on compare pour obtenir un résultat correct. La cohérence entre ces différentes versions présente alors le critère de validité précédemment mentionné. Pour les codes correcteurs d'erreurs, une information est ajoutée pour décider de la validité de la donnée transférée. Il s'agit également d'une redondance en terme d'information analysée.

Ainsi, la tolérance aux fautes est synonyme de redondance. Dans le présent manuscrit, nous désignerons par **duplication** une redondance consistant à utiliser deux copies identiques d'un même composant. La **réplication** est une redondance utilisant plus de deux copies identiques d'un même composant. Le terme **réplique** d'un composant désigne alors une copie identique de ce dernier. Si les composants implémentent la même fonctionnalité mais en utilisant différentes techniques, nous parlons alors de **diversification**. Dans ce cas, nous désignons par **versions** les différentes instances du composant diversifié.

II.3 CONCLUSION

Dans le cadre de notre travail, nous nous intéressons aux applications à criticité multiple dans le domaine de l'avionique. Nous avons ainsi :

- présenté les principales normes qui ont guidé le développement logiciel et matériel dans les fonctions avioniques, à savoir la norme ARP 4754 et le standard DO 178-B.
- présenté l'avionique modulaire (IMA), qui contrairement à l'avionique fédérée classique, permet l'implantation de plusieurs fonctions sur un même module matériel.
- discuté des aspects sécurité-immunité dans les appareils existants et futurs en se basant sur une ségrégation au niveau des réseaux et des domaines d'applications.

Ces applications avioniques font parties des systèmes critiques qui doivent être sûrs de fonctionnement. Nous avons donc :

- détaillé la terminologie relative au domaine de la sûreté de fonctionnement
- présenté les mécanismes de tolérance aux fautes, à savoir les mécanismes de détection et de recouvrement d'erreurs et ce à la fois pour des fautes d'origines accidentelle et malveillante.

Nous avons également constaté que la notion de criticité est primordiale dans les systèmes critiques, et qu'elle guide le développement des systèmes avioniques. Nous proposons dans le chapitre suivant (Section III.1) une vision cohérente du développement logiciel qui se base sur une classification par niveaux de criticité et de confiance. Cette vision par niveaux est exploitée dans différents modèles de sécurité-immunité pour permettre l'implantation d'architectures sûres aussi bien au niveau sécurité-innocuité qu'au niveau sécurité-immunité (Section III.2).

Chapitre III

CONSIDERATIONS IMMUNITE - INNOCUITE

INTRODUCTION

Nous avons présenté dans le chapitre II les différentes normes avioniques qui guident le développement d'applications critiques, et avons introduit la terminologie de la sûreté de fonctionnement et plus spécialement de la tolérance aux fautes. Il est clair que pour rendre un système critique sûr de fonctionnement, il faut le développer de manière à ce qu'il y ait le moins de bogues possibles, mais également faire de sorte qu'un tel système soit fonctionnel en cas d'activation de bogues résiduelles.

Remarquons que divers aspects sont mis en jeu dans un tel contexte, notamment les notions de criticité, validation, intégrité, tolérance aux fautes, confiance, sécurité-innocuité, sécurité-immunité etc. Ces notions sont utilisées dans différents standards et normes, mais il n'existe pas de vision générale cohérente de leur interdépendances.

Suite à ces constatations, nous proposons une vision cohérente de ces différentes notions en se basant sur le concept de niveaux utilisé dans les différents standards des systèmes critiques. La bonne définition des niveaux nous permet ensuite d'étudier en détail les systèmes multiniveaux, dont nous présentons les principaux modèles.

Ainsi, ce chapitre est présenté comme suit : dans un premier temps, nous détaillons notre vision du développement des logiciels critiques et proposons une taxonomie cohérente, ensuite nous présentons les principaux modèles qui ont traité de la problématique multiniveaux dans la littérature des systèmes critiques.

III.1 INTEGRITE ET SECURITE-INNOCUITE

Nous avons présenté dans le chapitre I la norme ARP 4754 relative au développement des systèmes en avionique. Rappelons que les systèmes avioniques font partie des systèmes critiques, et obéissent, de ce fait, aux mêmes exigences relatives à ces systèmes. Cependant, ces systèmes appartiennent à des domaines fonctionnels distincts (avionique, ferroviaire, nucléaire, etc). Ce fait explique qu'historiquement, chaque domaine s'est vu développer ses propres standards relatifs à la sûreté de fonctionnement (la DO-178B pour l'avionique, la EN 50126 [EN 50126] pour le ferroviaire, etc). Cependant, depuis quelques années, des efforts ont été réalisés pour l'homogénéisation de ces différentes normes. Nous proposons dans un premier temps d'en présenter deux illustrations : les critères SQUALE et la norme IEC 61508. Ces deux illustrations montrent qu'il y a de fortes similarités dans les approches utilisées pour concevoir des systèmes critiques. Nous proposons ainsi dans un second temps de présenter notre vision multiniveaux dans les systèmes critiques.

III.1.1 Retour sur les normes de développement des systèmes critiques

Dans cette section nous nous intéressons aux travaux visant à proposer une approche homogène de la gestion de la sûreté de fonctionnement lors de la conception de systèmes. En effet, ces travaux permettent de souligner les points communs entre les différents domaines traitant de systèmes critiques. Nous proposons alors de détailler les critères SQUALE et la norme IEC 61508.

III.1.1.1 Les critères SQUALE

Les critères SQUALE [Yves Deswarte et al. 1999] se veulent non seulement indépendants du domaine d'application, mais également génériques en prenant en compte l'ensemble des attributs de la sûreté de fonctionnement. Leur utilisation s'applique à l'ensemble du cycle de vie du système, et ne repose pas sur un modèle particulier de cycle de vie. Toutefois le modèle de référence repose sur l'hypothèse qu'un tel cycle comprend les phases de construction, de service opérationnel et de retrait de service. Dans chaque étape de construction, une distinction est faite entre les tâches de conception, les tâches d'implémentation et les tâches de vérification. Ceci détermine un canevas élémentaire sur lequel viennent se greffer, à chaque étape, les activités décomposées en quatre processus : évaluation des exigences de sûreté de fonctionnement, vérification de conformité, validation, assurance qualité (cf. figure III.1).

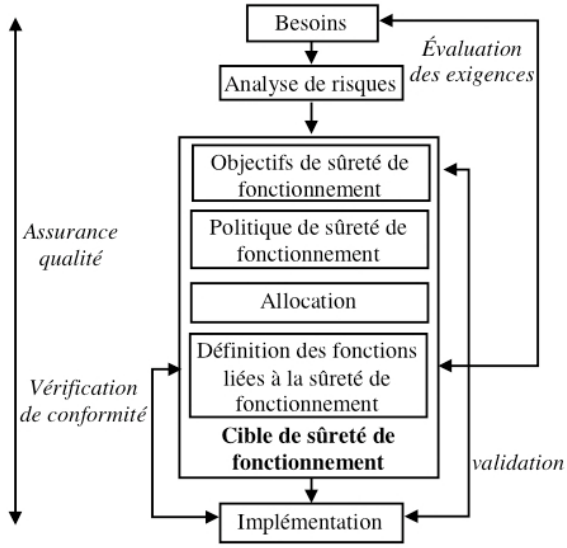


Figure III.1 Activités d'évaluation dans SQUALE

Dans le cadre des critères SQUALE, on quantifie l'importance de chaque attribut de la sûreté de fonctionnement dans le système considéré. Ainsi, on associe à chaque attribut un niveau de confiance visé variant de 0 à 4 : 1 représente le niveau de confiance le plus bas, et 4 le plus haut, 0 représentant l'absence d'exigence pour l'attribut. Ainsi un profil de sûreté de fonctionnement de combinaison A1, C0, R3, I3, S3, M2 correspond à des niveaux 1, 0, 3, 3, 3 et 2 respectivement pour la disponibilité, la confidentialité, la fiabilité, l'intégrité, la sécurité-innocuité et la maintenabilité.

En outre, les critères SQUALE définissent pour les activités d'évaluation des niveaux de rigueur avec laquelle les activités doivent être réalisées, de détail et d'indépendance (lien organisationnel entre ceux qui réalisent l'activité et les développeurs) plus ou moins élevés (de 1 à 3) en fonction des niveaux de confiance visés dans le profil de sûreté de fonctionnement.

III.1.1.2 La norme IEC 61508

La norme IEC 61508 [IEC 61508 2007] propose une approche générique, non spécifique d'un domaine d'application particulier, pour la spécification, la mise en oeuvre et l'évaluation de systèmes dits E/E/SEP (Systèmes Électriques / Électroniques / Électroniques Programmables). Les systèmes informatiques entrent donc dans le cadre des systèmes traitées par la IEC 61508. Cette norme utilise le concept de cycle de vie pour la sécurité-innocuité, les phases de ce cycle décrivant les principales activités à mener pour définir les exigences de sûreté, les mettre en oeuvre et s'assurer du respect des exigences tout au long du cycle de vie du logiciel. Pour chaque phase du cycle de développement, la norme spécifie les objectifs à atteindre, les exigences à satisfaire, la portée de ces objectifs, les entrées nécessaires à la mise en oeuvre de cette phase, et les délivrables à fournir à la fin de la phase pour montrer que les exigences sont satisfaites. Chaque phase doit être conclue par une activité de vérification répondant aux exigences préalablement définies dans le plan de la vérification.

Cette norme définit pour chaque système une intégrité de sécurité-innocuité (*safety integrity*) comme étant la *« probabilité pour qu'un système relatif à la sécurité exécute de manière satisfaisante les fonctions de sécurité requises dans toutes les conditions spécifiées et dans une période de temps spécifiée »*. L'intégrité, selon la norme IEC 61508 relève donc de

l'aptitude du système à remplir correctement ses fonctions. La norme introduit des niveaux d'intégrité de sécurité-innocuité, appelés SIL (*Safety Integrity Le*vel). Un SIL est un « niveau discret (parmi quatre possibles) permettant de spécifier les prescriptions concernant l'intégrité de sécurité des fonctions de sécurité à allouer aux systèmes E/E/PE relatifs à la sécurité ». Quatre niveaux d'intégrité sont définis, où le niveau 1 correspond aux systèmes non-critiques et le niveau 4 aux systèmes hautement critiques. Pour chaque niveau d'intégrité, la norme recommande des méthodes et techniques qu'il est souhaitable d'appliquer afin d'atteindre le niveau de sécurité-innocuité requis.

	Sollicitation du système		
Niveau de SIL	Rare (PFD[4])	Fréquent (PFH)[5]	Facteur de réduction de risque
4	$[10^{-5}\text{-}10^{-4}]$	$[10^{-9}\text{-}10^{-8}]$	De 10^4 à 10^5
3	$[10^{-4}\text{-}10^{-3}]$	$[10^{-8}\text{-}10^{-7}]$	De 10^3 à 10^4
2	$[10^{-3}\text{-}10^{-2}]$	$[10^{-7}\text{-}10^{-6}]$	De 10^2 à 10^3
1	$[10^{-2}\text{-}10^{-1}]$	$[10^{-6}\text{-}10^{-5}]$	De 10^1 à 10^2

Tableau 4 **Echelle des niveaux de SIL et facteur de réduction de risque associés à une fonction de sécurité**

Le développement de systèmes critiques consiste alors à réduire le plus raisonnablement possible le risque, cette réduction est qualifiée par ALARP (*As Low As Reasonably Practicable*). De ce point de vue, un niveau de SIL constitue un facteur de réduction de risque par rapport aux systèmes qui n'implémenteraient pas des fonctions de sécurité. Plus le niveau de SIL est élevé, plus le facteur de diminution de risque est important (cf. tableau 1).

Comme le montre le tableau 1, une réduction de facteur de risque ainsi qu'une probabilité de défaillance (à la demande ou par heure) est attribuée à chaque niveau de SIL. Cette vision multiniveaux en fonction des probabilités (et de sévérité) de défaillances est présente dans d'autres normes de systèmes critiques (par exemple les normes avioniques, comme nous l'avons vu dans le chapitre II). Cependant, la terminologie utilisée diffère d'une norme à l'autre. Ainsi, dans la IEC 61508 la notion de niveau d'intégrité de sécurité-innocuité (SIL) est utilisée, alors que dans la ARP 4754 il est question de niveau d'assurance de développement (DAL). Nous proposons dans ce qui suit de présenter notre approche multiniveaux qui vise à avoir une vision globale cohérente du développement des systèmes critiques, tout en faisant le lien avec les normes déjà existantes. Ce développement, comme nous allons le montrer, ne dépend pas uniquement des attributs sécurité-innocuité, mais également des attributs sécurité-immunité.

III.1.2 Considérations multiniveaux

Tout système est conçu et développé dans le but de réaliser une mission spécifique. Nous désignons par **tâche** la fonction du système en charge de réaliser la mission demandée. Pour être accomplie, une tâche doit être implémentée sur un support d'exécution que nous

[4] *Probability of Failure on Demand*, probabilité d'avoir une défaillance au moment d'une sollicitation
[5] *Probability of dangerous Failure per Hour*, probabilité de défaillance par heure.

désignons par **module**. Un module peut être de nature logicielle ou matérielle (ou les deux). Ainsi, pour réaliser la mission « commander un avion », une tâche de pilotage est mise en œuvre pour accomplir cette mission. Des modules sous forme de calculateurs sur lesquels s'exécutent des applications logicielles relatives au pilotage sont ensuite développés en prenant en compte les exigences de conception et les contraintes environnementales.

III.1.2.1 Criticité et confiance

Comme nous l'avons indiqué dans la section II.1, les systèmes avioniques font partie des systèmes critiques. Nous proposons de spécifier dans ce qui suit les notions de criticité et de confiance, et d'identifier leurs principales caractéristiques.

Criticité d'une tâche

La **criticité** est associée à une tâche en charge de réaliser une mission donnée. Une tâche est considérée critique si, en cas de défaillance de cette tâche, des conséquences catastrophiques peuvent arriver au système. Remarquons que cette définition correspond à la terminologie introduite dans les normes avioniques. Par conséquences catastrophiques, nous désignons les blessures physiques des utilisateurs humains du système (cf. tableau 1 et 2 du chapitre II). Ainsi, le *crash* d'un avion est considéré comme événement catastrophique.

La sévérité de défaillance d'une tâche peut dépendre des différentes phases de la mission attribuée à la tâche. Ainsi, la tâche de commande d'un train, par exemple, est très critique quand le train est en marche. Cette même tâche devient moins critique quand le train est à l'arrêt. Cependant, même si une tâche a une criticité variante en fonctions des phases de la mission, c'est la criticité la plus élevée qui doit être prise en compte dans l'étude du système global. Dans l'exemple que nous avons donné, la tâche de commande d'un train sera donc considérée comme critique lors de la conception et déploiement du système final. Ainsi, nous désignons par *FSL(T)* le niveau de criticité[6] d'une tâche *T* (*FSL : Failure Severity Level*)

Pour définir *FSL(T)*, différentes études doivent être réalisées pour évaluer le risque que courent les utilisateurs en cas de défaillance de cette tâche. Ces études fournissent alors un ensemble de scénarios qui identifient les défaillances potentielles de la tâche et, en fonction de la sévérité des défaillances, un niveau de criticité est attribué à la tâche. Il convient de souligner que l'évaluation de la criticité est un processus réalisé en prenant en compte les aspects environnementaux d'exécution de la tâche. En effet, la sévérité d'une défaillance dépend essentiellement de la nature de la tâche, mais également des différentes interactions avec les autres tâches.

Il est donc important de spécifier clairement les interactions d'une tâche avec son environnement (qui comprend tous les éléments extérieurs à la tâche) afin de lui attribuer le niveau de criticité adéquat. Dans une telle démarche, ce niveau de criticité est une propriété **intrinsèque** à la tâche. En effet, une fois toutes les interactions identifiées et toutes les actions de la tâche spécifiées, l'analyse de risques effectuée sur cette tâche permet de lui attribuer un niveau de criticité qui lui est propre. Par intrinsèque nous désignons ainsi le fait que ce niveau ne peut être modifié, sous peine de risquer la perte de la propriété de sécurité-innocuité du système critique ou l'augmentation du coût de production du système. En effet, accepter de diminuer le niveau de criticité d'une tâche implique que l'on considère que la sévérité de sa défaillance a diminué. Une telle diminution doit donc être accompagnée d'une justification, par exemple une réévaluation de la sévérité suite à une changement dans les contraintes environnementales. Augmenter le niveau de criticité, implique que l'on suppose que sa défaillance est plus sévère et qu'il faut alors déployer des mécanismes de réduction de risques

[6] Ce niveau correspond donc à la défaillance la plus sévère de la tâche en question.

supplémentaire pour y palier. Un tel déploiement ne peut se faire sans une augmentation dans le coût de production totale du système.

Cependant, si le niveau de criticité est intrinsèque à la tâche, il y est possible de répondre à ces exigences de criticité en passant par les niveaux de confiance d'un module.

Niveau de confiance d'un module

Rappelons qu'une tâche est implémentée sur un module (logiciel ou matériel) qui permet de réaliser la tâche. Ainsi, il est nécessaire, pour une tâche critique, de s'assurer qu'elle s'exécute sur un module qui remplisse correctement ses fonctions, donc un module de **confiance**. Nous désignons par $CL(M_T)$ le niveau de confiance que l'on attribue au module M_T réalisant une tâche T (*CL : Confidence Level*).

Le niveau de confiance est donc associé à un module et traduit la fidélité avec laquelle ce dernier s'exécute selon la spécification requise par la tâche. Ainsi, le niveau de confiance d'un module doit être au moins égal au niveau de criticité de la tâche qu'il réalise. Prenons l'exemple d'une tâche en charge de calculer une loi physique, en fonction de valeurs de pressions mesurées. Cette tâche est réalisée par un module. Ce module est dit de confiance s'il réalise correctement le calcul de la loi, mais également si ses capteurs mesurent correctement les valeurs de pression. Dans cet exemple, la tâche peut être divisée entre deux sous tâches : une pour la capture de la valeur de pression et une pour la réalisation du calcul. Il est donc possible d'utiliser deux modules afin de réaliser chaque sous tâche. Dans ce cas, le module de calcul est de confiance si à partir de mesures correctes, il fournit un calcul de loi correct. Cependant, pour la réalisation de la tâche (mesure + calcul), il est nécessaire que le module de mesure soit de confiance également. À défaut, le résultat final est incorrect, ce qui ne répond pas aux besoins de criticité de la tâche. Ainsi, il faut s'assurer que tous les modules associés à une tâche soient d'un niveau de confiance au moins égal au niveau de criticité de cette tâche.

Contrairement à une tâche qui est liée à la mission confiée au système, un module est lié à la conception et l'implémentation de mécanismes permettant la réalisation de cette tâche. La diversité des choix de conception et d'implémentation implique que le niveau de confiance d'un module peut être augmenté ou diminué. Nous proposons dans ce qui suit de présenter les différents paramètres dont dépend la confiance d'un module.

III.1.2.2 Paramètres de confiance

Les niveaux de confiance permettent de caractériser l'aptitude d'un module à réaliser correctement une tâche. Ainsi, pour modifier le niveau de confiance d'un module, il est possible d'intervenir à différents instants du cycle de vie de ce module. Considérons un cycle de vie de référence (cf. SQUALE, section III.1.1.1). Un tel cycle peut être divisé en trois phases :

- Phase de construction : durant cette phase, différents moyens peuvent être utilisés pour assurer la conception et le développement du module.
- Phase de service opérationnel : cette phase correspond à l'exécution du module dans son environnement de déploiement.
- Phase de retrait de service du module : qui correspond à mettre fin aux services fournis par le module.

Les deux première phases (construction et service opérationnels) sont directement liées à la réalisation de la tâche associée au module. Ainsi, c'est au niveau de ces deux phases que l'on peut intervenir pour augmenter et/ou diminuer le niveau de confiance d'un module.

Nous avons identifié trois principaux paramètres qui définissent le niveau de confiance d'un module : la validation, la crédibilité et l'intégrité. La validation intervient à la première phase de cycle de vie d'un module, à savoir à la construction. La crédibilité et l'intégrité interviennent à l'exécution du module, comme le montre la figure III.2. Le niveau de confiance du module M_T correspond alors à la confiance que l'on peut avoir dans ses sorties (O_{MT}) : $CL(M_T) = CL(O_{MT})$

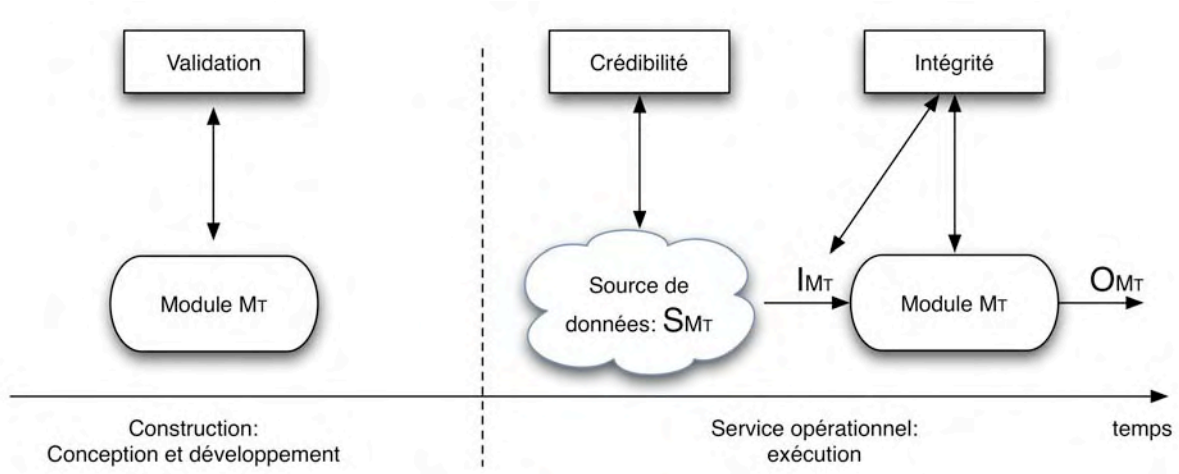


Figure III.2 Différents paramètres influençant le niveau de confiance d'un module

Remarquons que le niveau de confiance que l'on propose correspond aux SILs introduit par la norme IEC 61508 (cf. section III.1.1.2). Nous proposons de détailler dans ce qui suit les caractéristiques de chacun de ces paramètres qui influencent le niveau de confiance d'un module donné.

Niveaux de validation

Nous désignons par validation $V(M_T)$ d'un module M_T, les efforts fournis durant la phase de conception et de développement pour obtenir un module qui satisfait les exigences de la tâche qu'il implémente. Dans l'avionique, ces niveaux de validation sont connus par les DAL (cf. chapitre II, section II.1).

Ainsi pour développer un module au niveau de validation le plus élevé, il est nécessaire d'utiliser des techniques qui permettent de démontrer que le module répond bien aux exigences de sécurité-innocuité requises par la tâche. Dans la DO-178B, il est recommandé d'utiliser des méthodes formelles [John Rushby 1993] pour prouver le comportement correct du module.

La validation concerne également l'aspect matériel du module, comme nous l'avons déjà mentionné dans la section II.1 (à travers la norme DO-254 relative aux exigences de développement matériel). Une fois le module conçu et développé, il faut s'assurer que, durant sa phase opérationnelle, il réalise correctement les opérations relatives à la tâche qui lui a été affectée. Pour ce faire, il faut s'assurer :

- que les données d'entrée du module sont correctes
- que ce module correspond bien aux exigences de la phase de construction.

Le premier aspect est pris en compte par la notion de niveau de crédibilité, et le second par les niveaux d'intégrité, comme nous le détaillons dans ce qui suit.

Niveaux de crédibilité

La crédibilité correspond à la confiance que l'on peut avoir dans la source de données utilisées comme entrée pour un module donné. Une source peut être humaine (un opérateur en charge de communiquer manuellement des données au module par exemple) ou un autre module. Nous notons $Cr(S_{MT})$ la crédibilité des sources S_{MT} de M_T comme le montre la figure III.2.

Dans le cas où il s'agit d'une source humaine, le niveau de crédibilité peut correspondre par exemple à l'expérience acquise par cet opérateur lors de ses interactions avec des modules similaires.

Dans le cas où il s'agit d'un autre module M'_T, il faut fournir une justification technique du niveau de crédibilité des sorties de ce module. Ainsi le niveau de crédibilité des sources de M_T correspond au niveau de confiance de M'_T : $CL(M'_T)=Cr(S_{MT})$ puisque $S_{MT} = M'_T$. Reprenons l'exemple d'un capteur de pression en charge de mesurer la pression ambiante et de la communiquer à d'autres modules. La mesure de pression dépend de la fiabilité du matériel utilisé pour la capture, et il se peut que le module de mesure ne soit pas capable de fournir une mesure fiable de la pression (c'est-à-dire que la mesure fournie ne correspond pas à la valeur de la pression réelle). Dans un tel cas, le module de capture est considéré comme non crédible, et ce niveau de crédibilité doit être pris en compte lors du déploiement du système final (par exemple par l'utilisation de plusieurs capteurs, comme nous allons le voir ci-dessous).

Niveaux d'intégrité

L'intégrité telle qu'elle est définie dans le « Guide de la Sûreté de Fonctionnement » [Jean-claude Laprie et al. 1996] consiste à la non-occurrence d'altération inappropriée du système. Ces altérations peuvent porter aussi bien sur les parties matérielles du système que sur ses parties logicielles. Le premier type d'altération consiste à modifier un dispositif physique (comme par exemple la modification d'une carte à puce par faisceau ionique). Le deuxième type consiste à altérer soit le programme soit les données manipulées par ce programme.

Dans le cadre de nos considérations, nous nous intéressons tout particulièrement aux altérations logicielles, c'est-à-dire portant sur les programmes et les données. Ainsi, un module est dit intègre si le logiciel exécuté n'a pas été altéré (c'est-à-dire qu'il s'agit du même logiciel fourni par la phase de construction) et si les données manipulées ne sont pas altérées également (lors de leur transmission par exemple). Ainsi, le niveau d'intégrité d'un module traduit la non-altération de ce dernier et la non-altération des données qu'il manipule. Nous notons par $il\ (M_T)$ et $il(I_{MT})$ respectivement le niveau d'intégrité du module M_T et des données qu'ils reçoit : I_{MT}.

Notons que cette altération peut être d'origine accidentelle ou malveillante. Les altérations d'origine accidentelle doivent être prises en compte lors de la phase de construction du module afin de les tolérer à l'exécution. C'est pour cette raison que nous nous intéressons aux altérations malveillantes qui doivent être prises en compte lors de l'exécution du module, afin de les éliminer.

Afin d'assurer l'intégrité d'un module, il est possible de faire appel à des mécanismes de protection communément utilisés dans le domaine de la sécurité-immunité. Les critères communs [Common Criteria] introduisent le niveau d'assurance d'évaluation (*EAL : Evaluation Assurance Level*) pour décrire l'efficacité de tels mécanismes de protection. Il est donc possible d'attribuer un niveau d'intégrité à un module en s'appuyant sur le niveau EAL des mécanismes de protection implémenté pour protéger ce module contre les fautes malveillantes.

Ainsi, l'intégrité telle que nous l'avons introduite est fortement liée aux altérations accidentelles et surtout malveillantes. Différents modèles ont été introduits pour assurer l'intégrité dans le contexte de sécurité-immunité et nous proposons de les présenter plus en détail dans la section III.2.

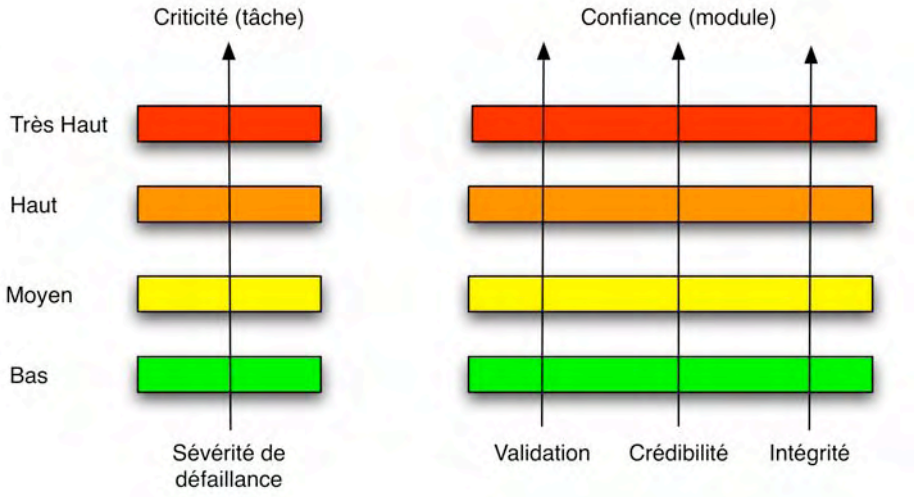


Figure III.3 Vision globale des niveaux de criticité et de confiance

La figure III.3 présente les différents niveaux introduits dans cette section qui se résument comme suit : à une tâche on affecte un niveau de criticité en fonction de la sévérité de défaillance de cette tâche. Pour satisfaire ce niveau de criticité, la tâche est développée sur un module auquel on attribue un niveau de confiance. Cette confiance dépend de la phase de construction (niveau de validation), des données reçues à l'exécution (niveau de crédibilité) et de la non-altération de ce module durant la phase opérationnelle (niveau d'intégrité). Ainsi, la confiance d'un module M_T est une fonction de la crédibilité de ses sources, de sa validation, de son intégrité et de l'intégrité des données qu'il manipule :

$$CL(M_T) = CL(O_{MT}) = f(Cr(S_{MT}), V(M_T), il(M_T), il(I_{MT}))$$

Si la source du module M_T est un autre module M'_T, cette définition s'applique récursivement, en remplaçant $Cr(S_{MT})$ par $CL(M'_T)$ comme indiqué précédemment.

Nous discutons dans le paragraphe suivant de la redondance et de son rôle dans l'attribution des niveaux de confiance d'un module donné.

Redondance et niveaux de confiance

Comme le montre la figure III.3, une tâche d'un niveau de criticité 'haut' doit être réalisée par un module de niveau de confiance au moins égal à celui du niveau de criticité de la tâche, c'est-à-dire le niveau 'haut'. Cette condition peut s'écrire sous la forme de l'inégalité suivante : une tâche T se réalisant par un module M doit satisfaire : $CL(M_T) \geq FSL(T)$. En effet, il est possible d'utiliser un module de niveau de confiance 'très haut' pour exécuter une tâche d'un niveau de criticité 'haut'. Cependant, rappelons que pour obtenir un niveau de confiance donné, il faut déployer les techniques nécessaires à la justification d'un tel niveau de confiance. Une telle justification ne peut se faire sans une augmentation considérable dans le coût de production du système.

Ainsi, d'un point de vue économique, il faut chercher à atteindre l'égalité entre le niveau de criticité et le niveau de confiance, afin d'optimiser les coûts de production. Cependant, nous pouvons réduire encore plus les coûts de production si nous faisons des hypothèses relatives à

l'implémentation du module M. En effet, comme nous l'avons présenté dans la section II.1.1.3, la réplication et la diversification peuvent être utilisées pour rendre le système plus sûr de fonctionnement. Or rendre le système plus sûr de fonctionnement implique que l'on a une meilleure confiance dans ce système. Ainsi, il est possible, à travers l'utilisation de la diversification (ou réplication) d'augmenter le niveau de confiance d'un module. Par conséquent, un module implémenté en utilisant la diversification a un niveau de confiance plus élevé que celui d'un module non diversifié.

Cette idée s'inscrit dans une discussion relative à la diversification et les SIL introduits dans la norme IEC 61508 (seconde partie, section 7.4.3), que nous proposons de détailler dans ce qui suit. En effet, cette discussion s'articule autour de ce point : **3+3=4 et 3+3=2**. La première addition est interprétée comme suit : si l'on utilise deux composants diversifiés d'un niveau de SIL 3, alors il est possible, en respectant certaines conditions d'obtenir un résultat d'un niveau de SIL 4. Les conditions, dans ce cas, portent essentiellement sur la qualité de la diversification et la notion d'indépendance des versions. La seconde addition traduit le fait que si l'on utilise deux composants d'un niveau de SIL 3, l'ensemble de ces deux composants n'est pas forcément aussi sûr de fonctionnement que chacun des deux composants pris à part. Ainsi, l'ensemble voit son niveau de SIL diminuer.

Les SIL ne sont autres que les niveaux de confiance que nous avons introduits. Ainsi, ces deux additions soulignent l'importance de l'utilisation de chaque composant dans la définition du niveau de confiance globale. Par conséquent, si un module M_T (réalisant une tâche T) est implémenté en utilisant des composants diversifiées C_i, alors il peut être admis d'utiliser une contrainte moins sévère sur le niveau de confiance des modules, par exemple : $\forall C_i\ CL(C_i) \geq FSL(T)-1$. Cette approche est intéressante parce qu'elle permet d'assurer le même niveau de confiance globale du module, tout en satisfaisant des contraintes moindres pour chaque composant. Une telle réduction a une importance économique considérable puisqu'elle permet de réduire le coût de production total du module.

***

Dans cette section, nous avons proposé une vision cohérente du développement de logiciels critiques en nous basant sur les normes et notions déjà existantes dans différents secteurs industriels (et spécialement en avionique). Cette vision se base sur une approche par niveau des différents critères de criticité et de confiance. Cependant, nous n'avons pas traité de la problématique de communication entre ces différents niveaux. Nous proposons donc de présenter, dans la section suivante, quelques travaux qui se sont intéressés à cette problématique.

III.2 INTEGRITE DANS LES SYSTEMES A CRITICITES MULTIPLES

Comme nous l'avons vu dans la section II.1.3, pour répondre aux exigences de sécurité-innocuité, il faut prendre en considération aussi les aspects sécurité-immunité. En effet, pour obtenir une confiance suffisante dans un module, il faut s'assurer qu'il est correctement développé, qu'il manipule des données de provenance sûre, mais qu'également il n'a pas été altéré (accidentellement ou surtout d'une façon malveillante). L'approche utilisée en avionique (présentée dans la section II.1.3.1) consiste à séparer les modules dans des domaines bien spécifiques, et d'interdire les communications inter-domaines. Cette approche s'inscrit dans le cadre des travaux sur des modèles d'intégrité, que nous présentons dans cette section.

III.2.1 Modèle Biba

La politique d'intégrité de Biba [Biba 1977] est issue de la politique multiniveau de Bell et Lapadula [Bell et Lapadula 1975] définie pour la confidentialité. L'intégrité dans ce modèle est prise dans la définition utilisée dans le domaine de sûreté de fonctionnement, à savoir comme la non altération accidentelle ou délibérée d'un composant. Comme cette politique, elle se fonde sur la notion de treillis. Biba propose plusieurs variantes et définit un certain nombre d'approches dans le cadre de la définition d'une politique obligatoire.

III.2.1.1 Définition du treillis

Dans ce modèle, les entités du système sont totalement ordonnées selon des classifications de sécurité, par exemple : NON-CLASSIFIÉ, CONFIDENTIEL, SECRET et TRÈS SECRET. De plus, à chaque entité est associé un compartiment qui se présente sous forme d'un ensemble de catégories. Ces compartiments permettent d'effectuer un partitionnement entre les différentes entités. Les catégories peuvent, par exemple, correspondre à certains types d'applications (simulation, commande ou contrôle temps-réel, …). Le niveau d'une entité comprend donc sa classification et le compartiment auquel cette entité appartient.

Cet ensemble est partiellement ordonné selon une relation de dominance que nous noterons $\angle$ et qui est définie de la manière suivante: si deux informations sont définies par leur classification et leur compartiment, respectivement (c, C) et (c', C'), alors le niveau n de l'une est dominé par le niveau n' de l'autre, noté $n \angle n'$, si et seulement si $c \leq c'$ et $C \subseteq C'$. L'ensemble des niveaux et de relation de dominance $\angle$ définissent ainsi un treillis.

III.2.1.2 Concepts et notations de la politique d'intégrité

Biba distingue dans sa politique deux types d'entités, les entités actives, dites *sujets* et les entités passives, les *objets*. Les sujets regroupent les entités actives telles que les processus ou les utilisateurs, et les objets, toutes les entités qui contiennent de l'information (telles que les fichiers). Le modèle se compose ainsi de :

- *S,* l'ensemble de sujets *s*;
- *O,* l'ensemble d'objets *o*;
- *I,* l'ensemble de niveaux d'intégrité;

- *il* : une fonction permettant d'affecter un niveau d'intégrité à tout sujet ou objet du système;
- $\angle$ la relation de dominance définie précédemment sur un sous-ensemble de *I ;*
- *obs* la relation définie sur un sous-ensemble de $S \times O$ définissant la capacité d'un sujet *s* de *S* d'observer un objet *o* de *O*;
- *mod* une relation définie sur un sous-ensemble de $S \times O$ définissant la capacité d'un sujet*s* de S de modifier un objet *o* de *O*;
- *inv* une relation définie sur un sous-ensemble de $S \times S$ définissant la capacité d'un sujet s_1 de *S* d'invoquer un autre sujet s_2 de *S*. Cette invocation correspond à un appel sans retour d'un service offert par s_2.

Il faut noter ici que chaque objet ou sujet est mono-niveau, c'est-à-dire qu'il appartient à un seul niveau d'intégrité à un instant donné. À partir de ces définitions, Biba construit les règles d'accès aux objets et sujets dans différentes politiques d'intégrité proposant des propriétés sensiblement différentes.

III.2.1.3 Spécification de la politique

Le but de cette politique est de s'assurer que l'intégrité d'un sujet ou objet ne pourra pas être altérée volontairement ou accidentellement par un sujet de niveau d'intégrité inférieur. Ceci est mis en place par les trois règles suivantes:

1. Règle d'observation : $\forall (s,o) \in (S \times O),\ obs(s,o) \Rightarrow il(s) \angle il(o)$
2. Règle de modification : $\forall (s,o) \in (S \times O),\ mod(s,o) \Rightarrow il(o) \angle il(s)$
3. Règle d'invocation : $\forall (s_1,s_2) \in (S \times S),\ inv(s_1,s_2) \Rightarrow il(s_2) \angle il(s_1)$

On distingue deux types de règles: celle portant sur les accès sujet-objet, et celle mettant en présence deux sujets:

- La règle d'observation permet de s'assurer qu'un sujet ne peut accéder à une information de niveau d'intégrité inférieur au sien. La règle de modification empêche un objet d'être corrompu par un sujet de niveau d'intégrité inférieur au sien.
- La règle d'invocation empêche la perturbation d'un sujet par un autre sujet de niveau inférieur. On peut noter que cette règle est similaire à la règle (2) qui concerne la modification d'un objet. Elle peut donc être interprétée comme la prévention contre la modification de l'état d'un autre sujet.

Dans cette approche, chaque entité du système a un niveau d'intégrité fixe, et détient des informations d'un niveau égal au sien. Toutefois une information de haute intégrité copiée dans une entité de faible intégrité est automatiquement considérée du niveau de cet objet. Ce comportement implique une dégradation du niveau d'intégrité des informations, ce qui est l'un des principaux inconvénients de cette politique. Quelques extensions ont été proposées pour ce modèle. Nous en citons par exemple le changement dynamique des niveaux des sujets. Dans cette extension, tout sujet peut observer n'importe quel objet du système, à condition d'hériter de son niveau d'intégrité. Ainsi, après chaque accès à un objet, le sujet risque de voir son niveau d'intégrité baisser, et à terme, tous les sujets du système risque de se retrouver au niveau le plus bas.

D'autre part, il est évident que les règles en elles-mêmes sont particulièrement contraignantes. On doit en effet vérifier que tous les accès nécessaires au bon déroulement d'une tâche donnée au sein d'une application seront autorisés.

III.2.2 Modèle de non interférence

Le modèle de [Dutertre et Stavridou 1999] s'intéresse à la problématique d'intégration de plusieurs composants logiciels (avec différents niveaux de confiance) sur un même module physique. Cette problématique se situe clairement dans le cœur des architectures IMA que nous avons présentées dans le chapitre précédent. Le modèle s'intéresse alors à l'indépendance entre les différents composants pour éviter toute altération d'un composant critique par un composant peu critique. Une telle indépendance est réalisée grâce à la non interférence entre les composants.

Ce modèle définit l'interface d'un composant avec son environnement comme étant composé d'un ensemble d'événements d'entrée *I* et de sortie *O*. Une exécution du système peut être décrite en enregistrant à chaque instant *t* l'élément de *I* reçu, s'il y en a un, et l'élément de *O* produit en sortie. Le symbole ⊥ est utilisé pour traduire l'absence d'événement d'entrée ou de sortie ($\perp \notin I$ et $\perp \notin O$). L'exécution du système est décrite par deux séquences $\sigma = (x_t)$ et $\tau = (y_t)$, où x_t est une entrée (c'est-à-dire $x_t \in I \cup \{\perp\}$) et y_t est une sortie ($y_t \in O \cup \{\perp\}$). Lorsque le composant est déterministe (ce qui est supposé ici être le cas), il existe une seule séquence τ pour une séquence σ donnée. Le comportement du composant peut être représenté par une fonction $F : Seq(I) \rightarrow Seq(O)$ où *Seq(I)* et *Seq(O)* correspondent à des séquences respectivement de $I \cup \{\perp\}$ et $O \cup \{\perp\}$.

Le système temps-réel supposé déterministe *G* est associé à ses entrées I_G et ses sorties O_G qui incluent les entrées et sorties de tous les composants présents. Le comportement de *G* est déterminé par la fonction $F_G : Seq(I_G) \rightarrow Seq(O_G)$. Soit un composant critique *M* défini par son domaine d'entrée $I_M \subset I_G$et son domaine de sortie $O_M \subset O_G$ ayant, dans un environnement isolé, un comportement F_M.

Soit une séquence $\sigma \in Seq(I_G)$. On note σ / I_M la séquence obtenue en remplaçant tous les éléments de σ non présents dans I_M par ⊥. On définit de manière similaire, pour une séquence de sortie τ, τ / O_M.

L'intégration du composant *M* dans le système *G* est dite sûre si:

$$\forall \sigma \in Seq(I_G), F_G(\sigma)/O_M = F_M(\sigma/I_M)$$

En d'autres termes, l'intégration est sûre si aucun composant du système n'influence l'exécution *idéale* de M, c'est-à-dire si aucun composant n'interfère sur l'exécution de M. Toutefois, cette condition étant extrêmement forte, l'approche consiste à la relâcher en acceptant en particulier de légères déviations dans le temps (déviations qui doivent être bornées), de sorte que la propriété est considérée comme vérifiée si elle est vraie avec un éventuel décalage temporel.

III.2.3 Modèle de dépendances causales

Le modèle de dépendances causales [D'ausbourg 1994] présente une façon originale de modéliser les flots d'information dans un système multiniveau. Ce modèle propose de décrire le système comme un ensemble de points *(o, t)*, où *o* est un objet et *t* un instant donné. Un point *(o, t)* représente donc un objet *o* à un instant *t*. Chaque point évolue dans le temps et cette évolution est due à une transaction élémentaire effectuée dans le système. Une transaction peut ainsi modifier un point de telle façon qu'à l'instant *t*, l'objet *o* possède la nouvelle valeur *v*. Cet instant *t* et cette nouvelle valeur *v* dépendent fonctionnellement de points précédents. Cette dépendance est nommée *dépendance causale*.

La dépendance causale du point *(o, t)* sur un point précédent *(o', t')* (où $t'<t$) est représentée par la relation suivante : *(o', t')* $\rightarrow$*(o, t)*. Le cône de causalité représentant tous les points dont dépend un point donné (en notant $\rightarrow^*$ la fermeture transitive de $\rightarrow$) est défini par:

$$cône(o,t) \quad = \quad \{ (o',t') \mid (o',t') \rightarrow^* (o,t) \}$$

On peut également construire l'ensemble des points dépendants d'un point donné en utilisant la définition suivante:

$$dep(o,t) \quad = \quad \{ (o',t') \mid (o,t) \rightarrow^* (o',t') \}$$

Les dépendances causales permettent de construire la structure des flots d'information à l'intérieur du système (cf. figure III.4). Du point de vue de la confidentialité, si un sujet *s* connaît le fonctionnement interne du système, alors il est capable de connaître le schéma interne des dépendances causales, et donc s'il peut observer un point de sortie x_0, alors il est capable de déduire des informations de cône(x_0). D'un point de vue de la sécurité-innocuité, toute erreur qui est détectée au point x_0 peut avoir pour cause une propagation d'erreur dans le cône(x_0). L'intégrité du point x_0 est donc conservée si aucune information erronée n'a été utilisée dans cône(x_0).

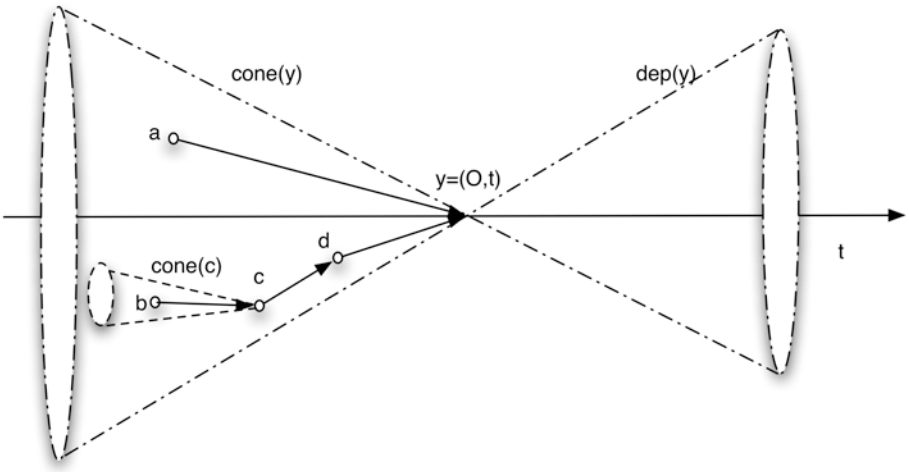


Figure III.4 Cônes de causalité et de dépendances

Un système est considéré sûr si un sujet ne peut observer directement ou indirectement que les objets qu'il a le droit d'observer directement. Si nous notons Obs_s l'ensemble des objets qu'un utilisateur *s* peut observer directement ou indirectement et R_s l'ensemble des objets que cet utilisateur est autorisé à observer directement, nous pouvons affirmer que le système est sûr si $Obs_s \subseteq R_s$.

Si l'on considère la mise en œuvre d'une politique de confidentialité multiniveau dans un tel modèle, d'Ausbourg explique que deux conditions sont nécessaires pour garantir la confidentialité. Sachant que, respectivement, un niveau de classification *l(x)* et un niveau d'habilitation *l(s)* est assigné à chaque point *x* et chaque sujet *s*, les deux conditions peuvent être exprimées comme suit, dans le cadre de la confidentialité:

- $\forall (s,x_0) \in Obs_s \quad \Rightarrow \quad l(x_0) \angle l(s)$, c'est-à-dire que l'habilitation d'un utilisateur doit dominer la classification de l'ensemble des points Obs_s qu'il peut observer
- $\forall x, \forall y, x \rightarrow y \quad \Rightarrow \quad l(x) \angle l(y)$, c'est-à-dire que si un objet *y* dépend causalement d'un autre objet *x*, alors le niveau de classification de *y* doit dominer celui de *x*.

On peut interpréter ces règles dans le cadre de l'intégrité, en définissant les règles duales (de même que Biba définit les règles duales de la politique de Bell et Lapadula) :

- $\forall(s,x_0) \in Obs_s \Rightarrow l(s) \angle l(x_0)$, c'est-à-dire que l'habilitation d'un utilisateur doit dominer la classification de l'ensemble des points Obs_s qu'il peut observer
- $\forall x, \forall y, x \rightarrow y \Rightarrow l(y) \angle l(x)$, c'est-à-dire que si un objet y dépend causalement d'un autre objet x, alors le niveau de classification de y doit dominer celui de x.

III.2.4 Modèle Clark & Wilson

Le modèle de Clark et Wilson [Clark et Wilson 1987] vise à préserver l'intégrité des informations dans un système à caractère commercial (c'est-à-dire à caractère non gouvernemental). Il introduit deux notions couramment utilisées dans les systèmes commerciaux : les transactions bien formées et la séparation des pouvoirs. La première stipule que les utilisateurs ne peuvent manipuler les informations qu'en utilisant des procédures sûres et complètement validées. La seconde impose que certaines opérations nécessitent l'intervention de plusieurs personnes pour pouvoir être menées à bien (principe de la séparation des pouvoirs).

Le modèle s'applique à certaines données identifiées dans le système, appelées "données contraintes" (CDI, *Constrained Data Items* en anglais). Toute autre donnée non contrainte est appelée UDI (*Unconstrained Data Item*). La politique d'intégrité que le concepteur doit appliquer au système se caractérise par l'utilisation de deux classes de procédures : les procédures de *vérification de l'intégrité* (IVP, *IntegrityVerification Procedure*) et les procédures de *transformation* (TP, *Transformation Procedure*). Le but des IVP est de confirmer que toutes les CDI du système sont conformes à la spécification de leur intégrité chaque fois qu'une IVP est exécutée. Les TP concrétisent la notion de transaction bien formée, leur rôle étant de manipuler les CDI tout en conservant leur intégrité.

Afin de maintenir l'intégrité des CDI, le système doit assurer que seules les TP autorisées puissent les manipuler. Si cette contrainte est vérifiée, on peut en déduire qu'à tout moment l'intégrité d'une CDI est conservée. En effet, l'intégrité d'une CDI pouvant être vérifiée à un moment donné (par une IVP), par récurrence toute suite autorisée de TP appliquée à cette donnée conservera son intégrité par la suite. Le rôle du système sera donc de maintenir une liste de relations (TP_i, ($CDI_{a,i}$, $CDI_{b,i}$, $CDI_{c,i}$,…)) où la liste de CDI définit quelles données peuvent être utilisées par cette TPi. L'ensemble de ces listes permet de déterminer également quelles TP ont le droit d'utiliser une CDI.

Bien que le système puisse assurer que les CDI ne seront manipulées que par les TP autorisées, il ne peut assurer que ces TP remplissent leur rôle. La validité de ces procédures doit donc être certifiée afin que l'ensemble du système puisse être considéré comme sûr. De même, la séparation des pouvoirs doit être définie de manière externe au système par un administrateur dont le rôle est de maintenir des relations du genre (Utilisateur, TPi, (CDIa,i, CDIb,i, CDIc,i,…)) qui précisent les données contraintes qu'un utilisateur est autorisé à manipuler par le biais de TPi..

Le point le plus intéressant de ce modèle dans notre cadre concerne les données non contraintes, c'est-à-dire non directement prises en compte dans le cadre de la politique d'intégrité. Toutefois ces données sont amenées à être utilisées par le système, ne serait-ce que parce que toute nouvelle donnée introduite est au départ non contrainte (par exemple une donnée introduite par un utilisateur). On doit donc pouvoir autoriser certaines TP à utiliser des UDI. Le modèle impose dans ce cas que la procédure de transformation utilisée produise une donnée intègre (CDI). La donnée obtenue est alors vérifiée par une IVP, assurant ainsi la cohérence globale du système.

Dans la mesure où on peut interpréter les deux types de données comme deux classes d'intégrité, on peut considérer que le modèle de Clark et Wilson définit deux niveaux d'intégrité, UDI et CDI. En suivant ce raisonnement, on s'aperçoit que le rôle des TP et des IVP est d'assurer la conservation de l'intégrité des données appartenant au plus haut niveau d'intégrité (les CDI), alors que l'association de TP et IVP bien précises peut permettre d'augmenter le niveau d'intégrité de certaines données (une CDI peut être produite à partir d'une UDI). Ceci constitue sans aucun doute le point majeur de ce modèle : il est possible de monter le niveau d'intégrité d'une donnée dynamiquement lorsque cela s'avère nécessaire. Cette propriété est exploité dans le cadre du modèle Totel pour pallier le problème de la dégradation de l'intégrité dans la politique de Biba.

III.2.5 Modèle Totel

Le modèle Totel propose une politique d'intégrité pour assurer la cohabitation d'entités logicielles ayant des criticités multiples [E. Totel, J.-P. Blanquart, et al. 1998]. Ce modèle présente l'avantage de permettre des communications bidirectionnelles entre les différentes entités logicielles, sans altérer leur niveau d'intégrité [E. Totel, Beus-dukic, et al. 1998]. Une telle communication est fort intéressante, puisqu'elle permet de se libérer des contraintes du modèle Biba, tout en gardant le même niveau d'intégrité. Nous proposons de présenter dans un premier temps les différentes composantes du modèle Totel, puis de présenter son architecture globale.

III.2.5.1 Définitions et notation

Les règles définies dans ce modèle se basent sur les éléments suivants [Eric Totel 1998]:

- Soit O l'ensemble des objets du système, qui se compose d'objets mono-niveau et d'objets multiniveau. Ces ensembles d'objets mononiveau (*SLO : Single Level Object*), et d'objets multiniveaux (*MLO : Multi Level Object*) étant disjoints, ils forment une partition de l'ensemble O, et $O = SLO \cup MLO$.
- Soit Γ l'ensemble des compartiments C du système, chaque compartiment étant défini comme un ensemble de catégories C_c. Ces catégories peuvent, par exemple, représenter les différents types d'applications qui sont présents dans le système.
- Soit Cl l'ensemble totalement ordonné des classifications c des objets (par exemple *TRÈS FIABLE, FIABLE, PEU FIABLE*), où le niveau de classification d'un objet dépend de la confiance que l'on porte dans son fonctionnement et les données qu'il gère.
- Soit I l'ensemble des niveaux d'intégrité avec $I = Cl \times \Gamma$, c'est-à-dire qu'un niveau d'intégrité est un couple composé d'une classification et d'un compartiment.;
- Soit $\angle$ la relation de dominance qui définit un ordre partiel sur I. Si deux niveaux d'intégrité sont définis par $i = (c, C)$ et $i' = (c', C')$, alors $i \angle i'$ si et seulement si $c \leq c'$ et $C \subseteq C'$. $\{I, \angle\}$ définit un treillis ;
- Soit $il : O \rightarrow I$, une fonction qui associe à tout objet o son niveau d'intégrité courant $il(o)$.

Le modèle Totel introduit alors deux types d'objets : des objets mononiveaux et des objets multiniveaux. Les objets multiniveaux peuvent être utilisés au sein d'un système d'exploitation dans le cas où un objet système doit gérer des objets ayant des niveaux d'intégrité distincts. Cependant, dans le cadre de nos travaux, nous nous intéressons aux objets mononiveaux. En effet, dans un contexte avionique, les composants se voient attribuer

un seul niveau de confiance (correspondant à un niveau de validation, crédibilité et intégrité donné). Nous proposons alors de présenter les règles de contrôle d'intégrité entre objets mononiveaux.

III.2.5.2 Flux de données entre les SLO

Comme nous l'avons précédemment mentionné, un niveau d'intégrité est affecté à chaque objet SLO. Ainsi, si *o* est un SLO, la description de son intégrité est réduite au singleton {*il(o)*}. La figure III.5 présente le modèle Totel pour une communication entre des SLO à niveaux d'intégrité distincts.

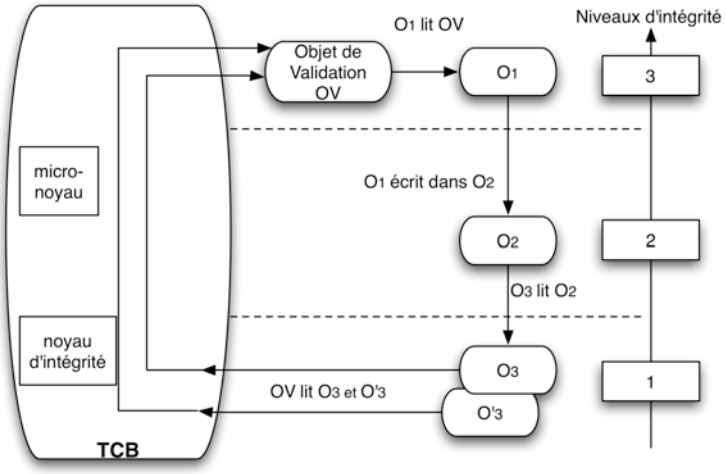


Figure III.5 Modèle Totel pour des objets mononiveaux

Dans la figure III.5, le sens des flèches correspond au sens des flux logiques d'informations entre les différents objets. Ainsi, si un objet O_3 lit un objet O_2, le sens logique de circulation de l'information est de O_2 à O_3. Remarquons que tous les flux descendant ne nécessitent aucun traitement spécial. En effet, un flux descendant respecte les règles introduites dans Biba et de ce fait, n'altère pas l'intégrité des objets.

La figure III.5 montre également un flux remontant depuis O_3 et O'_3 à destination de l'objet O_1. Un tel flux est exceptionnel, c'est pour cette raison qu'il doit être bien contrôlé pour éviter toute altération du niveau d'intégrité de l'objet O_1. Ainsi, ce flux passe par une TCB (*Trusted Computing Base*) en charge de remonter le flux d'information jusqu'à un objet de validation (OV). Un objet de validation est un objet spécialement conçu et développé pour contrôler les informations à destination d'un objet donné. Ainsi, un OV doit avoir le même niveau d'intégrité que celui de l'objet pour lequel il valide le flux d'information. Le modèle Totel ne spécifie pas comment implémenter un objet de validation, mais recommande d'utiliser des techniques de tolérance aux fautes pour assurer la validation des données reçues d'un niveau inférieur. C'est pour cette raison que sur la figure III.5 on représente deux objets O_3 et O'_3. Ces deux objets correspondent à deux entités redondantes qui peuvent être des copies (à l'identique) ou des variantes (diversifiées).

Ainsi, à travers l'utilisation d'entités redondantes, il est possible d'utiliser des mécanismes de tolérance aux fautes pour valider les informations en provenance de l'objet O_3 (et O'_3) et à destination de O_1. La TCB contient un noyau d'intégrité et un micro-noyau. Le permier est en charge de vérifier la bonne redirection des flux, depuis les ports d'entrées dans la TCB et jusqu'à leur envoi à un objet de validation. Le second noyau est en charge d'assurer la gestion de ressources entre les différents objets pour éviter qu'un objet peu critique, par son

occupation de ressources communes, altère le comportement d'un objet plus critique. Rappelons que cette fonctionnalité est l'une des principales caractéristiques des modules IMA en avionique.

Le modèle Totel présente ainsi l'avantage de permettre, sous certaines conditions, une remontée de flux d'un niveau peu critique à un niveau plus critique. Cette remontée est assurée par un objet de validation qui utilise des techniques de tolérance aux fautes, et éventuellement des entités redondantes.

***

Dans le cadre de nos travaux, nous nous intéressons à des entités redondantes sur une seule machine physique. Nous proposons alors l'utilisation de la virtualisation pour assurer une telle redondance. Dans le chapitre IV, nous présentons plus en détails cette technique de virtualisation et soulignons son intérêt vis-à-vis de la virtualisation.

Remarquons que l'approche Totel est très intéressante dans un contexte avionique. En effet, il serait intéressant d'enfin avoir une telle communication bidirectionnelle entre les différents domaines avioniques identifiés, tout en gardant les mêmes niveaux de confiance. L'utilisation de ce modèle et son adaptation au contexte avionique sont ainsi présentées dans les chapitres V et VI.

III.3 CONCLUSION

Dans ce chapitre, nous avons :

- présenté une vision cohérente du développement des systèmes critiques, en nous appuyant sur les normes et références dans le domaine.
- montré que l'approche multiniveau est commune aux standards existants.
- introduit une distinction entre le niveau de criticité (intrinsèque à une tâche) correspondant à la sévérité de la défaillance, et le niveau de confiance que l'on peut avoir dans un module réalisant la tâche .
- classé les paramètres de niveau de confiance en trois attributs : la validation (durant la phase de conception), la crédibilité (des sources à l'exécution), et l'intégrité (la non altération du module et de ses données).
- présenté différents modèles proposés pour étudier les interactions entre les différents niveaux d'intégrité.

Parmi les différents modèles présentés, nous avons vu que le modèle Totel présente une approche fort intéressante qui peut être exploitée dans le domaine de l'avionique. Ce modèle utilise des techniques de tolérance aux fautes pour augmenter le niveau de confiance que l'on peut avoir dans une application donnée. Dans le cadre de nos travaux, nous nous intéressons à des applications présentes sur un même module physique. C'est pour cette raison que nous proposons d'étudier plus en détail l'apport de la technique de virtualisation dans la tolérance aux fautes dans le chapitre suivant.

Chapitre IV

LA VIRTUALISATION POUR LA TOLERANCE AUX FAUTES

INTRODUCTION

Depuis quelques années, la virtualisation s'est imposée sur le marché des logiciels comme technique permettant l'exécution de plusieurs instances de système d'exploitation sur une seule machine physique.

Dans ce chapitre, nous proposons de faire le lien entre la virtualisation et la tolérance aux fautes. Pour ce faire, nous rappelons les principales caractéristiques d'un système tolérant aux fautes puis nous proposons de définir plus clairement la virtualisation.

À travers la définition de la virtualisation, nous présentons les différentes implémentations qui en découlent, et discutons une taxonomie envisageable pour les différents moniteurs de machines virtuelles.

Les implémentations proposées peuvent être utilisées dans des domaines disjoints, en fonction des propriétés qu'elles possèdent. Nous avons donc classé les cas d'utilisation des machines virtuelles par analogie à ce qui existe déjà dans le domaine de la tolérance aux fautes, tout en mettant l'accent sur l'utilisation de la virtualisation dans le cadre de systèmes tolérants aux fautes.

IV.1 RETOUR SUR LA TOLERANCE AUX FAUTES

Comme nous l'avons présenté dans le chapitre II (section II.2), la tolérance aux fautes se base essentiellement sur le principe de redondance, afin d'offrir la possibilité de tolérer une ou plusieurs fautes. Ces fautes peuvent avoir différentes origines (physique, conception, utilisation, malveillance), et en fonction de ces origines, la faute peut concerner aussi bien le matériel que le logiciel. Cependant, quelle que soit la nature de l'entité défaillante, la politique de tolérance aux fautes se base toujours sur la redondance. Dans ce qui suit, nous nous intéressons à dégager les propriétés qu'un système tolérant aux fautes doit avoir afin de permettre la détection et/ou le recouvrement de l'erreur produite par cette faute. Une fois ces propriétés établies, nous nous intéresserons ensuite aux différentes architectures tolérantes aux fautes que l'on peut déployer, en fonction des hypothèses de fautes.

IV.1.1 La redondance et les systèmes tolérants aux fautes

Un système informatique peut être schématiquement représenté comme un ensemble de composants logiciels s'exécutant sur un support matériel. Ainsi, chaque composant C peut s'exécuter sur un support S selon un schéma qui représente la brique élémentaire d'un système informatique (comme le montre la figure IV.1.A).

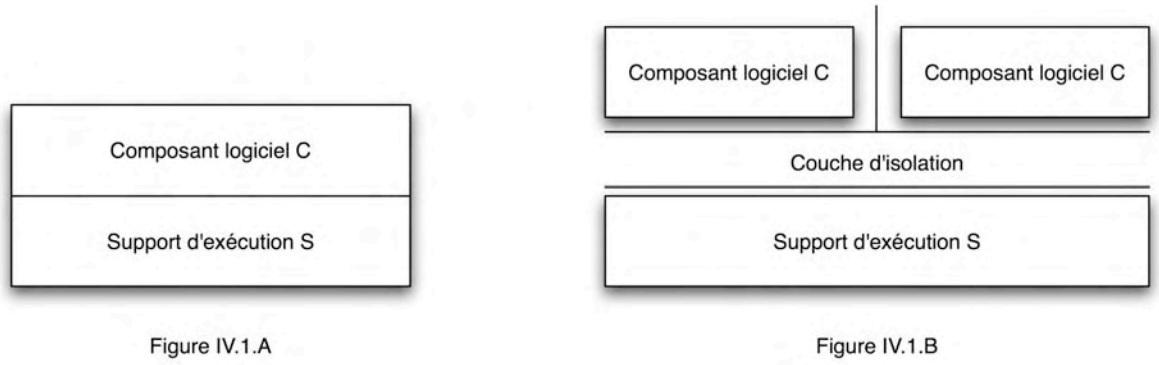


Figure IV.1 Brique élémentaire d'un système informatique

Dans le cas le plus simple, le composant C est unique, et le support d'exécution a pour rôle d'assurer la bonne exécution de C. Ce cas de figure correspond à ce qui existe dans les systèmes avioniques critiques. En effet, à chaque calculateur (support d'exécution) correspond une tâche logicielle spécifique réalisée par un seul composant. Cependant, il est possible d'avoir plusieurs composants logiciels qui s'exécutent sur un même support S, auquel cas, une nouvelle couche doit être ajoutée entre les composants et les supports afin d'assurer un multiplexage temporel entre les différents composants (cf. figure IV.1.B).

Prenons le cas simple dans lequel un seul composant s'exécute, et étudions de plus près comment rendre l'exécution de ce composant tolérante aux fautes. Il est clair que les mécanismes de tolérance aux fautes déployés dépendent des hypothèses de fautes que l'on considère pour le système en question. Si la faute concerne le matériel, une redondance des supports d'exécution doit être réalisée et le composant C est exécuté plusieurs fois sur des supports physiques différents. Cependant, beaucoup de fautes matérielles sont transitoires, elles peuvent être tolérées en exécutant plusieurs fois le même logiciel, sur le même matériel [Y. Deswarte et Lavictoire 1975]. Les résultats des différentes exécutions sont ensuite

comparés et selon un algorithme de décision (cf. chapitre II sur la tolérance aux fautes), un résultat final cohérent est fourni. Si la faute est logicielle, la redondance doit être réalisée au niveau des composants logiciels, ce qui implique l'utilisation de composants diversifiés. L'ensemble doit être coordonné afin de fournir un résultat global cohérent entre toutes les versions implémentées du composant C.

Quelle que soit la nature de la faute (logicielle ou matérielle, accidentelle ou malveillante), la redondance se base sur le principe d'utilisation de plusieurs instances (de composant ou de support) dans des zones dites d'indépendance. Ces zones d'indépendance permettent ainsi des exécutions multiples des différentes instances d'un système donné, en assurant une complète isolation entre les différents sites d'exécution. Une faute altérant une instance ne peut donc pas corrompre le fonctionnement des autres instances, ce qui conduit à la construction de zones dites de confinement d'erreur (cf. figure IV.2).

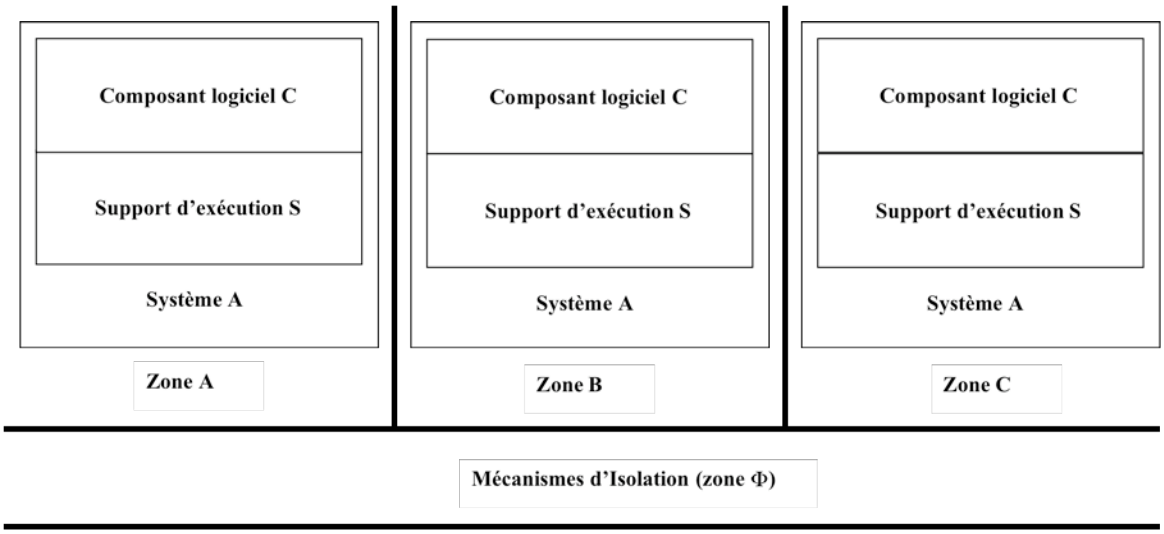


Figure IV.2 La redondance pour la tolérance aux fautes

L'exécution d'un même système dans plusieurs zones de confinement permet ainsi, à travers un algorithme de décision, de fournir un résultat correct (sous l'hypothèse que l'on ait évité toute faute de mode commun). L'algorithme de décision peut être implanté soit dans la couche assurant l'isolation (la couche de Mécanismes d'Isolation de la figure IV.2), soit dans une zone d'indépendance différente de celles des autres instances du système (dans une quatrième zone Φ, par exemple). Notons que cette zone doit être une zone d'indépendance, et ce quelle que soit le niveau d'implantation de ce mécanisme de décision.

Remarquons que dans la figure IV.2, les zones A, B et C contiennent des instances du même système, à savoir le même composant logiciel sur trois exemplaires du même support d'exécution. Ce type de redondance permet de tolérer les fautes physiques (liées au matériel ou à l'emplacement géographique du système, par exemple une coupure de courant dans une ferme de serveurs). Cependant, il est possible de tolérer des fautes liées à la conception du matériel ou du logiciel du système A (de la figure IV.2) en assurant une diversification entre les différentes instances du système.

Pour tolérer une faute matérielle, un même composant C est exécuté sur des supports d'exécution différents. Le composant logiciel est gardé le même dans ce cas, mais il doit être compilé différemment afin de prendre en compte le changement du support matériel sur lequel il va s'exécuter (cf. section IV.2).

IV.1.2 La redondance logicielle

L'utilisation d'un support matériel différent pour chaque instance du système est un mécanisme largement utilisé en avionique, dans les systèmes de commande de vol électrique. Cependant, un tel mécanisme, à lui seul, ne permet pas de tolérer les fautes qui peuvent survenir au niveau du composant logiciel. En plus, utiliser un matériel différent pour chaque instance peut s'avérer coûteux en terme d'investissement économique. Dans certains cas, la redondance matérielle est tout simplement impossible parce que l'utilisation de plusieurs machines nécessite un aménagement physique approprié de l'emplacement des machines, et ceci n'est pas toujours possible (dans les avions par exemple). Il est donc impossible de réaliser une tolérance aux fautes matérielles, ce qui implique que lors de la conception du système, il faut faire en sorte que le matériel unique soit suffisamment fiable pour exécuter les composants qui y sont installés. Il convient donc de ne pas installer des composants très critiques sur un matériel qui ne peut être diversifié.

L'utilisation de plusieurs machines en parallèle pose des problèmes d'interface homme-machine : il est difficile de demander à un opérateur de comparer par exemples des affichages différents des mêmes informations sur différents écrans, et d'entrer plusieurs fois des commandes équivalentes mais différentes (par des souris ou des claviers) sur différentes machines. Ceci implique qu'il est en pratique impossible d'implanter des instances diversifiées d'interfaces homme machine. L'implantation d'une IHM tolérante aux fautes suppose que l'on ne doit en aucun cas demander directement à l'opérateur de réaliser plusieurs entrées sur des IHM redondantes (même si dans certains critique de la navette spatiales, certaines commandes doivent être entrées en redondance par plusieurs pilotes, sur des IHM distinctes). Il faut utiliser alors un mécanisme qui permet à la fois une diversification logicielle et une interaction unique avec l'opérateur.

Dans le cadre de ce manuscrit, nous nous intéressons en premier aux fautes malveillantes atteignant le logiciel. Ceci s'inscrit dans le cadre de cas d'étude identifiés avec Airbus et dans lesquels l'utilisation d'une seule machine physique est dictée par des besoins fonctionnels. L'architecture de tolérance aux fautes est donc basée sur une redondance logicielle, dans laquelle plusieurs exemplaires d'un même composant logiciel sont exécutés sur un seul support matériel.

Pour réaliser une telle redondance, le même principe précédemment décrit est utilisé, à savoir qu'une couche assurant l'indépendance entre les différentes zones doit être implémentée, et dans chaque zone, seul le composant logiciel est exécuté (cf. figure IV.3).

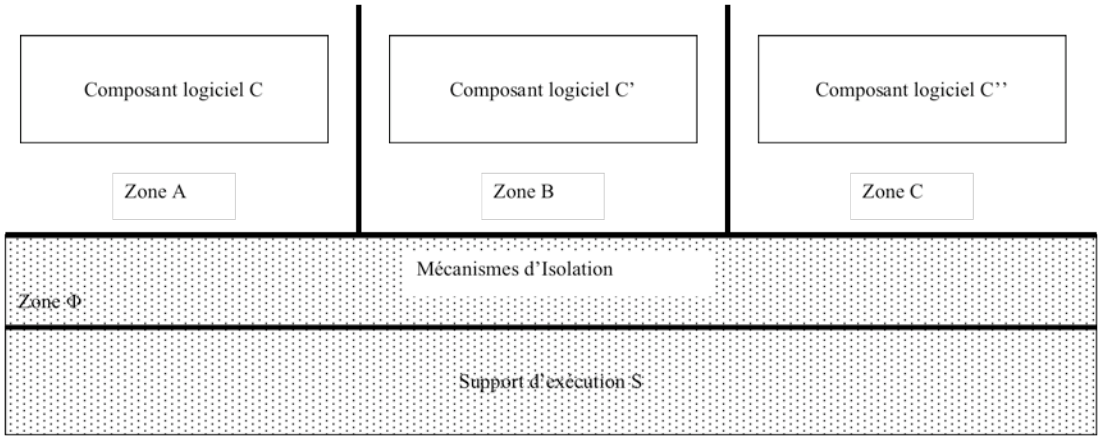


Figure IV.3 Redondance logicielle sur un même support matériel

Remarquons que le support d'exécution ne fait plus partie d'une zone d'indépendance particulière, mais qu'il est accessible par les différents composants logiciels des zones A, B et C. Ces composants logiciels sont ainsi diversifiés, mais accomplissent la même tâche. Ainsi, par exemple pour une tâche accomplissant le calcul de paramètres de vol, le composant C peut être implémenté en utilisant le langage C, le composant C' en utilisant le langage Pascal et le composant C'' en utilisant le langage Ada. Chez Airbus, une telle diversification logicielle a été utilisée depuis les premiers A320, et les composants ont même été développés par des équipes différentes, utilisant chacune des algorithmes différents, afin de tolérer au mieux les fautes de conception.

Il est important de souligner à nouveau que les algorithmes de décision, même s'ils ne sont pas présentés sur la figure IV.3, doivent être implémentés soit dans la couche d'isolation soit dans une autre zone d'indépendance.

Remarquons également qu'un système (tel que celui présenté dans la figure IV.2) ne comporte plus un composant C sur un support S dans une même zone d'indépendance, mais un composant C, isolé dans une zone d'indépendance, et un support S commun à plusieurs composants C. Jusqu'ici, nous avons fait abstraction des différentes interactions qui peuvent exister entre les composants, les mécanismes d'isolation et les supports d'exécution. Dans la section suivante, nous détaillons ces interactions qui présentent la base d'une technique devenue de plus en plus répandue : la virtualisation.

IV.2 LA VIRTUALISATION

IV.2.1 Origines de la virtualisation

Les systèmes informatiques sont des systèmes de plus en plus complexes, offrant des fonctionnalités des plus en plus conviviales aux utilisateurs, à travers l'introduction de bibliothèques graphiques étendues ou des services logiciels à large spectre d'action, tout cela dans un environnement d'utilisation intuitive à l'utilisateur final. Une telle complexité a toujours été gérée à travers l'introduction de niveaux d'abstraction qui permettent de s'abstraire des fonctionnalités propres de chaque composant, matériel ou logiciel, en définissant des interfaces de programmation appropriées. Ces interfaces concernent aussi bien les couches logicielles que les couches matérielles. L'introduction du concept des Architectures de Jeux d'Instructions, aussi connues sous le terme ISA (*Instruction Set Architecture*) dans la famille IBM 360 dans les années soixante, a permis d'assurer une meilleure portabilité des logiciels d'une famille de processeurs à une autre. Ces jeux d'instructions permettent de séparer la partie logicielle de la partie matérielle d'une machine (cf figure IV.4). Ainsi, il suffit d'avoir une couche qui s'occupe de la traduction des instructions logicielles en instructions matérielles pour que l'application en question soit opérationnelle sur la machine physique en question. Cette idée fut à l'origine de l'utilisation des machines virtuelles, et a conduit à leur évolution telle qu'on les connaît actuellement.

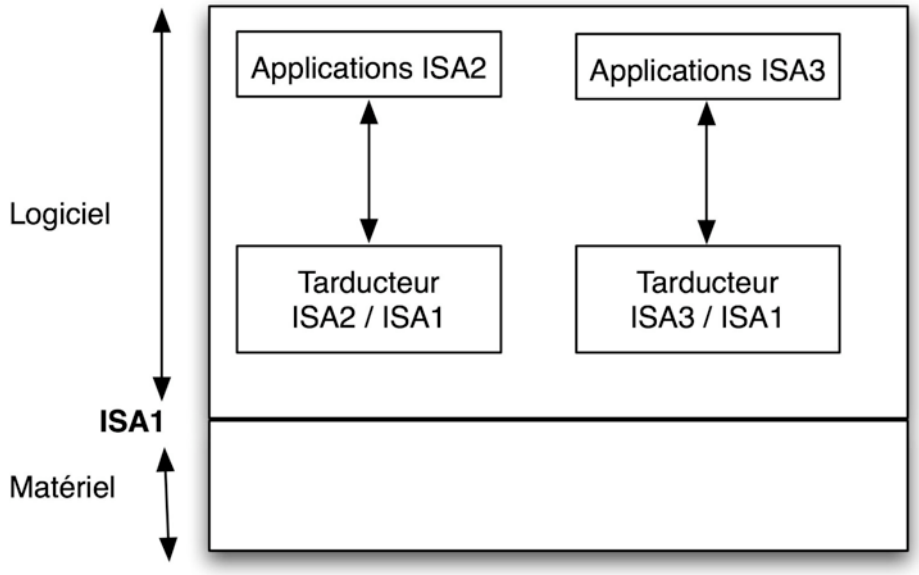


Figure IV.4 Utilisation de traducteur ISA dans les IBM 360

IV.2.2 Définition simple de la virtualisation

La virtualisation est une technique qui est bien répandue de nos jours et est présente dans divers domaines. Wikipédia, l'encyclopédie en ligne, définit la virtualisation comme « l'ensemble des techniques matérielles et/ou logicielles qui permettent de faire fonctionner sur une seule machine plusieurs systèmes d'exploitation et/ou plusieurs applications, séparément les uns des autres, comme s'ils fonctionnaient sur des machines physiques distinctes » [Wikipédia].

Cette définition est plutôt de haut niveau et ne permet pas de refléter les mécanismes qui se cachent derrière la mise en œuvre d'un tel parallélisme d'exécution entre les différents systèmes d'exploitation et/ou applications. Cependant, cette définition souligne l'importance de la séparation des différents systèmes exécutés. En effet, cette séparation permet de garantir des propriétés d'isolation importantes que nous détaillerons dans ce chapitre. Dans ce qui suit, nous présentons d'abord une vue plus détaillée des différents types d'abstraction communément utilisés dans les architectures logicielles afin d'en déduire les différents types de virtualisation qui peuvent en découler.

IV.2.3 Interfaces d'abstraction logicielles

Les couches matérielles sont accessibles aux logiciels par les instructions ISA [Tanenbaum 2001]. La couche logicielle est composée de deux entités principales : les applications et le système d'exploitation (cf. figure IV.5). Certaines instructions sont disponibles aux applications, ces instructions sont définies par l'architecture de jeu d'instruction utilisateur (*user ISA*). D'autres instructions sont disponibles uniquement pour le système d'exploitation, et sont appelées instructions système (*system ISA*). Certaines instructions peuvent être utilisées aussi bien par les applications que par le système d'exploitation, comme le montre la figure V.5.

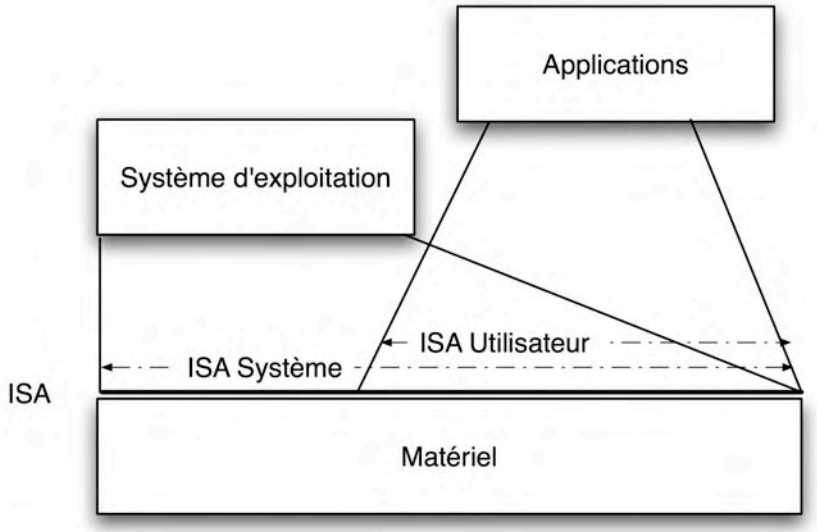


Figure IV.5 Architecture de jeu d'instructions (ISA)

La couche applicative contient également des bibliothèques qui servent d'interface intermédiaire supplémentaire entre les applications et le matériel. Ces bibliothèques peuvent fournir une abstraction directe du matériel, mais elles peuvent également servir d'interface entre les applications et le système d'exploitation (cf. figure IV.6). L'interaction entre le système et le matériel passe par différentes entités logicielles qui servent à gérer des ressources et à commander des périphériques spécifiques, comme le gestionnaire mémoire, le gestionnaire des pilotes et l'ordonnanceur.

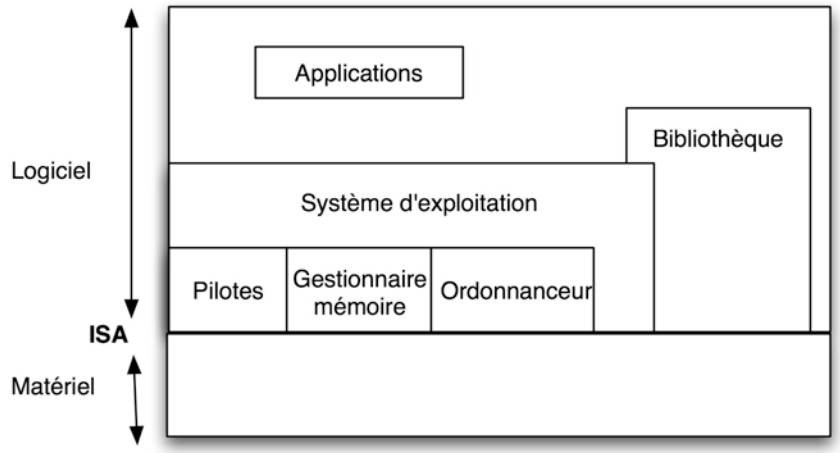


Figure IV.6 Architecture logicielle d'une machine

La couche matérielle peut également être découpée en plusieurs entités, cependant ce découpage n'est pas traité dans le cadre de ce manuscrit. Ainsi, en fonction des différentes entités logicielles présentes dans une machine, les applications ont différentes possibilités pour interagir avec le matériel. Ces interactions peuvent être groupées en deux catégories : les Interfaces Binaires d'Application (*ABI* :*Application Binary Interface*) et les Interfaces de Programme d'Application (*API* :*Application Program Interface*).

IV.2.3.1 Interface binaire d'application (ABI)

Les ABI donnent aux applications un accès direct aux ressources matérielles. Ces interfaces peuvent être subdivisées en deux catégories : les architectures de jeu d'instruction utilisateurs (*user ISA*) présentées dans la section précédente et les appels systèmes (*system call*). Les appels systèmes présentent une suite d'opérations que le système effectue pour répondre à un besoin spécifique d'une application. Un appel système est implémenté de manière à passer la main au système d'exploitation. Cet appel ressemble à un appel de procédure (*procedure call*) dans lequel la procédure appelée est une routine spécifique du système d'exploitation (cf. figure IV.7). Les arguments de l'appel système sont passés soit dans des registres prédéfinis du processeur, soit dans des zones mémoires dédiées à cet effet, en fonction des spécifications de l'interface système.

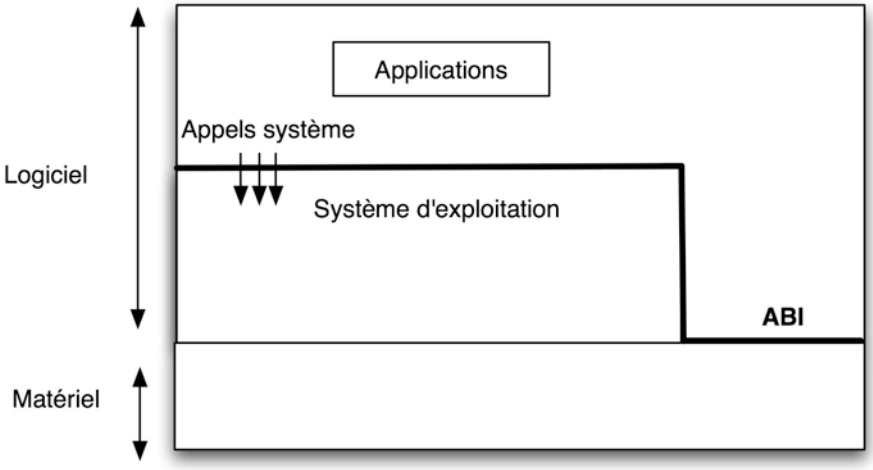


Figure IV.7 Interface Binaire d'Application

IV.2.3.2 Interfaces de Programme d'Application

Les API, encore connues sous le nom « interfaces de programmation », sont un ensemble de fonctions, procédures, instructions systèmes mises à la disposition du programmeur d'application sous la forme d'une bibliothèque (*library*) (cf. figure VI.8). Ces interfaces sont souvent définies par rapport à un langage de programmation de haut niveau (*High-Level Language*). Par exemple, la programmation d'une application graphique peut s'avérer très compliquée si le programmeur doit se soucier de tous les détails d'implémentation de chaque objet graphique. Grâce à des bibliothèques graphiques (OpenGL ou Direct3D, par exemple), l'instanciation d'objets graphiques se fait plus facilement en manipulant les fonctions fournies par les interfaces de programmation.

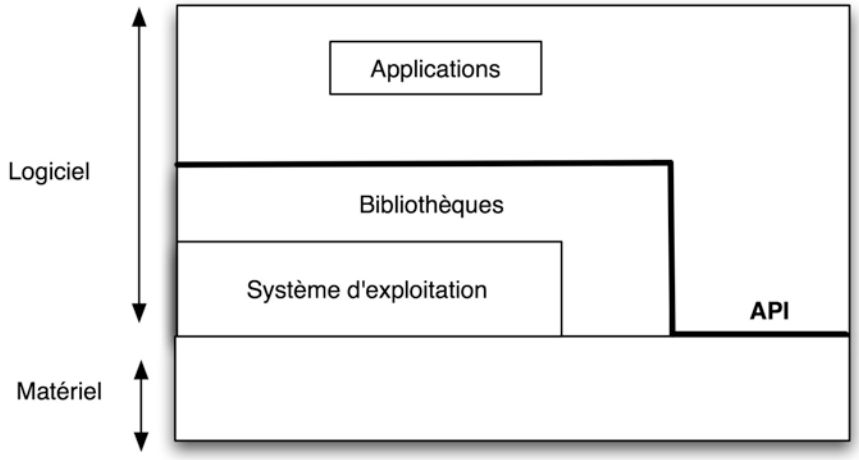


Figure IV.8 Interface de Programme d'Application

Les fonctions ainsi fournies par ces interfaces comportent donc des appels à des services disponibles sur le système d'exploitation et permettent ainsi de fournir une abstraction de ces services au niveau des applications. Ainsi, un appel d'une interface de programmation comporte un ou plusieurs appels systèmes contenus dans une interface binaire d'application. Dans certains cas, les API ont pour rôle de traduire les instructions provenant d'un langage de programmation haut niveau vers des instructions prises en charge par le système d'exploitation, ce qui peut être vu comme un simple *wrapper* du système d'exploitation en question.

IV.2.4 Virtualisation et Interfaces Logicielles

Les différentes interfaces logicielles ayant été décrites, nous pouvons définir la virtualisation par rapport aux différentes couches logicielles installées sur une machine physique. La virtualisation est considérée alors comme un mécanisme permettant à une entité logicielle (une application unique ou tout un système d'exploitation) de s'exécuter sur n'importe quel support de la même façon que sur le support pour lequel elle a été conçue. Dans la section IV.1.1, le terme support d'exécution a été présenté comme constitué du matériel. Dans ce qui suit, nous définissons le support d'exécution d'un logiciel comme étant l'ensemble des couches sur lesquelles s'exécute le logiciel en question. Le support peut donc être le matériel si nous nous intéressons au système d'exploitation comme entité logicielle. Il peut être le système et le matériel si l'entité logicielle en question est une simple application. Prenons quelques exemples pour clarifier cette définition. Une application conçue pour Windows, peut s'exécuter sur un Linux, grâce à une couche de virtualisation : ici, la virtualisation fournit un support d'exécution et une API Windows au-dessus du support d'exécution Linux. Un

système d'exploitation de type MacOS X conçu pour s'exécuter sur un processeur de type PowerPC peut s'exécuter sur un matériel Intel, grâce à une couche de virtualisation, qui dans ce cas fournit à MacOS X un support d'exécution qui émule le matériel et le jeu d'instruction du PowerPC sur un support matériel Intel.

D'après cette définition, nous constatons déjà qu'il y a une première classification des différents types de virtualisation ; un premier type concerne les applications (connu sous le nom de machine virtuelle de processus) et le second concerne les systèmes d'exploitation (connu sous le nom de machine virtuelle de système).

IV.2.4.1 Machine virtuelle de processus

La figure 9 représente le principe de fonctionnement d'une machine virtuelle de processus.

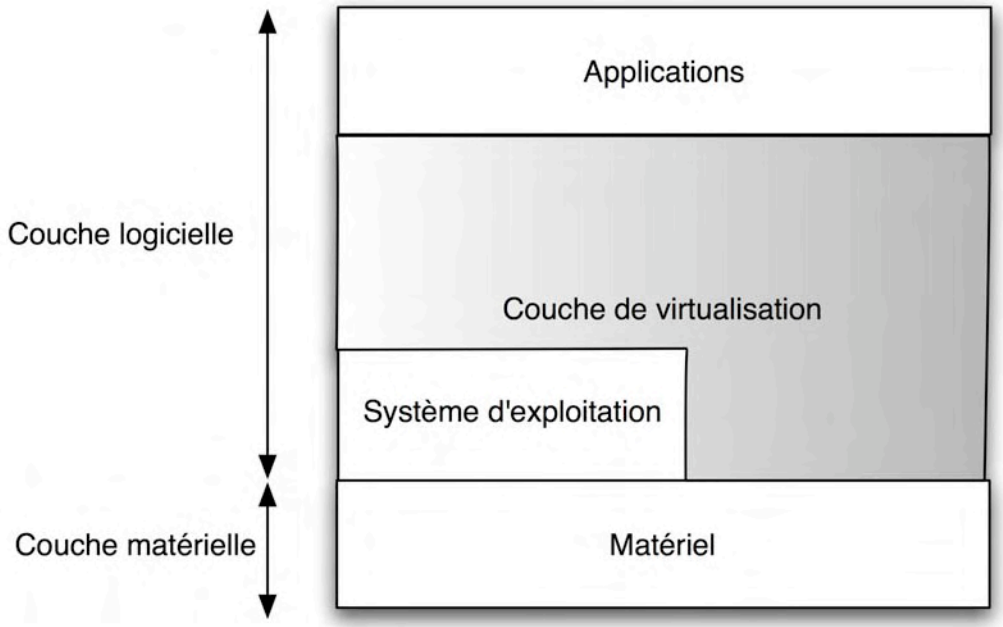


Figure IV.9 Machine Virtuelle de processus

Les applications de la figure IV.9 sont appelées applications invitées et l'ensemble système d'exploitation et matériel est appelé machine hôte. La couche de virtualisation fournit à l'application une API virtuelle, permettant ainsi un large spectre d'utilisation que nous détaillons dans ce qui suit :

La multiprogrammation

Une des utilisations les plus répandues de machines virtuelles de processus est l'implémentation de systèmes d'exploitation multitâches. En effet, chaque tâche s'exécute sur le système comme si elle était la seule à s'exécuter. Le système d'exploitation fournit à chaque processus applicatif un espace d'adressage mémoire et de disque et il ordonnance les accès de chaque processus au CPU (ou autre périphérique partagé). La tâche a une vision « virtuelle » du matériel dont elle dispose, puisqu'elle considère que ce matériel est toujours libre pour exécuter ses instructions, et le système peut être vu comme une couche de virtualisation fournissant une abstraction du matériel, différente de l'état réel du matériel (un processeur qui est occupé alors qu'il est vu comme libre par la tâche applicative).

L'émulation

Comme présenté dans la section IV.2.1, un traducteur de jeu d'instructions permet à une application conçue pour un matériel spécifique de s'exécuter sur un matériel différent. Autrement dit, une application conçue pour un premier jeu d'instruction est exécutée sur une autre machine présentant un jeu d'instruction différent. Une illustration de ce concept est l'émulateur FX !32 qui permet à des applications x86 Windows de s'exécuter sur des machines Alpha [Chernoff et Hookway 1997].

Les langages de haut niveau

L'exemple de l'émulateur FX !32 précédemment décrit permet à un code développé pour une plateforme largement utilisée (par exemple Windows sur PC) d'être exécutée sur une plateforme moins utilisée (Plateforme Alpha). Cette propriété de portage est intéressante. Cependant pour avoir une même application qui s'exécute sur une architecture MIPS, SPARC ou PowerPC, il faudrait implémenter des émulateurs qui se basent sur le même principe que FX!32. Il est clair qu'une telle solution n'est pas facile à déployer vu qu'elle dépend de chaque plateforme d'exécution.

Les langages de haut niveau (*High-Level Languages*) présentent au développeur une abstraction de développement indépendante de la plateforme sous-jacente. En effet, une chaîne de compilation classique passe par une génération de code intermédiaire, en se basant sur une analyse lexicale, syntaxique et sémantique du code source. Ces analyses dépendent du contenu du code source et non pas de la nature des couches logicielles inférieures (interfaces binaires, applicatives ou jeux d'instructions), puis ce langage intermédiaire est lui-même compilé pour générer du code exécutable sur une plateforme donnée. Les langages de haut niveau peuvent donc être vus comme offrant au développeur un ensemble de règles de programmation qui ne dépendent pas de la plateforme d'exécution. Pour certains de ces langages, le compilateur ne génère pas de code exécutable, mais s'arrête à la génération du code intermédiaire, et ce code intermédiaire ainsi généré est interprété ensuite par une machine virtuelle spécifique de la plateforme. Ainsi ces langages nécessitent effectivement la présence d'une machine virtuelle qui traduit le code intermédiaire en code binaire propre au matériel d'exécution. Cette approche a été introduite dans les années 80 avec l'environnement de programmation Pascal [Bowles 1980]. Le langage CLI (*Common Language Infrastructure*) développé par Microsoft est la base du développement du *cadriciel* (pour traduire *framework*) .Net [Box et Sells 2002]. CLI est un langage de programmation de haut niveau qui se base sur le principe de *bytecode* comme code intermédiaire, qui peut être exécuté sur des plateformes différentes. Les architectures de machines virtuelles Java de Sun se basent sur le même principe : les programmes écrits en Java génèrent du *bytecode* exécutable sur des plateformes différentes.

IV.2.4.2 Machine virtuelle de système

Dans de nombreux travaux récents, on désigne par virtualisation l'utilisation de machine virtuelle de système. Il s'agit donc d'exécuter tout un système d'exploitation sur une architecture de jeu d'instruction qui peut ne pas être la même que celle fournie le matériel. Dans ce cas, la virtualisation consiste à remplacer l'ISA réelle du matériel en fournissant soit la même ISA et on parle dans ce cas de virtualisation classique des systèmes d'exploitation, soit une ISA différente, auquel cas on parle d'émulation matérielle ou encore virtualisation d'un système global. La figure IV.10 représente une classification des différentes machines virtuelles selon une taxonomie proposée par Smith et Nair [Smith et Nair 2005].

Ce type de virtualisation étant celui qui concerne le présent manuscrit (à savoir l'utilisation simultanée de plusieurs systèmes d'exploitation sur un matériel unique), nous proposons donc de détailler les différentes implémentations possibles dans la section suivante. Ces implémentations ont des caractéristiques distinctes, se basant sur des propriétés liées à la disposition des couches d'abstraction logicielle précédemment décrites. Nous proposons alors de citer quelques possibilités d'utilisation de ces implémentations pour répondre à des cas d'étude (de la simple isolation à l'équilibrage des charges en passant par des systèmes de détection d'intrusion).

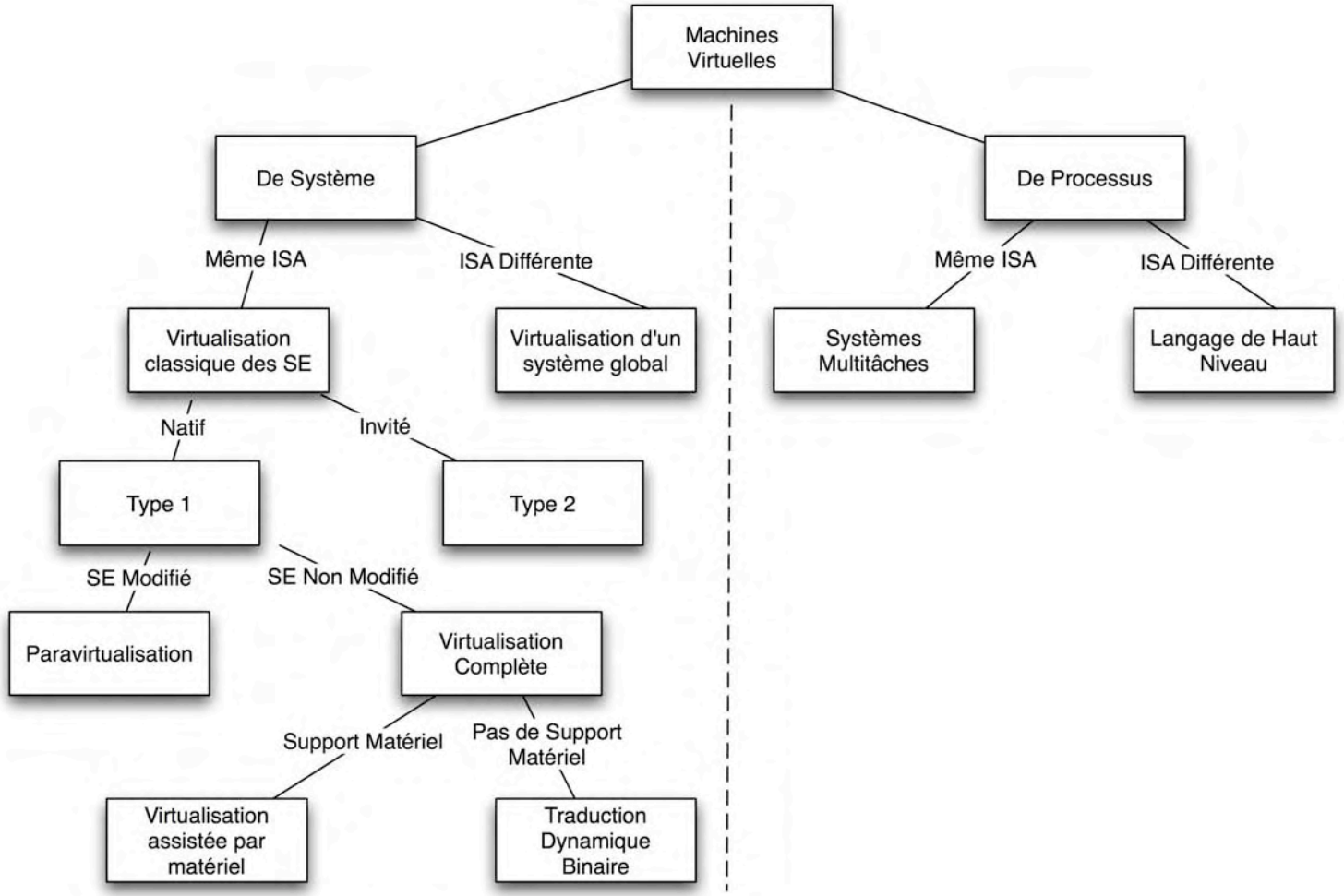


Figure IV.10 Différents types de machines virtuelles [Smith et Nair 2005].

IV.3 IMPLEMENTATION ET UTILISATION DE LA VIRTUALISATION

IV.3.1 Implémentations de machines virtuelles de système

Différentes implémentation de machines virtuelles de système ont été proposées. Nous proposons de présenter dans ce qui suit les principales implémentations : les isolateurs, la virtualisation complète, la virtualisation assistée par matériel et la paravirtualisation.

IV.3.1.1 Isolateur

Les isolateurs sont des espaces mémoires utilisateurs servant comme environnements d'exécution multiples d'un même noyau. Cette technique est aussi appelée virtualisation au niveau des systèmes d'exploitation (*operating-system level virtualization*), et peut être considérée comme une sorte de bac à sable (*sandbox*) qui permet de fournir une isolation forte pour exécuter plusieurs ensembles d'application en parallèle (cf. figure IV.11).

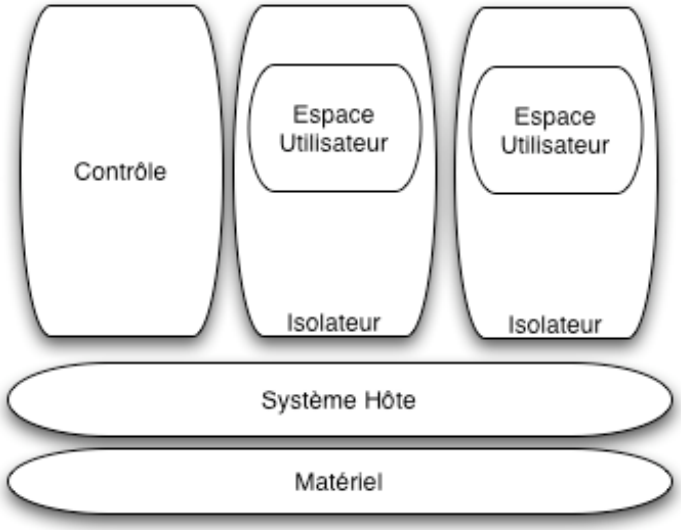


Figure IV.11 Implémentation des Isolateurs

L'avantage de cette technique est qu'elle présente un faible surcoût en terme de charge CPU. Cependant les opérations permises à chaque espace utilisateur sont limitées, afin d'éviter toute interférence avec les autres espaces utilisaeurs. Certains systèmes d'exploitation implantent ce type de virtualisation, citons par exemple Linux-VServer, OpenVZ/Virtuozzo et Solaris Zones. Ce type de virtualisation est de plus en plus utilisé pour des serveurs départementaux, où sur une même machine physique on peut implémenter plusieurs services indépendants de fichiers, de courrier électronique, de Web, etc. En général, la défaillance (*plantage*) d'un de ces services, n'a pas de conséquence pour les autres.

L'utilisation de ce type d'isolation au niveau des espaces utilisateurs peut être étendue à l'utilisation de noyaux en mode utilisateur. Dans ce cas un noyau hôte se charge de l'ordonnancement des différents noyaux en espace utilisateur. L'isolateur de la figure IV.11 est alors remplacé par un noyau de système d'exploitation. Cette technique est notamment utilisée dans *User Mode Linux* [Hoxer et al. 2002], coLinux, Adeos, L4Linux. Elle permet en

particulier aux développeurs de noyaux de système de déboguer leur implémentation. Ce type de virtualisation est alors une virtualisation classique des SE de type 2 (selon la figure IV.10), contrairement à l'isolateur, qui est toujours implémenté sur un système hôte, et n'est pas directement implémenté sur le matériel.

IV.3.1.2 Virtualisation complète

La virtualisation complète (*full virtualization*) consiste à émuler l'intégralité d'une machine physique pour le système invité. Le système invité n'est pas modifié et il exécute son code comme s'il était sur une véritable machine physique. Le logiciel chargé d'émuler cette machine physique est appelé machine virtuelle et son rôle consiste à traduire les instructions du système invité en instructions pour le système hôte.

En effet, comme le montre la figure IV.12, la machine virtuelle est considérée, par le système hôte comme n'importe quel autre processus applicatif, tel qu'un éditeur de texte ou un navigateur Internet. Cependant, rappelons qu'un système d'exploitation s'exécute dans un mode privilégié du processeur, et qu'il a, par conséquent, accès à des instructions privilégiées, réservées au système d'exploitation. Toutefois, considérer une machine virtuelle comme un simple programme applicatif nécessite que le système hôte accomplisse les routines système provenant de la machine invitée (machine virtuelle). La machine virtuelle émule donc de manière logique (c'est-à-dire avec du code) tout le matériel classique de l'architecture de l'ordinateur cible. Ainsi, sur la figure IV.12, seul le système hôte a un accès direct à la couche matérielle, il est en charge de traduire toutes les instructions système provenant des machines virtuelles.

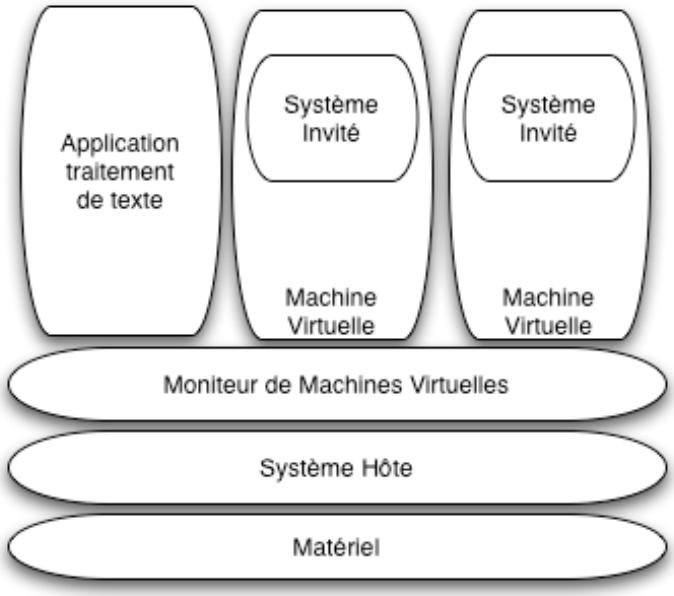


Figure IV.12 Virtualisation Complète

En pratique, le disque dur de la machine virtuelle est la plupart du temps géré comme un fichier volumineux, (de l'ordre de plusieurs Go) par le système hôte, alors que la mémoire vive dont le système invité dispose est réservée, généralement au moment de l'instanciation des machines virtuelles, par le moniteur de gestion des machines virtuelles. La gestion du reste des périphériques physiques varie considérablement selon les implémentations.

Cependant, on retrouve généralement au moins une carte réseau générique, une carte graphique générique et un clavier 105 touches standard.

L'utilisation de périphériques basiques s'explique par le fait qu'il y a toujours un nombre minimal d'opérations supportées par toute catégorie de matériel sur un ordinateur : la vitesse de transfert la plus lente pour un disque dur, la résolution d'affichage la plus faible pour une carte graphique, etc. Or comme le comportement de ces périphériques est entièrement implémenté de manière logicielle par la machine virtuelle, émuler un périphérique avec le minimum de fonctionnalités permet de limiter la quantité de code à développer pour en couvrir le comportement. C'est la raison pour laquelle les cartes graphiques et les cartes réseaux sont la plupart du temps au niveau des standards en vigueur dans les années quatre-vingt.

Ce n'est toutefois pas la seule raison de l'émulation de périphériques de base. En effet, la plupart des évolutions ultérieures du matériel visent à améliorer les performances des périphériques, par exemple en augmentant le débit du disque dur ou la résolution supportée par la carte graphique. Cependant, les optimisations de performances sont dans ce cas sans objet, car elles ne se répercutent de toute manière pas sur un matériel physique en mesure de les supporter. La rapidité du disque dur virtuel est par exemple limitée par la vitesse d'accès au fichier le représentant sur le système hôte. Notons que dans ce cas de virtualisation, le matériel ne prend pas en considération l'instanciation d'une machine virtuelle par le système hôte. De ce fait, les optimisations matérielles fournies à la machine virtuelle se trouvent limitées par la nature de l'abstraction utilisée pour implémenter ce matériel (comme l'implémentation du disque sous la forme d'un fichier).

Le système s'exécutant dans la machine virtuelle est un système d'exploitation à part entière, tel qu'on pourrait en installer sur une machine physique. Les systèmes peuvent donc être de type COTS, tels que Microsoft Windows, GNU/Linux, MacOS X, etc. Cette propriété est assurée par le fait que les systèmes invités sont inchangés, comme nous l'avons précédemment indiqué.

Le système invité peut à son tour exécuter n'importe quel programme prévu pour ce système, pour autant qu'il ne nécessite pas de matériel non fourni par la machine virtuelle (c'est-à-dire pas de carte graphique spécifique ou autre périphérique peu répandu). Il est même possible d'installer une autre couche de virtualisation telle qu'un isolateur par exemple, pour autant que cet isolateur fasse appel à des instructions prises en charge par la machine hôte.

La traduction à la volée des instructions du système invité est néanmoins une opération complexe et fort coûteuse en terme de temps de calcul. En effet, la machine virtuelle ne peut pas, dans la plupart des cas, exécuter directement les instructions du système invité sur le système hôte. Les instructions de gestion de la mémoire vive, par exemple, doivent être interprétées par la machine virtuelle pour aboutir au résultat attendu, car seul le système hôte peut gérer les composants matériels de la machine physique.

La machine virtuelle doit donc implémenter en logiciel une gestion complète de la mémoire de l'invité, en utilisant les couches d'abstraction de l'hôte pour accéder à la mémoire vive. Cet empilement de couches réduit significativement les performances, surtout en cas de montée en charge (c'est-à-dire quand la mémoire est utilisée de manière intensive : lecture, écriture, déplacement de données, etc.).

Les performances de la machine virtuelle dépendent directement des performances de la couche d'abstraction du système hôte et de la qualité de l'émulation implémentée pour le matériel. La virtualisation complète consiste donc à avoir une séparation nette entre la machine virtuelle et le système hôte ce qui représente un avantage par rapport à la sécurité-

immunité. En effet, comme la machine virtuelle est considérée comme un simple programme, on peut très facilement limiter la quantité de mémoire qui lui est allouée, le temps CPU qu'elle peut utiliser, sa priorité par rapport aux autres programmes... Certains systèmes hôtes offrent des possibilités de configuration encore plus fines, par exemple, dans le cas d'une machine multi-cœur, il est possible de dédier à chaque machine virtuelle un ou plusieurs cœurs.

Cependant, du fait de l'empilement de couches d'abstraction et de l'impossibilité pour la machine virtuelle d'accéder directement au matériel, les performances du système invité sont assez éloignées de celles d'un système *natif*. Selon les implémentations, diverses solutions sont utilisées pour accélérer les machines virtuelles, par exemple en passant directement au processeur physique la plupart des instructions destinées au processeur virtuel. Cela accélère la vitesse de calcul du système invité. Il reste cependant le problème des Entrées/Sorties (E/S), c'est-à-dire les accès au disque, à la mémoire vive (accessible comme un périphérique par Unix et ses variantes), à la carte graphique, à la carte réseau, etc. Les E/S sont beaucoup plus difficiles à optimiser, car chaque système d'exploitation a ses propres routines lui permettant de traiter les exceptions générées par le matériel. Il est clair que si l'on veut optimiser les E/S des systèmes invités de manière à ce qu'il y ait le moins de déroutement possible pour le système hôte, deux solutions peuvent être envisagées. La première solution consiste à dérouter directement les appels du système invité au matériel, ce qui suppose que le matériel soit capable d'identifier l'appartenance d'un appel à un système donné, et décider de passer la main ou pas au moniteur de machines virtuelles. Ceci est le principe de la virtualisation assistée par le matériel. La seconde solution consiste à faire de sorte que le système invité fasse le moins d'appels possibles aux primitives qui nécessitent un déroutement par le système hôte. Pour cela, le système invité doit être *conscient* de la présence d'une sous couche de virtualisation. Ce principe est expliqué dans le paragraphe suivant, et est connu sous le nom de paravirtualisation.

IV.3.1.3 Virtualisation assistée par le matériel

Certains nouveaux processeurs (AMD-V ou Intel VT) offrent la possibilité de traiter toutes les instructions en mode non privilégié, et d'assurer en même temps le bon fonctionnement global du Moniteur de Machines Virtuelles (MMV) et des systèmes invités. Rappelons que l'architecture traditionnelle des processeurs dans laquelle seuls deux modes d'exécution sont utilisés conduit à une concurrence entre le moniteur de machine virtuelle et les machines virtuelles instanciées. Ces nouvelles familles de processeurs introduisent un autre mode d'exécution qui correspond au mode hyperviseur ou encore *ring*-1. Chez AMD, ce mode est désigné par SVM (*Secure Virtual Machine*).

Au démarrage de la machine physique, le processeur fonctionne en mode SVM pour donner la main au MMV. Ce moniteur a la possibilité d'exécuter des instructions qui lui sont propres sur le processeur, et lui permettant d'instancier des machines virtuelles et de les paramétrer. Ainsi, à partir de ce mode, le MMV peut spécifier la zone mémoire à attribuer à chaque machine virtuelle, ainsi que les canaux visibles utilisés pour les entrées-sorties pour cette machine virtuelle. Une fois ce paramétrage exécuté, le processeur passe en mode *ring*-0 et les machines virtuelles sont exécutées dans l'espace d'adressage attribué et ont la possibilité d'exécuter directement leurs instructions sur le processeur. Ce dernier, se basant sur les premiers paramétrages de la machine virtuelle, exécute les traductions nécessaires entre les ressources virtuelles et les ressources physiques. Cette traduction, habituellement exécutée par le moniteur de machines virtuelles, est réalisée par le matériel. Cette technologie offre au MMV la possibilité de définir des '*traps*' (sous forme de conditions sur les instructions utilisées par le système virtualisé) pour qu'il récupère la main et exécute éventuellement des

vérifications supplémentaires sur le code exécuté par le système invité [Lacombe et al. 2009]. Remarquons également que cette nouvelle architecture matérielle permet de modéliser et d'implémenter plus facilement les machines virtuelles. Ceci permet donc d'éviter le surcoût important introduit par la traduction d'instructions exécutée par le MMV. Il est à noter que les systèmes d'exploitation non modifiés (pour être adaptés aux architectures de virtualisation) continueront à faire appel au MMV pour le traitement des erreurs, exceptions et interruptions générées par les applications ou les systèmes invités. Dans ce cas, une opération similaire au changement de contexte habituel a lieu pour permettre le passage en mode *ring*-1 et le traitement de ces tâches par le MMV. IBM a utilisé ce type de virtualisation dans ses systèmes d'exploitation de type AIX (dans les années 80) pour gérer, à travers un moniteur de ressources virtuelles la notion de multi-tâche et multi-utilisateur [IBM 1986].

IV.3.1.4 Paravirtualisation

La paravirtualisation est très proche du concept de la virtualisation complète, dans le sens où c'est toujours un système d'exploitation complet qui s'exécute sur le matériel émulé par une machine virtuelle, cette dernière s'exécutant au dessus d'un système hôte. Toutefois, dans une solution de paravirtualisation, le système invité est modifié pour être exécuté par la machine virtuelle. Les modifications effectuées visent à rendre le système émulé *conscient* du fait qu'il s'exécute sur une machine virtuelle. Ainsi, il pourra collaborer plus étroitement avec le système hôte, en utilisant une interface spécifique, au lieu d'accéder au matériel virtuel via les couches d'abstraction. Au final, l'architecture obtenue est plus performante que l'empilement des couches d'abstraction d'une virtualisation complète.

Le terme *paravirtualization* a été mentionné pour la première fois dans [Whitaker et al. 2002], où les auteurs définissent la paravirtualisation comme la modification sélective de certaines parties de l'architecture virtuelle pour améliorer les performances, la réactivité sous forte charge et la simplicité de conception. L'idée de la paravirtualisation est toutefois plus ancienne que cela. Les premiers gros systèmes utilisant une architecture de virtualisation avaient déjà une technologie similaire, dès les années soixante-dix [IBM 1972].

En pratique, un système paravirtualisé possède quelques pilotes de périphériques et sous-systèmes modifiés, qui lui permettent de communiquer directement avec le système hôte, sans avoir à passer par une couche d'abstraction. Les pilotes paravirtualisés échangent directement des données avec le système hôte, sans avoir à passer par une émulation du comportement du matériel. Certaines parties du système hôte doivent généralement être modifiées pour tirer profit de la paravirtualisation, en particulier la gestion de la mémoire et la gestion des E/S.

La figure IV.13 décrit la structure d'une machine virtuelle et d'un système hôte supportant la paravirtualisation. Les pilotes non modifiés interagissent toujours avec le matériel émulé par la machine virtuelle, alors que les pilotes modifiés communiquent directement avec le système hôte. La simplification qui en résulte permet au système invité de collaborer plus efficacement avec l'hôte : les parties critiques du système communiquent presque directement avec le système hôte, en contournant les couches d'abstraction virtuelles (c'est-à-dire le matériel émulé). Le reste de l'architecture est inchangé : la machine virtuelle est toujours une application utilisateur et le système d'exploitation hôte est toujours le seul à avoir un accès privilégié au matériel (cf. Figure IV.13).

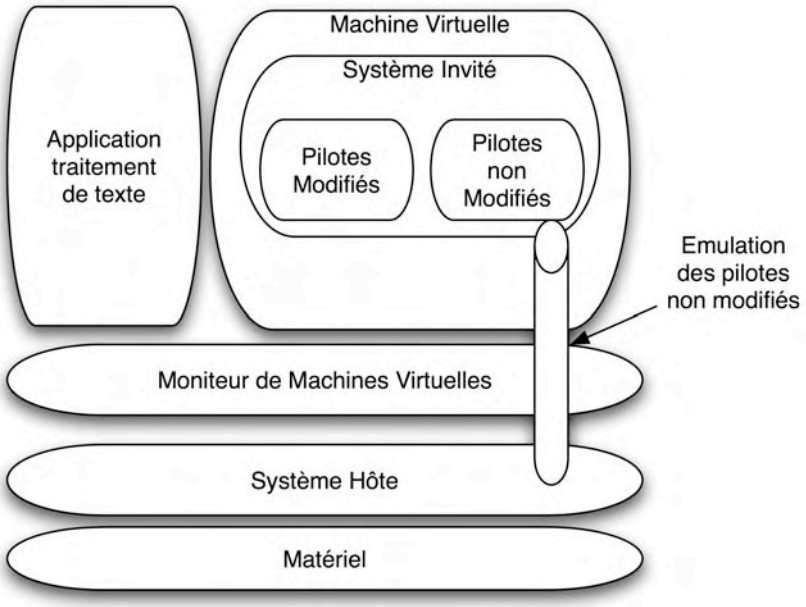


Figure IV.13 Principe de la paravirtualisation

Les détails des optimisations varient selon les implémentations, mais il s'agit en général d'utiliser des appels systèmes ou des instructions spécifiques pour communiquer avec le système hôte. Le type d'actions paravirtualisées varie également selon les implémentations, mais on retrouve en général tout ce qui est déplacement de données entre l'hôte et l'invité (accès disque, transfert réseau, etc.) et gestion de la mémoire.

La paravirtualisation apporte un gain de performance important, du fait du contournement des couches d'abstraction. En effet, comme le système invité collabore activement avec le système hôte, il ne se comporte plus comme un système d'exploitation à part entière s'exécutant directement sur du matériel. Au contraire, il adapte son comportement pour que les accès au matériel, souvent difficiles à interpréter de manière efficace par la machine virtuelle, soient transformés en des appels directs au système hôte. De plus, étant donné que seules les couches de bas niveau du système invité ont été modifiées, toutes les applications qui pouvaient fonctionner dans une architecture de virtualisation complète peuvent aussi être utilisées dans une architecture paravirtualisée.

Toutefois, cette augmentation des performances est restreinte à certains systèmes. En effet, comme le système invité doit être modifié pour être paravirtualisé, il faut bien évidemment que l'on ait la possibilité de réaliser cette opération de portage. Or, cela nécessite à la fois l'accès au code source du système d'exploitation et la permission du détenteur des droits de le modifier. Si cela ne pose aucun problème pour un système libre (notamment GNU/Linux et les systèmes BSD), il n'en va pas de même pour les systèmes propriétaires, tels que Microsoft Windows et Mac OS. L'usage de la paravirtualisation est donc généralement limité aux systèmes libres, à moins d'utiliser une solution de virtualisation propriétaire compatible avec un seul système d'exploitation invité, comme c'était le cas pour les premières versions du logiciel VirtualPC commercialisé par Microsoft.

IV.3.1.5 Combinaisons des implémentations de machines virtuelles

Les sections précédentes ont montré comment la virtualisation complète, la paravirtualisation ou la virtualisation assistée par matériel permettent d'implémenter une couche de

virtualisation au dessus d'un système hôte qui est en charge d'héberger les différentes machines virtuelles à travers la couche de virtualisation qu'il implémente. La couche de virtualisation est désignée dans ce cas comme Moniteur de Machines Virtuelles de type 2. En effet, ces MMV ne sont pas en contact direct avec la matériel, et de ce fait, l'architecture logicielle globalement déployée est complexe dans le sens où un système hôte doit toujours être implémenté.

Cependant, il est possible d'avoir des solutions de virtualisation complète, de paravirtualisation ou encore de virtualisation assistée par le matériel en utilisant un MMV implémenté directement sur le matériel. Dans ce cas, on parle de moniteur de type 1, ou encore *hyperviseur* (cf. figure 14).

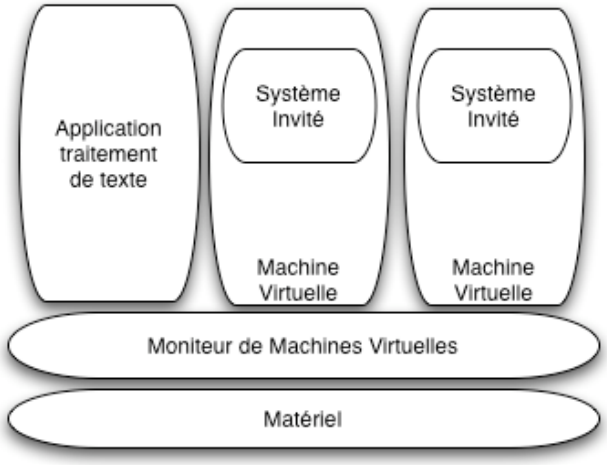


Figure IV.14 Moniteur de Machines Virtuelles de Type 1

Un exemple de tels hyperviseurs est fourni par Xen. En effet, cet hyperviseur libre offre la possibilité de déployer des machines virtuelles non modifiées telles que Windows XP ou bien certaines distributions de Linux paravirtualisées.

Les dernières implémentations des hyperviseurs prennent toutes en charge la virtualisation assistée par le matériel, parce que ce type de technique assure une bonne isolation entre les différentes machines virtuelles, et cette isolation est directement assurée par le matériel.

Nous remarquons ainsi, qu'en fonction de la position de la couche de virtualisation ainsi que de l'interaction entre les différentes couches logicielles, on peut définir plusieurs types de virtualisation. Chaque implémentation a ses propriétés qui peuvent être exploitées pour déployer des architectures logicielles qui répondent à des exigences variées. Dans la section suivante, nous mentionnons quelques utilisations de la virtualisation en mettant l'accent sur les propriétés de sécurité.

IV.3.2 Exemples d'utilisation des techniques de virtualisation système

La virtualisation système est vue comme un moyen adéquat permettant d'augmenter la fiabilité d'une machine à travers l'implémentation de services tels que le réseau ou la sécurité à un niveau inférieur à celui d'un système d'exploitation classique [Garfinkel et Rosenblum 2005]. En effet, la taille d'un hyperviseur (ou Moniteur de Machines Virtuelles) est bien inférieure à celle d'un système classique en terme de lignes de codes (LDC). Un hyperviseur est d'une taille de l'ordre de 10^4 LDC alors qu'un système d'exploitation a une taille de l'ordre de 10^6 LDC [Garfinkel et Rosenblum 2003]. Il est donc plus facile de s'assurer que les

services implémentés par le MMV sont corrects ou pas. Cette vérification a permis pour certaines implémentations d'hyperviseur d'être certifiées [Kaiser et Wagner] à un niveau assez élevés d'EAL dans les critères communs (cf. chapitre III). Cette propriété de validation est importante et peut être exploitée dans un contexte d'applications plus au moins critiques, ce qui est le cas des applications que nous traitons dans ce manuscrit.

La virtualisation système, de par son niveau d'abstraction logiciel, permet d'avoir un contrôle plus au moins fin des systèmes virtualisés, en fonction de l'implémentation de l'hyperviseur. Ce contrôle, essentiellement basé sur l'isolation et l'interception des instructions système, est utilisé dans différents domaines, de la simple isolation à la détection et recouvrement d'erreurs du système. La détection d'erreurs peut être vue comme une surveillance du système virtualisé (*monitoring*), en se basant sur des techniques de détections d'anomalies. Le recouvrement consiste à modifier le comportement d'un système pour remédier à des erreurs (d'une façon proactive ou réactive). Dans ce qui suit, nous détaillons chacune des ces utilisations et présentons quelques exemples d'implémentation que l'on trouve dans la littérature.

IV.3.2.1 Isolation

L'isolation entre différentes machines virtuelles est une propriété fondamentale que doit vérifier un hyperviseur pour empêcher toute forme de communication non contrôlée entre les machines virtuelles. On parle ainsi d'une isolation spatiale (ou isolation des différentes ressources) et temporelle (garantie d'accès aux ressources partagées). Rappelons que quelle que soit la nature de cette isolation, il s'agit dans les deux cas d'une problématique de gestion de ressources communes.

Ce sont ces propriétés d'isolation qui permettent d'implémenter un partitionnement dont l'efficacité varie selon les méthodes déployées lors de l'implémentation de l'hyperviseur. Ainsi, un hyperviseur sûr doit assurer une isolation totale entre les différentes machines.

Si l'isolation est parfaite, le fonctionnement erroné d'une machine virtuelle (par exemple, en raison de fautes logicielles résiduelles) n'affectera pas celui des autres machines virtuelles s'exécutant en parallèle. Prenons l'exemple d'une application dans laquelle un bogue produit une boucle infinie. La machine virtuelle sur laquelle l'application s'exécute est ralentie voire même arrêtée. Cependant, le processeur n'est pas bloqué par cette boucle infinie, vu que l'hyperviseur assure un ordonnancement qui empêche qu'une machine virtuelle monopolise la CPU.

Dans le cas de fautes malveillantes, en particulier visant un déni de service, la machine virtuelle contaminée doit être confinée dans son espace virtuel et ne peut en aucun cas altérer le fonctionnement des autres machines virtuelles (d'une manière similaire au cas de la boucle infinie dans l'exemple précédent). Le processus attaquant se voit donc ralentir sa propre machine, sans aucun effet sur les machines voisines. Cependant, si l'attaque porte sur le MMV, et que cette attaque aboutit, la propriété d'isolation n'est plus vérifiée et l'architecture globale déployée n'est plus sûre. En effet, une telle attaque change la propriété d'intégrité que nous avons présentée dans la section IV.1 (zone Ø d'indépendance).

Ainsi, quelque soit l'hypothèse de fautes (conception ou malveillante), la virtualisation offre un cadre d'isolation parfait pour développer ou tester un système d'exploitation et/ou une application spécifique, en se basant sur la technique de bac à sable (*sandboxing*). *User Mode Linux* (cité dans la section IV.3.1.1) est un MMV conçu pour permettre aux développeurs de noyaux de systèmes de contrôler et tester le fonctionnement des nouveaux noyaux déployés.

Dans [Joshi et al. 2005], il est proposé d'analyser le comportement d'un système, après l'application d'une rustine (*patch*). La machine virtuelle sert donc de bac à sable et le MMV peut suivre le fonctionnement de la machine virtuelle en question selon des recommandations données par les développeurs de la rustine.

Dans certains cas, le temps nécessaire pour développer une rustine de sécurité (temps calculé entre le moment de la découverte de la vulnérabilité et le moment de déploiement de la rustine) se mesure en mois [Vache 2009]. En 2005, il fallait en moyenne plus de 130 jours à Microsoft pour publier une rustine critique liée à une vulnérabilité qui lui avait été notifiée [Washington Post 2006]. L'alternative d'arrêter le système en attendant la publication d'une rustine appropriée est économiquement inacceptable pour les entreprises. En même temps, maintenir les machines en opération peut s'avérer dangereux pour le fonctionnement global du système et du réseau, surtout qu'il faut beaucoup moins de temps pour développer une attaque exploitant une nouvelle vulnérabilité (quand elle est connue) que pour développer une rustine.

IV.3.2.2 Observation d'activités de systèmes virtualisés

L'hyperviseur ou le MMV, de par le niveau d'abstraction qu'il représente, a la possibilité d'isoler toutes les machines virtuelles qu'il instancie. Les techniques et mécanismes mis en œuvre pour assurer cette isolation peuvent être aussi utilisés pour implémenter des mécanismes d'observation. En effet, le MMV s'assure que les opérations effectuées par une machine virtuelle respectent le partitionnement défini (par le MMV) lors du lancement de cette machine virtuelle. Cette technique peut être facilement étendue de manière à ce que toutes ces opérations soient observées et qu'un journal (sous forme de *Log* par exemple) soit maintenu tout au long de l'exécution de la machine virtuelle. Nous ne parlons donc plus de simple mécanismes d'isolation entre machines virtuelles, mais plutôt d'isolation et de surveillance (*monitoring*) des machines instanciées. Dans ce qui suit, nous présentons différentes utilisations de ces techniques d'observation, dans le cadre de prévention et/ou tolérance aux fautes.

Traçage d'activité d'un système

L'observation d'activités d'un système est généralement assurée par des sondes implémentées par des programmes s'exécutant sur le système en cours d'observation. Il est clair que ce type d'observation dépend principalement de l'intégrité du système en question. Imaginons que le système soit corrompu, les sondes peuvent être détournées et leur résultat d'observation ne conduit pas forcément au dépistage du fonctionnement défaillant du système observé. Dans ReVirt [Dunlap et al. 2002], l'observation d'un système est faite au niveau d'un MMV. Le système est installé dans une machine isolée, et les événements sont enregistrés par une sonde au niveau de l'hyperviseur. Dans le cas de ReVirt, les événements en question sont assez nombreux pour permettre de relancer la machine virtuelle à partir d'un point sûr. La relance de la machine dans ce cas ne fait pas partie de l'observation proprement dite, c'est plutôt un mécanisme de recouvrement d'erreurs, en présence d'attaques. *Backtracer* [King et Chen, 2003] se base sur un principe similaire, dans lequel les objets (tels que les processus, les fichiers…) et les événements systèmes (tels que les opérations de lecture, écriture, duplication de processus par *fork*) sont sauvegardés par un moniteur installé dans l'hyperviseur. De telles observations peuvent servir de base pour analyser le comportement défaillant d'un système suite à une attaque, et de remonter à la source de cette attaque.

Détection d'intrusion

L'observation des activités du système peut être utilisée pour implémenter des Systèmes de Détection d'Intrusion (ou encore *IDS : Intrusion Detection System*). Les SDI ont pour but d'observer le comportement d'un système pour déterminer s'il est victime d'une attaque. Les SDI peuvent détecter aussi bien les attaques réussies (et qui ont conduit à une intrusion) qu'une tentative d'attaque. Ils se basent soit sur la détection d'anomalie, soit sur la détection de symptômes d'attaques (généralement par identification de *signatures* caractéristiques d'attaques).

Il existe deux principales familles de SDI: les SDI réseaux (*Network IDS*) et les SDI hébergés (*Host-based IDS*). Dans le cas d'un SDI réseau, le détecteur analyse le trafic réseau en provenance (et de destination) du système observé. De ce fait, le SDI se trouve isolé du système observé, ce qui réduit son efficacité dans le cas où une attaque a abouti. En plus, dans le cas d'utilisation de paquets réseaux chiffrés, la tâche de détection de trafic malicieux devient plus compliquée. Dans une grande famille d'attaques, la machine infectée ne génère pas de trafic anormal, mais se contente de modifier le comportement interne de la machine, auquel cas une détection basée sur le réseau est peu efficace. Par contre, avoir une visibilité plus fine du comportement du système, à savoir l'analyse des différents appels système, peut s'avérer plus utile quant à la détection des éventuelles intrusions. Ainsi, les sondes installées par un SDI hébergé permettent d'avoir des informations plus précises sur le fonctionnement interne du système observé. Notons qu'une fois le système infecté, l'intégrité des sondes et du SDI peut être mise en question, et les résultats de l'analyse de ces détecteurs ne sont plus fiables.

L'implémentation d'un SDI au niveau d'un hyperviseur permet d'avoir les avantages aussi bien des SDI réseaux que des SDI hébergés, tout en évitant leurs inconvénients respectifs. Dans X-SPY [Jansen et al. 2008], les sondes du SDI sont implémentées dans une machine virtuelles sûre appelée SSVM (*Secure Service Virtual Machine*) afin de contrôler des machines appelées PVM (*Production Virtual Machine*). L'implémentation d'une sonde de SDI sur la même machine physique que le système observé mais sur une machine virtuelle différente permet au détecteur d'avoir une observation fine du comportement de la machine virtuelle observée tout en étant isolé de cette machine (grâce à la propriété d'isolation du MMV).

Quelle que soit la nature du SDI (réseau ou hébergé), ces systèmes de détection d'intrusion se basent sur :

- La **détection d'anomalies** : elle consiste à comparer le comportement d'un système avec une référence de comportement normal. La référence de comportement normal est généralement obtenue par observation du système pendant une phase d'apprentissage, pendant laquelle on suppose qu'il n'y a pas d'attaque. Pendant la phase opérationnelle, on compare le comportement observé à celui enregistré lors de la phase d'apprentissage. Dans [Laureano et al. 2004], cette approche est utilisée par un moniteur de machines virtuelles pour enregistrer le comportement normal d'un système d'exploitation durant une première phase, puis pour détecter les appels systèmes susceptibles de faire partie d'une attaque.
- La **détection par signature** : les symptômes relatifs à une attaque qui exploite la vulnérabilité en question peuvent servir de référence à l'hyperviseur pour arrêter tout trafic ou flot d'exécution conduisant à la réalisation de l'attaque. Dans ce cas, on combine aussi bien des techniques d'isolation simple que des techniques d'observation d'activité de systèmes, présentées dans la section suivante.

Dans les mécanismes de tolérance aux fautes, nous distinguons des mécanismes de détection et de recouvrement de fautes. Nous pouvons faire une analogie entre ces mécanismes de tolérance aux fautes et les utilisations de la virtualisation. En effet, à travers l'isolation, les techniques d'observation assurent la détection d'erreurs, et nous avons vu que les systèmes de détections d'intrusions peuvent être facilement implémentés grâce à la virtualisation. Dans ce qui suit, nous détaillons le rôle de la virtualisation dans le contrôle des systèmes virtualisés, ce qui peut être comparé au recouvrement d'erreurs dans les systèmes tolérants aux fautes.

IV.3.2.3 Contrôle de systèmes virtualisés

Par contrôle d'un système virtualisé, nous entendons le changement de fonctionnement de ce dernier pour qu'il réponde à une spécification donnée. Prenons l'exemple de détection d'intrusion décrite dans la section précédente. Cette technique permet de détecter un comportement potentiellement dangereux. La simple détection, assurée par les mécanismes d'observation fournis par l'hyperviseur, ne peut être efficace sans la mise en œuvre de mécanismes de recouvrement permettant de compléter cette détection. Nous pouvons, par exemple, imaginer une politique interdisant toute communication avec la machine virtuelle désignée comme infectée, le recouvrement consistant alors à bloquer l'exécution de cette machine virtuelle. Nous présentons ici quelques exemples illustrant le rôle que peut avoir un MMV pour contrôler le fonctionnement d'un système, notamment pour ajouter des mécanismes de tolérance aux fautes tels que la sauvegarde (et la reprise) et le rajeunissement ou des mécanismes d'équilibrage de charge pour éviter l'encombrement des serveurs par exemple.

Rajeunissement

Au niveau d'un MMV, une machine virtuelle est vue comme un processus dans un système d'exploitation classique. Les opérations sur les processus (création, duplication, arrêt, redémarrage,…) sont également possibles sur des machines virtuelles. Le rajeunissement (*rejuvenation*) logiciel consiste à relancer l'application afin de repartir d'un état initial stable. Pour un système d'exploitation, le redémarrage nécessaire au rajeunissement risquerait de provoquer une indisponibilité pendant une période qui peut être trop longue pour les applications concernées (typiquement plusieurs minutes). La virtualisation, dans ce cas, constitue un bon moyen pour assurer la relance de la machine sans qu'il y ait rupture de service. En effet, le MMV peut lancer périodiquement (période à définir en fonction des besoins environnementaux) une instance d'une machine virtuelle, à partir d'une image stable. Durant toute la phase de démarrage de la nouvelle machine virtuelle, la machine virtuelle opérationnelle continue à livrer ses services, et n'est arrêtée qu'à la fin du démarrage de la seconde machine.[H.P. Reiser et R. Kapitza 2007]. La même technique peut être utilisée lors de l'application de rustine, la machine mise à jour remplace l'ancienne machine une fois que la mise à jour est accomplie, afin d'éviter la rupture du service.

Le rajeunissement est une technique qui ne modifie pas le comportement d'un système, mais se contente de le relancer à partir d'un état stable. Dans ce qui suit, nous nous intéressons à la virtualisation comme contrôleur plus fin du comportement de la machine virtuelle.

Sauvegarde et reprise des activités d'un système

ReVirt est un exemple d'implémentation d'hyperviseur permettant d'implémenter des mécanismes d'observation comme nous l'avons indiqué dans la section précédents. Les observations faites sur le système virtualisé sont sauvegardées, de manière à ce que le

fonctionnement d'un système puisse être reproduit à partir de ces journaux d'observation. De plus, un mécanisme de reprise peut être facilement implémenté [Joshi et al. 2005] en faisant appel aux journaux pour restaurer n'importe quelle étape d'exécution.

La technique de reprise dans les machines virtuelles peut être utilisée par les développeurs pour déboguer le système lors de la phase de développement. Ce concept est décrit par King [King, Dunlap, et al. 2005] comme une « machine virtuelle à voyager dans le temps » (*TTVM : Time-Traveling Virtual Machine*). L'hyperviseur est utilisé dans ce cas pour rejouer l'exécution d'une certaine machine virtuelle, grâce aux journaux d'observation.

En développant davantage la notion de voyage temporel des machines virtuelles, Backtracer [King, Mao, et al. 2005] permet de tenter de remonter les différentes instructions qui ont conduit à une attaque détectée, et ce jusqu'à la machine source. Ceci est notamment possible grâce aux observations et aux liens de causalités qui peuvent être déduits des différents SDI.

La sauvegarde et reprise sont des techniques classiques de tolérance aux fautes, qui peuvent être implémentées en utilisant la virtualisation. Cependant, il est possible d'utiliser également la virtualisation pour l'évitement de fautes, comme nous le montrons dans le paragraphe suivant.

Equilibrage de charge

L'informatique dans les nuages (plus connue sous le nom de *cloud computing*) consiste à l'exploitation, à distance, de ressources matérielles fournies par des clusters de machines. L'utilisation de calcul sur grille ou encore de la sauvegarde à distance en sont des exemples. Les serveurs assurant de tels services ont des montées en charges importantes pouvant conduire à des dénis de service. La virtualisation a été proposée comme technique pour permettre de dupliquer, à la volée, les machines virtuelles implémentant le service afin de répondre au mieux aux besoins des clients [Lagar-Cavilla et al. 2009]. La duplication est ainsi optimisée afin de réduire le surcoût (*overhead*). Cette technique est particulièrement efficace si les services requis ne font pas appel aux Entrées/Sorties, ce qui est souvent le cas des systèmes informatiques dans les nuages.

IV.4 CONCLUSION

Nous avons vu dans ce chapitre que la virtualisation présente un niveau d'abstraction logicielle qui se situe entre les applications ou le système d'exploitation et le matériel. Cependant, cette abstraction peut être interprétée de différentes manières, ce qui donne un large spectre d'implémentations de Moniteurs de Machines Virtuelles. Nous avons donc proposé une classification des différents MMV en nous basant sur ce qui existe déjà dans la littérature. Ainsi, nous parlons de machines virtuelles de processus (si on s'intéresse à virtualiser le fonctionnement d'une application) et de machines virtuelles de systèmes (si le MMV virtualise tout un système d'exploitation), le second type étant celui le plus souvent désigné par le terme virtualisation.

Il est à noter que chaque implémentation a ses propres caractéristiques qui peuvent être utilisées pour répondre à des exigences fonctionnelles précises. Dans le cadre de ce manuscrit, nous nous sommes plus intéressés aux techniques de tolérance aux fautes utilisant la virtualisation. Ainsi, nous avons détaillé quelques cas d'utilisation, que nous avons classés en trois grandes catégories : la simple isolation, l'observation et le contrôle des systèmes virtualisés. Ces trois classes peuvent être comparées au notion de confinement, détection et recouvrement dans le vocabulaire de la tolérance aux fautes.

Dans le cadre de notre étude, la virtualisation désigne la virtualisation système, et elle est utilisée pour assurer l'isolation nécessaire entre machines virtuelles, afin d'implémenter un mécanisme de contrôle de flux entre entités de criticités multiples. Le MMV constitue donc un élément primordial dans une telle architecture, puisqu'il doit être suffisamment sûr pour héberger des machines virtuelles critiques et sûres. Nous avons montré dans ce chapitre que les MMV peuvent être utilisés pour implémenter des mécanisme de sécurité, comme les Systèmes de Détection d'Intrusions (*SDI*). Nous proposons dans le chapitre suivant de détailler notre approche dans l'utilisation de la virtualisation pour implémenter une architecture tolérante aux intrusions.

Chapitre V

ARCHITECTURES DE SECURITES POUR LES APPLICATIONS CRITIQUES EMBARQUEES

INTRODUCTION

Nous avons présenté dans le chapitre III quelques considérations sur les aspects immunité et innocuité et le rapprochement qu'on peut trouver entre, d'une part la criticité d'une tâche, et d'autre part la confiance dans les modules réalisant cette tâche.

Cette confiance dépend de plusieurs critères, et il est possible, au moyen de techniques de tolérance aux fautes, d'augmenter le niveau de confiance d'un module donné. Dans le cadre de nos travaux, nous nous situons dans un contexte avionique, dans lequel des tâches de multiples niveaux de criticité coexistent dans un même environnement, et doivent interagir. Il est donc primordial de respecter les niveaux de criticité de ces tâches lors du déploiement des modules les implémentant dans un environnement réel d'exécution.

L'environnement d'exécution et la nature des tâches étudiées dans ce manuscrit ont les caractéristiques fonctionnelles et environnementales suivantes :

- Les applications ont des niveaux de criticité hétérogènes. Certaines applications sont considérées comme critiques et appartiennent au monde avionique, d'autres le sont moins, et peuvent être implantées par des composants sur étagère (*Commercial Off-The-Shelf : COTS*).
- Les applications sont distribuées et communicantes : certaines applications sont hébergées dans des modules avioniques, d'autres sont fournies par des plateformes du monde ouvert et implémentées sur des machines extérieures à l'avion. Toute forme de communication doit donc respecter les contraintes liées à l'hétérogénéité des criticités (et doit donc suivre l'un des modèles de contrôle de flux présentés dans le chapitre III).
- Les applications présentent des interfaces homme-machine permettant à des opérateurs humains de spécifier certains paramètres nécessaires au déroulement de la tâche implantée par l'application en question.

Chacune de ces caractéristiques nous impose les contraintes spécifiques suivantes :

- L'hétérogénéité des niveaux de criticité nous contraint à assurer une isolation complète entre ces niveaux, afin d'éviter qu'une tâche peu critique ne perturbe le fonctionnement d'une tâche critique.
- La communication entre tâches de criticités hétérogènes présente une violation de la propriété d'isolation précédemment mentionnée. Cependant, une telle communication est nécessaire dans certains cas de figure, comme ceux que nous avons identifiés dans les cas d'étude qui seront présentés au chapitre VI. Nous nous basons sur le modèle Totel (chapitre III) afin de permettre de telles communications, tout en préservant les propriétés d'intégrité. L'aspect distribué nous conduit à une adaptation du modèle Totel afin de répondre à cette contrainte.
- L'interface homme-machine empêche de diversifier directement l'application finale. En effet, diversifier une IHM implique que l'opérateur doit reproduire une action autant de fois que le nombre de variantes présentes dans le système. Une telle interaction n'est pas souhaitable puisqu'elle introduit, de par sa mauvaise ergonomie, une nouvelle source d'erreurs liées à l'opérateur [A Avizienis et al. 2004].

Nous proposons dans ce chapitre d'établir dans un premier temps les hypothèses de fautes et de fonctionnement. Ces hypothèses montrent qu'il est important d'identifier clairement les

objets et les flux d'information. Nous proposons alors ensuite de formaliser le modèle Totel en nous basant sur le modèle de dépendances causales, et ainsi caractériser les flux de données et les contraintes de contrôle de flux. Nous discuterons enfin des contraintes fonctionnelles et applicatives qui ont guidé notre phase de conception de l'architecture finale.

V.1 HYPOTHESES DE FAUTES ET DE FONCTIONNEMENT

Nous avons vu dans le chapitre IV que l'utilisation d'une redondance logicielle au sein d'une seule machine physique (servant comme support d'exécution) est rendue possible grâce à la technique de virtualisation dont nous avons présenté les caractéristiques principales vis-à-vis des propriétés de tolérance aux fautes. Cependant, il est nécessaire de souligner que toute phase de conception d'un système tolérant les fautes doit être accompagnée d'une analyse concernant les hypothèses de fautes que l'on cherche à tolérer. Une fois ces hypothèses établies, une étude détaillée de chaque technique de tolérance aux fautes adoptée est nécessaire afin de s'assurer qu'elle est réalisable et n'introduit pas plus de fautes qu'elle n'en tolère. En effet, prenons l'exemple d'un simple chien de garde (*watchdog*) en charge de lever une exception si une application ne donne pas de réponse au bout d'un délai *d* préalablement fixé. Il est clair que *d* dépend aussi bien de l'application que de son environnement. Ainsi, si la valeur de *d* n'est pas choisie en prenant en compte ces paramètres internes et externes, le chien de garde risque soit de générer beaucoup de fausses alarmes (ce qui est potentiellement bloquant pour le reste du système), soit de ne pas générer d'alarmes assez rapidement, alors que l'application est défaillante (ce qui risque d'avoir des conséquences encore plus catastrophiques sur le système).

À travers cet exemple, nous voyons qu'il est important d'adapter les mécanismes choisis aux hypothèses de fautes et aux besoins fonctionnels. Ainsi, nous proposons de présenter dans un premier temps les hypothèses de fautes que nous traitons dans le cadre de ces travaux. Nous détaillerons ensuite une problématique liée à la redondance logicielle, à savoir le déterminisme.

V.1.1 Hypothèses de fautes de l'architecture déployée

Avant d'établir les hypothèses de fautes, nous présentons d'abord notre utilisation de la virtualisation pour assurer la tolérance aux fautes (cf. figure V.1). La virtualisation permet aussi bien l'isolation (cf. IV.3.2.1) que la détection (IV.3.2.2) et le recouvrement (IV.3.2.3) des erreurs.

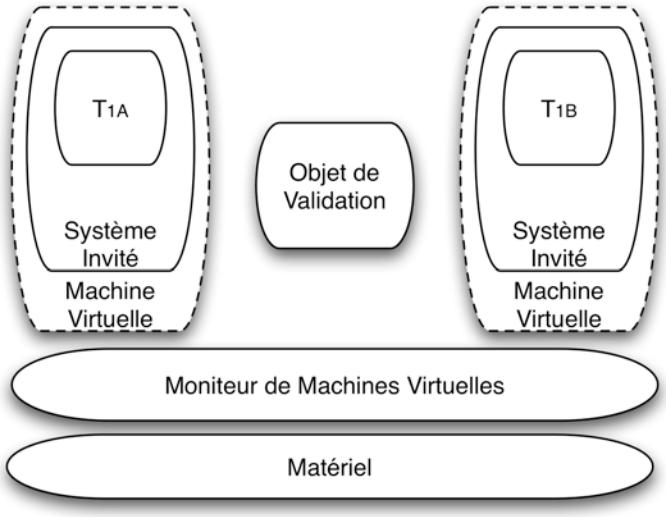


Figure V.1 Mise en oeuvre de la redondance avec la technique de virtualisation

La figure V.1 représente deux tâches, T_{1A} et T_{1B}, qui sont deux versions qui remplissent les mêmes fonctions, mais qui sont implantées sur deux systèmes d'exploitation différents (par exemple une première tâche développée pour s'exécuter sur Windows et une deuxième sur Linux). Nous avons précédemment indiqué que le moniteur de machines virtuelles pouvait assurer aussi bien l'isolation que la détection et le recouvrement. Cependant, dans l'architecture représentée par la figure V.1, ce moniteur n'assure que l'isolation. En effet, les tâches de détection et de recouvrement sont déléguées à l'objet de validation (OV). Ainsi, le contrôle de la validité des deux versions est réalisé au niveau de l'OV qui a pour rôle d'implanter le mécanisme de décision d'adjudication approprié afin de valider les résultats. Remarquons que nous ne spécifions pas, pour l'instant, l'emplacement exact de l'OV. Diverses possibilités d'implantation peuvent être envisagées (cf. V.3). Cependant, d'un point de vue conceptuel, la validation doit toujours être effectuée au niveau d'une entité suffisamment sûre de fonctionnement (cf. Zone Ø de la section IV.1). Il est donc important d'identifier clairement les zones sûres, en fonction des fautes que nous cherchons à tolérer.

Afin d'établir exhaustivement les hypothèses de fautes, nous nous basons sur la classification des types de fautes que nous avons présentée dans II.2. Cette classification distingue principalement deux types de fautes : les fautes accidentelles et les fautes délibérées. Le premier type de fautes est soit d'origine physique externe (rayonnements cosmiques pour les systèmes des satellites par exemple), soit d'origine humaine (faute de conception, d'utilisation). Le deuxième type est soit bienveillant (suite à une décision économique, on accepte de ne pas implanter des mécanismes de protection trop coûteux), soit malveillant (ce qui est le cas des attaques sur les systèmes informatiques).

Dans ce qui suit, nous nous intéressons aux différents types de fautes qui peuvent survenir au niveau des différentes couches présentées dans la figure V.1, à savoir au niveau du matériel, du moniteur de machines virtuelles, du système virtualisé et de l'application diversifiée.

V.1.1.1 Fautes au niveau du matériel

Dans le cadre de nos travaux, nous faisons l'hypothèse que l'exécution se fait au niveau d'un matériel sûr de fonctionnement. Nous faisons ainsi confiance à la couche matérielle et nous ne traitons pas le cas de fautes accidentelles ou délibérées qui peuvent altérer le fonctionnement du matériel. Cependant, il est possible d'étendre les travaux actuels pour qu'ils prennent en compte des fautes qui peuvent concerner la couche matérielle. Par exemple, l'utilisation d'un TPM (*Trusted Platform Module*) [Balacheff et al. 2003] peut s'avérer utile contre certaines modifications malveillantes (par exemple une attaque du BIOS).

Rappelons que la plupart des fautes matérielles sont fugitives ou au mieux transitoires, ce qui les rend difficilement identifiables par logiciel. Ainsi, même si la tolérance de ces fautes matérielles n'est pas l'objectif principal de nos travaux, il convient de noter que les techniques que nous mettons en place sont susceptibles d'être efficaces pour tolérer les fautes matérielles. Cet aspect est similaire à celui présenté dans [Gray 1990] dans lequel un système conçu pour tolérer les fautes matérielles est prouvé qu'il tolère également certaines fautes logicielles.

V.1.1.2 Fautes au niveau du moniteur de machines virtuelles

Le moniteur de machines virtuelles est une entité logicielle clé dont dépend le bon fonctionnement de toute l'architecture que nous cherchons à déployer.

En effet, remarquons que c'est ce moniteur qui sert d'interface entre le matériel et les machines virtuelles. Il est donc nécessaire de s'assurer que le moniteur se comporte d'une façon sûre qui ne mette pas en défaut le fonctionnement des machines virtuelles (et par

conséquent les applications qui sont exécutées sur ces machines). Au niveau du moniteur, les fautes accidentelles envisagées sont les fautes humaines (dont les fautes de conception). Les fautes délibérées envisagées sont essentiellement des fautes malveillantes, les fautes bienveillantes devant être signalées par leur créateur, sinon elles seraient considérées malveillantes. Cela dit, quelle que soit la nature de la faute, nous faisons l'hypothèse, dans le cadre de nos travaux, que le moniteur est digne de confiance (développé à un niveau supérieur ou égal à celui des applications recevant des flux d'information de l'architecture), qu'il est robuste vis-à-vis des modifications malveillantes et que sa réalisation a obéi à des règles strictes de développement, de test et de validation afin d'atteindre le niveau de confiance recherché.

La nécessité de cette hypothèse s'explique également par le fait que le moniteur de machines virtuelles implante les mécanismes d'isolation et qu'il doit, de ce fait, éviter toute interférence entre les différentes machines virtuelles. En effet, dans le cas où l'isolation ne serait plus assurée, il est facile d'imaginer une attaque venant corrompre chaque machine virtuelle, mettant ainsi en défaut l'hypothèse d'absence de faute de mode commun affectant les entités redondantes en détaut, ce qui rendrait notre architecture moins sûre.

V.1.1.3 Fautes au niveau des systèmes invités

Par systèmes invités, nous désignons la couche logicielle située entre l'application diversifiée et le moniteur de machines virtuelles. Dans le cadre de nos travaux, les applications sont développées sous Java, ce qui implique que le système virtualisé est composé d'un système d'exploitation classique (Windows, Linux…) et d'une machine virtuelle Java installée sur chaque système d'exploitation.

À ce niveau, nous prenons en compte aussi bien les fautes accidentelles que délibérées. En effet, ces systèmes virtualisés sont des composants sur étagère (*COTS*), et peuvent de ce fait contenir des bogues. De plus, les fautes malveillantes sont très fréquentes pour ce genre de systèmes (sous formes de vers, virus, rootkits…) ce qui rend l'exécution des applications peu sûre de fonctionnement.

Nous faisons l'hypothèse que de tels systèmes sont sources de potentielles fautes pour les applications qui s'y exécutent, et il nous faut ainsi prendre en compte ces fautes dans l'architecture globale que nous proposons. En conséquence, pour pallier ce type de fautes, nous utilisons plusieurs systèmes d'exploitation différents. Nous nous basons donc sur la diversification de ces systèmes et faisons l'hypothèse que si une faute (accidentelle ou délibérée) est activée sur une machine virtuelle, la probabilité pour qu'elle soit activée également sur une autre machine virtuelle, en se traduisant par une défaillance de mode commun, est très faible. Nous faisons donc l'hypothèse que les systèmes invités (et les machines virtuelles Java qui y sont installées) sont suffisamment diversifiés pour ne pas avoir de faute de conception commune. Nous supposons également qu'il est difficile de corrompre plusieurs systèmes invités (et/ou machine virtuelle Java) pour y introduire une faute provoquant les mêmes erreurs. Dans ce cas, une corruption (ou altération de fonctionnement) d'une machine virtuelle est confinée dans cette machine (cf. chapitre IV) ou conduit à l'altération du résultat délivré par la machine qui sera donc différent du résultat (correct ou non) fourni par une autre machine, et l'objet de validation détectera une erreur.

Avec ces hypothèses, une attaque réussie sur une machine virtuelle exécutant un système est confinée au niveau de cette machine virtuelle, puis détectée au niveau de l'objet de validation permettant de déclencher une alarme.

V.1.1.4 Fautes au niveau de l'objet de validation et son environnement d'exécution

Remarquons que notre hypothèse est basée essentiellement sur la propriété d'isolation que doit garantir le moniteur de machines virtuelles, et sur la propriété de détection de l'objet de validation. Il est donc nécessaire de s'assurer du bon fonctionnement de cet objet afin d'avoir confiance dans le résultat final. Pour cela, il faut que l'OV soit réalisé en imposant les mêmes exigences sur le développement, le test et la validation que pour le moniteur de machines virtuelles, de façon à obtenir le même niveau de confiance, et qu'il est aussi robuste que le MMV vis-à-vis des attaques malveillantes.

L'objet de validation étant une entité logicielle, il faut également s'assurer qu'il s'exécute sur un environnement d'exécution qui n'altère pas le processus de validation. Cet environnement peut varier en fonction de la difficulté de validation et de capture des flux à contrôler. Nous proposons de détailler, dans la section V.3.1, les possibilités d'implantation envisageables pour un tel objet de validation.

V.1.1.5 Fautes au niveau des applications

Dans notre travail, nous considérons que les applications représentent les entités logicielles en charge d'effectuer des tâches de calcul et de permettre d'envoyer des informations à d'autres applications à bord de l'avion. Les applications que nous traitons sont des applications situées dans des machines extérieures à l'avion, et devant interagir avec des entités logicielles à bord. L'originalité de notre approche consiste à fournir aux entités de bord un résultat sûr (c'est-à-dire d'un niveau de crédibilité au moins équivalent au niveau de criticité de la tâche qui utilise ce résultat), même si la machine externe est potentiellement corrompue (cf. hypothèses précédentes).

Il est possible de faire l'hypothèse que les applications puissent contenir des fautes accidentelles (bogues de conception) ou être sujettes à des attaques ciblées. Dans ce cas, il est nécessaire d'implanter une diversification fonctionnelle au niveau de chaque tâche (T_{1A} et T_{1B} de la figure V.1), à savoir un développement utilisant des langages de programmation différents avec des algorithmes diversifiés, voire avec des équipes de développeurs différentes. Dans ce cas, nous pouvons considérer qu'un bogue ou une attaque sur T_{1A} a peu de chance de donner le même résultat erroné qu'une attaque sur T_{1B}, ce qui permet de tolérer le dysfonctionnement de chacune des instances diversifiées [Traverse et al. 2004].

Cependant, dans le cadre de nos travaux, nous considérons que ces applications ont été développées par l'avionneur, et qu'elles ont subi le même cycle de développement que les applications à bord, ce qui leur garantit un niveau de confiance suffisant. Néanmoins, l'environnement d'exécution de ces applications n'est pas sûr, ce qui altère potentiellement leur résultat. Ainsi, nous utilisons le même code source des applications, implanté sur différentes machines virtuelles. Cette redondance ne tolère pas des fautes de conception ou des attaques spécifiques aux applications, mais tolère les fautes qui proviennent du système virtualisé servant de support d'exécution à l'application, à savoir le système d'exploitation et les API qui y sont installées.

V.1.1.6 Synthèse des hypothèses de fautes

Dans les sections précédentes, nous avons discuté les différentes hypothèses de fautes relatives à chaque couche logicielle introduite dans notre architecture. Cette dernière se base sur la virtualisation pour assurer une diversification logicielle permettant d'augment le niveau de confiance global attribué à la machine sur laquelle cette architecture est déployée.

Rappelons que la confiance correspond à un niveau d'intégrité, crédibilité et validation justifiés pour réaliser une tâche d'une criticité donnée. La figure V.2 présente ainsi les composants de confiance dans l'architecture finale proposée.

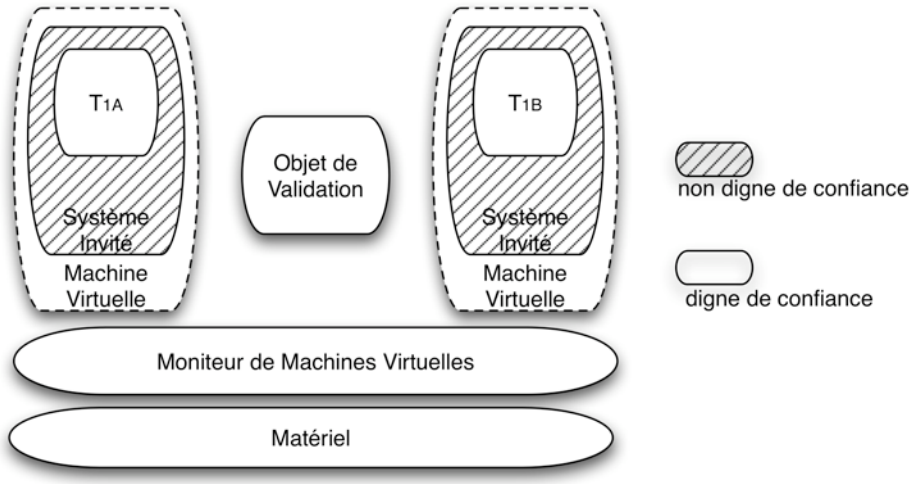


Figure V.2 Composants non dignes de confiance

La figure V.2 met l'accent sur le fait que seuls les systèmes invités sont considérés comme non dignes de confiance, comme nous l'avons précédemment détaillé. Notre architecture vise donc à tolérer les fautes au niveau de ces systèmes virtualisés. C'est pour cette raison que les couches inférieures (matériel et MMV) doivent être de confiance, sinon les systèmes invités pourraient être compromis (par des fautes en provenance de ces couches). L'objet de validation est également de confiance, puisqu'il s'agit de l'entité logicielle en charge de fournir le résultat final des applications T_{1A} et T_{1B} (qui sont également dignes de confiance).

V.1.2 Déterminisme des versions et validation

Les systèmes tolérant aux fautes se basent sur la redondance, qui permet d'implanter des mécanismes de comparaison et de décision. Les exemples du doublement et comparaison et du masquage par vote majoritaire (ou TMR, *Triple Modular Redundancy*) permettent d'illustrer le fonctionnement de tels systèmes [Jean-claude Laprie et al. 1996]. Remarquons que pour pouvoir fournir un résultat correct après comparaison des différentes versions redondantes, il faut avoir établi le bien fondé de la comparaison, selon les deux propriétés suivantes :

- La comparaison se fait entre deux entités comparables. Dans le cas contraire, le résultat de la comparaison n'a aucune signification.
- La comparaison se fait entre deux entités présentant les mêmes informations à valider. En effet, si l'on compare des informations différentes, le résultat est (presque) toujours négatif, et le comparateur générera des alarmes en continu.

La première propriété concerne la nature des entités redondantes. Il est donc nécessaire d'étudier au préalable les entités comparées afin de donner une sémantique au processus de comparaison. Par exemple, si une entité redondante fournit un entier comme résultat et que l'autre fournit une chaîne de caractères, il est clair que la comparaison de ces deux résultats échouera.

La seconde propriété relève des instants de la comparaison : il faut s'assurer qu'en l'absence de fautes les exécutions redondantes fournissent les mêmes informations, malgré l'indépendance des exécutions. Cette propriété est donc liée au déterminisme des exécutions redondantes. Dans le contexte de composants logiciels répliqués, [David Powell 1991] définit le déterminisme comme suit : *« Un groupe de répliques est déterministe, si, en l'absence de fautes, et étant donné le même état initial et les mêmes messages d'entrée pour chaque réplique, le groupe fournit le même ensemble ordonné de messages de sortie »*.

Dans [Poledna 1994], cette définition est étendue pour prendre en considération les contraintes temporelles (en bornant le temps de réponse du système) afin de pouvoir effectuer une comparaison valable. En effet, il est possible d'implanter des versions donnant exactement les mêmes valeurs dans le même ordre, sans pour autant garantir que ces résultats soient fournis au même instant pour toutes les versions. Cette définition du déterminisme en fonction de tels intervalles temporels permet de s'affranchir des contraintes de simultanéité qui rendraient la comparaison impossible en pratique. Dans ce cas, la comparaison n'est effectuée qu'au bout d'un certain délai à respecter, pour permettre à toutes les versions de fournir leur résultat.

Ainsi, afin d'assurer une comparaison correcte, il est nécessaire de garantir le déterminisme des entités redondantes, qui permet de confirmer que l'on compare effectivement les mêmes informations dans des intervalles de temps donnés. Cependant, cette propriété de déterminisme est difficile à prouver, vu les différentes sources d'indéterminisme qui peuvent survenir dans le cade de système tolérant aux fautes que nous déployons. Dans ce qui suit, nous proposons de lister les sources d'indéterminisme susceptibles d'affecter un tel système.

V.1.2.1 Sources d'indéterminisme

Les sources d'indéterminisme sont nombreuses dans un système et ne peuvent être listées exhaustivement. Nous proposons de lister les principales sources identifiées dans [Poledna 1994] :

- ***Entrée incohérentes*** : prenons l'exemple d'un capteur de température qui envoie une mesure de température à chaque entité redondante. Si ce capteur est peu fiable, il est possible que la valeur envoyée ne soit pas la même pour toutes les entités redondantes, ce qui conduit à un résultat potentiellement différent pour chaque entité redondante.
- ***Ordres incohérents*** : prenons l'exemple de files d'attente de messages au niveau de chaque entité redondante. Dans le cas d'entités redondantes distribuées, il est possible que ces messages n'arrivent pas dans le même ordre pour chaque entité redondante, ce qui pourrait conduire à un traitement différent au niveau de chaque entité redondante.
- ***Informations d'appartenance incohérentes*** : si les différentes entités redondantes sont amenées à communiquer entre elles, il faut qu'elle aient toutes la même liste des entités présentes. À défaut, il est possible qu'un destinataire d'une information ne la reçoive pas (s'il ne fait pas partie de la liste connue de l'émetteur), conduisant ainsi à un traitement différent.
- ***Structures non déterministes d'un programme*** : il est clair que l'utilisation de fonctions aléatoires peut conduire à un comportement non déterministe des différentes entités redondantes. De plus, certains langages de programmation (Ada par exemple) ont des structures non déterministes (par exemple, la fonction « *select* » qui effectue un choix arbitraire parmi *N* possibilités).
- ***Informations locales*** : l'accès à une variable locale à une entité redondante et non accessible aux autres entités redondantes peut conduire à un comportement non

déterministe. Un exemple de telle variable est la valeur de l'horloge d'un système. En effet, les valeurs fournies par une horloge sont strictement croissantes. L'interrogation de cette horloge par deux entités redondantes fournit deux instants différents, formant ainsi une source de non déterminisme.

- ***Échéances*** (*timeouts*) : dans certains cas, le calcul des échéances se base sur la lecture d'une horloge locale, ce qui correspond au cas précédent. Cependant, même si l'on suppose qu'une horloge globale soit fournie à chaque entité redondantes et que cette horloge soit sûre de fonctionnement, il est possible qu'une entité décide avant les autres que l'échéance est atteinte (par exemple en raison de différences de vitesse de traitement), causant ainsi une différence de traitement.
- ***Décision dynamique d'ordonnancement*** : cette source d'indéterminisme est très répandue dans les systèmes d'exploitation classiques basés sur un ordonnancement préemptif des différents processus et fils d'exécution (*threads*). Il en découle ainsi un comportement différent, selon qu'un processus ait eu la main avant ou après une modification d'une variable globale par exemple (problématiques d'ordre d'accès aux ressources, d'exclusion mutuelles, etc.)
- ***Précision des calculs*** : il est clair que la représentation mathématique de certains nombres n'est pas infiniment précise et est, de ce fait, limitée par la capacité de la machine. Dans certains cas, la précision de la machine conduit à représenter de façon différente deux résultats de calculs équivalents, ce qui met en question la validité de la comparaison (dans certains cas, on affirmera que deux nombres sont égaux, alors qu'on affirmera le contraire dans d'autres configurations) [Brillant et Knight 1989].

Remarquons que les sources d'indéterminisme sont nombreuses, et dépendent clairement de la nature de la redondance implantée et de la nature des entités redondantes. Ainsi, en fonction de la cause de l'indéterminisme, il est parfois possible d'adopter un mécanisme de renforcement de déterminisme adéquat.

V.1.2.2 Renforcement du déterminisme

Comme nous l'avons précédemment indiqué, il est nécessaire d'identifier les causes d'indéterminisme potentielles au sein des entités redondantes en fonction de l'architecture redondante, afin d'adopter les mécanismes de renforcement de déterminisme adéquats. Dans le cadre de nos travaux, les applications redondantes s'exécutent sur deux systèmes d'exploitation de type *COTS*, utilisant chacun un ordonnanceur préemptif (donc source d'indéterminisme). Sur chaque système virtualisé (cf. figure V.1), plusieurs applications peuvent s'exécuter en même temps. Cette exécution parallèle implique que le traitement d'une tâche peut dépendre de l'ordonnanceur de chaque machine virtuelle (et également de l'ordonnanceur du MMV).

L'approche que nous considérons est similaire à celle présentée dans [Rodrigues et al. 2003], à savoir que nous ne modifions pas le code source des applications redondantes, mais implantons des mécanismes de contrôle et de renforcement de déterminisme comme une enveloppe (*wrapper*) entourant l'application [Salles et al. 1999]. Cette enveloppe permet d'extraire les données abstraites que nous cherchons à valider à travers les mécanismes de comparaison. Comme nous le verrons au Chapitre VI, nous nous intéressons essentiellement aux données de sortie de chaque entité redondante de l'application redondante. Ainsi, au niveau du *wrapper*, nous implantons les mécanismes nécessaires à la capture de ces sorties, à leur identification en fonction de leur origine, puis à leur envoi à l'objet de validation qui est en charge de comparer les différentes sorties et de les valider.

Dans le cadre de notre étude, nous nous sommes limités à des applications ayant un fil d'exécution principal (*monothread*) en charge d'effectuer toutes les opérations séquentiellement, d'une manière similaire à [Fraga et al. 1997]. Cependant, il serait intéressant d'étendre cette hypothèse en considérant des applications redondantes avec plusieurs fils d'exécution (*multithread*). En effet, dans [Napper et al. 2003] et [Friedman et Kama 2003], le déterminisme d'une entité redondante *multithreadée* est renforcé au niveau de la machine virtuelle Java. Les causes d'indéterminisme traitées dans ce cas sont les accès à l'horloge, et les primitives de synchronisation (de type verrou, variables de condition, etc.). La nouvelle machine virtuelle Java proposée implante alors des mécanismes de synchronisation déterministes et un accès contrôlé à l'horloge du système.

Le déterminisme peut également être renforcé au niveau du système d'exploitation, comme c'est le cas pour MARS [Kopetz et al. 1989]. Cependant, ces approches nécessitent une modification importante du système d'exploitation hôte ou de la machine virtuelle Java, ce qui présente un problème vis-à-vis de l'évolution du système (et de la JVM).

Nous pouvons envisager une approche de *wrapper* implanté au niveau d'un intergiciel en charge d'assurer le déterminisme des applications Java *multithreadées* qui s'exécutent dans la machine virtuelle hôte. Divers algorithmes de synchronisation déterministes (LSA *Loose Synchronization Algorithm* [Basile et al. 2002] et PDS : *Preemptive Deterministic Scheduling* [Basile et al. 2003], etc.) ont été étudiés dans [Domaschka et al. 2008]. Cette étude a permis de déceler, d'une part, la forte liaison qui existe entre la nature des interactions applicatives et d'une autre part les algorithmes à implanter pour renforcer le déterminisme.

Ainsi, il est important de spécifier clairement les interactions applicatives que nous cherchons à contrôler afin de rendre sûres de fonctionnement les applications exécutées. Dans le cas de nos travaux, ces interactions engendrent des flux d'information entre des entités qui n'ont pas forcément le même niveau de criticité. Pour contrôler ces flux correctement, en respectant le modèle Totel décrit dans le chapitre III, nous proposons dans la section suivante une formalisation de ce modèle, en se basant sur le principe de dépendance causale.

V.2 FORMALISATION DU MODELE TOTEL

Le modèle Totel est une représentation de mécanismes de contrôle d'intégrité, basé sur une identification des flux d'information existant dans un système donné [E. Totel, J.-P. Blanquart, et al. 1998]. Les entités présentes dans ce modèle sont les suivantes :

- Les objets, qui représentent un module implémentant une tâche donnée. Chaque objet peut consommer et/ou produire des flux de données vers d'autres objets du système.
- Les flux d'informations qui consistent en un envoi d'informations (à travers un canal dédié) d'un objet producteur vers un objet consommateur. Si nous faisions l'analogie entre un canal d'information et un tuyau servant à transporter des fluides, le flux d'informations que nous considérons correspondrait au fluide envoyé à une extrémité du tuyau et reçu à l'autre extrémité.
- Les objets de validation (OV) : ce sont des objets spéciaux dans le sens où ils ont été introduits dans le modèle Totel pour permettre la validation d'un flux, et par conséquent, permettre une remontée d'information d'un niveau peu critique à un niveau plus critique.
- La Base de Traitement de Confiance (*Trusted Computing Base* ou *TCB*) : cette base est chargée d'assurer l'isolation entre les différents objets et objets de validation présents dans le système. Cette isolation doit assurer que l'utilisation d'une ressource commune par différents objets n'altère en aucun cas le fonctionnement de ces objets. Comme nous l'avons précédemment mentionné, dans la littérature, il est question d'isolation spatiale (un objet ne peut pas accéder à la mémoire d'un autre) et d'isolation temporelle (les activités d'un objet ne peuvent pas perturber celles d'un autre) [J. Rushby 2000]. Rappelons que dans nos travaux, nous référons à ces propriétés par la notion de gestion de ressources.
- Les niveaux d'intégrité : à chaque composant (objet, objet de validation et TCB), un niveau d'intégrité est attribué. Ce niveau d'intégrité correspond à la confiance que l'on peut avoir dans le fonctionnement du composant en question (cf. chapitre III). Ces niveaux sont attribués statiquement lors de la spécification du système. Dans le cadre de notre travail, nous nous sommes intéressés aux objets mono-niveaux. Les objets multi-niveaux introduits dans le modèle Totel pourront faire l'objet d'une extension possible.

Le modèle Totel se base, comme d'autres modèles de contrôle d'intégrité, sur l'interdiction de tout flux de provenance d'un objet de bas niveau d'intégrité vers un objet plus critique. Les seuls flux permis sont les flux descendants (c'est-à-dire, vers un niveau égal ou inférieur). En effet, permettre à une entité peu digne de confiance de communiquer avec une entité critique conduit potentiellement à la dégradation du niveau d'intégrité de cette dernière. Cette dégradation peut être vue dans le temps comme suit : si une communication ascendante a eu lieu à un instant t, alors à l'instant $t' > t$, l'entité la plus critique peut être corrompue par l'information reçue. Ce type de dépendance temporelle est facilement interprétable dans le cadre du modèle de dépendance causale (d'Ausbourg 1994).

Dans le modèle Totel, les objets de validation (*OV*) constituent une exception à cette règle dans le sens où ils sont les seules entités du système auxquelles des flux de provenance d'un niveau d'intégrité moindre sont acceptés. Il est important de s'assurer de la validité des objets de validation puisqu'ils constituent les seules entités autorisées à remonter un flux d'un

niveau peu critique vers un niveau critique. À travers une formalisation basée sur le modèle de dépendance causale, nous proposons une démarche qui permet de décider de la validité des OV, en fonction d'hypothèses établies sur les flux d'informations.

Les flux d'informations étant la composante principale du modèle Totel, nous nous proposons donc dans un premier temps de faire un rapprochement entre ces flux et le modèle de dépendance causale (*MDC*) pour spécifier, dans un second temps, les règles de contrôle d'intégrité de Totel dans le formalisme du MDC. Nous fournissons ensuite une représentation des mécanismes d'isolation introduits par la TCB afin de permettre l'implantation de mécanismes de tolérance aux fautes et d'indépendance d'exécution, et proposons une extension du modèle Totel, relative à l'indépendance des versions.

V.2.1 Dépendances causales et flux d'informations

Un flux d'informations correspond à la transmission de données d'un objet *O1* vers un objet *O2*. L'aspect temporel de l'envoi des données n'est pas pris en compte dans le modèle Totel, dans lequel seules les sources et destinations des flux sont représentées. Cependant, il est intéressant d'introduire la composante temporelle puisque la validation d'une donnée est effectuée à un instant *t* donné, ce qui implique que cette validation dépendra de ce qu'on a pu observer durant les instants antérieurs à *t*.

V.2.1.1 Dépendance causale directe et indirecte

Le modèle de dépendance causales (MDC) permet d'introduire la notion de causalité entre les différents objets du système en les indexant temporellement. Rappelons que ce modèle décrit un système par un quadruplet <*O,T,D,V*> dans lequel *O* est l'ensemble des objets, *T* est l'horloge discrète du système, *D* est le domaine de valeurs des objets et *V* une fonction de *O* x *T* dans *D* qui attribue à chaque objet , à un instant donné, une valeur dans l'ensemble des valeurs D. Pour chaque point $y = (O_1, t_1) \in OxT$ du système, on définit deux ensembles : *cône*(*y*) et *dep*(*y*).

Ces deux ensembles représentent deux cônes. Nous avons donc trouvé plus judicieux de les noter ***cause(y)*** et ***dep(y)*** (au lieu de *cône(y)* et *dep(y)*). Le premier ensemble est appelé cône de causalité et correspond à tous les points du système dont dépend *y* d'une manière directe ou indirecte. La dépendance causale introduite dans *cause(y)* correspond à une influence potentielle de tout objet appartenant à ce cône sur le point *y*. Le deuxième ensemble est appelé cône de dépendance, et regroupe les points du système qui dépendront de y dans des instants postérieurs à t_1.

Ainsi, tous les points de *cause*(*y*) ont eu, à des instants antérieurs à *t*, la possibilité d'influencer *y*, et ce, en modifiant directement *y* ou en modifiant de proche en proche des points du système qui conduisent à la modification de *y*. La modification indirecte est définie par une fermeture transitive de la relation de dépendance causale introduite dans le Chapitre III.

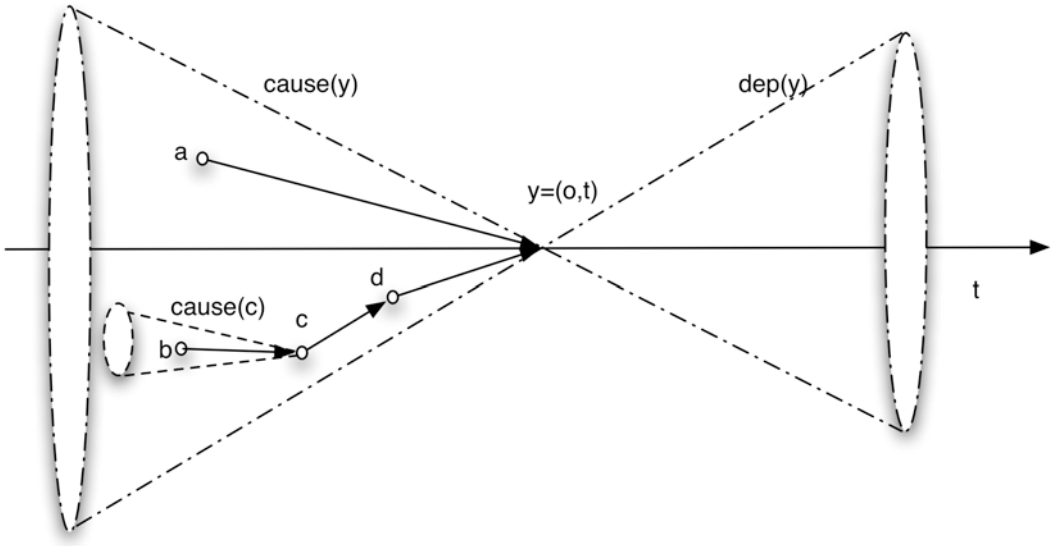


Figure V.3 Dépendances directes et indirectes

Dans la figure V.3, nous considérons une observation d'un point *y* correspondant à un objet *o* à un instant *t*. À ce point correspond l'ensemble {*a*,*b*,*c*,*d*} des points de l'ensemble *cause*(*y*) situés à des instants antérieurs à t. Les flèches représentent une relation de causalité directe entre deux points du système. Ainsi, en reprenant les notations du MDC, *y* dépend causalement de manière directe de l'ensemble *X* où *X*={*a*,*d*}, et on note alors $X \rightarrow y$. Similairement, $\{c\} \rightarrow d$ et $\{b\} \rightarrow c$, ce qui implique que *b* est un élément de *cause*(*c*), et que *c* est un élément de *dep*(*b*) (cf. figure V.3). Par fermeture transitive de la relation de causalité, *y* dépend potentiellement de *c* et *b*, mais d'une façon indirecte.

Dans ce qui suit, nous nous intéressons à la notion de flux de données entre deux points du système, et nous en proposons une interprétation dans le cadre du MDC.

V.2.1.2 Flux et causalité

Comme nous l'avons précédemment mentionné, un flux est un transfert de données d'un objet source O_1 vers un objet destination O_2. Autrement dit, $O1$ peut potentiellement influencer l'exécution de O_2 à travers ce flux. Notons par le couple $((O_1,t_1),(O2,t_2))$ le flux de données de provenance de O_1 à l'instant $t_1$et arrivant à O_2, à l'instant *t2,* où $t_1<t_2$. Dans ce qui suit, nous discutons la relation d'équivalence suivante, établie entre l'existence d'un flux de $O1$ à $O2$ et l'existence d'une dépendance causale entre $O1$ et $O2$:

$$\exists((O_1,t_1) \rightarrow (O_2,t_2)) \Leftrightarrow (O_1,t_1) \in cause(O_2,t_2), \text{ avec } t_1 < t_2$$

L'implication directe de l'équivalence est facile à établir. En effet, s'il existe un flux entre O_1 et O_2, alors, à l'instant t_1 de l'envoi des données, on peut définir un point $y = (O1,t_1)$. À $t_2 > t_1$, O_2 reçoit les informations de provenance de O_1, et se situe donc dans le cône de dépendance de O_1, c'est-à-dire que $(O_2,t_2) \in dép(O_1,t_1)$ ce qui est équivalent à $(O_1,t_1) \in cause(O_2,t_2)$.

La réciproque est également facile à démontrer. Supposons qu'il existe $t_1 < t_2$ tel que $y = (O_1,t_1) \in cause(x)$, $x=(O_2,t_2)$. Comme *cause(x)* est la fermeture transitive de la relation de dépendance causale, alors on peut trouver un point *w* de OxT tel que $y = (O_1,t_1) \in cause(w)$ et $w \rightarrow (O_2,t_2)$. Ainsi, de proche en proche, on construit des ensembles de points en dépendances directes les uns par rapport aux autres, jusqu'à construire un ensemble Z de

points z tel que $(O_1,t_1) \rightarrow z, \forall z \in Z$. La succession de ces ensembles constitue un flux de (O_1,t_1) à (O_2,t_2) à travers lequel des données peuvent être acheminées de O_1 à O_2.

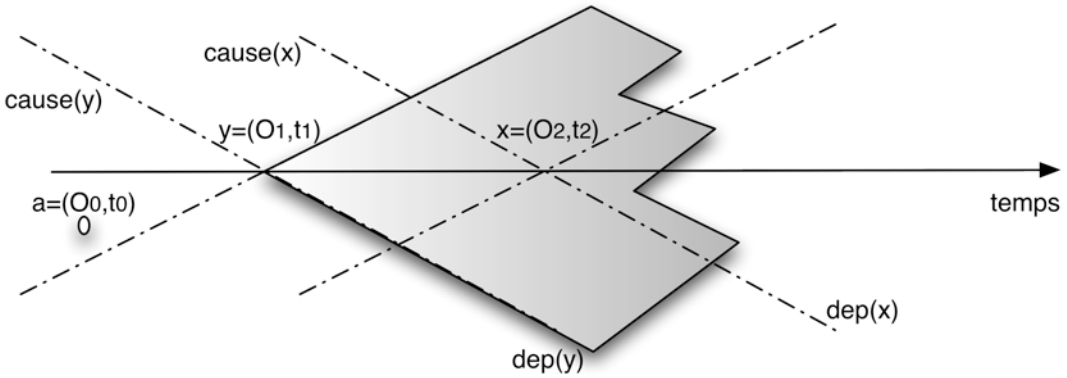


Figure V.4 Flux et cône de dépendances

Ainsi, selon la figure V.4, le point $x = (O_2,t_2)$ est dans le cône de dépendances de y. Si on se place à l'instant t_1, on ne peut pas vérifier qu'une donnée en provenance de O_1 atteindra ou non O_2 dans un instant futur t. Cependant, on peut affirmer l'existence d'un chemin possible de O_1 à O_2 pouvant servir à acheminer un tel flux. En revanche, si on se place à l'instant $t_2 > t_1$, on peut vérifier la provenance des flux reçus et confirmer (ou pas) par conséquent la présence de ces flux. L'équivalence entre l'existence d'un flux et la dépendance causale nous permet de manipuler les mécanismes de contrôles de flux, en nous basant sur le modèle de dépendances causales.

Cependant, il convient de rappeler que nous nous intéressons aux dépendances potentielles, et que l'existence d'un flux (et donc d'une dépendance causale) n'implique pas l'existence d'une influence réelle du producteur du flux sur son consommateur. En effet, reprenons l'exemple de la figure V.4. Le point *a* appartient à *cause(y)* et *y* appartient à *cause(x)*. Il existe alors un flux de *a* vers *y* et de *y* vers *x*, ce qui implique un flux de *a* vers *x*. Supposons qu'à un instant t3, situé entre les instants t_1 et t_2, la valeur de l'objet O_1 soit réinitialisée. Cette réinitialisation implique que les valeurs de O_1 dans des instants postérieurs à t_3 dépendent de cette réinitialisation. Dans ce cas, le point *x*, même s'il dépend causalement de *a*, comme nous l'avons précédemment indiqué, n'est pas réellement influencé par ce dernier. Autrement dit, à partir de t_3, le point *a* n'a plus aucune influence réelle sur le point *x*. Cette propriété peut être exploitée dans un contexte de contrôle de flux, afin de se baser sur les influences réelles au lieu des flux d'informations, comme nous allons le détailler dans la section V.2.3.

V.2.2 Contrôle d'intégrité : utilisation du modèle Totel avec les dépendances causales

Le modèle de dépendance causale introduit les cônes de causalité et de dépendances pour chaque point du système. Un point est un couple objet/instant qui permet de situer un objet durant l'exécution du système. Le MDC présente les objets comme étant des entités passives, observables et/ou modifiables par des **sujets** actifs. Ainsi, les flux d'informations dans le MDC concernent l'évolution des états des objets au court du temps, ce qui permet d'avoir un contrôle plus dynamique comme nous allons le montrer dans les sections suivantes.

Comme indiqué précédemment, le cône de causalité sert à définir l'ensemble des points qui ont le droit de modifier un point donné du système. Cette propriété peut être retenue

intuitivement comme suit : « Si à un instant *t*, un objet *A* dépend causalement d'un objet *B*, alors *B* a eu la possibilité de modifier *A* à un instant t' < t ». Autrement dit, *B* a la possibilité de modifier *A*, et par conséquent d'altérer son intégrité. Les cônes de causalité sont alors liés à l'intégrité des objets en question. Une règle de contrôle de niveau d'intégrité peut donc être exprimée sous forme de contrainte sur les différents cônes de causalité.

Les cônes de dépendance sont moins intuitifs à appréhender. En effet ils traitent des instants futurs par rapport à un instant donné. Dans le MDC, ce cône définit l'ensemble des objets qui ont la possibilité d'être influencés par l'objet en question, et donc d'observer son comportement. L'autorisation (ou l'interdiction) de l'observation relève de la confidentialité. En effet, à partir de l'autorisation d'observation, on peut définir l'ensemble des sujets ayant le droit d'observer la donnée, ce qui permet de définir des niveaux de confidentialité pour chaque objet, et pour chaque sujet effectuant des opérations de modifications ou observations sur les objets.

Dans ce qui suit, nous nous intéressons au contrôle d'intégrité. Nous rappelons dans un premier temps les règles de contrôle d'intégrité, nous présentons quelques conventions de notations dans le modèle Totel, puis nous exprimons les remontées de flux, à travers l'utilisation du MDC.

V.2.2.1 Rappel des règles de contrôle d'intégrité

Dans le MDC, on distingue les objets faisant partie d'un système de sujets extérieurs au système et intervenant pour effectuer des opérations d'observation ou de modification sur ces objets. Chaque objet et sujet se voit attribuer un niveau d'intégrité, en fonction d'une politique de sécurité donnée. Les règles de contrôle d'intégrité se traduisent facilement en inégalité entre ces niveaux : un sujet *S* ne peut modifier un objet *O* que si le niveau d'intégrité de *O* est inférieur ou égal à celui de *S*. Le niveau de S doit donc être supérieur au niveau de tous les objets qu'il peut modifier. En prenant comme échelle celle des niveaux d'intégrité, cette propriété correspond à une règle classique de contrôle d'intégrité qui consiste à n'autoriser que les flux descendants (c'est-à-dire d'un niveau critique vers un niveau moins critique).

Nous nous proposons d'étendre cette propriété en intégrant les propriétés vues dans le modèle Totel. Dans ce dernier, il n'y a pas de distinction entre sujet et objet : un objet est une entité active qui peut éventuellement modifier et/ou observer un autre objet. Comme nous l'avons précédemment mentionné, nous ne traitons que le cas d'intégrité, donc l'altération des objets les uns par les autres. Dans le cadre du modèle Totel, deux types de flux sont considérés :

- Les flux descendants, ce sont les flux classiquement autorisés dans les politiques de contrôle de flux (comme le modèle de dépendances causales par exemple).
- Les flux ascendants, ce sont les flux classiquement interdits dans ces politiques. En effet, permettre à un composant d'un faible niveau d'intégrité d'envoyer des données vers des composants plus critiques est dangereux pour ces derniers puisqu'ils peuvent propager une erreur (plus probable dans un objet de faible niveau), et pourrait conduire à un dysfonctionnement dont les conséquences peuvent être catastrophiques (cf. chapitre III). Cependant, si ce composant peu critique est diversifié, et que les résultats sont communiqués par les différents exemplaires diversifiés via des canaux bien spécifiques, à des entités qui valident ces résultats en appliquant des techniques de tolérance aux fautes, alors le résultat ainsi obtenu est plus digne de confiance et peut être utilisé par les composants critiques.

Dans le MDC, les flux descendants sont permis, et les flux ascendants sont interdits. Cette politique est implantée comme suit : « Tout flux de communication est interdit, sauf s'il est

descendant ». En utilisant la notation de ce modèle, on obtient alors : $x_i \in M_s \Rightarrow n_i(x_i) \leq n_{hi}(s)$, où M_s désigne l'ensemble des objets que peut modifier un sujet s, n_{hi} désigne le niveau d'intégrité du sujet s, et n_i désigne le niveau d'intégrité d'un objet x_i pouvant être modifié par s. En effet, un sujet s possède un niveau d'habilitation relatif à son intégrité (n_{hi}) et ne peut modifier que des objets qui sont d'un niveau d'intégrité moindre.

Permettre une remontée de flux est un challenge intéressant à étudier, nous proposons dans ce qui suit d'établir des conventions de notation dans le cadre du modèle Totel, afin de pouvoir modéliser ensuite le contrôle de tels flux ascendants, en utilisant le MDC.

V.2.2.2 Conventions de notations dans le modèle Totel

Afin de permettre la formalisation des flux remontants, il convient d'établir des conventions de notations permettant d'identifier clairement les différentes entités présentes dans le modèle Totel:

1. Notion d'objet : nous notons ainsi O l'ensemble des objets du système. Dans le modèle Totel, tous les objets sont actifs, et leur état est accessible à travers l'invocation de méthodes spécifiques. Dans le cadre de nos travaux, nous nous intéressons à des objets d'un haut niveau d'abstraction, c'est-à-dire des objets assez complexes (par exemple un objet regroupant plusieurs objets), réalisants des tâches non élémentaires. Il est possible de raffiner cette définition pour ramener les objets aux différents composants d'un système d'exploitation, mais cela ne fait pas partie de notre étude.
2. Notion d'objet de validation : les objets de validation implémentent des mécanismes de tolérance aux fautes afin de valider les données qu'ils reçoivent des objets de niveau d'intégrité moindre. Dans le cadre de nos travaux, nous nous intéressons à deux types de mécanismes de tolérance aux fautes : le contrôle de vraisemblance, et la réplication. Dans le premier cas, une seule instance d'un objet est nécessaire pour effectuer le contrôle, et nous noterons OV_{lh} l'ensemble de ces objets de validation. Dans le second cas, il est nécessaire d'implanter au moins deux instances de l'objet dont on veut valider les résultats afin d'utiliser les mécanismes basés sur la comparaison. Nous noterons $OV_{rép}$ l'ensemble de ces objets de validation. Quelle que soit la nature de l'objet de validation, cet objet est développé pour valider une donnée de provenance et de destination connues. Nous notons ainsi $src(OV)$ et $dest(OV)$ les fonctions qui à un objet de validation OV attribuent respectivement l'objet (ou les objets) source(s) et l'objet de destination.
3. Notion de TCB : la TCB, selon la définition [NCSC 1987] représente l'ensemble minimal de composants de confiance en charge d'assurer des propriétés de sécurité. Dans le modèle Totel, la TCB est en charge d'assurer l'isolation spatiale et l'isolation temporelle entre les différents objets du système. Dans ce manuscrit, nous désignerons toujours cette entité par TCB, tout en gardant en tête l'idée que cette base est en charge d'assurer l'isolation et les communications également.
4. Notion d'étiquetage des objets : chaque objet représenté dans le modèle doit avoir un identifiant unique (par exemple, O_1, O_2…) , permettant de le distinguer des autres objets du système. Les TCB et les objets de validation sont également identifiés d'une façon non ambiguë, en leur associant une étiquette unique. Ils sont respectivement notés :TCB_1, TCB_2… et OV_1, OV_2… Cependant, certains objets sont redondants voire diversifiés, afin de permettre l'implantation de mécanismes de tolérance aux fautes. Dans ce cas, nous ajoutons une étiquette pour identifier les différentes instances. Ainsi, nous pouvons avoir l'ensemble des objets suivants :

$\{O_1, O_{2A}, O_{2B}, O_3\}$, dans lequel O_{2A} et O_{2B} sont deux versions accomplissant la même fonctionnalité. En fonction des hypothèses de fautes et de la nature de l'architecture globale déployée, ces deux objets peuvent être de simples répliques ou deux objets complètement diversifiés (cf section V.2.2.3).

5. Notion de niveau d'intégrité : comme dans la notation du modèle Totel, nous noterons *il* la fonction qui, à un objet du système, attribue un niveau d'intégrité. Ainsi, $il(O_1)$ renvoie la niveau d'intégrité de l'objet O_1. Par définition, la TCB a le niveau d'intégrité de l'objet le plus critique avec lequel elle interagit. Un objet de validation a le niveau d'intégrité de l'objet destination pour lequel il valide un flux de données. Ainsi, $il(OV_1)=il(dest(OV_1))$.

Ces notations nous permettent d'identifier clairement les entités mises en jeu dans le modèle Totel. Rappelons que nous ne faisons pas de différence entre sujet et objet, comme c'est le cas dans le MDC. En effet, un objet dans le modèle Totel peut observer et modifier d'autres objets. Dans ce qui suit, nous ne représenterons que des objets, sur lesquels nous établissons des règles de contrôle de flux.

V.2.2.3 Objets de validation par vraisemblance

Le MDC tel qu'il a été proposé dans [D'ausbourg 1994] permet des contrôles d'intégrité qui sont similaires à ceux de la politique Biba [Biba 1977], et ne permet pas, de ce fait, une remontée de flux, tout en évitant la dégradation des niveaux d'intégrité des composants les plus critiques. Afin de permettre une remontée de flux, nous avons vu qu'il est nécessaire d'introduire des objets de validation développés dans le but de valider des données spécifiques pour un objet de haut niveau d'intégrité (objet renvoyé par la fonction *dest*).

Rappelons que, dans un premier temps, nous nous plaçons dans une vision pessimiste des flux, à savoir une vision qui admet l'existence d'une réelle influence en cas d'existence d'un flux (cf. section V.2.2.1). Soient O_1 et O_2 deux objets tels que $il(O_1)<il(O_2)$. Intéressons nous à un flux de O_1 vers O_2, noté ($O_1 \rightarrow O_2$), ce qui implique qu'il existe deux instants t_1 et t_2 tel que $(O_1,t_1) \in cause(O_2,t_2), t_1 < t_2$ (t_2 représente alors l'instant de réception du flux et t_1 l'instant de son émission).

Dans ce paragraphe, nous étudions le contrôle de flux remontant effectué par des objets de validation de vraisemblance. Soit OV un objet de validation dans OV_{lh}. Le flux de O_1 (émis à l'instant t_1) à O_2 (reçu à l'instatant t_2) est permis si et seulement si :

$$\exists OV \in OV_{lh}, O_1 \rightarrow OV, dest(OV) = O_2, \ \forall O \in dep(O_1,t_1) \cap cause(O_2,t_2), il(O) = il(O_2) \text{ et } \left(dep(O_1,t_1) \cap cause(O_2,t_2)\right) \subset \left(OV_{lh} \cup OV_{rép}\right).$$

La première partie de la condition nécessite l'existence d'un objet de validation de vraisemblance propre à O_2, qui est en dépendance causale directe avec O_1. La deuxième partie suppose que tout objet commun entre le cône de dépendance de O_1 et de causalité de O_2 est du niveau d'intégrité de O_2. La troisième partie implique que les seuls objets communs entre le cône de dépendance de O_1 et celui de causalité de O_2 sont des objets de validation (cf. figure V.5).

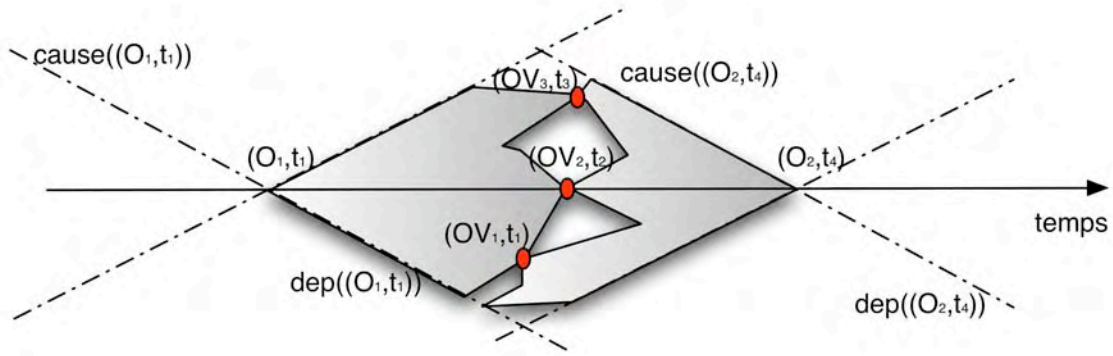


Figure V.5 Objets de validation et cônes de causalité

D'après les règles du MDC, tous les objets du cône *dep((O1,t1))* sont des objets ayant le même niveau d'intégrité que *((O1,t1)*, ou des objets de validation (exception que nous ajoutons selon le modèle Totel). Dans la figure V.5, ce sont les objets de validation OV_1, OV_2 et OV_3. La règle que nous avons établie suppose donc qu'au moins un de ces objets de validation est un objet de validation de vraisemblance propre à l'objet O_2 (pour valider des données à destination de O_2).

V.2.2.4 Objets de validation par réplication

Ces objets de validation implantent des mécanismes de tolérance aux fautes basés sur la réplication. Dans ce qui suit, nous nous intéressons à formaliser les besoins dans le cadre de deux instances, sachant que l'extension à N instances est facile à réaliser. Soient O_{1A} et O_{1B} deux instances logicielles réalisant une même tâche, mais en étant diversifiées. Ces deux instances possèdent le même niveau d'intégrité ($il(O_{1A}) = il(O_{1B})$). Notons par X l'ensemble des objets dépendants à la fois de O_{1A} et O_{1B}, $X = dép(O_{1A},t_1) \cap dép(O_{1B},t_2), \forall t_1,t_2 \in \Re$. On note par $X|_n$ l'ensemble des objets de X ayant un niveau d'intégrité n ; $X|_n = \{x \in X \mid il(x) = n\}$. Soit O_2 un objet tel que $il(O_2)>il(O_{1A})$. O_2 reçoit un flux de O_{1A} et O_{1B} si et seulement si $\exists OV \in OV_{rép} \cap X|_{il(O_2)}$, $O_{1A} \rightarrow OV, O_{1B} \rightarrow OV$ et $dest(OV)=O_2$ (cf. figure V.6).

Dans la figure V.6, nous avons délimité l'ensemble X, et représenté uniquement l'objet *O2* et l'objet de validation lui correspondant. Cet ensemble peut contenir d'autres objets, qui peuvent interagir entre eux, mais leurs interactions sont toujours régies par les règles de contrôle de flux que nous avons précédemment citées. Ainsi, les objets qui dépendent à la fois de O_{1A} et O_{1B} (donc des objets de l'ensemble X) sont des objets qui ont un niveau d'intégrité inférieur ou égal à celui de O_{1A} (ce qui correspondrait à des flux descendants), avec pour seules exceptions les objets de validation. Dans le cas où un objet est d'un niveau d'intégrité supérieur à celui de O_{1A}, la règle que nous venons de mentionner ci-dessus est appliquée, et un objet de validation par redondance doit être implanté, en respectant cette règle, afin de valider tout flux possible de O_{1A} et O_{1B} vers cet objet de niveau supérieur. Nous appliquons ainsi cette règle de manière récursive jusqu'à ce qu'on couvre tous les flux ascendants.

Notons que les dépendances causales entre l'objet de validation et les deux instances sont des dépendances causales directes (cf. V.1.1.1). En effet, l'objet de validation est directement lié, grâce à la TCB aux objets qui lui fournissent les données à valider.

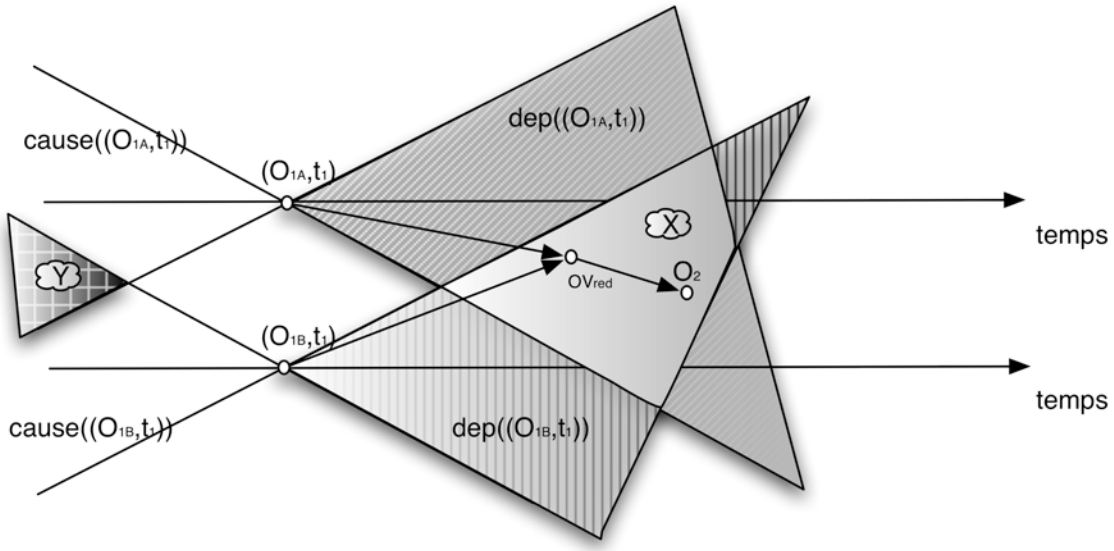


Figure V.6 Cônes de dépendances pour un objet de validation par redondance

Remarquons que nous n'avons pas traité, jusqu'à présent, la problématique de l'indépendance des versions O_{1A} et O_{1B}. Cependant, afin de garantir une validation correcte en utilisant la redondance, il est nécessaire d'avoir des données provenant de sources indépendantes. L'indépendance implique l'absence de fautes de mode commun. Nous supposons que les deux entités redondantes sont suffisamment différentes et s'exécutent également sur des plateformes suffisamment différentes pour éviter toute faute (de conception ou autre) de mode commun. Nous nous intéressons alors au cas où les deux entités redondantes (O_{1A} et O_{1B}) ont une source de données commune. Cette source, si elle est corrompue, peut altérer aussi bien O_{1A} que O_{1B}, ce qui peut mettre en échec la validation de l'OV (par exemple si la validation se base sur une simple comparaison des données des deux instances). Dans ce cas, le flux ascendant serait permis même si les deux exemplaires sont corrompus. Dans la section suivante, nous discutons de l'indépendance des répliques, et proposons d'éventuelles extensions du modèle Totel pour avoir plus de flexibilité quant à cette propriété.

V.2.3 Validité des résultats de l'objet de validation

Nous proposons de discuter dans cette section de la validité des résultats envoyés par l'objet de validation à l'objet pour lequel il valide les données. Cette validité est décidable dans certains cas, sous des hypothèses que nous proposons de détailler, et qui concernent les **intersections** de cônes de causalité, **le niveau de confiance** des éléments de cette intersection et **le degré d'observation** que nous avons sur le système. Cette décidabilité est résumé dans la figure V.7.

La séparation des différentes hypothèses se base sur l'accès aux informations suivantes:

- Les dépendances causales de chaque objet : ceci implique que nous avons accès aux cônes de causalité de chacune des entités redondantes, afin de pouvoir analyser l'intersection de ces deux cônes.
- Le niveau de confiance de chaque élément dans cette intersection, ainsi que le niveau de confiance de chacune des entités redondantes.
- Les influences réelles qui sont permises par une observation des interactions réelles des objets au cours du temps.

Ainsi, si nous avons uniquement accès au passé causal, on ne peut affirmer que l'objet de validation remplit bien sa mission que si l'intersection des deux cônes est vide. En effet, dans la figure V.6, nous avons présenté la zone Y qui correspond à l'intersection des deux cônes de causalité de O_{1A} et O_{1B} à l'instant $_{t1}$. $Y=cause((O_{1A},t_1)) \cap cause((O_{1B},t_1))$. Ces cônes contiennent les points dont dépendent à la fois O_{1A} et O_{1B} à l'instant où leurs informations sont transmises à l'objet $OV_{rép}$. Nous nous intéressons dans ce qui suit à la validité des sorties de l'objet de validation, en fonction du contenu de l'ensemble *Y*. En effet, dans le cas où *Y* est un ensemble vide, aucune source commune entre les deux instances n'existe. Ceci implique l'indépendance de leurs sources, et donc la validité de la validation. Nous pouvons, par conséquent, avoir confiance dans le résultat renvoyé par l'objet de validation.

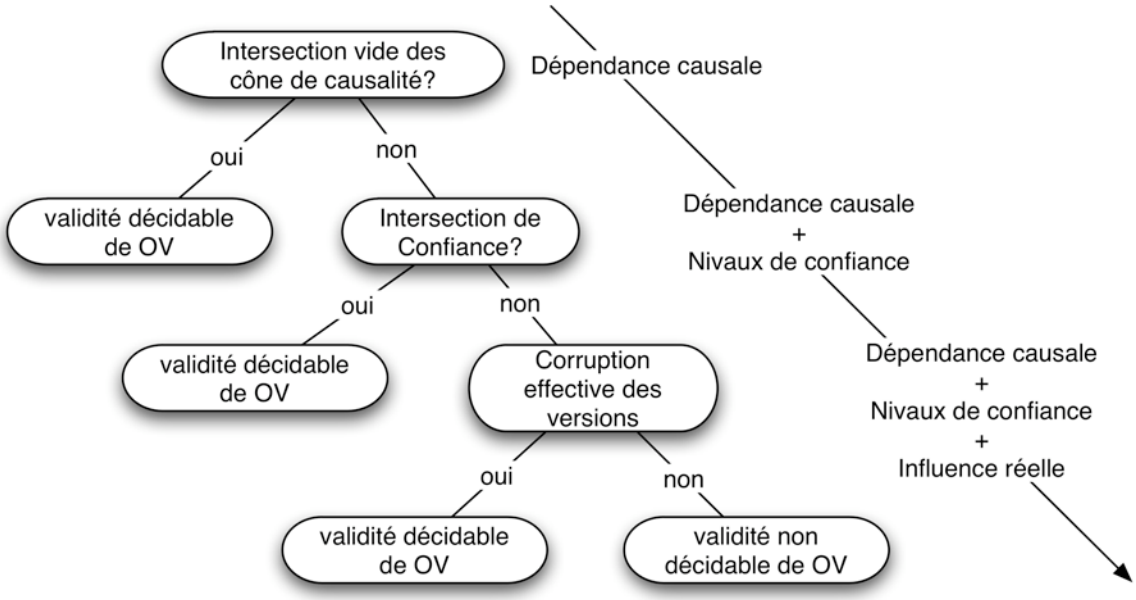


Figure V.7 Hypothèses de validité des objets de validation

Dans le cas où *Y* n'est pas vide, nous étudions alors le niveau de confiance des différents objets de *Y*, ce qui est le but de la section suivante.

V.2.3.1 Niveau de confiance et validation des flux

Il est clair que dans le cas où Y n'est pas vide, l'indépendance des versions n'est plus assurée, et nous pouvons considérer que les flux comparés par l'objet de validation peuvent tous deux être contaminés par des erreurs dans *Y*. Cependant cette hypothèse est trop restrictive et ne permet pas de prendre en considération les niveaux de confiance de certains objets présents dans *Y*. En effet, comme nous l'avons présenté dans le chapitre III, les niveaux que l'on attribue aux différents objets du système correspondent à des niveaux de confiance, obtenus par des méthodes de développement, de validation, et de vérification de crédibilité et d'intégrité [Laarouchi et al. 2009a]. Il serait donc judicieux de tenir compte des niveaux des objets qui seraient présents dans Y. Ainsi, on pourra autoriser une source commune d'information si elle est d'un niveau au moins égal à celui de l'objet destinataire du flux ascendant. Si l'on reprend la notation de la section V.1.2.4, cette condition s'exprime comme suit : $Y=Y|_n$, $n \geq il(O_2)$ où O_2 représente l'objet qui reçoit un flux ascendant venant des objets diversifiés O_{1A} et O_{1B} (cf. Figure V.8). Cette condition implique que tout objet commun aux instances est d'un niveau au moins supérieur ou égal au niveau de O_2.

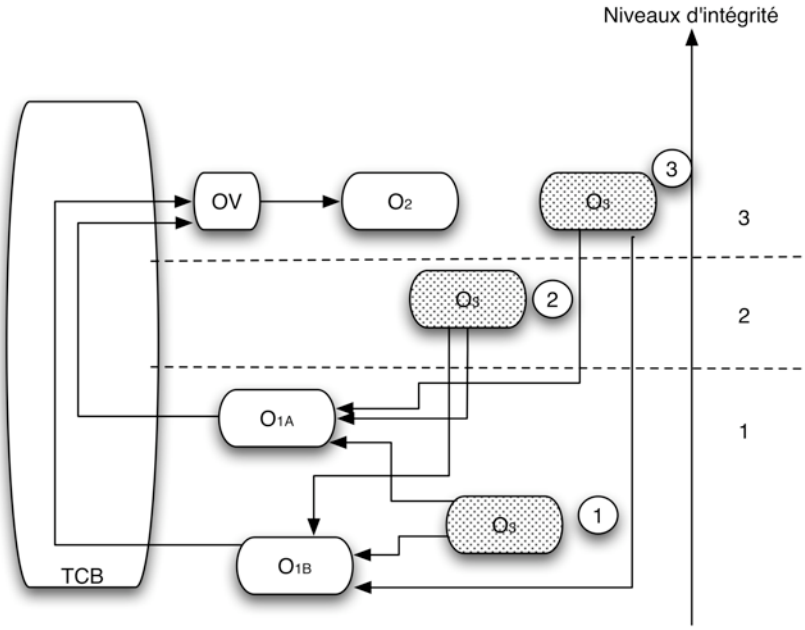


Figure V.8 Indépendance des répliques et niveaux de confiance

Dans la figure V.8, nous représentons différents cas de figure pour la mise en oeuvre d'un objet O_3 à différents niveaux d'intégrité. L'objet O_3 est une source commune pour les deux instances O_{1A} et O_{1B}, et nous discutons dans ce qui suit les propriétés de validation par l'*OV* en fonction du niveau de confiance attribué à cette source commune :

- Le cas (1) de la figure V.8 correspond à l'implantation d'un objet O_3 au niveau d'intégrité 1. Dans ce cas, la confiance dans cet objet est faible, influant ainsi sur le niveau de crédibilité que l'on peut avoir dans les données envoyées par cet objet. Dans un tel cas, l'ensemble Y (qui est réduit à O_3) contient des éléments peu crédibles. Ainsi, si O_3 est incorrect, il peut, à travers la corruption simultanée des deux instances O_{1A} et O_{1B}, envoyer des données erronées à O_2, sans que l'objet de validation ne les détecte (par exemple si l'objet de validation est basé sur une simple comparaison comme nous l'avons précédemment indiqué). Dans un tel cas, il faut arrêter le fonctionnement de l'objet de validation et de lever une exception, ou ajouter d'autres hypothèses, comme nous le montrons dans la section V.2.3.2.
- Le cas (2) de la figure V.8 correspond à un objet O_3 implanté au niveau d'intégrité supérieur à celui de O_{1A} et O_{1B} mais inférieur à celui de O_2 Dans ce cas, O_3 est plus crédible que les deux instances O_{1A} et O_{1B}, cependant, il est toujours moins digne de confiance que O_2. Comme les données de provenance O_3 peuvent, à travers O_{1A} et O_{1B} influencer le comportement de O_2, ce cas de figure présente un potentiel danger, et l'objet de validation doit également lever une exception comme c'est le cas dans le paragraphe précédent.
- Dans le cas (3) de la figure V.8, O_3 est implanté au même niveau que O_2. Dans ce cas, les données de provenance O_3 sont acceptées. Cette hypothèse constitue un enrichissement dans le modèle initial, puisqu'elle permet une remontée de flux par des objets de validation basée sur la redondance, même si les instances diversifiées ont des sources d'information communes. Cependant, il est nécessaire de s'assurer que les flux ($O_3 \rightarrow O_{1A}$) et ($O_3 \rightarrow O_{1B}$) ne sont pas modifiés par d'autres objets présents dans les niveaux 1 et 2.

Ainsi, nous pouvons nous baser sur les niveaux de confiance pour décider de la validité des résultats de l'OV, même si les instances diversifiées ont des sources d'information communes. Dans la section suivante, nous faisons une hypothèse supplémentaire quant à l'observabilité des flux réels pour établir la validité de l'OV.

V.2.3.2 Observations réelles et validation des flux

Comme nous l'avons indiqué lors de la discussion de l'équivalence dans le paragraphe V.2.1.2, nous avons considéré que l'existence d'un flux entre deux objets correspond à l'existence d'une influence réelle de l'objet émetteur sur l'objet récepteur du flux. Cependant, si l'on suppose que les flux dans un système sont observables, nous pouvons retracer chaque flux et établir le niveau d'intégrité de sa source.

Ainsi, si l'on reprend les deux cas (1) et (2) de la Figure V.8, l'observation des activités du système peut tracer tout flux d'origine O_3. Dans le cas où cet objet enverrait un flux vers une seule instance de O_1 (soit O_{1A} ou O_{1B}), nous pouvons faire l'hypothèse que dans ce cas, il n'y pas de fautes de mode commun et que l'objet de validation remplira correctement sa tâche de validation.

Notons que dans ce cas, nous faisons l'hypothèse de l'observabilité de tout flux entre tous les objets du système. Cette hypothèse paraît forte, cependant elle est réaliste. En effet, prenons l'exemple d'une architecture MILS (*Multiple Independent Levels of Security and Safety*) [Alves-Foss et al. 2006]. Dans ces architectures, des partitions sont introduites et toute communication entre ces dernières sont observables. Afin d'implanter notre mécanisme de contrôle de flux ascendants, nous pouvons établir l'hypothèse d'observabilité entre les différentes partitions, et pour l'intérieur des partitions, établir l'hypothèse de visibilité du niveau de confiance (présenté dans la section précédente) afin de traiter un nombre plus important de cas de flux réels et éviter la non décidabilité de la validité du résultat de l'OV (cf. figure V.7).

Remarquons qu'à travers les notions de confiance et d'observabilité, nous avons pu traiter plus de flux réels dans le sens où nous sommes capables de décider de la validité de l'OV dans un spectre plus large de cas possibles de flux réels.

Dans cette section, nous avons réalisé une analyse des besoins architecturaux du modèle Totel, et avons pu établir le lien avec le modèle de dépendance causale. À ce point, nous considérons que nous avons clarifié les différents concepts manipulés dans le modèle Totel, et avons fourni assez d'éléments pour qu'à partir du MDC nous puissions effectuer des opérations automatiques d'analyse de flux.

En effet, nous avons spécifié les règles de contrôle de flux et proposé quelques extensions possibles pour traiter plus de flux réels. Ces règles peuvent être représentées sous différentes formes. Par exemple, le passage du MDC à un graphe orienté est facile à réaliser (il suffit de représenter la relation de dépendance entre deux objets comme arc entre deux nœuds du graphe). Une fois un tel graphe construit, nous pouvons réaliser une analyse globale du graphe pour détecter les flux remontants par exemple, ou les OV de réplication recevant des données de versions ayant des sources communes.

D'autres travaux ont été réalisés au niveau de l'analyse des politiques de sécurité pour générer automatiquement des architectures respectant ces politiques. Dans le projet POK [Delange et al. 2008] une représentation sous AADL [Rugina et al. 2008] d'une politique de sécurité est appliquée à un code non sûr de fonctionnement (cf. figure V.8, reprise de [Pok Project]). Le code de l'application est ainsi analysé puis modifié pour qu'il respecte les spécifications AADL de la politique de sécurité. Comme extension de nos travaux, nous pouvons envisager

une représentation sous AADL du modèle Totel, en respectant le formalisme que nous avons présenté dans cette section. Cette représentation permettrait alors, en étant intégrée dans l'architecture POK, de déployer des applications qui permettent des remontées de flux de manière sûre.

***

Nous avons vu dans cette section que l'identification claire des flux et des différents objets manipulés est importante pour permettre une bonne implantation des mécanismes de sécurité. Tout en suivant les hypothèses de fautes que nous avons présentées dans la section V.1, nous proposons dans la suite de ce chapitre de discuter les différents niveaux envisageables d'implantation de mécanismes de contrôle de flux. Nous nous intéressons ensuite à l'implantation de ces mécanismes dans un environnement distribué.

V.3 ANALYSE FONCTIONNELLE ET CONCEPTION

Dans la section V.1, nous avons présenté les hypothèses de fautes concernant les applications et les supports d'exécution. Nous avons aussi présenté la notion d'un objet de validation en charge de valider les flux en provenance d'objets diversifiés de niveau inférieur. Dans cette section, nous proposons d'étudier plus en détail les caractéristiques de la capture des flux entre les différentes versions, puis de discuter du niveau de placement de l'objet de validation. Nous traiterons ensuite de la problématique d'un contexte d'exécution distribué et des différentes propriétés d'isolation qui doivent y être associées.

V.3.1 Niveaux d'identification et de validation des flux

Comme nous l'avons vu dans la section V.1, une bonne identification des flux doit être réalisée afin d'en permettre un contrôle complet, en vue de décider de leur validité avant de les laisser passer ou de les bloquer (en fonction des règles établies dans la section V.2). Dans le paragraphe suivant, nos présentons les différents niveaux possibles d'identification des flux dans un système et discutons des avantages et inconvénients de ces niveaux d'identification.

V.3.1.1 Niveaux d'identification et de capture

Nous nous intéressons dans ce paragraphe à identifier les différents niveaux logiciels de capture des flux. Rappelons qu'un flux consiste en un envoi d'informations depuis un objet du système vers un autre. Comme nous l'avons précédemment mentionné, nous traiterons des objets d'applications logicielles. Dans ce cas, les flux seront identifiés comme tout échange d'information entre les applications du système global.

Il convient de noter que nous supposons que tous les objets sont déjà identifiés, permettant ainsi de suivre clairement tout flux existant. Le cas où des objets existent et ne sont pas identifiés ou déclarés est plus difficile à gérer, puisque de tels objets représenteraient une menace potentielle pour les objets les plus critiques. Cependant, une telle menace peut être évitée en identifiant clairement les flux remontants permis, et en implémentant les objets de validation qui leur correspondent. Un flux remontant en provenance d'un objet non identifié est considéré comme flux illégal et est donc bloqué par la TCB.

Dans ce qui suit nous nous intéressons aux différents niveaux d'identification des flux relatifs à l'envoi d'informations entre objets applicatifs identifiés dans l'architecture globale.

Au niveau de l'objet

Effectuer l'identification des flux au niveau de l'objet implique que cet objet ait été spécialement conçu pour l'architecture tolérante aux fautes que nous proposons, c'est-à-dire une architecture implémentant directement le modèle Totel. Ainsi, chaque objet identifie clairement les autres objets avec lesquels il communique puis étiquette tous les flux dont il est la source avec un identifiant qui lui est propre.

Il est intéressant d'identifier les flux à ce niveau, puisqu'elle implique la prise en compte, par l'application, de la collaboration entre les objets diversifiés et les objets de validation, permettant ainsi de réduire la complexité de déploiement d'une redondance logicielle. Cependant, notons qu'il est nécessaire de modifier le comportement des objets applicatifs pour qu'ils prennent en charge de telles identifications et capture des flux, ce qui sous nous oblige à avoir accès au code source des objets. Dans certains cas, ce code n'est pas toujours accessible, et spécialement dans le cas d'utilisation de composants sur étagère (*COTS*).

Au niveau d'un intergiciel

L'implantation au niveau d'un intergiciel (*middleware*) consiste à réaliser l'identification et étiquetage des flux d'une manière transparente aux objets communicants. Cette propriété est similaire à ce que nous avons présenté dans la section V.1.2.2 concernant le renforcement du déterminisme d'une façon transparente aux versions [Rodrigues, Castro, et Liskov 2003]. En effet, l'utilisation d'un intergiciel sous forme de *wrapper* peut renforcer le déterminisme en détectant les flux, les identifiant puis les transférant aux objets de validation.

Dans le cadre de nos travaux, nous détaillerons dans le chapitre suivant l'implémentation de cet intergiciel en nous basant sur des objets applicatifs existant chez Airbus et étudierons les contraintes d'implantation qui en découlent. Notons toutefois qu'un tel niveau de capture et d'identification se fait d'une façon complètement transparente, permettant ainsi la réutilisation de composants sur étagères ou d'une façon plus générale l'utilisation d'applications dont le code source n'est pas facilement accessible.

Au niveau du système virtualisé

La capture au niveau système suppose que dans chaque système virtualisé (c'est-à-dire le système "invité", hébergé dans une machine virtuelle), un module est ajouté afin d'identifier la source de chaque flux entre les différents objets, puis de l'étiqueter correctement et l'envoyer à l'objet de validation. Rappelons que par système virtualisé (cf. Figure V.1), nous désignons la couche logicielle située entre l'application et le MMV installé directement sur le matériel. Cette couche contient donc le système d'exploitation virtualisé, mais également les différentes API que l'application utilise. Ainsi, le module qu'il faut développer à ce niveau peut être soit sous la forme d'un module noyau classique, soit sous la forme d'une extension de l'API que l'application utilise. Par exemple, dans le cas où l'API est une machine virtuelle Java, l'implantation à ce niveau implique une modification de la machine virtuelle Java pour y ajouter tous les mécanismes de capture, d'indentification et de routage des flux.

Par rapport à une capture au niveau intergiciel, ce niveau présente l'avantage d'être plus proche du matériel, et donc plus exhaustif quant au processus de capture. Cependant, la modification d'un système d'exploitation ou d'une API aussi riche comme celle d'une machine virtuelle Java est une tâche compliquée, qui doit être maintenue ou refaite à chaque mise à jour du système d'exploitation, de l'API ou de la machine virtuelle Java, une fois l'architecture déployée.

Au niveau de l'hyperviseur

La capture au niveau de l'hyperviseur permet de contrôler tous les flux à destination du matériel, tout en identifiant leur source. Il est donc possible d'implanter notre mécanisme d'identification à ce niveau. Cependant, l'ajout d'un tel module au niveau de l'hyperviseur doit se faire sans altérer le fonctionnement de ce dernier. En effet, le MMV est une entité qui doit être sûre de fonctionnement, et l'ajout d'une fonctionnalité telle que la capture des flux nécessiterait une revalidation du MMV pour montrer qu'il est toujours sûr.

L'étude de ces différents niveaux montre que le déploiement de la capture des flux soulève la problématique de validation de ce mécanisme, et la validation de la couche qui l'implémente. Dans le chapitre VI, nous présenterons plus en détail les mécanismes de capture, en nous basant sur des cas d'étude réels. Ces cas d'étude nous permettront d'étudier plus spécifiquement la sémantique des flux, en vue de leur comparaison et validation. En effet, l'aspect sémantique est important puisqu'il fait partie des sources de non déterminisme qu'il

faut traiter afin d'avoir une validation correcte. Cette validation dépend également des niveaux d'implantation de chaque objet de validation, ce qui est traité par le prochain paragraphe.

V.3.1.2 Niveaux de validation

Nous discutons dans ce paragraphe des différents niveaux envisageables pour l'implantation de l'objet de validation. Rappelons que l'objet de validation doit avoir le même niveau d'intégrité que l'objet pour lequel il valide les données. Afin d'assurer une bonne exécution de cet objet, il faut que le support d'exécution de cet objet soit aussi de confiance, sinon nous tombons dans les mêmes conditions que nous avons précédemment décrites, à savoir une entité sûre qui s'exécute sur une plateforme non sûre. Dans ce qui suit, nous présentons trois niveaux où il est possible d'implanter l'objet de validation.

Implantation au niveau du MMV

La capture de flux au niveau du MMV peut s'accompagner d'une comparaison par l'objet de validation. En effet, le MMV est une entité sûre de fonctionnement, et donc respecte la condition que nous avons citée ci-dessus. Cependant, comme nous l'avons indiqué dans le paragraphe V.3.1.1, l'implantation d'une nouvelle fonctionnalité dans le MMV doit être du même niveau de confiance (donc développé et validé par des méthodes adéquates) et nous devons nous assurer qu'une telle implantation ne modifie pas le comportement de ce moniteur. Une telle tâche peut s'avérer longue et coûteuse d'un point de vue économique, surtout si la comparaison par l'objet de validation porte sur des primitives qui ne sont pas gérées par le MMV de base (par exemple la gestion d'un serveur X graphique). En effet, pour faciliter leur validation, les moniteurs sont généralement minimaux et ne contiennent que les primitives basiques. L'intégration d'un objet de validation complexe dans le MMV nécessiterait de mettre en œuvre des méthodes de développement et de validation coûteuses.

Implantation au niveau d'un binaire

Dans certains hyperviseurs, il est possible d'exécuter directement du code binaire sur l'hyperviseur [Kaiser et Wagner], sans avoir à passer par un système d'exploitation invité. Ce niveau d'implantation est intéressant dans le sens où il ne modifie pas le MMV. Cependant, l'objet de validation ainsi implémenté doit comprendre également toutes les bibliothèques n'existant pas dans le MMV et dont l'objet de validation aurait besoin pour des fonctionnalités spécifiques. Il est clair que de telles bibliothèques doivent également être validées au niveau d'intégrité de l'objet de validation. Une telle tâche peut s'avérer coûteuse également, comme c'est le cas pour le niveau d'implantation précédent.

Implantation au niveau d'une machine virtuelle sûre

L'implantation au niveau d'une machine virtuelle sûre (c'est-à-dire comportant un système invité d'un niveau de confiance suffisant) permet à l'objet de validation d'utiliser toutes les bibliothèques du système invité, sans avoir à les revalider, ni les développer comme c'est le cas des niveaux précédents.

Une telle implantation ne modifie pas le MMV, et ne modifie pas le système invité non plus. En effet, l'objet de validation est implémenté sous la forme d'une application qui s'exécute sur le système sûr. Ce dernier, de par sa nature, permet une telle exécution tout en assurant son intégrité.

Notons également que de tels systèmes d'exploitation existent, et sont déjà utilisés en avionique, il serait donc profitable de les intégrer dans notre architecture globale.

En résumé, nous avons étudié les différents niveaux d'identification des flux et d'implantation des objets de validation, et nous avons opté pour une identification au niveau d'un intergiciel et une validation au niveau d'un système d'exploitation sûr implanté dans une machine virtuelle s'exécutant sur un MMV sûr également. Ainsi, avec une telle architecture, nous pouvons remonter les informations d'une entité peu critique vers une entité plus critique. Cependant, cette remontée assurée par la TCB pose le problème de validation de cette TCB et également de son implantation dans un environnement distribué.

V.3.2 La TCB : entre complexité, validation et environnement distribué

La TCB, telle que nous l'avons présentée dans la section V.2, joue un rôle primordial dans le contrôle du sens des flux. Elle permet les flux descendants, contrôle les objets émetteurs de flux remontants et décide de les envoyer aux objets de validation correspondant ou de les bloquer. Il est donc important de s'assurer que cette TCB remplit bien son rôle, et c'est pour cette raison qu'elle doit être validée au niveau de l'objet le plus critique pour lequel elle envoie des flux. D'un autre côté, cette TCB est potentiellement complexe puisqu'elle doit implémenter des mécanismes de communication communs avec des objets de bas niveau de confiance (par exemple des systèmes d'exploitation sur étagères). Or il est connu dans le monde du développement que les applications les plus critiques doivent être le plus simple possible [Yves Deswarte et al. 1999]. Nous avons donc clairement un paradoxe : la TCB doit être digne de confiance et complexe également (cf. Figure V.9).

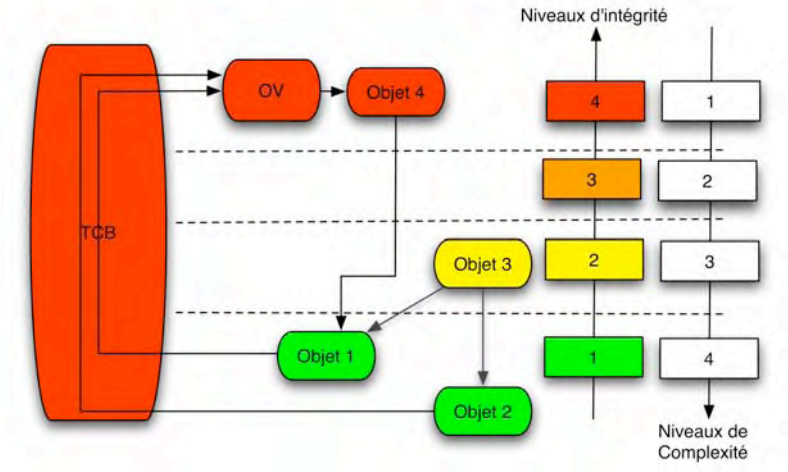


Figure V.9 Niveaux de complexité et d'intégrité des objets dans le modèle Totel

Ainsi, dans la figure V.9, nous remarquons que la TCB est du niveau d'intégrité 4 (le même que celui de l'objet 4), en même temps, la TCB est en communication avec des objets de niveau de complexité 4 (objets 1 et 2).

V.3.2.1 Décomposition de la TCB

Pour palier ce paradoxe apparent, nous pouvons jouer sur le niveau de complexité et de validation de la TCB, en réduisant le niveau d'intégrité maximal des objets contrôlés par cette TCB. Autrement dit, nous réduisons la portée de la TCB pour qu'elle ne contrôle plus des flux vers des entités très critiques (par rapport au niveau de l'objet le moins critique). Dans le cas de la Figure V.9, nous pouvons réduire cette portée en supposant qu'une TCB ne peut contrôler que les flux d'un niveau *n* à un niveau *n+1,* comme le montre la Figure V.10.

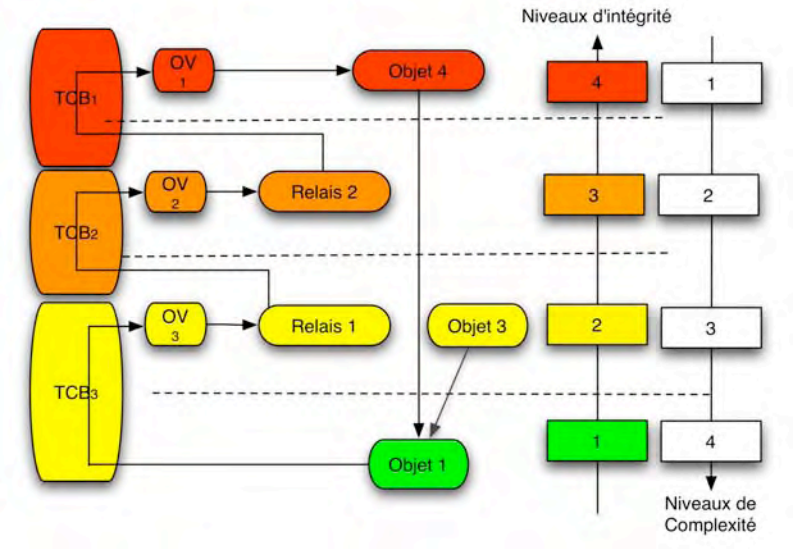


Figure V.10 Décomposition de la TCB

Dans la figure V.10, nous n'avons représenté que l'objet 1 au niveau le plus bas pour ne pas encombrer le schéma. Cependant, le principe reste le même : chaque TCB est en charge de contrôler les flux d'un niveau *n* à un niveau *n+1*. Avec une telle hypothèse, chaque TCB ne gère que des niveaux d'intégrité et de complexité homogènes, ce qui permet d'implémenter et valider une telle TCB : soit le niveau de complexité de la TCB est élevé et le niveau d'intégrité visé est faible, soit le niveau d'intégrité visé est élevé et le niveau de complexité est faible. Afin d'assurer cette décomposition de la TCB, nous introduisons des relais (*proxies*) développés au niveau d'intégrité correspondant au niveau *n+1*, où *n* correspond au niveau d'intégrité le plus bas contrôlé par une TCB donnée (cf. figure V.10). Notons que pour respecter la formalisation présentée dans la section V.2, nous avons ajouté des objets de validation propres à chaque relais, afin de valider les flux de provenance un niveau inférieur. Nous pouvons implanter dans ces objets de validation des mécanismes de contrôle qui sont propres aux mécanismes de communication de la TCB [Laarouchi et al. 2008]. Ainsi, si la TCB utilise des trames IP pour envoyer les données, nous pouvons imaginer que l'objet de validation de ces trames est en charge d'analyser cette trame pour en extraire l'information envoyée, avant de la transmettre au relais correspondant, qui peut servir de passerelle pour adapter ces trames aux protocoles du réseau (plus simple) géré par la TCB du niveau supérieur. Nous avons exploité cette idée pour adapter l'architecture à un contexte distribué, comme nous le montrons dans le paragraphe suivant.

V.3.2.2 Adaptation au contexte distribué

La décomposition d'une TCB en plusieurs entités en charge de valider des flux remontant offre un bon moyen pour déployer une architecture de communication à multiples niveaux d'intégrité dans un environnement distribué. En effet, la TCB dans notre étude est en charge d'assurer les communications en contrôlant les flux observés. Cette observation des flux dépend alors de la nature des objets observés. Nous avons vu que, dans le cadre d'applications implantées sur une seule machine physique, différents niveaux sont envisageables pour assurer cette observation des flux, et donc l'implantation d'une partie de la TCB à ce niveau. Si nous changeons l'hypothèse concernant les applications contrôlées, il est très possible d'avoir d'autres choix d'implantation pour la TCB. En effet, prenons l'exemple d'applications avioniques certifiées, s'exécutant sur des calculateurs certifiés et utilisant le réseau avionique. Dans ce cas, le proxy peut prendre la forme d'une passerelle entre les réseaux avioniques et

les autres réseaux, chaque TCB ne gérant qu'un seul réseau lié à un domaine d'application, avec un niveau de confiance homogène.

Dans le cadre de nos travaux, nous nous sommes limités à l'étude d'une TCB assurant une remontée de flux du monde ouvert vers le monde avionique, en nous basant sur la technologie de virtualisation. Néanmoins, nous offrons ici un cadre théorique permettant, à travers l'utilisation de relais, d'adapter l'architecture déployée à des objets distribués, utilisant des technologies d'implantations hétérogènes. Dans la section suivante, nous présenterons plus en détail une telle extension, en nous basant sur des cas d'étude identifiés chez Airbus.

V.4 CONCLUSION

Dans ce chapitre, nous avons :

- défini les hypothèses de fautes dans une architecture redondante basée sur la virtualisation ;
- étudié les contraintes liées à l'indéterminisme dans les versions implantées. L'indéterminisme, s'il n'est pas clairement identifié et étudié, présente une source de dysfonctionnement de toute architecture tolérante aux fautes basée sur la réplication ;
- établi nos hypothèses concernant l'indéterminisme dans le cadre de notre étude ;
- présenté une formalisation du modèle Totel en nous basant sur le modèle de dépendance causale (MDC). À travers cette formalisation, nous avons clairement identifié les entités mises en jeu dans le modèle Totel et spécifié les règles d'interaction enter ces entités pour permettre une communication bidirectionnelle sûre entre des objets à niveaux de criticité disjoints ;
- discuté des différents niveaux d'implantation pour la capture et la validation des flux d'information, en fonction des hypothèses établies sur les objets applicatifs ;
- proposé une décomposition de la TCB pour permettre sa validation et son adaptation à un contexte distribué.

Nous avons ainsi étudié les différents aspects d'une architecture permettant d'assurer l'innocuité et l'immunité des applications les unes par rapport aux autres. Cette architecture de *sécurités* (ArSec) prend en considération :

- l'aspect malveillant de certaines applications,
- le contexte distribué d'exécution,
- l'aspect distribué des applications,
- l'hétérogénéité des niveaux de criticité des applications.

Nous proposons dans le chapitre suivant d'appliquer ces principes architecturaux à deux cas d'étude sur des applications avioniques distribuées.

Chapitre VI

CAS D'ETUDE ET MISE EN OEUVRE

INTRODUCTION

Nous avons présenté dans le chapitre précédent les contraintes liées à l'implantation de mécanismes de contrôle de flux entre composants de différents niveaux de criticité, dans un contexte d'exécution distribuée. Le présent chapitre s'intéresse à la mise en œuvre de tels mécanismes dans le cadre des cas d'étude que nous avons traités.

Dans un premier temps, nous présentons les deux cas d'études envisagés [Laarouchi et al. 2009b] : l'ordinateur de maintenance et l'ordinateur du pilote. Pour chaque cas, nous présentons les services envisagés et détaillons ensuite les flux de données, pour en extraire ceux qui doivent répondre à un contrôle plus strict (notamment la création d'un objet de validation spécifique, l'ajout d'un canal dans la TCB, etc.) avant d'être permis.

Dans un deuxième temps, nous détaillons la mise en œuvre de l'architecture proposée dans le chapitre précédent, à travers l'utilisation de la virtualisation comme support permettant l'implémentation de mécanismes de tolérance aux fautes cohérents avec le modèle Totel. L'utilisation de la redondance et de la virtualisation nous a conduit à résoudre certains problèmes liés d'une part, aux besoins applicatifs propres à chaque cas d'étude, et d'autre part, à l'environnement de développement et d'exécution de ces cas d'étude.

Nous présentons enfin le prototype que nous avons développé pour prendre en charge les opérations de maintenance, tout en assurant des propriétés de sécurité, à la fois aux sens de l'immunité et de l'innocuité. Ce prototype concerne essentiellement le premier cas d'étude, mais il pourrait être adapté pour répondre aussi aux besoins de l'autre cas.

VI.1 1ER CAS D'ETUDE : L'ORDINATEUR DE MAINTENANCE

Les opérations de maintenance ont une influence primordiale sur le temps d'escale d'un avion dans un terminal d'aéroport. En effet, un appareil ne peut décoller si des fonctions essentielles présentent un dysfonctionnement pouvant mettre en danger la sécurité des passagers.

Les opérations de maintenance peuvent être divisées en deux catégories : une inspection globale de la coque de l'appareil, pour détecter d'éventuelles traces d'usure ou d'anomalies sur la structure physique, et une inspection des composants internes de l'appareil.

La première catégorie nécessite une intervention directe de l'équipage ou d'un agent de maintenance pour réaliser les différentes observations, et il est hors de notre étude de proposer une optimisation pour réduire la durée de cette intervention. La seconde catégorie nécessite une intervention au niveau des équipements de bord, qui ne sont pas toujours facilement accessibles par l'opérateur de maintenance. Nous nous intéressons à ce cas de figure et détaillons dans ce qui suit le cadre d'utilisation et les analyses de flux d'information qui en découlent, ainsi que la correspondance avec le modèle Totel. Pour des raisons de confidentialité, nous ne détaillerons pas le rôle fonctionnel de chaque application mise en jeu dans un scénario de maintenance, mais nous traiterons les applications en fonction de leurs niveaux de criticité respectifs.

VI.1.1 Cadre d'utilisation

Il convient de noter que les cas d'étude qui servent de support pour les présents travaux ne sont actuellement implémentés sur aucun appareil en cours d'exploitation. Ainsi, nous proposons de détailler dans un premier temps la tâche de maintenance classique telle qu'elle est réalisée actuellement dans les appareils, et de proposer des scénarios qui pourraient dans l'avenir faire un usage plus étendu des technologies nouvelles.

VI.1.1.1 Tâches de maintenance classique

Afin d'analyser l'état des équipements à bord de l'appareil, des tests sont effectués au moyen de sondes spécifiques, permettant ainsi de recueillir des informations précises quant à l'état de l'équipement testé. Les résultats renvoyés sont présentés à l'opérateur de maintenance, au niveau d'un terminal dédié (c'est notamment le cas pour l'A380) afin d'afficher des détails relatifs à chaque test effectué : nature de l'équipement, nature du test, nature de la faute identifiée, code de la faute, code de la procédure de traitement de la faute, etc.

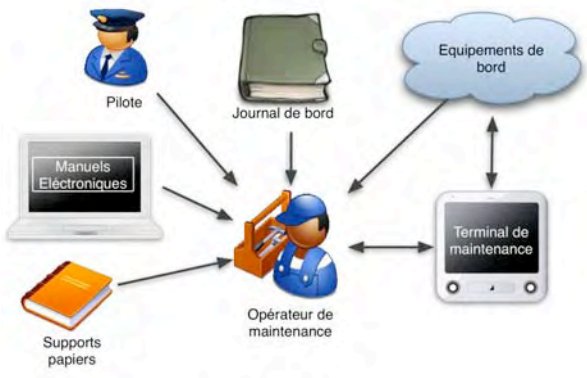


Figure VI.1 Scénario classique d'opération de maintenance

Ces analyses peuvent être guidées par les informations recueillies durant le vol. En effet, durant le vol, les équipements envoient périodiquement des informations relatives à leur état interne de fonctionnement. Le pilote est alerté (sous forme de signal sonore ou avertissement visuel) pour les défaillances les plus importantes. Tout disfonctionnement est enregistré dans un journal de bord (cf. figure VI.1). Le journal de bord est ainsi analysé durant la phase de maintenance, afin de classer les incidents éventuels selon leur gravité, et agir en conséquence.

Actuellement, durant les opérations de maintenance, l'opérateur analyse les différents rapports (journal de bord, rapport d'équipements, etc.) en s'appuyant sur des manuels de maintenance. Ces manuels sont généralement sous forme de documents papiers, que l'agent de maintenance doit avoir avec lui lors de sa mission. Ces documents sont progressivement mis sous format électronique, permettant ainsi à l'agent de les consulter directement sur un ordinateur portable. Cependant, cet ordinateur n'est pas connecté au réseau de l'avion. Il sert uniquement de support électronique de consultation des manuels, comme nous le montre la figure VI.1.

Après avoir affiché sur son ordinateur portable la procédure de maintenance indiquée par ces manuels, l'opérateur lance la batterie de tests qui est demandée dans cette procédure, par le biais d'un équipement dédié à bord de l'avion : le terminal de maintenance. La présence d'un opérateur humain durant la phase de maintenance est primordiale pour la prise de certaines décisions. Prenons l'exemple d'un calculateur signalé comme défaillant. On peut soit empêcher le décollage de l'avion tant que le remplacement n'est pas effectué, soit, grâce aux redondances, permettre le décollage sous certaines conditions, par exemple que ce calculateur puisse être remplacé dans un futur proche. Ce genre de décision nécessite l'intervention d'un opérateur humain, qui en fonction des erreurs signalées, doit choisir l'action à accomplir afin d'y répondre, tout en prenant en compte des contraintes extérieures notamment des contraintes économiques. Nous nous plaçons dans ce cadre opérationnel et proposons d'optimiser l'opération de maintenance en la mettant en œuvre sur un support électronique unique, tout en permettant à l'agent de maintenance de contrôler le bon déroulement de ces opérations, en se basant sur son expérience, et sur des tests intermédiaires qu'il pourra lancer sur les équipements de l'avion. Dans le paragraphe suivant, nous proposons de détailler le scénario fonctionnel d'une telle opération de maintenance.

VI.1.1.2 Opérations de maintenance avec support électronique

Dans le cadre des opérations classiques de maintenance, l'opérateur est contraint de servir d'interface de communication entre ses manuels électroniques (contenant en particulier les procédures à suivre pour tester un équipement) et le terminal de maintenance sur lequel il effectue les actions dictées par ces procédures. Ce mode de fonctionnement permet d'éviter la connexion directe de l'ordinateur portable à l'avion, annulant ainsi tout risque de remontée de flux d'information depuis ce portable vers des équipements critiques. Dans un tel cas, nous pouvons dire que l'opérateur humain joue le rôle d'un « objet de validation » du modèle Totel, avec un niveau d'intégrité équivalent à celui du terminal de maintenance. En effet, c'est l'opérateur qui choisit de valider le suivi d'une procédure de maintenance depuis son manuel, en l'appliquant sur le terminal de maintenance. Dans la figure VI.2, nous présentons une autre approche de la maintenance.

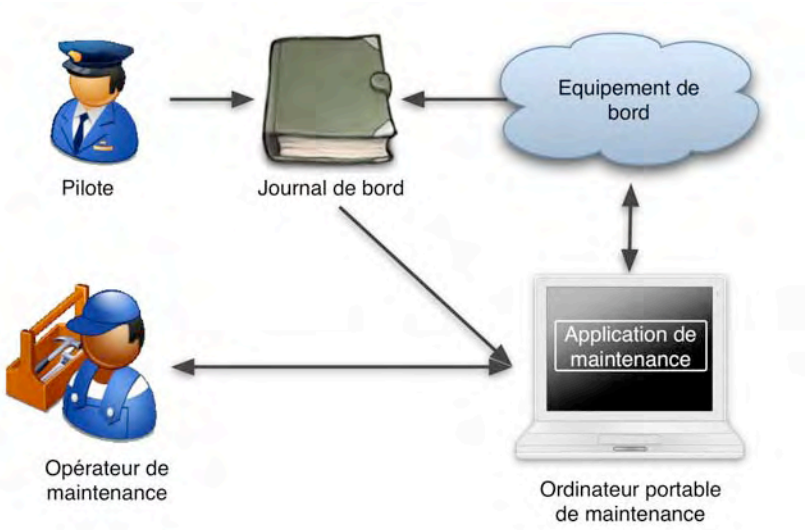


Figure VI.2 Maintenance avec support électronique

Dans cette approche, les opérations de maintenance bénéficient d'une meilleure intégration des moyens électroniques. En effet, les manuels de maintenance (aujourd'hui sous forme électronique ou papier) décrivant les procédures à accomplir sont pré-enregistrés sur l'ordinateur de maintenance. Lorsqu'il est connecté à l'avion, l'ordinateur de maintenance récupère les rapports d'anomalies provenant du journal de bord, les classifie et indique pour chaque anomalie la nature de la faute qui lui correspond, et les actions à réaliser pour la traiter. Dans chaque procédure, des liens vers des applications de maintenance spécifiques sont fournis, permettant à l'opérateur de lancer directement les tests et opérations nécessaires pour réaliser ces procédures. Remarquons que dans ce cas, l'opérateur humain n'a plus à définir chacune des actions exigées par la procédure, mais il se contente de suivre la procédure, de vérifier la validité des informations qui s'affichent et de veiller au bon déroulement des actions en cours. Dans certains cas, il peut être demandé à l'opérateur de se déplacer dans l'appareil et de réaliser lui-même certaines actions sur les équipements. Par exemple, dans certaines procédures où les volets de l'avion sont actionnés, et il est demandé à l'opérateur de se déplacer pour vérifier que les volets ont été bel et bien actionnés, puis de valider cela au niveau de l'ordinateur de maintenance. Ceci est facilité si la connexion entre l'ordinateur portable de maintenance et l'avion est une liaison sans fil.

Une telle assistance à l'opérateur dans les opérations de maintenance conduit à un gain de temps considérable, rendant ainsi l'appareil opérationnel plus rapidement. Cette propriété fait l'objet d'une forte demande et est fort appréciée par les compagnies aériennes qui cherchent à réduire au maximum le temps d'escale. En effet, dans la plupart des aéroports, la compagnie doit payer des frais supplémentaires en cas de dépassement de la plage horaire qui lui est assignée. En plus, l'image de la compagnie est mise en jeu lors d'un dépassement de ces plages, puisque les retards induits conduisent à une mauvaise appréciation de la part des clients.

La réaffectation du rôle de l'opérateur humain qui en résulte, en l'allégeant des tâches répétitives et peu valorisantes, permet également de limiter les erreurs pouvant survenir suite à une faute d'interaction. En effet, la fréquence de telles fautes peut être augmentée par le contexte du déroulement des opérations de maintenance, caractérisé par de fortes pressions de la part des compagnies pour assurer la continuité du service de l'appareil contrôlé.

Cependant, il est clair que l'opérateur ne joue plus le rôle d'un objet de validation entre les manuels de maintenance et l'avion, comme dans les opérations de maintenance classique. Il faut donc s'assurer que la communication entre l'ordinateur de maintenance et les équipements de bord n'introduit pas de risque d'altération du fonctionnement des entités critiques à bord. Pour cela, nous analysons dans ce qui suit les flux fonctionnels de données, pour appliquer ensuite le modèle Totel sur les modules en question.

VI.1.2 Analyse des flux de données

L'utilisation d'un ordinateur de maintenance en connexion directe avec l'avion pose la problématique suivante : l'ordinateur, de type portable, est fourni par la compagnie et peut être considéré comme un ordinateur personnel de l'agent de maintenance. Il peut donc être déplacé et connecté à des réseaux du monde ouvert, et potentiellement être connecté à l'Internet, par exemple pour du courrier électronique ou de la navigation sur le Web. Cette hypothèse de fonctionnement expose le système d'exploitation et les applications qui y sont installées à des attaques (des vers, des chevaux de Troie, etc.), diminuant de la sorte le degré de confiance que l'on peut accorder aux sorties de cet ordinateur. Pourtant, cet ordinateur de maintenance doit pouvoir être connecté à l'avion, afin de réaliser des tests et éventuellement procéder à des remplacements d'entités logicielles à bord.

Comme nous l'avons mentionné dans le chapitre III, les niveaux de criticité dépendent uniquement de la nature de la tâche accomplie. C'est aux concepteurs et développeurs de cette tâche sur un module spécifique de prouver que le niveau de confiance obtenu pour ce module est au moins égal à celui de la criticité de la tâche.

Dans le cadre avionique où nous nous plaçons, on distingue quatre niveaux de criticité (cf. figure VI.3) :

- Niveau 4 : niveau des systèmes de contrôle de l'appareil (*Flight Control* en anglais) : ce niveau est le plus critique et contient les applications en charge de piloter l'avion (applications CDVE : Commandes De Vol Électriques).
- Niveau 3 : niveau des systèmes d'opération et de maintenance de l'appareil (*Aircraft Control*)
- Niveau 2 : niveau du système d'information de la compagnie (*Aircraft Information System*)
- Niveau 1 : niveau des moyens mobiles ou encore appelé le monde ouvert, dans lequel nous plaçons l'ordinateur de maintenance. Ce niveau correspond aux tâches non critiques, ou aux modules d'exécution ayant le plus bas niveau de confiance (cf. III.2.).

Dans chaque niveau, une ou plusieurs tâches sont implantées. Ces tâches peuvent communiquer entre elles, selon le schéma présenté dans la figure VI.3. Les flèches représentent les flux logiques qui peuvent exister entre différentes tâches, comme c'est le cas dans la représentation du modèle Totel (cf. chapitre III).

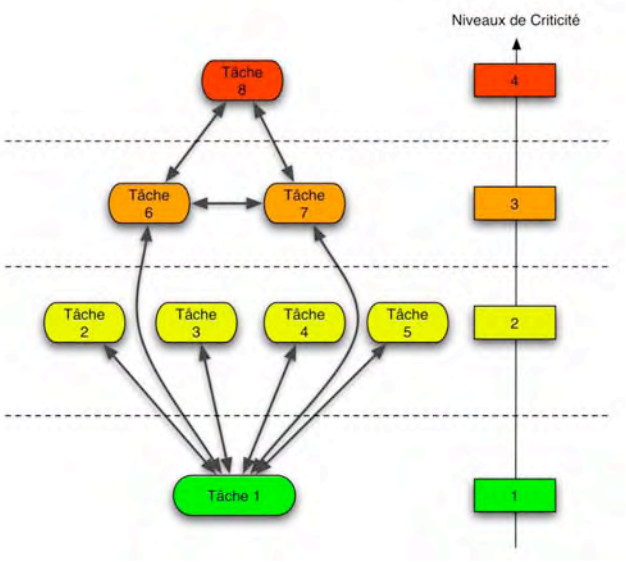


Figure VI.3 Les tâches de maintenance et leurs niveaux de criticité

Ainsi, la tâche 1 (de niveau de criticité 1), qui correspond ici à l'application installée sur l'ordinateur de maintenance, envoie des informations vers des tâches de niveau 2 (les tâches 2, 3, 4 et 5) et également vers des tâches de niveau 3 (tâches 6 et 7). Ces deux dernières communiquent avec une tâche de la plus haute criticité (tâche 8).

Il est clair qu'un contrôle de flux s'impose afin de réaliser de telles communications tout en évitant que la tâche 1 n'altère le fonctionnement des autres tâches. En effet, à défaut de tels mécanismes de protection, une attaque réalisée au niveau de l'ordinateur de maintenance serait susceptible de corrompre une tâche du domaine des systèmes de contrôle de l'appareil à travers une corruption intermédiaire du domaine des systèmes d'opération et de maintenance de l'appareil.

Dans la section suivante, nous proposons de faire une correspondance entre le modèle Totel et le cas d'étude des opérations de maintenance. Cette correspondance peut être directement établie par un analyseur statique des différents flux fonctionnels, sur la base de ce que nous avons présenté dans le chapitre 5. Nous présentons dans ce qui suit le résultat d'une telle analyse.

VI.1.3 Application du modèle Totel à l'ordinateur de maintenance

Nous avons présenté dans la figure VI.3 les flux de données fonctionnels entre les différentes tâches de maintenance. Afin de permettre une remontée de flux, il est nécessaire d'implanter des objets de validation pour chaque objet recevant un flux de données d'un niveau de criticité moindre. Ainsi, nous obtenons la représentation dans le modèle Totel décrite par la figure IV.4.

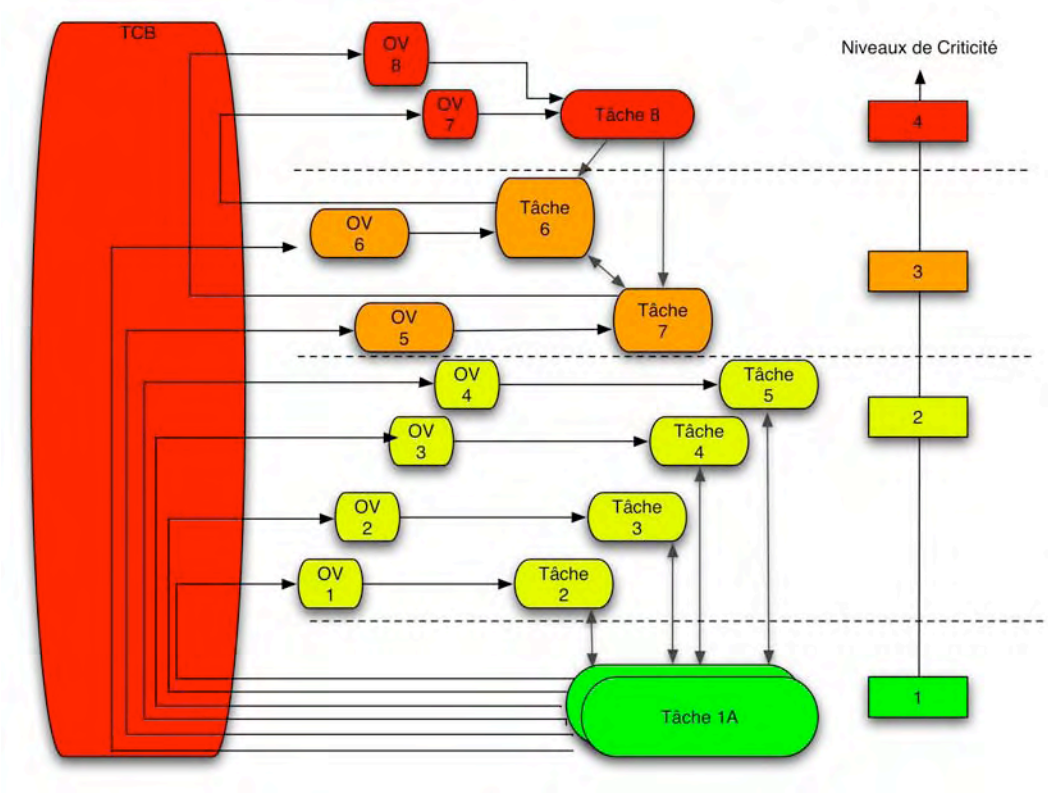


Figure VI.4 Contrôle des flux ascendants des opérations de maintenance

Remarquons que, dans la figure IV.4, la TCB est en charge de canaliser les flux remontants de tous les niveaux de criticité vers des tâches spécifiques. Dans ce cas, la TCB prend en charge la gestion de flux en provenance du monde ouvert, ce qui, potentiellement, induit la prise en charge de protocoles et mécanismes de communication complexes et donc coûteux à valider pour obtenir un niveau de confiance élevé. En même temps, la TCB communique avec des tâches du du domaine des systèmes de contrôle de l'appareil, il faut donc s'assurer que cette TCB a le même niveau d'intégrité, ce qui est fort difficile à réaliser si elle intègre des mécanismes de communication complexes. Dans ce contexte, il peut être préférable d'utiliser le principe d'éclatement de la TCB que nous avons présenté dans la section V.2. Dans ce cas, l'éclatement a pour rôle de réduire la portée de chaque élément de TCB et d'implanter chacun de ces éléments sur des composants physiques ayant le niveau d'intégrité requis pour cette entité.

Notons également la présence de deux instances de la tâche 1. Ceci est dû au fait que l'on utilise des objets de validation qui se basent sur la redondance. En effet, afin de valider une donnée, on peut se baser sur deux types de mécanismes. Dans le premier, on effectue une validation sur une donnée à source unique, auquel cas, nous parlons de contrôle de vraisemblance. Ce genre d'objet de validation est implanté par l'*OV7* de la figure VI.4. Le second type d'OV nécessite plusieurs sources de données redondantes, afin d'appliquer des techniques de tolérance aux fautes par comparaison sur ces entrées, ce qui est le cas de la tâche 1. Notons aussi que pour que la comparaison soit efficace, les deux instances doivent se comporter différemment en cas de faute, ce qui nécessite une certaine diversification.

La figure VI.4 ne met pas en évidence l'aspect distribué de l'implantation finale. Cependant, la tâche 1 est installée sur l'ordinateur de maintenance, c'est-à-dire sur une machine différente de celle qui exécute les autres tâches. Les tâches de niveau supérieur ou égal à 2 sont installées à bord, et pour certaines, elles sont assurées par une même machine physique. Les

flux représentés sont donc des flux fonctionnels et ne prennent pas en compte la distribution géographique des différentes tâches.

Notons également que l'opérateur de maintenance n'est plus en charge de valider directement des données à l'entrée des équipements de bord, son rôle se concentre sur la supervision des procédures de maintenance. Il faut donc s'assurer que toutes les sorties produites par l'ordinateur de maintenance soient validées, avant d'être envoyées à un niveau supérieur. Par sorties, nous désignons aussi bien les communications vers le réseau de l'avion, que les affichages sur l'écran de l'ordinateur de maintenance. En effet, alors que les premières sorties concernent l'envoi direct des données à des tâches plus critiques, les sorties graphiques ont pour but de tenir informé l'agent de maintenance du bon déroulement des opérations et de lui afficher les choix pour lesquels il doit prendre une décision. Ainsi, par extension du modèle, on peut considérer que l'agent de maintenance est une entité possédant un haut niveau de confiance et que par conséquent, toute information destinée à cet agent doit être vérifiée et validée avant qu'elle ne lui soit communiquée.

Ainsi, il est important de vérifier aussi bien les sorties vers l'avion que les sorties graphiques de l'interface homme-machine (IHM). Le prototype que nous avons développé dans le cadre de ces travaux se base sur la validation de ces deux types de sorties avant de les envoyer au réseau avionique ou de les afficher. Nous présenterons dans ce qui suit le détail d'une telle implantation, et décrirons un scénario d'attaque plausible qui fait intervenir l'affichage de l'IHM et également les données envoyées à l'avion.

VI.2 2EME CAS D'ETUDE : L'ORDINATEUR DU PILOTE

Alors que le premier cas d'étude présenté concerne l'utilisation d'un ordinateur portable de maintenance qui constitue une évolution d'un système existant déjà sur l'A380 avec un terminal de maintenance fixé dans le *cockpit*, le second cas d'étude que nous traitons présente un fonctionnement qui n'est implémenté sur aucun avion existant et concerne de ce fait, les futures générations d'avions.

VI.2.1 Cadre d'utilisation

L'ordinateur du pilote est utilisé pour calculer certains paramètres définissant le profil de décollage. Nous présentons dans un premier temps le scénario de calcul de ces paramètres puis nous proposons la nouvelle utilisation envisagée dans le cadre de ce cas d'étude.

VI.2.1.1 Calcul classique du profil de décollage

Le profil de décollage correspond au calcul de certains paramètres nécessaires au bon déroulement du décollage (par exemple : vitesse de décollage, vitesse limite au-delà de laquelle l'abandon du décollage n'est plus possible, etc.). Ces vitesses dépendent de deux types de données :

- Les données aéroportuaires : ces données dépendent de l'aéroport dans lequel se trouve l'appareil. Les données en question peuvent être :
  - fixes : comme l'orientation et la longueur de la piste de décollage.
  - variables : comme l'état de la piste ou les conditions météorologiques lors du décollage.
- Les données de l'avion : ces données sont essentiellement fournies par les compagnies aériennes qui exploitent l'appareil en question. Elles peuvent être :
  - fixes : comme le type de l'avion
  - variables : comme la masse totale ou le centrage (la répartition des masses dans l'avion)

Ces différentes données sont recueillies par le pilote comme le montre la figure VI.5.

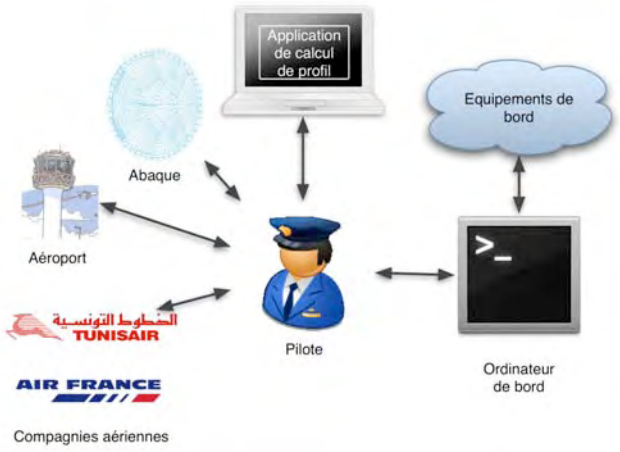


Figure VI.5 Scénario de calcul du profil de décollage

L'aéroport envoie au pilote les différentes données aéroportuaires, sous format non électronique en général. De même, la compagnie envoie les données propres à l'avion au pilote, soit par un service aéroportuaire spécifique, soit d'une manière directe. Une fois ces données récupérées, le pilote procède au calcul du profil de décollage dans une phase préparatoire au vol. Traditionnellement, ce calcul se fait à l'aide d'un abaque qui permet de donner manuellement les paramètres du profil. Depuis peu de temps, le pilote peut être assisté dans cette tâche par une application logicielle de calcul de profil de décollage installée sur un ordinateur portable (cf. figure VI.5). Le pilote entre alors manuellement les données récupérées de l'aéroport et de la compagnie aérienne dans l'application qui lui donne en sortie les paramètres de décollage. Ces paramètres sont ensuite récupérés par le pilote et entrés manuellement dans l'ordinateur de commande de vol.

Comme pour l'opérateur de maintenance, le pilote joue ici le rôle d'un « objet de validation » humain en charge de remonter le niveau de confiance que l'on peut avoir dans les données calculées par les applications d'assistance fournies par les compagnies (ou les données calculées à partir de l'abaque). Nous proposons dans le paragraphe suivant de détailler une nouvelle utilisation de l'ordinateur portable du pilote.

VI.2.1.2 Nouvelle utilisation de l'ordinateur du pilote

La recopie, par le pilote, du profil du décollage depuis l'application de calcul installée sur l'ordinateur du pilote (ou de l'abaque) vers l'ordinateur de bord est source de fautes d'interaction, comme c'est le cas pour l'ordinateur de maintenance. Nous proposons que la tâche de transfert du profil vers l'ordinateur de bord soit réalisée électroniquement, comme le montre la figure VI.6.

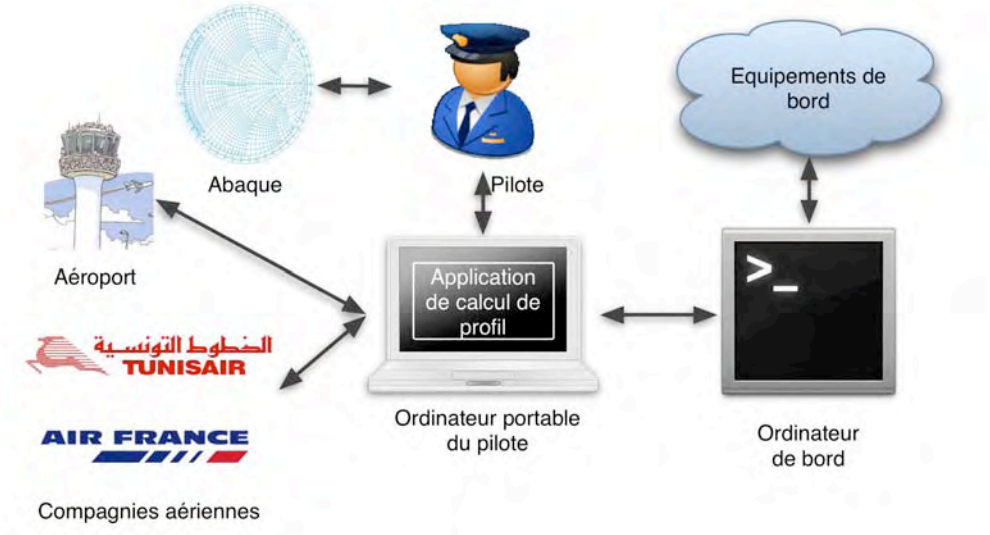


Figure VI.6 Nouvelle utilisation de l'ordinateur du pilote

L'ordinateur du pilote contient l'application de calcul de profil de décollage. Plusieurs moyens de connexion entre cet ordinateur et le sol sont envisageables pour récupérer les paramètres de l'aéroport et des compagnies : une connexion WiFi avec authentification, une connexion par fil direct à des ports Ethernet spécifiques fournis par l'aéroport, un périphérique de stockage externe (type USB par exemple) que le pilote récupère auprès des autorités aéroportuaires et qu'il branche sur son ordinateur, etc.

Une fois les données récupérées par l'application, le profil est calculé, puis le résultat est transmis aux applications de contrôle de l'appareil installées sur l'ordinateur de bord. L'application, quant à elle, n'est pas développée par le constructeur de l'avion, qui ne contrôle ni son processus de développement ni sa validation. C'est pour cette raison que nous considérons qu'elle est du niveau d'intégrité le plus bas (celui du monde ouvert).

Ceci est renforcé par le fait que l'ordinateur du pilote est un ordinateur portable que le pilote peut garder avec lui lors de ses escales, et en disposer pour une utilisation personnelle (consulter ses courriels, naviguer sur Internet, etc…). Il faut donc prendre en considération cet aspect multi usage de la machine dans le cadre de l'architecture finale que nous proposons pour ce cas d'étude.

L'utilisation de l'abaque présente dans la figure VI.6 sert à une validation supplémentaire éventuelle réalisée manuellement par le pilote dans le cas où il aimerait vérifier que les résultats transmis à l'avion correspondent bien à une fourchette de valeurs prédites par le calcul manuel utilisant l'abaque. Cette utilisation, même si elle n'est pas représentée dans le cadre du modèle Totel, peut faire partie d'un processus de validation supplémentaire à intégrer au niveau de l'objet de validation (comme nous le montrons dans le chapitre V). Dans le paragraphe suivant, nous proposons d'analyser les flux de données mis en jeu lors de telles communications inter niveaux de criticité.

VI.2.2 Analyse des flux de données

Dans la figure VI.7, nous considérons les différents flux mis en jeu dans le scénario proposé de calcul du profil de décollage par l'ordinateur du pilote.

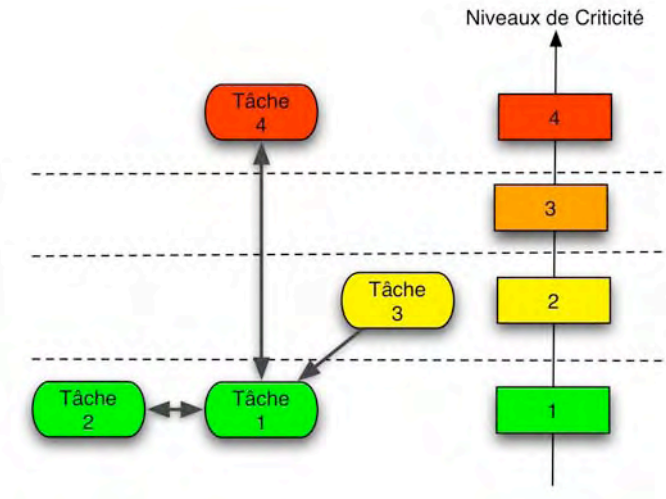


Figure VI.7 Flux de données lors du calcul du profil de décollage

La tâche 1 (celle qui calcule le profil de décollage) est en charge de récupérer des données des tâches 2 et 3. La tâche 2 est du niveau 1 (le même que celui de la tâche 1), le flux entre ces deux tâches ne nécessite pas de contrôle particulier. Dans le cas réel, nous pouvons supposer que la tâche 2 correspond à une application développée par la compagnie aérienne afin de fournir à la tâche 1 les données spécifiques de l'avion (cf. paragraphe IV.1.3.2), et donc que cette tâche n'a pas été spécialement validée pour correspondre à un niveau plus élevé d'intégrité. La tâche 3, par contre, est développée au niveau 2. Dans le cas réel, on peut faire correspondre cette tâche à une application de l'aéroport en charge de récupérer les paramètres aéroportuaires puis de les communiquer aux ordinateurs des pilotes. Ce scénario est réaliste,

dans le sens où on peut supposer que certains logiciels utilisés par l'aéroport ont subi des contrôles plus stricts que ceux utilisés par la compagnie. Ceci étant dit, la communication entre les tâches 3 et 1 est uniquement descendante (de la tâche 3 vers la tâche 1), ce qui non plus, ne nécessite pas de contrôle particulier.

La communication entre les tâches 1 et 4, par contre, nécessite des contrôles supplémentaires. En effet, il s'agit d'une communication directe depuis le monde ouvert et jusqu'au plus haut niveau de criticité (système de commandes de vol). Une telle communication doit être strictement contrôlée et il faut donc s'assurer que les objets de validation que l'on met en place, ainsi que la TCB à implanter, sont bien en mesure d'effectuer le contrôle de données provenant d'un niveau de criticité 1 à un niveau de criticité 4. Ce *saut* de niveaux est difficile à implanter dans le cas réel, comme nous allons le voir dans la section suivante, où nous proposons une correspondance entre le modèle Totel et les flux identifiés dans les deux cas d'étude.

VI.2.3 Application du modèle Totel à l'ordinateur du pilote

Comme nous l'avons indiqué dans la figure VI.7, la tâche 1 est celle qui implante l'application du calcul de profil qui sera utilisée par le pilote. Afin d'augmenter le niveau de confiance que l'on a dans le résultat de cette tâche, on la duplique comme le montre la figure VI.8.

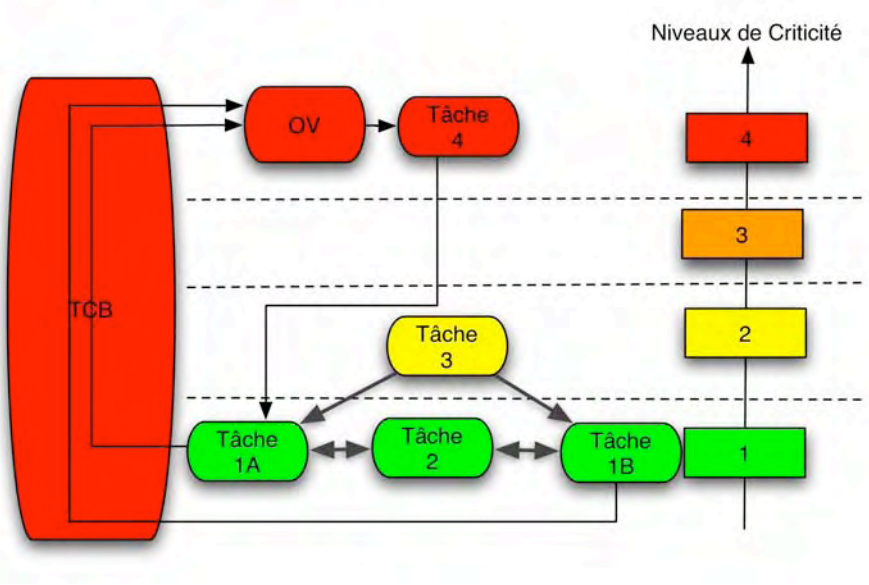


Figure VI.8 Contrôle des flux ascendants de l'ordinateur du pilote

La redondance que nous proposons est une diversification logicelle. En effet, assurer une redondance matérielle et demander au pilote de transporter plusieurs ordinateurs portables s'avère encombrant et peu opérationnel. Ainsi, deux instances d'application de calculs de profil de décollage sont installées sur une seule machine physique. Nous détaillerons dans la section suivante le type de diversification que l'on vise dans le cadre de ces travaux. (diversification de développement, de compilateur, de plateforme d'exécution, etc.).

Dans ce cas d'étude, nous retrouvons la même problématique que dans le cas précédent, à savoir faire suivre les communications des tâches critiques aux différentes instances de l'application diversifiée. Remarquons que nous considérons que la tâche 2 a le même niveau de criticité que l'application de calcul de profil. Comme nous l'avons précédemment indiqué

(cf. section VI.2.2), cette tâche pourrait correspondre aux applications développées par les compagnies pour envoyer les données spécifiques de l'avion à l'application de calcul du profil de décollage. On peut imaginer que cette dernière application implante un mécanisme de contrôle fonctionnel des données qui proviennent de la compagnie. Ce contrôle (fondé, par exemple, sur une base de valeurs précédemment enregistrée) permet de vérifier la validité de ces données avant de les utiliser dans le calcul du profil de décollage.

Dans ce cas, nous avons un *saut* de niveau important entre les sorties des instances 1A et 1B d'une part, et l'entrée de l'objet de validation OV d'autre part. Pour les raisons que nous avons précédemment mentionnées, il est plus judicieux d'éclater la TCB en plusieurs parties et d'assurer une implantation par niveaux de criticité basée sur des *proxies*, comme nous l'avons montré dans le cas d'une implantation dans un environnement distribué.

Ce type de solution peut être guidé par les architectures de réseau déjà existantes dans l'avion. Ainsi, un élément de la TCB peut prendre la forme d'une passerelle entre des réseaux existant (IP sur Ethernet au niveau 1, AFDX au niveau 2, bus, etc. [Moir et Seabridge 2003]) et assurer l'envoi d'informations d'une tâche spécifique utilisant un protocole donné vers une tâche plus critique utilisant un protocole différent (cf. figure VI.9).

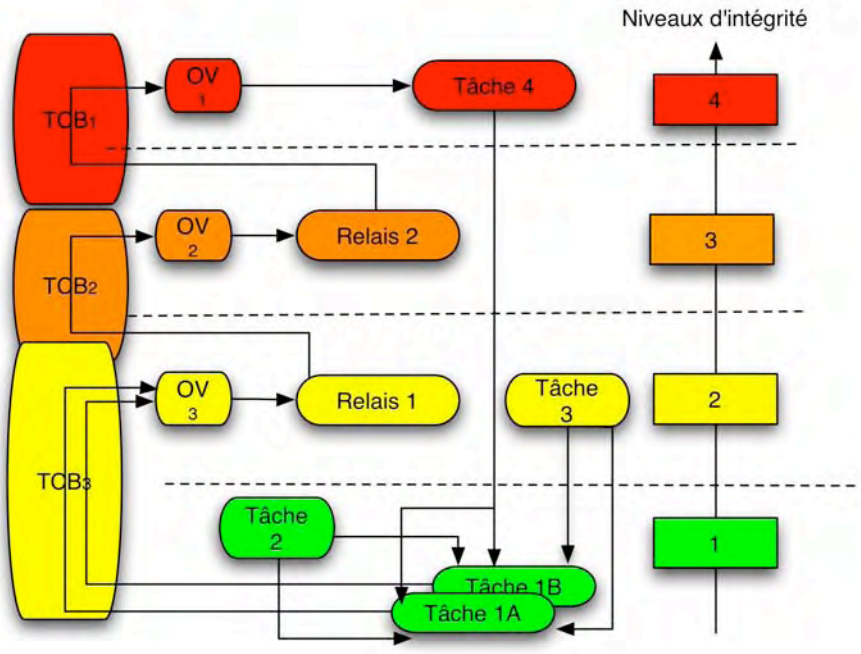


Figure VI.9 Utilisation de relais pour le contrôle des flux ascendants

Dans le cadre de cette étude, cependant, nous nous limitons à définir la communication de l'ordinateur du pilote (du monde ouvert, de niveau 1) vers l'avion (au niveau du système d'informations de l'avion, de niveau 2).

Nous remarquons qu'au niveau des flux opérationnels, les tâches de maintenance et de calcul de profil de décollage sont similaires. Dans la section suivante, nous détaillons l'implantation du prototype que nous avons développé pour augmenter le niveau de confiance de l'application de maintenance. Concernant la tâche de calcul de profil de l'ordinateur de pilote, nous proposons donc une solution technique similaire à celle que nous avons développée.

VI.3 CONSIDERATIONS DE MISE EN ŒUVRE

Nous avons présenté deux cas d'étude dans lesquels des flux fonctionnels ascendants sont requis par des applications critiques. Dans cette section, nous nous intéressons à l'implantation d'un prototype permettant de tels flux, tout en préservant l'intégrité des composants critiques.

L'implémentation du prototype dépend de la nature des applications qui génèrent ces flux. En effet, il est nécessaire de capturer ces flux afin de les valider avant de les envoyer à l'application critique qui en a besoin. Cette capture dépend de la nature des hypothèses que nous faisons vis-à-vis du code et de l'exécutif de l'application en question, ainsi que de l'architecture basée sur la virtualisation que nous avons présentée dans la section V.1. Ainsi, nous présentons dans un premier temps les différents niveaux de capture que nous considérons puis nous discuterons des hypothèses de travail que nous avons établies. Dans ce qui suit, nous détaillons ces besoins et essayons d'en déduire des recommandations pour le développement des futures applications dans le Monde Ouvert.

VI.3.1 Capture des flux ascendants des applications redondantes en vue de leur validation

Sur l'A380, une application de maintenance est installée sur une machine fixée dans le poste de pilotage. Cette machine n'est pas transportable, et n'a aucune connexion avec le monde ouvert. Nous nous basons dans un premier temps sur cette application existante, afin d'établir les contraintes de déploiement de notre architecture.

L'application existante est développée en utilisant le langage Java, l'interface homme-machine est implantée en utilisant des objets graphiques fournis par la bibliothèque *Swing* de la machine virtuelle Java, cette machine virtuelle s'exécutant sur Windows XP. Le système d'exploitation, aussi bien que l'application sont téléchargés depuis l'avion au moment du lancement de l'ordinateur de maintenance.

VI.3.1.1 Implémentation de la diversification par virtualisation

Afin de rendre plus sûre l'exécution de cette application A, nous la dupliquons en l'implantant sur un autre système d'exploitation (l'application A est représentée par les instances A_{1A} *et* A_{1B} dans la figure VI.10). Pour ce faire, nous adoptons l'architecture décrite dans le chapitre V, en utilisant la virtualisation. Dans le cadre de ce prototype, nous avons utilisé Xen comme hyperviseur permettant d'assurer l'isolation et l'ordonnancement des différentes machines virtuelles.

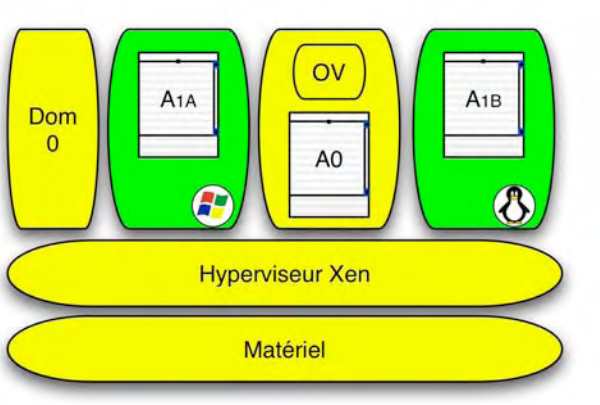


Figure VI.10 La diversification en utilisant la virtualisation

Remarquons que nous faisons confiance aux couches inférieures aux systèmes d'exploitation, à savoir l'hyperviseur et le matériel, et également au domaine 0 de Xen, chargé du contrôle de l'hyperviseur, et à une quatrième machine virtuelle (comme nous l'avons précédemment indiqué dans les hypothèses de travail de la section V.1). Cette dernière exécutera un système d'exploitation certifié au moins au niveau Système d'Information de l'Avion. C'est cette hypothèse qui nous permet d'avoir ensuite confiance dans le résultat de la validation effectuée par l'objet de validation (*OV*).

Dans l'exemple de la figure VI.10, l'instance A_{1A} est installée sur un système Windows, et l'instance A_{1B} est installée sur un système d'exploitation de type Linux. Le résultat est fourni par une autre application (*A0*) qui est en charge de contrôler les sorties des deux instances. Chacune des instances A_{1A} et A_{1B} est exécutée sur une JVM propre au système d'exploitation de la machine virtuelle sur lesquelles elles sont installées, et toutes deux utilisent les bibliothèques *Swing* pour gérer les objets graphiques. Rappelons que les sorties des instances redondantes sont les sorties au niveau de l'utilisateur (écran, imprimante), et au niveau du réseau (communication avec l'avion).

Afin de valider les flux de données issus des deux instances de l'application par un objet de validation, nous devons nous poser les deux questions suivantes :

- À quel niveau peut-on capturer ces flux ?
- Est-ce que le niveau de capture choisi permet d'assurer une comparaison fiable des deux flux ?

La première question est d'ordre pratique et est liée à l'architecture de diversification et à la nature de l'application diversifiée. La seconde concerne la comparabilité des flux capturés dans le sens où la comparaison est suffisamment précise pour détecter les erreurs tout en minimisant le risque de fausses alertes. Cet aspect définit l'efficacité de la détection d'erreurs que nous implantons dans notre prototype. Nous avons envisagé diverses interfaces pour réaliser la capture (cf. figure VI.11) que nous discuterons et comparons dans ce qui suit.

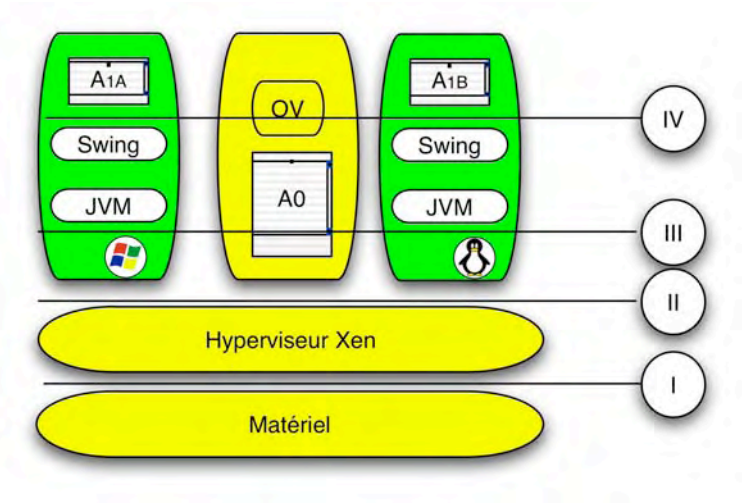


Figure VI.11 Différents niveaux d'interfaces possibles de capture de flux

VI.3.1.2 Capture au niveau du matériel

Cette capture se base sur les flux reçus au niveau de l'interfaçage des périphériques réels de sortie, à savoir le pilote des périphériques (dans notre cas, on s'intéresse au pilote de la carte graphique et au pilote de la carte réseau). Ce niveau, désigné par (I) dans la figure VI.11, est situé à l'interface entre l'hyperviseur Xen et le matériel. En effet, toute sortie réelle des machines virtuelles passe nécessairement par l'hyperviseur avant d'être envoyée au matériel.

Dans ce cas, les flux que nous comparons prennent la forme de séquences de bits en entrée des registres des pilotes de périphériques. À ce niveau, la comparaison serait du type bit-à-bit entre les deux flux. Cependant, nous n'avons aucune sémantique significative permettant d'interpréter les données qui sont comparées. En effet, prenons l'exemple d'un affichage graphique d'une chaîne de caractères. Ce qui est envoyé à la carte graphique correspond à une séquence de pixels en charge de représenter chaque caractère de la chaîne à afficher. Il suffit d'un bit différent dans les deux flux pour que la comparaison échoue, même si un pixel diffèrent n'induit pas une différence visible dans le caractère final affiché.

Ainsi, se placer à ce niveau pour comparer les flux risque de conduire à de fausses alertes inutiles car correspondant à des différences minimes dans les bits comparés, sans que ce soit forcément significatif au niveau de l'affichage final. Pour pallier cela, nous avons envisagé d'utiliser des logiciels de reconnaissance de caractères. Ces logiciels seraient en charge de convertir les registres de la carte graphique en caractères, ce qui faciliterait la comparaison des deux flux. Cependant, cette solution doit répondre aux exigences suivantes :

- Comment identifier les flux relatifs à chaque machine virtuelle ?
- Est-ce que de telles applications sont validables aux moins au niveau du système d'information de l'avion (le même niveau que l'hyperviseur)
- Quel est le degré de précision de ces applications dans la reconnaissance des caractères qui s'affichent à l'écran ?

Pour répondre à la première question, on peut imaginer une identification des flux liées à la position des pixels sur l'écran. Les deux autres questions portent sur la nature des applications que nous considérons. Mais de toute façon, quelque soit leur nature, cette solution ne permet pas de résoudre le problème de comparaison des sorties non textuelles (par exemple

directement graphiques : flèches, cases, etc.). Compte tenu de ces contraintes, nous ne préconisons pas ce niveau pour la capture.

VI.3.1.3 Capture au niveau de l'hyperviseur

À ce niveau, nous comparons les sorties des machines virtuelles vers l'hyperviseur (cf. (II) de la figure VI.11). La comparaison dépend clairement du type d'hyperviseur ainsi que de la nature de la virtualisation. Nous nous plaçons dans le cadre d'une virtualisation complète (dans laquelle le système virtualisé ne coopère pas avec l'hyperviseur, contrairement à la paravirtualisation). Le système envoie des séquences de bits aux périphériques de sortie, et ces séquences sont captées par l'hyperviseur (à travers les périphériques virtuels) avant d'être transférées aux périphériques réels. Intéressons-nous aux deux types de sorties que nous traitons dans nos cas d'étude : le réseau vers l'avion et l'affichage graphique.

Concernant le réseau vers l'avion, nous avons considéré dans le cadre de notre démonstrateur une communication basée sur IP sur Ethernet. L'hyperviseur Xen implante des réseaux virtuels que l'on peut facilement configurer afin de capturer les flux IP, puis les comparer au niveau d'un objet de validation. À ce niveau, la comparaison des trames une à une peut conduire à de fausses alertes qui pénaliseraient le processus de validation. En effet, chaque trame Ethernet et générée par un système d'exploitation différent, ce qui implique que les numéros de séquence des trames peuvent changer d'un système à un autre. Il est donc nécessaire de traiter ces flux afin d'en extraire les données applicatives que nous souhaitons comparer.

Concernant l'affichage graphique, la capture des flux consiste à récupérer les flux graphiques tels qu'ils sont gérés par l'hyperviseur Xen avant de les comparer. Xen utilise la technique *VNC* (*Virtual Network Computing*) [Richardson et al. 1998] qui est équivalente à une connexion *ssh*, mais en mode graphique. Ainsi un serveur VNC est installé sur chaque machine virtuelle, puis les flux vers le serveur VNC sont transférés au niveau d'un client VNC installé au niveau de l'hyperviseur, qui s'occupe de l'affichage. Nous avons envisagé de capturer ces flux VNC, et de les comparer. Les flux VNC utilisent le protocole RFB (*Remote Frame Buffer*) [Richardson 2007] qui consiste à échanger des séquences de pixels entre le client et le serveur, représentant l'affichage graphique du serveur. Dans certaines optimisations, seules les différences avec les trames RFB précédentes sont calculées, afin de minimiser la bande passante utilisée dans un tel échange graphique. Toutefois, VNC est basé, comme la solution précédente, sur un échange de pixels, ce qui complique la comparaison puisqu'il risque de conduire à un nombre important de fausses alertes, comme dans le cas de la capture au niveau du matériel. Nous avons donc rejeté aussi cette solution.

VI.3.1.4 Capture au niveau des appels des machines virtuelles Java

Pour cette capture, nous nous plaçons au niveau des appels système générés par les machines virtuelles Java (cf. (III) de la figure VI.11). Rappelons que les applications que nous utilisons sont des applications Java utilisant la bibliothèque *Swing* pour gérer les objets graphiques de l'interface homme-machine. La comparaison à ce niveau consisterait donc à identifier, dans un premier temps, les appels systèmes que la machine virtuelle Java a générés pour effectuer une opération sur un périphérique de sortie, puis de comparer les appels. Cependant, vu la diversification que nous imposons au niveau des systèmes d'exploitation utilisés, ces appels ne peuvent pas être directement comparés, puisque leur syntaxe varie selon les systèmes d'exploitation, mais aussi parce que les fonctions fournies par les appels systèmes diffèrent d'un sytème à l'autre. Il est donc nécessaire d'implanter un traducteur sémantique en charge de capturer chaque appel système puis de le traduire dans une syntaxe qui permettrait une comparaison entre les différents appels. Une telle tâche est fort laborieuse et conduirait à une

implantation finale dépendant étroitement des systèmes d'exploitation sous-jacents, ce qui serait très pénalisant vis-à-vis de l'évolution de ces systèmes.

Nous avons donc opté pour une capture située à un niveau d'implantation supérieur, à savoir à l'entrée des bibliothèques graphiques Java.

VI.3.1.5 Capture au niveau des entrées graphiques des bibliothèques java

Cette capture est représentée par (IV) sur la figure VI.11 et consiste à intercepter les appels de méthodes de la machine virtuelle Java, permettant de commander un périphérique de sortie (d'une façon similaire à la capture des appels systèmes précédemment décrite). Cette technique consiste donc à envelopper (ou entourer d'un *wrapper*) chacune des applications puis transférer ces appels vers la machine virtuelle sûre en charge d'effectuer la comparaison et la validation.

Capturer les flux à ce niveau présente l'intérêt d'obtenir une valeur sémantique relative aux appels de méthodes dans une syntaxe unique (à travers l'utilisation des appels de l'API Java) pour interpréter les flux de provenance les deux machines virtuelles. C'est ce niveau de capture que nous avons choisi dans notre implantation finale du prototype. Mais le choix de ce niveau de capture implique une interaction plus au moins importante avec l'application diversifiée. Dans la section suivante, nous détaillons ces interactions en fonction de la visibilité du code de l'application.

VI.3.2 Interactions avec les applications diversifiées

Rappelons que dans le cadre de notre étude, les sorties auxquelles nous nous intéressons sont les sorties graphiques et les sorties vers le réseau. Les sorties réseaux sont plus faciles à gérer car elles ne génèrent pas d'interaction directe avec un opérateur humain, rendant ainsi leur duplication (et leur comparaison) moins problématique. La diversification d'une IHM quant à elle est plus compliquée à gérer, et nous proposons dans la section suivante de présenter la solution que nous préconisons dans le cadre de notre étude. Nous détaillerons ensuite les implications que peut avoir l'application sur les solutions finales à retenir.

VI.3.2.1 Gestion de l'interaction avec l'opérateur

La diversification basée sur la virtualisation conduit à l'implantation de plusieurs interfaces graphiques (correspondant à plusieurs machines virtuelles) sur un seul écran physique, étant donné que l'application de départ (application A_{1A} de la figure VI.11) est une IHM graphique. Opérer une simple réplication implique que l'opérateur humain doit interagir avec plusieurs fenêtres normalement identiques qu'il devrait comparer lui-même avant d'interagir dans chacune de ces fenêtres de la même façon. Une telle utilisation serait lourde et potentiellement source d'erreur d'utilisation. Dans certains cas, il n'est tout simplement pas possible pour l'opérateur de réaliser la même tâche plusieurs fois, pour des contraintes temporelles par exemple.

Ainsi, il est nécessaire d'avoir une seule interface homme machine, tout en assurant la redondance que nous avons décrite précédemment. Pour ce faire, nous considérons que l'affichage doit être réalisé par la machine virtuelle sûre, afin de s'assurer que l'exécution corresponde bien aux exigences fonctionnelles. Pour cela, nous nous basons sur le fait qu'une interface homme machine suit le modèle vue-contrôleur [Michaloski et al. 2000], dans lequel on distingue l'affichage (les objets graphiques) et les actions activées par l'IHM (par exemple les traitements effectués si l'on clique sur un bouton).

La diversification de l'application consiste donc à diversifier le cœur de l'IHM tout en gardant une seule interface. Le cœur communique avec l'interface graphique en utilisant un modèle événementiel, qui permet de commander les changements d'affichage et la mise à jour des informations affichées à l'opérateur. La comparaison effectuée entre les deux instances de cœur d'IHM des machines virtuelles non sûres portent donc dans un premier temps sur les objets graphiques mis en œuvre (boutons, zones de texte, leur position dans l'écrans, etc.), puis sur le contenu de ces objets (le texte affiché dans une zone de texte, etc.). Toute interaction avec les objets graphiques doit donc être capturée afin de permettre une validation fiable de l'affichage effectué.

Dans le cadre de nos travaux, nous avons dû adapter notre solution finale à des contraintes liées à cette interaction homme machine, et également à la visibilité du code de l'application diversifiée. Dans le cas présent, nous nous sommes intéressés aux applications existantes, et avons envisagé différentes pistes pour l'implantation du contrôle de flux à l'entrée des bibliothèques graphiques, en fonction des contraintes liées au code de l'application. Dans le paragraphe suivant nous décrivons ces contraintes, et proposons des recommandations pour les futures applications.

VI.3.2.2 Cas des applications existantes

Afin d'implémenter l'architecture que nous avons décrite dans la section IV.3, nous devons établir des hypothèses quant à la visibilité du code de l'application vis-à-vis des lectures et/ou écritures. En effet, il est plus compliqué de contrôler une application dont on ne maîtrise pas le code source qu'une application que l'on développe spécifiquement. Nous pouvons considérer les différentes hypothèses suivantes :

- **Hypothèse 1** : Nous n'avons aucun accès au code source, seuls les fichiers binaires (*bytecode*) sont fournis.

Dans ce cas, la capture des flux ne peut se faire que par modification de la machine virtuelle Java afin de transférer tous les appels vers les périphériques de sortie à l'objet de validation. L'avantage de cette solution est qu'aucun accès au code de l'application n'est nécessaire. Ceci est utile dans le cas où les applications en question ont été développées par des prestataires de services, rendant l'accès aux sources compliqué, voire même impossible. Cependant, cette solution n'est pas compatible avec les évolutions des machines virtuelles Java et des systèmes d'exploitation sous-jacents. En effet, à chaque mise à jour de la JVM, nous devons vérifier que les mécanismes de capture que nous avons implantés sont toujours valides et opérationnels. Ceci est fort contraignant, surtout que cela suppose un suivi technique assez fin lors du déploiement final de l'application. Nous pouvons envisager une optimisation liée au nombre des méthodes à capturer, en analysant le code binaire afin d'en extraire les méthodes réellement appelées, et ne traiter que ces méthodes. Le suivi de l'application dans ce cas se limiterait au fait de vérifier si les mises à jour affectent ou non les méthodes identifiées. Nous avons testé cette méthode dans le cadre de notre prototype (en modifiant des classes internes à la machine virtuelle Java) et avons bien vérifié qu'elle était fort contraignante. De ce fait, cette solution ne devrait être retenue que pour un nombre limité d'applications afin de réduire le suivi technique.

- **Hypothèse 2** : Nous avons un accès en lecture seule au code source de l'application.

Ce cas correspond en particulier à des logiciels certifiés, qu'il serait très coûteux de modifier. Dans ce cas, il est possible de reprendre le code et de le recompiler en utilisant un autre compilateur que celui de la JVM. Nous avons envisagé l'utilisation du compilateur *AspectJ* [Adrian Colyer et al. 2005] qui nous permet de détecter les aspects relatifs aux opérations de sorties effectuées par l'application, puis d'ajouter des fonctions permettant de transférer ces

opérations d'une manière similaire à ce qui est fait dans [Lawall et Muller 2000]. L'objectif de la programmation orientée aspect est de pouvoir séparer les modules ayant des objectifs différents, et en particulier de séparer les considérations techniques des considérations métiers. Nous avons retenu cette solution parce qu'elle présente deux avantages importants : la non modification du code de la JVM, et la non modification du code source. L'inconvénient de cette approche est la nécessité d'une recompilation du code. Cependant, ceci est envisageable dans un processus industriel pour permettre l'évolution de certaines applications. Nous donnerons dans la section VI.4.3 les détails de cette solution.

- **Hypothèse 3** : Nous avons un accès en lecture et en écriture du code source de l'application.

Ceci signifie que nous avons un contrôle complet sur le code et que nous pouvons l'adapter en fonction de nos besoins architecturaux. Il est donc envisageable de précéder tous les appels des méthodes de sortie par des appels à des méthodes en charge de transférer les données de sortie à l'objet de validation. Il est également envisageable de remplacer l'appel des fonctions standard de sortie par des appels de fonctions spécifiques qui implantent les mêmes interfaces d'appels. Cependant, l'inconvénient de cette solution est qu'il n'est toujours pas possible d'avoir un contrôle complet sur le code, surtout s'il n'a pas été développé localement.

Dans la section suivante, nous présentons l'implantation finale du démonstrateur que nous avons développé pour illustrer la validité de notre approche. Nous nous sommes basés sur l'implantation actuelle des opérations de maintenance sur un terminal de maintenance à bord de l'avion, et avons proposé un prototype qui prenne en charge les principales caractéristiques de ces applications quant à l'affichage et à la communication avec les équipements de l'avion.

VI.4 MISE EN ŒUVRE ARSEC

La figure VI.12 représente l'architecture globale du démonstrateur que nous avons développé dans le cadre de ces travaux.

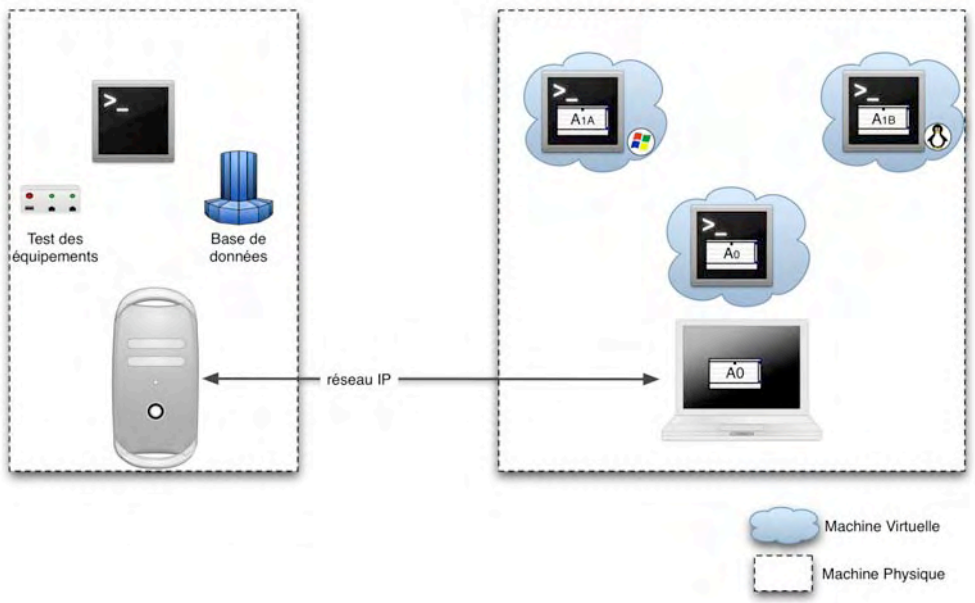


Figure VI.12 Principe général du prototype ArSec

L'application *A0* est sûre et est implantée sur une machine sûre, comme nous l'avons précédemment expliqué. Cette application n'a aucune fonctionnalité directe, elle est uniquement en charge de capter les événements des entrées-sorties graphiques, puis de les transférer vers/depuis les applications A_{1A} et A_{1B}. Ces dernières ont été initialement développées pour assurer un affichage graphique direct. Dans le cadre de ce prototype, A_{1B} a le même code que A_{1A}, mais s'exécute sur une machine virtuelle différente.

Dans ce qui suit, nous détaillons la capture des appels de la bibliothèque *Swing* en utilisant *AspectJ*.

VI.4.1 Interception des appels *Java Swing* et *Awt*

Afin d'avoir une meilleure idée sur l'implantation du mécanisme de contrôle, nous proposons de détailler d'abord les classes ajoutées respectivement dans les machines virtuelles non sûres et dans la machine sûre .

VI.4.1.1 Description d'une machine virtuelle non sûre

La bibliothèque *Swing* est une évolution de *AWT* (*Abstract Windows Toolkit*). Certains objets graphiques de *Swing* continuent à faire appel à des méthodes de *AWT*. Il est donc important de capturer les deux types d'appel. Intéressons nous à présent au mécanisme de détection et transfert de ces appels au sein d'une machine virtuelle. La figure VI.13 présente l'architecture déployée sur chaque machine virtuelle.

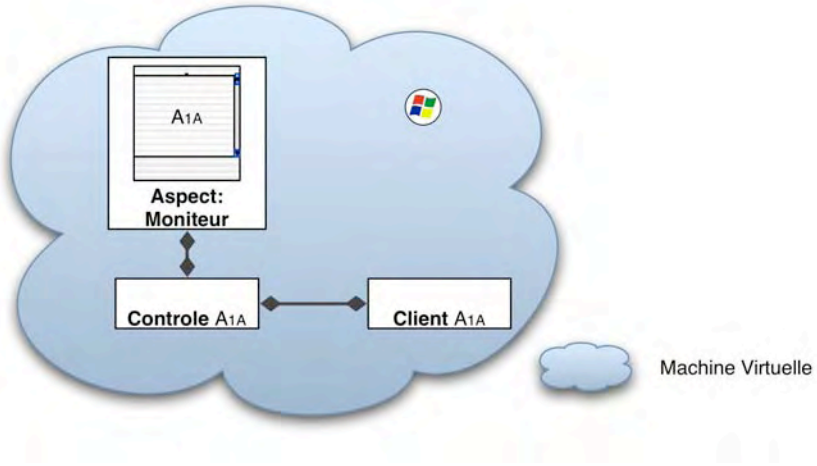


Figure VI.13 Capture des flux au niveau d'une machine virtuelle non sûre

La machine virtuelle contient donc les éléments suivants :

- L'application A_{1A} (ou A_{1B}) en charge d'effectuer les opérations de maintenance.
- Un Moniteur qui représente l'aspect. Il est en charge de définir les méthodes à capturer et les actions à réaliser une fois la capture faite.
- Une classe Contrôle A_{1A}. Cette classe est en charge de sauvegarder les objets qui gérent les sorties de A_{1A}.
- Une classe Client A_{1A}. Cette classe est en charge de la communication avec la machine virtuelle sûre.

VI.4.1.2 Description d'une machine virtuelle sûre

La machine virtuelle sûre contient les éléments suivants, comme le montre la figure VI.14 :

- Une classe Serveur 1 (et 2) : cette classe est en charge de la communication avec chacune des machines virtuelles non sûres.
- Une classe Contrôle 1 (et 2) : cette classe est en charge de sauvegarder les informations relatives à l'application A_{1A} (respectivement A_{1B}), et récupérées par le Serveur 1.
- Un objet de validation : en charge de comparer les états des deux applications A_{1A} et A_{1B} en vérifiant l'état de chaque classe de contrôle.
- Une classe *A0* : en charge d'implanter l'interface graphique, et est contrôlée par l'objet de validation.
- Une classe d'interface avec l'avion, en charge de sauvegarder l'état de communication de la machine de maintenance avec l'avion.

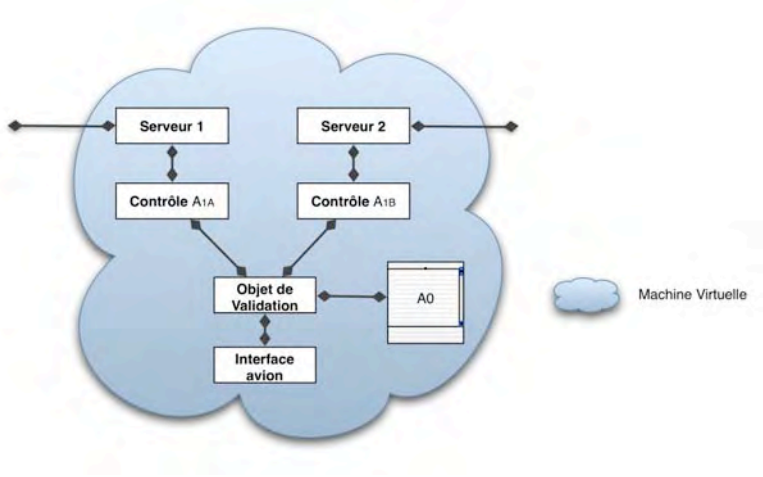


Figure VI.14 Description d'une machine virtuelle sûre

La capture des sorties se fait donc au niveau des aspects implantés dans chaque virtuelle non sûre. Dans ce qui suit, nous détaillons les différents niveau de capture réalisés.

VI.4.1.3 Capture des sorties et exécution

Cette capture est réalisée lors de l'instanciation de l'application. Nous distinguons alors deux phases d'exécution :

- Phase d'instanciation de l'interface graphique et réseau : durant cette phase, l'application instancie ses différents objets graphiques et réseau. Le moniteur les capture et les envoie à l'application de contrôle locale. Cette classe instancie un client qui envoie alors une notification au serveur qui lui est attribué au niveau de la machine virtuelle sûre afin qu'il prenne en compte cette action de création. Le code de la figure VI.15 illustre le rôle de l'aspect dans la capture d'un événement lié à la création d'une connexion à une base de données dans un équipement à bord de l'avion.
- Phase d'attente d'interaction en entrée: durant cette phase, tous les objets de l'interface graphique et réseau sont instanciés. L'application *A0* se met donc en attente d'une interaction en entrée afin de la transférer à chacune des applications A_{1A} et A_{1B}, elles-mêmes en attente de cette interaction. Prenons l'exemple d'un bouton graphique dans *A0*. L'*Action Performed* attribué à ce bouton définit les actions à effectuer si l'on clique sur le bouton. Dans *A0*, cette fonction implante le transfert de l'événement « clic » aux autres applications implantées sur les machines virtuelles non sûres. Ce sont ensuite les applications A_{1A} et A_{1B} qui prennent en charge l'action réelle à entreprendre (action qui a été définie dans le cœur de l'IHM).

```
//Pour se connecter à  la base de données
public pointcut CaptureGetConnection( String url, String user, String password) : call (Connection
java.sql.DriverManager.getConnection(String, String , String)) && args(url, user, password)  &&! this(application.Serveur) &&
this(virtualHost.MainApplication));

Connection around( String url, String user, String password) : CaptureGetConnection ( url, user, password) && !this(asp.AMoniteur){
        System.out.println("[ AspectJ ]On capture une connection à la base de données");
        Connection connect =  this.reseau.addGetConnection(url, user, password);
        this.tabObject.addObject(connect, "connection");
        System.out.println("[ AspectJ ]: Je vais renvoyer un null comme connection");
        return connect;
}
```

Figure VI.15 Aspect de capture d'événement de connexion

VI.4.2 Contrôle au niveau d'une machine virtuelle sûre

Afin de rendre sûres les sorties graphique et réseau de l'ordinateur de maintenance, nous faisons l'hypothèse que le matériel et l'hyperviseur sont au moins du niveau d'intégrité du monde avionique, comme nous l'avons montré dans le chapitre V. Nous faisons également l'hypothèse que les sorties sont correctes au niveau de la machine virtuelle sûre..

Dans l'implémentation actuelle, nous utilisons une JVM installée sur la machine virtuelle sûre. L'application *A0* est donc développée en utilisant la bibliothèque *Java Swing*. Nous supposons donc qu'il est possible de valider cette JVM au niveau d'intégrité de l'avionique. Cette hypothèse est réaliste, dans le sens où nous n'utilisons pas toutes les primitives de la JVM. En effet, l'application *A0* représente simplement un squelette graphique, et n'accomplit aucun traitement. Nous pouvons donc imaginer un processus de validation en charge d'augmenter uniquement le niveau de confiance des bibliothèques de sortie de la JVM, à savoir *Swing*, *AWT* et les bibliothèques réseau, sans avoir à valider la totalité d'une JVM.

Il est également envisageable d'implanter une application en charge de traduire les appels vers des méthodes *Swing* et réseau en appels à des méthodes d'une bibliothèque graphique qui existerait déjà dans le système d'exploitation validé. Mais même si une telle bibliothèque graphique existe, il peut être difficile de valider le mécanisme de traduction que l'on doit implanter. Nous n'avons pas avancé sur cette piste, mais nous tenons juste à mentionner que cela peut être envisagé dans le cas où la validation des bibliothèques de sortie de la JVM s'avèrerait trop difficile.

Une fois les niveaux de confiance correctement attribués à chaque entité logicielle de l'architecture, nous nous sommes intéressés à la problématique de déterminisme, que nous avons déjà décrite dans le chapitre V. Ci-dessous nous décrivons l'implémentation que nous avons faite dans le cadre du projet ArSec.

VI.4.3 Déterminisme dans ArSec

Sans solution spécifique au problème du déterminisme, il est fort probable qu'une proportion importante des comparaisons au niveau de l'objet de validation échouent, ce qui ne mettrait certes pas en danger la sécurité (puisque aucune sortie n'est validée, et ne provoque donc aucune altération du comportement des composants avioniques), mais serait rédhibitoire pour la disponibilité (un arrêt étant provoqué à chaque fausse alerte levée). Il est donc important de vérifier ces propriétés de déterminisme (présentées dans la section V.1.3) pour assurer le bon fonctionnement du système.

Dans le cadre de nos travaux, nous supposons que le comportement de l'application est déterministe et prévisible (chaque fois que l'application est lancée, elle exécute le même code et effectue les mêmes opérations, et si les mêmes entrées sont les mêmes, elle fournit les mêmes sorties). Cependant, il existe plusieurs sources de non-déterminisme dans l'architecture globale. En effet, les systèmes d'exploitation peuvent introduire, à travers leurs ordonnanceurs à préemption, des exécutions non déterministes des applications qu'ils contrôlent. Ainsi, il est possible que l'exécution d'une application soit retardée à cause d'une CPU occupée, ce qui pourrait induire le déclenchement d'un *timeout*. Les communications réseaux sont également une source de non-déterminisme, du moment que l'on ne maîtrise pas le temps requis par chaque paquet IP pour arriver à destination. La gestion interne des objets dans la JVM peut aussi être source de non-déterminisme, puisque les objets graphiques peuvent être traités par des processus légers (*threads*) différents, et que la politique d'ordonnancement de ces processus n'est pas obligatoirement déterministe.

Donc, même si l'application elle-même est prédictible et déterministe, toutes les sources de non-déterminisme que nous avons cités peuvent conduire à un comportement non déterministe de notre application. Ce comportement non déterministe induira donc un échec lors du processus de comparaison au niveau de la machine virtuelle sûre. En particulier si au moment de la comparaison des deux flux de provenance les deux machines virtuelles non sûres, on n'est pas certain que les flux sont bien parvenus au comparateur et qu'ils correspondent à la même exécution.

Pour mieux assurer un déterminisme au niveau de la comparaison, nous avons choisi de placer un point de contrôle (*checkpoint*) de la capture de chaque flux de sortie. Ces points de contrôle permettent de suivre de plus près l'exécution de l'application en suivant les messages issus des machines diversifiées. Pour ce faire, les messages envoyés sont des messages à base événementielle, qui ne contiennent pas d'objet Java, mais des événements relatifs aux actions qui ont eu lieu au niveau des applications A_{1A} et A_{1B} (cf. figure VI.12). Ainsi, les classes *Contrôle* A_{1A} (et *Contrôle* A_{1B}) des figures VI.12 et VI.13, sauvegardent les objets de sortie internes à chaque application (A_{1A} et A_{1B}) et les messages échangés avec la machine sûre permettent d'identifier aussi bien la nature de l'action (par exemple changer la valeur affichée dans une zone de texte) que l'objet sur lequel l'action porte.

Afin de mieux illustrer ces mécanismes, nous présentons dans la section suivante une description détaillée de l'exécution du prototype ArSec.

VI.4.4 Description du démonstrateur ArSec

La figure VI.16 présente une exécution du démonstrateur.

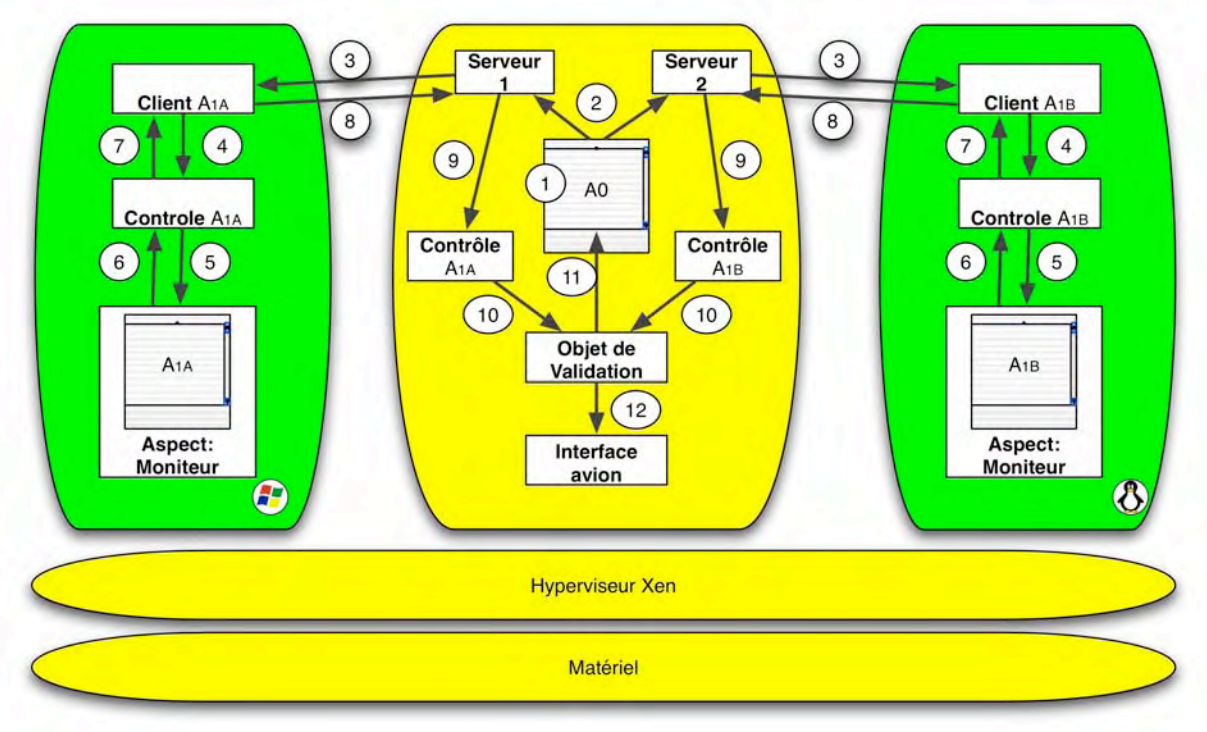


Figure VI.16 Exécution du démonstrateur ArSec

Nous nous intéressons dans cette exécution à un événement graphique (un clic sur un bouton de calcul par exemple) qui génère des actions au niveau de l'application principale. Nous supposons la phase d'instanciation effectuée (cf. VI.4.3.1) et nous nous plaçons dans la phase

d'attente d'interaction extérieure de l'application. Dans ce qui suit, nous détaillons les différents points d'exécution :

- (1) L'événement graphique est effectué au niveau de l'interface graphique *A0*.
- (2) Cet événement est transféré vers les applications serveurs, en charge de toutes les communications avec les autres machines virtuelles.
- (3) Chaque serveur transfère le message contenant la nature de l'événement (clic par exemple) et l'identifiant de l'objet sur lequel l'événement porte (le bouton sur lequel on a cliqué).
- (4) Chaque client transfère ce message à la classe de contrôle qui identifie l'objet local en question.
- (5) La classe de contrôle réalise le traitement correspondant à l'événement sur l'objet local, ce qui implique une activation d'une série d'actions au niveau de chaque instance.
- (6) Chaque instance opère les actions réelles relatives à l'événement en question (par exemple lancer un calcul, etc.). Ces actions comportent ensuite une interaction avec les périphériques de sortie, et sont donc capturées par les aspects qui les envoient aux classes de contrôle.
- (7) La classe de contrôle crée un message événementiel relatif à l'événement capturé par l'aspect et l'envoie à la classe client.
- (8) Chaque classe client envoie le message à la classe serveur correspondante, sur la machine virtuelle sûre.
- (9) Chaque serveur transfère ce message à la classe de contrôle locale de la machine virtuelle sûre, afin de mettre à jour l'état d'exécution de chaque instance.
- (10) L'objet de validation compare l'état d'exécution, et vérifie que les deux instances sont au même niveau d'exécution et qu'elles réalisent les mêmes actions (en contrôlant les événements reçus un par un). Si un événement manque, l'objet de validation observe un temps d'attente afin de tolérer les décalages temporels qui peuvent survenir lors du passage en réseau. Si le délai est dépassé, ou que les paramètres des événements ne sont pas les mêmes, ceci implique une divergence dans les exécutions, ce qui est potentiellement indicateur d'un dysfonctionnement (voire d'une malveillance) au niveau d'une machine virtuelle. Dans ce cas, l'exécution est arrêtée et une alerte est levée.
- (11) Si le résultat de la comparaison est positif et que l'événement concerne une sortie graphique, cette sortie est réalisée sur A0 et devient donc visible à l'opérateur.
- (12) Si le résultat de la comparaison est positif et que l'événement concerne une sortie vers l'avion, alors cet événement est envoyé à l'interface réseau de l'avion qui se charge de le traduire en requête interprétable par les équipements avioniques.

Dans le scénario déroulé ci-dessus, on s'est intéressé à une entrée graphique et nous avons suivi le déroulement de l'exécution. Il est possible de remplacer une entrée graphique par une entrée du réseau, et la procédure reste la même, à savoir le transfert vers les applications diversifiées, puis la comparaison des résultats.

VI.5 CONCLUSION

Dans ce chapitre nous avons présenté :

- Un cas d'étude des opérations de maintenance à bord.
- Un cas d'étude du calcul du profil de décollage sur l'ordinateur portable du pilote.

Ces deux cas d'étude ont en commun une nécessité de remontée de flux d'un niveau peu critique (du monde ouvert) vers un niveau plus critique (le niveau avionique). Nous avons ainsi réalisé les actions suivantes :

- Une analyse de tous les flux de données et l'établissement d'une correspondance entre les cas d'étude et le modèle Totel
- Une mise en œuvre d'un démonstrateur permettant de prendre en charge les flux ascendants.

Ces flux ont pour origine les périphériques de sortie, nous avons proposé alors la démarche suivante :

- L'analyse des différents niveaux de capture des flux et leur classification en fonction de leur difficulté d'implantation et de leur degré d'expressivité pour réaliser une comparaison.
- La séparation d'une application qui implémente une IHM en deux parties : le cœur de l'IHM et l'interface.
- L'implémentation des cœurs des IHM dans des machines virtuelles diversifiées, et l'implémentation de l'interface dans une machine virtuelle sûre.

Nous avons également établi des hypothèses de fautes et avons pu tester la validité de notre approche lors de l'implémentation d'un scénario d'attaque visant la JVM.

Ces travaux de mise en œuvre peuvent être étendus dans différentes directions :

- Vérification du démonstrateur dans un environnement de déploiement réel (à savoir la participation au développement et au déploiement de l'application de calcul de profil),
- Prise en charge d'applications non déterministes,
- Utilisation d'un analyseur statique de flux afin d'implémenter directement le modèle Totel sur une architecture logicielle existante.

VI.6 RECOMMANDATIONS POUR LES APPLICATIONS FUTURES

Dans le cadre de nos travaux, nous avons eu la possibilité d'avoir un accès en lecture seule au code source de l'application de maintenance. Nous avons donc opté pour la solution proposée dans l'hypothèse 2 du paragraphe VI.3.2.2.

Grâce aux différentes pistes que nous avons envisagées, nous pouvons émettre des recommandations pour les développeurs des futures applications. En effet, l'hypothèse 3 (du paragraphe VI.3.2.2) est fort intéressante parce qu'elle réduit considérablement l'effort de capture des flux. Ainsi, tout appel vers des méthodes de *Swing* seraient remplacés par des appels vers une bibliothèque *AirbusSwing* fournie par Airbus. Cette bibliothèque implanterait les mêmes interfaces que la bibliothèque *Swing* standard, mais comporterait des fonctions en charge de transférer les appels graphiques vers l'objet de validation. Ceci a l'avantage d'être facile à mettre en œuvre, mais également à évoluer en fonction des mises à jour des systèmes d'exploitation, des compilateurs et des JVM, une fois l'application déployée. En effet, s'il y a une modification importante au niveau de la JVM, il suffit d'adapter *AirbusSwing* et de l'envoyer aux clients finaux pour remplacer les versions existantes.

Nous recommandons également de respecter, lors du développement d'une application graphique, la séparation entre le cœur et l'affichage d'une IHM. Ceci permet de faciliter la capture des données fonctionnelles utilisées lors de la comparaison par l'objet de validation. Il n'est donc plus nécessaire d'implanter deux IHM diversifiées, mais il suffit d'implanter une seule interface graphique tout en diversifiant les cœurs d'IHM.

Chapitre VII

CONCLUSION GENERALE

VII.1 BILAN

Nous avons abordé, dans le cadre de cette thèse, la problématique relative aux aspects sécurité-innocuité et sécurité-immunité lors des interactions entre composants ayant des niveaux de criticité hétérogènes. Nous nous sommes placé dans un environnement avionique, qui présente traditionnellement des contraintes élevées en sécurité-innocuité, et avons proposé une architecture sûre permettant à des entités peu critiques de communiquer avec des entités plus critiques. La démarche que nous avons suivie dans le cadre de ces travaux est la suivante :

- Nous avons étudié, dans un premier temps, les caractéristiques des environnements avioniques (section II.1). Ces environnements sont caractérisés par de fortes exigences quant à la sécurité (au sens innocuité) et la disponibilité des fonctions avioniques. Cette étude nous a permis de prendre conscience des nouvelles composantes technologiques et économiques qui mettent en évidence l'émergence d'une nouvelle catégorie de menaces dans le monde avionique : les malveillances. Nous nous sommes alors penché sur les travaux réalisés dans le domaine de la sécurité-immunité pour l'avionique et plus précisément sur les nouvelles normes et les propositions des groupes de travail visant à intégrer les techniques de sécurité-immunité dans les systèmes avioniques.
- Au vu de ces nouvelles menaces, nous avons examiné les techniques proposées dans le domaine de sûreté de fonctionnement informatique (section II.2) et en particulier pour la tolérance aux fautes. Ces techniques offrent un large spectre de solutions potentielles pour tolérer aussi bien les fautes malveillantes qu'accidentelles.
- Nous avons constaté que différentes normes relatives à la sûreté de fonctionnement des systèmes critiques introduisent la notion de niveaux pour classifier les systèmes selon leur criticité (section III.1). Nous avons alors proposé une définition cohérente des différents types de niveaux introduits dans ces normes, à savoir la criticité, l'intégrité, la validation et la crédibilité. Pour ce faire, nous avons défini le niveau de confiance comme étant une fonction des niveaux d'intégrité, de validation et de crédibilité qui permet de répondre à des exigences de criticité.
- Nous avons ensuite présenté quelques modèles qui ont été développés pour des systèmes ayant des composants à niveaux d'intégrité hétérogènes (section III.2). Nous avons retenu le modèle Totel car c'est le seul modèle qui permette une remontée de flux (d'un niveau peu critique vers un niveau plus critique) sans altérer les niveaux d'intégrité de chaque composant. Ce modèle recommande l'utilisation de techniques de tolérance aux fautes pour augmenter la confiance dans certaines informations et ainsi permettre des flux remontants.
- Les contraintes de déploiement (notamment l'utilisation d'un ordinateur portable pour des applications dans un environnement de bas niveau d'intégrité) nous ont guidé vers l'utilisation de la technologie de virtualisation pour assurer une diversification logicielle, dans le but d'augmenter le niveau de confiance des sorties de cette machine. Nous avons alors étudié cette nouvelle technologie et présenté ses potentielles utilisations dans le cadre de la tolérance aux fautes (chapitre IV).
- Afin d'implémenter les mécanismes de tolérance aux fautes nécessaires, nous nous sommes intéressés à établir les hypothèses de fautes que nous traitons (section V.1). Nous avons ensuite proposé une formalisation du modèle Totel en nous basant sur les

cônes de causalité et de dépendance introduits dans le modèle de dépendance causale. Nous avons proposé également une extension à ce modèle afin de définir les contraintes imposées aux objets de validation utilisés dans le modèle Totel (section V.2).

- Le contexte d'exécution de notre architecture étant distribué, nous avons alors proposé une adaptation du modèle Totel au contexte distribué (section V.3).
- Afin d'illustrer la validité de notre approche, nous avons développé un démonstrateur (chapitre VI et annexe). Ce démonstrateur se base sur des besoins fonctionnels requis par l'avionneur et pour lesquels des flux de données remontants ont été identifiés. Nous nous sommes intéressé aux sorties graphiques de cette machine ainsi qu'aux sorties réseau afin de les valider (et donc pouvoir les transférer soit à l'utilisateur soit à l'appareil).

L'utilisation de la virtualisation tout en gardant un affichage unique sûr a notamment constitué une problématique originale et intéressante à traiter. Elle nous a permis d'étudier en détail les architectures d'un système d'exploitation et d'étudier les différents niveaux d'intervention pour implémenter les mécanismes de tolérance aux fautes.

VII.2 PERSPECTIVES

Nous avons émis, dans la section VI.6, des recommandations relatives au développement d'applications futures (notamment les applications dont il est nécessaire de rendre sûre l'exécution sur des plateformes de type COTS). Il est clair que l'intégration de notre démonstrateur dans un réel environnement d'exécution nous aurait donné l'opportunité de tester notre architecture dans un contexte industriel. Ainsi, réaliser une telle intégration s'avère une perspective fort intéressante d'investigation.

Dans le cadre de notre prototype et pour des raisons de confidentialité, nous nous sommes limités à deux niveaux de criticité : le monde ouvert et le premier niveau avionique. Cependant, notre réflexion porte également sur les niveaux avioniques plus élevés. Une extension possible de nos travaux consisterait alors à développer les relais et les TCB en charge de remonter les flux jusqu'aux niveaux les plus critiques, en se basant sur les technologies utilisées en avionique.

Dans cette même optique, il serait intéressant de voir comment adapter notre architecture pour d'autres communications à flux remontants, étant donnée l'utilisation croissante de services embarqués à bord des appareils nécessitant des interactions avec des composants du monde ouvert.

Nous avons présenté, dans ce manuscrit, la problématique du déterminisme, et avons établi certaines hypothèses relatives aux applications traitées. Il serait intéressant d'étendre ces hypothèses, notamment pour traiter le renforcement du déterminisme entre applications intrinsèquement non déterministes.

Remarquons que l'architecture que nous déployons suppose que la machine est démarrée directement en mode sécurisé (dans lequel l'hyperviseur Xen est lancé). Assurer une sécurité de bout en bout (en utilisant une *TPM Trusted Platform Module* par exemple) au cours du démarrage est une piste d'investigation importante, puisqu'elle peut contribuer à assurer la bonne exécution de l'architecture déployée.

Finalement, nous avons proposé une formalisation du modèle Totel avec une extension envisagée pour définir les contraintes sur les objets de validation utilisés. Inclure cette formalisation dans le cadre d'un outil automatique d'analyse de flux de données (par exemple au niveau d'un noyau de système d'exploitation) constitue une extension fort intéressante à envisager.

REFERENCES BIBLIOGRAPHIQUES

Adrian Colyer, Andy Clement, George Harley, et Matthew Webster. 2005. *Eclipse AspectJ, Aspect-Oriented Programming with AspectJ and the Eclipse Aspect Development Tools*. 1er éd. Eclipse.

Alena, R.L., J.P. Ossenfort, K.I. Laws, A. Goforth, et F. Figueroa. 2007. Communications for Integrated Modular Avionics. Dans *Aerospace Conference, 2007 IEEE*, 1-18. Big Sky, Montana, USA.

Alves-Foss, Jim, Paul W. Oman, Carol Taylor, et W. Scott Harisson. 2006. The MILS architecture for high-assurance embedded systems. Dans *International Journal of Embedded Systems*, Volume 2:pages 239-247.

ARINC 653. 1997. Avionics Application Standard Software Interface. *Aeronautical Radio Inc.*

ARINC 811 . 2005. COMMERCIAL AIRCRAFT INFORMATION SECURITY CONCEPTS OF OPERATION AND PROCESS FRAMEWORK. *Aeronautical Radio Inc.*

ARP 4754. 1996. Certification Considerations for Highly-Integrated Or Complex Aircraft Systems .

ARP 4761 . Guidelines and Methods for Conducting the Safety Assessment Process on Civil Airborne Systems and Equipment.

Avizienis, A, JC Laprie, B Randell, et C Landwehr. 2004. Basic Concepts and Taxonomy of Dependable and Secure Computing. *IEEE TDSC* 1, no. 1: 11-33.

Avizienis, A. 1985. The N-Version Approach to Fault-Tolerant Software. *IEEE Trans. Softw. Eng.* 11, no. 12: 1491-1501.

Balacheff, Boris, Liqun Chen, Siani Pearson, David Plaquin, et Graeme Proudler. 2003. *Trusted Computing Platforms, TCPA technology in context*. Pearson. HP Invent.

Basile, Claudio, Zbigniew Kalbarczyk, et Ravi Iyer. 2003. A Preemptive Deterministic Scheduling Algorithm for Multithreaded Replicas. *In Proceedings of the IEEE International Conference on Dependable Systems and Networks* : 149--158.

Basile, Claudio, Keith Whisnant, Zbigniew Kalbarczyk, et Ravi Iyer. 2002. Loose Synchronization of Multithreaded Replicas. Dans *Proceedings of the 21st IEEE Symposium on Reliable Distributed Systems*, 250. IEEE Computer Society.

Bell, D. Elliott, et Leonard J Lapadula. 1975. Secure Computer Systems: Mathematical Foundations. *MITRE Co., technical report M74-244*.

Biba, K. J. 1977. Integrity Considerations for Secure Computer Systems. *MITRE Co., technical report ESD-TR 76-372*.

Bowles, K.L. 1980. *Beginner's Guide for the UCSD Pascal System*.

Box, Don, et Chris Sells. 2002. *Essential .NET, Volume 1: The Common Language Runtime*.

Brillant, Susan S., et John C. Knight. 1989. The Consistent Comparison Problem in N-Version Software. Dans *IEEE Transactions on Software Engineering*, Volume 15:1481-1485.

Chernoff, Anton, et R.J. Hookway. 1997. DIGITAL FX!32: Running 32-Bit x86 Applications on Alpha NT. Dans *USENIX Windows NT Workshop*. Seattle.

Clark, DD, et DR Wilson. 1987. A Comparison of Commercial and Military Computer Security Policies. Dans *in IEEE Symposium on Security and Privacy*, 184-194. Oakland: IEEE Computer Society Press.

Common Criteria. Common Criteria for Information Technology Security Evaluation. *V3.1, CCMB-2006-09-001, CCMB-2006-09-002, CCMB-2006-09-003*. http://www.commoncriteriaportal.org/ .

Conmy, Philippa, et John McDermid. 2001. High level failure analysis for Integrated Modular Avionics. Dans *Proceedings of the Sixth Australian workshop on Safety critical systems and software - Volume 3*, 13-21. Brisbane, Australia: Australian Computer Society, Inc.

D'ausbourg, Bruno. 1994. Implementing secure dependencies over a network by designing a distributed security subsystem. Dans *Computer Security — ESORICS 94*, 247-266.

Delange, J., J. Hugues, L. Pautet, et B. Zalila. 2008. Code Generation Strategies from AADL Architectural Descriptions Targeting the High Integrity Domain. Dans *In 4th European Congress ERTS*. Toulouse, France. http://penelope.enst.fr/aadl/wiki/Papers.

Deswarte, Y., et J. Lavictoire. 1975. MARIGNAN-A method for correcting intermittent failures. Dans *International Symposium on Fault-Tolerant Computing FTC-5*. Paris, France.

Deswarte, Y., et D. Powell. 2006. Internet Security: An Intrusion-Tolerance Approach. *Proceedings of the IEEE* 94, no. 2: 432-441. doi:10.1109/JPROC.2005.862320.

Deswarte, Yves, Laurent Blain, et Jean-Charles Fabre. 1991. Intrusion Tolerance in Distributed Computing Systems. Dans *Proceedings of Symposium on Research in Security and Privacy*, 441-451. Oakland, California, USA.

Deswarte, Yves, Mohamed Kaâniche, Pierre Corneillie, et John Goodson. 1999. SQUALE Dependability Assessment Criteria. Dans *Computer Safety, Reliability and Security*, 71.

Deswarte, Yves, Karama Kanoun, et Jean-claude Laprie. 1998. Diversity against accidental and deliberate faults. *Computer Security, Dependability, and Assurance: From Needs to Solutions*: 171--181. doi:10.1.1.27.9420.

DO-254. 2000. Design Assurance Guidance for Airborne Electronic Hardware, DO-254. *Radio Technical Commission for Aeronautics (RTCA)*.

DO178-B. 1992. Software Considerations in Airborne Systems and Equipment Certification. *Radio Technical Commission for Aeronautics (RTCA) European Organization for Civil Aviation Electronics (EUROCAE), DO178-B*. doi:10.1.1.1.6299.

Domaschka, Jörg, Thomas Bestfleisch, Franz J. Hauck, Hans P. Reiser, et Rüdiger Kapitza. 2008. Multithreading strategies for replicated objects. Dans *Proceedings of the 9th ACM/IFIP/USENIX International Conference on Middleware*, 104-123. Leuven, Belgium: Springer-Verlag New York, Inc.

Dunlap, G.W., S.T. King, S. Cinar, M.A. Basrai, et P.M. Chen, 2002. ReVirt: Enabling Intrusion Analysis through Virtual-Machine Logging and Replay. Dans *SIGOPS Operating System Review 36 (SI)*, 211-224.

Dutertre, Bruno, et Victoria Stavridou. 1999. A Model of Noninterference for Integrating Mixed-Criticality Software Components. Dans *IN 7th International Working*

Conference on Dependable Computing for Critical Applications DCCA-7. San Jose, CA. http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.38.9844.

ED-127. 2009. Aeronautical Information System Security Part 2: Airworthiness Security Process Specification. *Draft of The European Organization for Civil Aviation Equipment*.

EN 50126. CENELEC EN 50126 Railway Applications - The Specification and Demonstration of Reliability, Availability, Maintainability and Saftey (RAMS) - IHS, Inc. *European Committee for Electrotechnical Standardization (CENELEC)*.

Fraga, J., C. Maziero, L. C. Lung, et O. G. Loques Filho. 1997. Implementing Replicated Services in Open Systems Using a Reflective Approach. Dans *Proceedings of the 3rd International Symposium on Autonomous Decentralized Systems*. IEEE Computer Society.

Friedman, Roy, et Alon Kama. 2003. Transparent fault-tolerant Java virtual machine. Dans *In proceedings of 22nd IEEE Symposium on Reliable Distributed Systems (SRDS 2003)*, 319-328.

Garfinkel, Tal, et Mendel Rosenblum. 2003. A Virtual Machine Introspection Based Architecture for Intrusion Detection. *In Proc. Network and Distributed Systems Security Symposium*: 191--206. doi:10.1.1.11.8367.

---. 2005. When Virtutal is Harder than Real: Security Challenges in Virtual Machine Based Computing Environments. Dans *10th Workshop on Hot Topics in Operating Systems (HotOS-X)*.

Gray, Jim. 1990. A Census of Tandem System Availabilility Between 1985 and 1990. Dans *IEEE Transactions on Reliability*, 39, n°2:409-432.

Hoxer, H.J., K. Buchacker, et V. Sieh. 2002. Implementing a User Mode Linux with Minimal Changes from Original Kernel. Dans *9th International Linux Sysem Technology Conference*.

IBM. 1972. IBM: VM History and Heritage References. http://www.vm.ibm.com/history/.

---. 1986. The IBM RT Personal Computer. *http ://www.oldcomputercollection.com/docs/ibm rt facts.pdf*.

IEC 61508. 2007. Functional safety of electrical /electronic /programmable electronic safety-related systems. Text. http://www.etas.com/fr/glossary_information_from_a_to_z-4319.php.

Jansen, Bernhard, Harigovind V. Ramasamy, Matthias Schunter, et Axel Tanner. 2008. Architecting Dependable and Secure Systems Using Virtualization. Dans *Architecting Dependable Systems V*, 124-149. Springer-Verlag.

Joshi, A., S.T. King, G.W. Dunlap, et P.M. Chen, 2005. Detecting Past and Present Intrusions Through Vulnerability-Specific Predicates. Dans *20th ACM Symposium on Operating Systems Principles (SOSP)*, 91-104.

Kaiser, Robert, et Stephan Wagner. The PikeOS Concept: History and Design. Dans *SYSGO Embedding Innovations White Paper*.

King, S.T., et P.M. Chen, 2003. Backtracing Intrusion. Dans *19th ACM Symposium on Operating Systems Principles (SOSP '03)*, 51-76.

King, S.T., G.W. Dunlap, et P.M. Chen, 2005. Debugging Operating Sysetms with Time-Traveling Virtual Machines. Dans *Annual USENIX Technical Conference*, 1-15.

King, S.T., Z.M. Mao, D.G. Luchetti, et P.M. Chen, 2005. Enriching Intrusion Alerts through Multi-Host Causality. Dans *Network and Distributed System Security Symposium (NDSS)*.

Kopetz, H., A. Damm, C. Koza, M. Mulazzani, W. Schwabl, C. Senft, et R. Zainlinger. 1989. Distributed fault-tolerant real-time systems: the Mars approach. *Micro, IEEE* Volume 9, no. 1: 25-40.

Laarouchi, Youssef, Yves Deswarte, David Powell, et Jean Arlat. 2008. Safety and Security Architectures for Avionics. Dans *Doctoral Consortium (DCSOFT 2008)*, the 3rd International Conference on Software and Data Technologies (ICSOFT 2008): Porto, Portugal, Juillet.

Laarouchi, Youssef, Yves Deswarte, David Powell, Jean Arlat, et Eric de Nadai. 2009a. Criticality and Confidence Issues in Avionics. Dans *12th European Workshop on Dependable Computing (EWDC)*, EWDC'09: Toulouse, France.

---. 2009b. Connecting Commercial Computers to Avionics Systems. Dans *Proceedings of the 28th IEEE/AIAA Digital Avionics Systems Conference (DASC), IEEE CS Press*. Orlando, FL, USA, Octobre.

Lacombe, Eric, Vincent Nicomette, et Yves Deswarte. 2009. Une approche de virtualisation assistée par le matériel pour protéger l'espace noyau d'actions malveillantes . Dans *Symposium sur la Sécurité des Technologies de l'Information et des Communications (SSTIC 09)*. Rennes.

Lagar-Cavilla, H.A., J.A Whitney, A. Scanell, P. Patchin, S. Rumble, E. de Lara, M. Brundo, et M. Satyanarayanan. 2009. SnowFlock: Rapid Virtual Machine Cloning for Cloud Computing. Dans *4th ACM European Conference on Computer Systems (EuroSys)*, 1-12. Nuremberg, Germany.

Laprie, Jean-Claude, Christian Béounes, Karama Kanoun, et Jean Arlat. 1990. Definition and Analysis of Hardware- and Software-Fault-Tolerant Architectures. *Computer* 23, no. 7: 39-51.

Laprie, Jean-claude, Yves Deswarte, Jean Arlat, David Powell, Karama Kanoun, Mohamed Kaaniche, Yves Crouzet, et al. 1996. *Guide de La Sûreté De Fonctionnement*. Cepadues.

Laureano, Marcos, Carlos Maziero, et Edgar Jamhour. 2004. Intrusion Detection in Virtual Machine Environments. Dans *30th EUROMICRO Conference'04*, 520-525.

Lawall, Julia, et Gilles Muller. 2000. Efficient Incremental Checkpointing of Java Programs. Dans *In International Conference on Dependable Systems and Networks (DSN'2000)*. New York.

Lee, P. A., et T. Anderson. 1990. *Fault Tolerance: Principles and Practice.* Éd. J. C. Laprie, A. Avizienis, et H. Kopetz. Springer-Verlag New York, Inc.

Libin Dong, R. Melhem, D. Mosse, S. Ghosh, W. Heimerdinger, et A. Larson. 1999. Implementation of a transient-fault-tolerance scheme on DEOS-a technology transfer from an academic system to an industrial system. Dans *Proceedings of the Fifth IEEE*

Real-Time Technology and Applications Symposium, 56-65. doi:10.1109/RTTAS.1999.777661.

Michaloski, John, Sushil Birla, et Jerry Yen. 2000. Software Models for Standardizing the Human-Machine Interface Connection to a Machine Controller. Dans *Proceedings of the World Automation Congress' 2000*. Maui. doi:10.1.1.14.1166.

Moir, Ian, et Allan Seabridge. 2003. *Civil Avionics Systems*. Professional Engineering Publishing.

Namjoo, M. 1983. CERBERUS-16: An Architecture for a General Purpose Watchdog Processor. Dans *IEEE Computer Society Press, 13th Int. Symposium on Fault Tolerant Computing (FTCS-13)*, 316-325. Milan, Italie.

Napper, Jeff, Lorenzo Alvisi, et Harrick Vin. 2003. A Fault-Tolerant Java Virtual Machine. *In Proceedings of the international conference on Dependable Systems and Networks (DSN 2003), DCC SYMPOSIUM*: 425-434.

NCSC. 1987. NCSC, Trusted CNetwork Interception of Trusted Computer Security Evaluation Criteria. Dans *National Computer Security Center (USA), rapport technique NCSC-TG-005*.

Olive, M.L., R.T. Oishi, et S. Arentz. 2006. Commercial Aircraft Information Security-an Overview of ARINC Report 811. Dans *25th Digital Avionics Systems Conference, 2006 IEEE/AIAA*, 1-12. doi:10.1109/DASC.2006.313761.

Peterson, W. W., et E. J. Weldon. 1972. Error-Correcting Codes. Dans *MIT Press*. Cambridge, MA, USA.

Pok Project. POK Project. Dans *http://penelope.enst.fr/aadl/wiki/PokPresentation*.

Poledna, Stefan. 1994. Replica Determinism in Distributed Real-Time Systems: A Brief Survey. *Real-Time Systems* Volume 6, no. 3: 289-316.

Powell, David. 1991. Chapter 6.4: Replicated Software Components. Dans *Delta-4: A generic architecture for dependable computing*, ESPRIT Research Reports:100-104. Springer Verlag.

Rabéjac, Christophe. 1995. Auto-surveillance logicielle pour applications critiques: méthode et mécanismes. Thèse en informatique, Institut National Polythechnique de Toulouse.

Randell, B. 1975. System structure for software fault tolerance. Dans *Proceedings of the international conference on Reliable software*, 437-449. Los Angeles, California: ACM. doi:10.1145/800027.808467.

Reiser, H.P., et R. Kapitza. 2007. Hypervisor-Based Efficient Proactive Recovery. Dans *26h IEEE International Symposium on Reliable Distributed Systems (SRDS)*, 83-92.

Richardson, Tristan. 2007. The RFB Protocol. Dans .

Richardson, Tristan, Quentin Stafford-Fraser, Kenneth R. Wood, et Andy Hopper. 1998. Virtual Network Computing. Dans *IEEE Internet Computing*, Volume2, Number 1:

Rodrigues, Rodrigo, Miguel Castro, et Barbara Liskov. 2003. BASE: using abstraction to improve fault tolerance. *ACM Transactions on Computer Systems* Volume 21, no. 3: 236-269.

Rugina, Ana-Elena, Peter Feiler, Karama Kanoun, et Mohamed Kaaniche. 2008. Software dependability modeling using an industry-standard architecture description language. Janvier 29.

Rushby, J. 2000. Partitioning in Avionics Architectures: Requirements, Mechanisms, and Assurance. Dans *NASA report*. CR-1999-209347.

Rushby, John. 1993. Formal Methods and the Certification of Critical Systems. *CSL Technical Report SRI-CSL-93-7*.

Salles, Frédéric, Manuel Rodriguez, Jean-Charles Fabre, et Jean Arlat. 1999. MetaKernels and Fault Containment Wrappers. Dans *Proceedings of the Twenty-Ninth Annual International Symposium on Fault-Tolerant Computing*, 22-29. Madison, Wisconsin, USA: IEEE Computer Society.

Smith, Jim, et Ravi Nair. 2005. *Virtual Machines: Versatile Platforms for Systems and Processes*. 1er éd. Morgan Kaufmann, Juin 17.

Tanenbaum, Andrew. 2001. *Architecture de l'ordinateur : Cours et exercices*. Dunod.

Totel, E., Lj. Beus-dukic, J.-P. Blanquart, Y. Deswarte, et D. Powell. 1998. Integrity Management in GUARDS. Dans *Proceedings of IFIP International Conference on Distributed Systems Platforms and Open Distributed Processing (Middleware'98)*, 105-122. The Lake District, UK.

Totel, E., J.-P. Blanquart, Y. Deswarte, et D. Powell. 1998. Supporting multiple levels of criticality. Dans *28th Annual International Symposium on Fault-Tolerant Computing*, 70-79. Washington, DC, USA.

Totel, Eric. 1998. Politique d'intégrité multiniveau pour la protection en ligne de tâches critiques. Thèse en informatique, Institut National Polythechnique de Toulouse, rapport LAAS N° 98533 .

Traverse, Pascal, Isabelle Lacaze, et Jean Souyris. 2004. Airbus Fly-By-Wire: A Total Approach To Dependability. Dans *Building the Information Society*, 191-212.

Vache, Géraldine. 2009. Environment characterization and system modeling approach for the quantitative evaluation of security. Dans *28th Int. Conf. on Computer Safety, Reliability and Security (SAFECOMP'09)*. Hamburg, Germany.

Washington Post. 2006. A Time to Patch. *http://blog.washingtonpost.com/securityfix/2006/01/a_time_to_patch.html*.

Whitaker, A., M. Shaw, et S. Gribble. 2002. Lightweight virtual machines for distributed and networked application. Dans *USENIX Annual Technical Conference*.

Wikipédia. Virtualisation. *http://fr.wikipedia.org/wiki/Virtualisation*.

ANNEXES TECHNIQUES

INTRODUCTION

Nous proposons de détailler dans ces trois annexes, quelques aspects techniques relatifs à l'implémentation du prototype ArSec. Dans ce prototype, nous nous sommes basés sur le premier cas d'étude, à savoir l'ordinateur de maintenance, et avons simulé le comportement de l'avion par une seconde machine, en charge de répondre aux requêtes lancées depuis l'ordinateur de maintenance.

L'annexe A présente le déroulement de l'exécution du prototype, depuis le lancement des machines virtuelles, jusqu'au déploiement des applications diversifiées, et l'instanciation de l'interface de l'opérateur.

L'instanciation de l'interface passe par une récupération des appels de fonctions d'affichage des différentes machines virtuelles (avec AspectJ). Un exemple d'une séquence de capture est fourni par un diagramme de séquences présenté dans l'annexe B.

Une fois le démonstrateur déployé, nous avons procédé à une injection d'attaques visant à évaluer la robustesse de notre démonstrateur vis-à-vis de certaines attaques réussies sur une architecture non sécurisée. Nous proposons de détailler ces attaques et les résultats obtenus dans l'annexe C.

ANNEXE A : EXECUTION DU PROTOTYPE

Nous proposons de détailler dans cette annexe, les étapes de déploiement du prototype ArSec, depuis son lancement et jusqu'à son arrêt. Par soucis de clarté, le code sur les machines virtuelles sera sur fond vert, et le code sur la machine sûre sera sur fond jaune (en respectant les codes de couleurs du chapitre VI)

L'architecture finale consiste en une application serveur sur la machine sûre, et un client sur chacune des machines virtuelles. Pour synchroniser le lancement des applications sur les machines virtuelles, nous avons implanté un serveur qui ne servira qu'à l'instanciation.

Dans le script de démarrage, nous lançons par SSH l'exécution du fichier jar sur chacune des machines virtuelles.

```
ssh root@192.168.122.132  java -jar /root/ArSec_Windows.jar
ssh root@192.168.122.20   java -jar /root/ArSec_Linux.jar
```

Les deux machines virtuelles ouvrent alors un socket en attendant l'instruction de lancement de la machine sûre :

```
Socket serveur = new ServerSocket(port, 100);
this.connexion = serveur.accept();
```

Nous lançons ensuite l'archive (*.jar*) contenant le programme serveur de la machine sûre

```
java -jar ArSec_Superviseur.jar
```

Celui-ci se connecte aux machines virtuelles pour lancer respectivement les applications

```
Socket client = new Socket(InetAddress.getByName(this.serveurDeChat),port);
ObjectOutputStream  sortie =new ObjectOutputStream(new
                        BufferedOutputStream(client.getOutputStream()));
sortie.flush();
Message m = new Message(0,0,"Application");
sortie.writeObject(m);
sortie.flush();
```

Lorsque la communication est établie, les machines virtuelles lancent l'application et ferment cette communication :

```
MainApplication application = new MainApplication();
sortie.close();
entree.close();
connexion.close();
```

Une fois les applications lancées, nous nous intéressons à l'affichage principal, réalisé au niveau de la machine virtuelle sûre (à travers l'instanciation d'objets graphiques *Java Swing*). Prenons l'exemple d'instanciation d'un cadre (*frame*). Cette instanciation étant la première instanciation d'objet graphique dans l'application.

```
public MainApplication(){

      frame = new JFrame("Application Principal");
      ...
```

L'appel à cette méthode est alors intercepté par l'aspect suivant :

```
public pointcut CaptureFrame( String name) :
call(javax.swing.JFrame.new(String))  && args(name);

JFrame around ( String name) : CaptureFrame(name) {
      this.testConnexion();
      JFrame maFrame = new JFrame(name);
      this.reseau.addFrame(maFrame);
      return maFrame;
}
```

Nous voyons que l'action définie ci-dessus va remplacer la méthode initiale (en utilisant l'action *around* d'AspectJ). La première action de ce code inséré au niveau de l'aspct est de tester si une connexion a déjà été établie sur le serveur de la machine sûre, et si ce n'est pas le cas, le faire, comme c'est le cas ici. La manière de se connecter est la même que la précédente pour le lancement de l'application.

Nous allons ensuite créer un *frame* avec le nom défini lors de l'appel méthode, puis appeler une méthode de la classe *reseau* qui s'occupe de préparer les messages à envoyer à la machine sûre :

```
// Envoie d'un message de création de frame
public void addFrame(Jframe frame){
      this.tabObject.addObject(frame,"frame");
      Message mess = new Message(0,tabObject.getObjectId(frame),"frame");
      mess.argNom = frame.getName();
      this.client.envoyerDonner(mess);
}
```

Cette méthode enregistre ce nouveau *frame* dans la liste d'objets pour pouvoir le récupérer plus tard. On prépare ensuite un nouveau message, en mettant en paramètre le nom du *frame*. Puis on l'envoie à travers le client qui s'occupe des communications entre machines virtuelles (selon le fonctionnement général présenté dans le chapitre VI).

En revenant au code précédent (celui de l'aspect) on retourne à l'application le *frame* instancié au niveau de l'aspect. Rappelons que cette opération est complètement transparente à l'application.

Du côté machine virtuelle sûre, les serveurs (un pour chaque machine virtuelle) enregistrent les messages qui arrivent dans une liste :

```
      Message message = (Message) entree.readObject();
      this.tabMessage.putMessage(message);
```

Dans le comparateur, nous avons mis une temporisation pour attendre que la phase d'instanciation soit terminée, afin d'éviter d'avoir des problèmes de synchronisme au départ.

```
      try{Thread.sleep(15000);} catch(InterruptedException e){}
```

L'application continue son exécution normale, et à chaque fois qu'une méthode graphique est appelée (une méthode héritant de *swing* ou *awt*, les deux librairies graphiques de java utilisées dans le programme), elle est capturée et un message est envoyé à la machine sûre.

Une fois la temporisation terminée, la comparaison des messages reçus commence. Nous faisons maintenant une boucle infinie en réception tant que le *socket* ne se ferme pas. Nous vérifions s'il y a un nouveau message à analyser sur le premier serveur (message non exécuté dont l'*id* est le suivant dans la liste de tel *thread*). Lorsque c'est le cas, nous cherchons le message possédant le même identifiant (*id*) sur l'autre jusqu'à épuisement du timeout. Lorsque le message est trouvé, nous les comparons, et suivant le résultat de la comparaison, nous exécutons ou non le message :

```
//Tant qu'il n'y a pas d'erreur on boucle
while (arret==false){
      //On test si il y a un nouveau message sur le premier serveur
```

```
	while (this.serveur.tabMessage.nextId()!=-1 && arret == false){
		System.out.println("Nouvel élément détecté sur serveur 1");
		int idTabMessage2 = this.serveur2.tabMessage.nextId();
		if(idTabMessage2 != -1)  arret = Analyse();
		else System.out.println("ATTENTION : Nouveau message mais pas
dans le bon ordre, attendre nouveau message sur serveur 2");
	}
	//On temporise
	try{Thread.sleep(100);} catch(InterruptedException e){}
}
```

Si les JVM n'ont pas été modifiées, l'application fonctionne correctement, et traite le message :

```
	public void TraiterMessage (Message mess){
			if(mess.natureMessage.equals("frame")) NouveauFrame(mess);
		else if(mess.natureMessage.equals("panel")) NouveauPanel();
		...
	}

	//Execution d"un nouveau frame
	private void NouveauFrame(Message mess){
		System.out.println("création d'un frame");
		JFrame monFrame = new JFrame (mess.argNom);
		tabComparateur.addObject(monFrame, "frame");
		System.out.println("Frame créé!");
	}
```

La fonction *TraiterMessage* est assez longue car elle doit pouvoir interpréter tous les messages qui arrivent, et il y en a plus de 50 possibles. Dans le code ci-dessud cas, nous présentons le traitement relatif à la réception d'un message contenant l'instruction *frame*. Dans ce cas, on crée un *frame* ayant le nom se trouvant en paramètre du message, puis on l'ajoute à la liste d'objets de la machine sûre. Nous possédons donc trois listes d'objet, une pour chaque machine, les trois listes devant être similaires.

Prenons maintenant l'exemple d'un bouton. A son instanciation, à l'instar du *frame*, un message va être envoyé à la machine sûre avec l'instruction *button*. La machine sûre exécute ce message de cette manière:

```
	//Si c'est un nouveau bouton
	} else if(mess.natureMessage.equals("button")){
		JButton monBouton = new JButton(mess.argNom);
		//On déclare ici les actions à réaliser lors d'un click
		ActionListener actionEnvoi = new ActionListener() {
			public void actionPerformed(ActionEvent e) {
				traiteEvent(e);
			}
		};
		monBouton.addActionListener(actionEnvoi);

		tabComparateur.addObject(monBouton, "button");
		System.out.println("Bouton créé!");
	}
```

Dans ce cas, il faut penser au retour, c'est-à-dire quand le manipulateur clique sur le bouton « fictif » de la machine sûre. Ce clic doit être envoyé aux machines virtuelles. Nous supposons ici qu'une action soit associée à chaque bouton, ce qui d'un premier abord parait évident. On crée donc un bouton, on lui associe un *ActionListener*, puis on l'enregistre dans la liste d'objets.

Une fois l'instanciation terminée et tous les messages traités, on peut commencer à utiliser l'application. La majorité des interactions du manipulateur avec l'application se fait par des boutons. Prenons donc l'exemple d'un clic effectué. Ceci déclenche l'*ActionListener* :

```
	public void traiteEvent(ActionEvent e){
		int id = this.tabComparateur.getObjectId(e.getSource());
		Message msgEnvoi = new Message (0,id,"boutonPresse");
		serveur.envoyerDonner(msgEnvoi);
		serveur2.envoyerDonner(msgEnvoi);
	}
```

Nous récupérons ici l'*id* du bouton qui a subi le clic, et envoyons un message aux deux machines virtuelles avec comme paramètre cet *id*. Ce message est reçu par les clients, qui analysent et exécutent le message :

```
	m=(Message)entree.readObject();
	//message lorsqu'on a cliqué sur un bouton
	if (m.natureMessage.equals("boutonPresse")){
		int id =m.idAppelant;
		JButton bouton = (JButton) tabObject.gettabobject(id);
		bouton.doClick();
	} else ...
```

Lorsque ce message est reçu au niveau de chaque machine virtuelle, on récupère, dans la liste d'objets, le bouton se situant à l'*id* indiqué, puis on exécute un clic dessus à travers la méthode *doClick()* . Ceci déclenche l'action associée à ce bouton dans l'application. Toutes les actions modifiant les éléments graphiques sont de nouveaux capturés par l'aspect qui avertit la machine sûre.

Ceci est valable pour tout ce qui correspond aux éléments *swing* et *awt*, c'est-à-dire à la sortie graphique. C'est également valable pour les sorties au niveau réseau. En effet, les machines virtuelles ne sont pas considérées comme sûres, et ne peuvent donc en aucun cas se connecter à l'avion. Il faut alors intercepter toutes les méthodes concernant une connexion à une base de données, ou une connexion via *socket*. Prenons l'exemple d'une connexion par *socket*, voici comment elle est interceptée :

```
//Création d'un socket
public pointcut newSocket(String ip, int port):
	call(java.net.Socket.new(String, int)) && args(ip,port);

Socket around(String ip, int port) : newSocket(ip, port){
	Socket monSocket = null;
	this.reseau.addSocket(ip, port, monSocket);
	return monSocket;
}
```

Nous voyons ici que nous créons un socket *null*, que nous retournons à l'application. L'application aura l' « impression » d'être connectée alors qu'en fait il n'en est rien. Un message est envoyé à la machine sûre pour lui dire de se connecter, avec les paramètres de connexions nécessaires :

```
	public void addSocket(String ip, int port, Socket monSocket){
		//On enregistre l'objet qui est null, et son id pour le
retrouver plus tard, l'objet étant null on ne peut pas le comparer
		this.tabObject.addObject(soc, "socket");
		this.tabObject.id_socket = this.tabObject.id -1;
		Message mess = new Message(0,-1,"newSocket");
		mess.argNom = ip;
		mess.argTaille = port;
		this.client.envoyerDonner(mess);
```

```
	}
```

De l'autre coté, la machine sûre exécute cette action :

```
//Reception d'un message de création de socket
	} else if(mess.natureMessage.equals("newSocket")){
		try{
			Socket soc = new Socket(mess.argNom, mess.argTaille);
			this.tabComparateur.addObject(soc, "socket");
			System.out.println("Socket ajouté avec succès!");
		} catch (Exception e){
			System.out.println("problème de création de socket");
		}
}
```

Comme pour les composants graphiques, nous créons le *socket* avec les paramètres reçus par le message avant de l'enregistrer dans la liste d'objet.

Enfin, quand l'application est fermée (en cliquant sur le bouton EXIT), cette méthode aussi est capturée, et on ferme ainsi l'application à la fois sur la machine sûre et sur les machines virtuelles :

```
public pointcut CaptureExit(int etat) : call(void
java.lang.System.exit(int)) && args(etat);

before(int etat) : CaptureExit(etat){
	this.reseau.sendExit(etat);
}
```

Avant de réaliser l'exit, on envoie un message à la machine sûre pour fermer l'application :

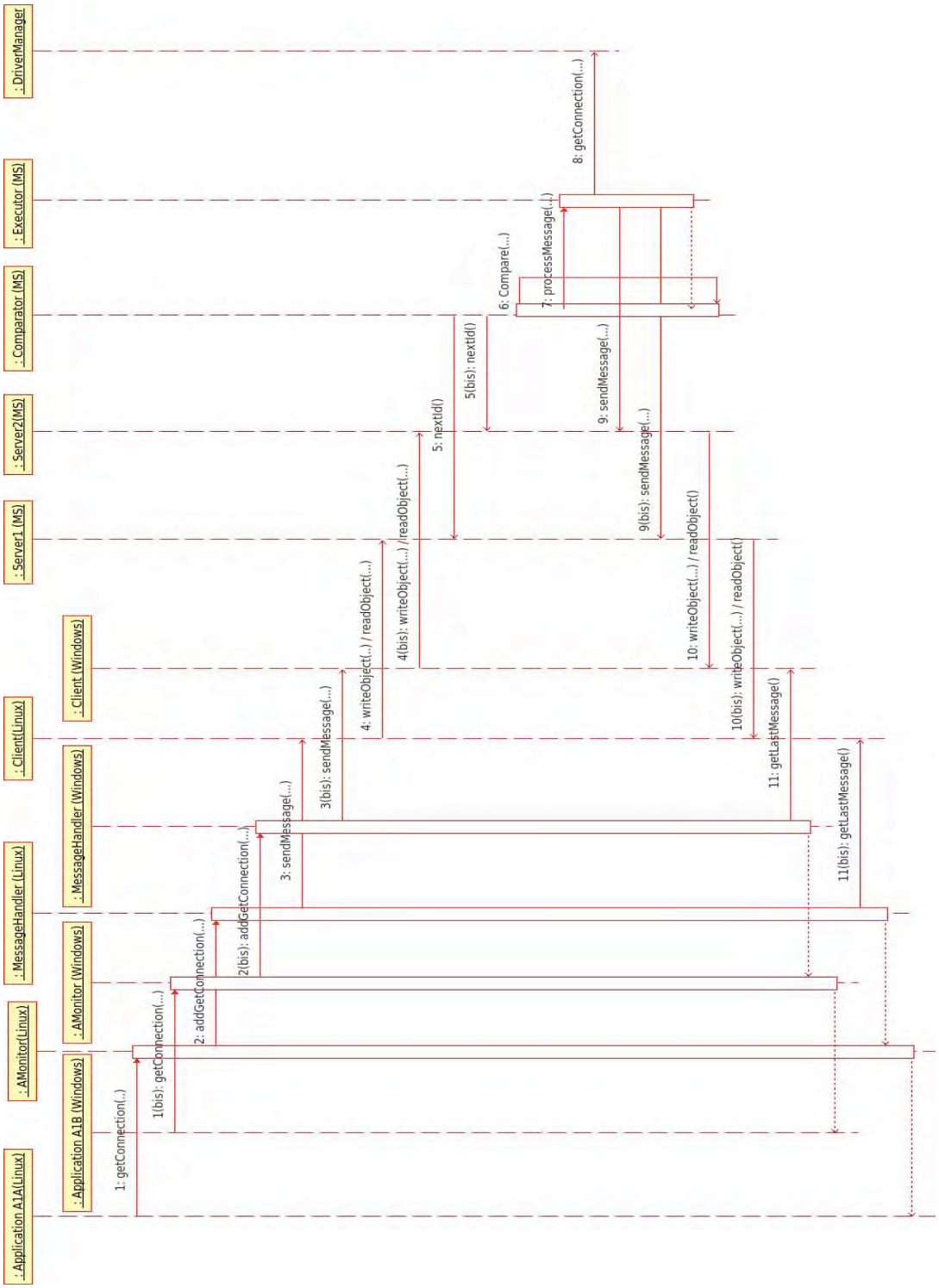
```
	public void sendExit(int status){
		Message message = new Message (0,-1,"exit");
		message.argId = status;
		this.client.envoyerDonner(message);
	}
```

De l'autre côté, le message est exécuté s'il est trouvé sur les deux serveurs :

```
//Réception d'un message pour exécuté un exit
	else if(mess.natureMessage.equals("exit")){
		System.exit(mess.argId);
	}
```

A ce moment là, tout s'arrête, l'application s'est terminée correctement.

ANNEXE 2 : EXEMPLE DE CAPTURE D'UNE FONCTION DE CONNEXION



ANNEXE 3 : EVALUATION PAR SIMULATION D'ATTAQUES

Nous venons de voir l'exécution normale du prototype. Pour ce faire, nous avons procédé par injection de fautes malveillantes. Ce ne sont pas des attaques réelles, mais elles sont imaginées pour notre démonstrateur, pour simuler ce qu'une attaque réussie sur l'une des machines virtuelles pourrait tenter de faire.

Inversion gauche/droite

Nous allons inverser les affichages gauche/droite afin d'induire l'opérateur en erreur.

Pour les tests : Lors de la réalisation de tests, par exemple sur les ailes, nous avons introduit une faute qui change « gauche » en « droite » et inversement au niveau des noms des boutons. De ce fait, lorsque l'opérateur cherche à réaliser un test sur l'aile gauche, il clique sur le bouton gauche, mais en fait, le nom aura été modifié et l'action associée à ce bouton est l'exécution d'un test sur l'aile droite.

Dans la liste des erreurs : Le manipulateur a accès au PFR (*Post Flight Report*) où sont consignées toutes les erreurs et alertes survenu durant le vol. Il pourra ainsi voir tout ce qu'il a à réparer. J'ai donc modifié une JVM afin que toutes les entrées contenant « gauche » soient transformées en « droite » et inversement. Ainsi, lorsque le manipulateur verra une succession d'erreur sur l'aile droite par exemple, il voudra réparer à droite, alors que le problème est à gauche.

De telles fautes pourraient avoir de graves conséquences, l'opérateur ne détectant pas des fautes qui resteront non résolues. Bien évidemment, si l'on souhaite réaliser une attaque de ce type, il faudrait choisir une des deux options, sinon les deux fautes se masqueraient l'une l'autre : on verrait une erreur sur l'aile gauche par exemple (alors que le problème est sur l'aile droite), et on demanderait donc de réaliser un test sur l'aile gauche, mais ce test serait réalisé sur l'aile droite, et la panne serait bien détectée.

Effacer les erreurs les plus graves

En continuant sur le *Post Flight Report*, où sont listées les erreurs, les informations présentes sont : la date, l'heure, la description de la faute et sa sévérité. La sévérité s'étend entre 0 et 5 (5 étant le niveau de sévérité maximum). Nous pouvons donc imaginer qu'une attaque pourrait être de masquer les erreurs les plus graves. Ainsi l'opérateur ne les verra pas, et ne les réparera donc pas, ce qui pourrait être très critique. Nous avons donc modifié une nouvelle JVM, et les lignes de la liste ayant une sévérité égale à 5 ne sont pas affichées par cette JVM.

Détection de fautes

Nous avons décrit comment introduire les fautes, et quel type de fautes nous allons tester, maintenant voyons comment cela se passe en pratique.

Nous allons d'abord tester l'application seule, indépendamment. Nous allons l'exécuter en utilisant une JVM modifiée, par exemple celle qui inverse gauche/droite. Si le manipulateur cherche à consulter la liste des erreurs, il obtiendra cela :

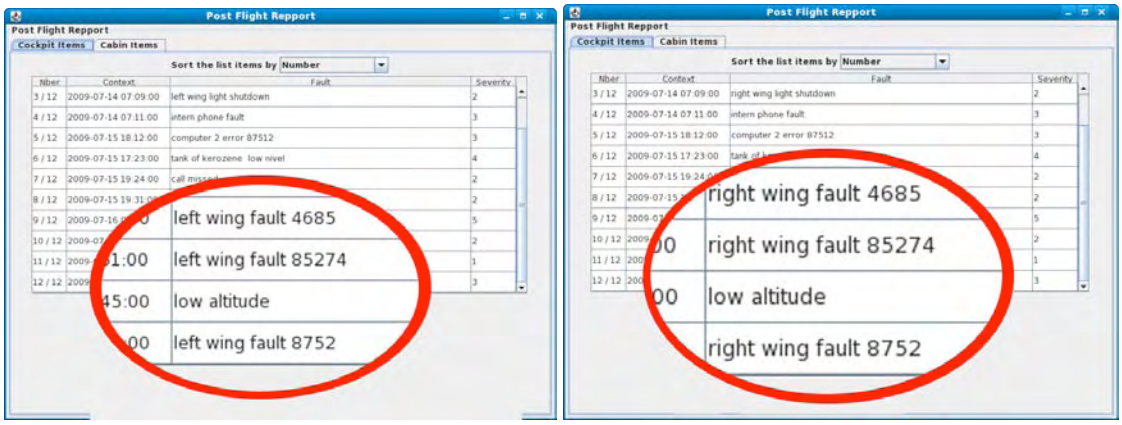


Figure C.1 - Liste d'erreurs modifiée droite/gauche

À gauche de la figure C.1, il y a la liste normale, où l'on peut voir qu'il y a des erreurs sur l'aile gauche. Et à droite, on voit la même liste, mais disant que l'erreur se situe à droite, ce qui est une fausse information générée par la JVM. Le manipulateur va donc penser qu'il y a une erreur sur l'aile droite. Il va donc vérifier cela en lançant des tests sur cette aile.

Comme la JVM est modifiée, si nous faisons un test sur l'aile droite, nous voyons que l'avion lance le test sur l'aile gauche, mais il n'y a eu aucun disfonctionnement de l'application, elle continue à fonctionner normalement (cf. figure C.2).

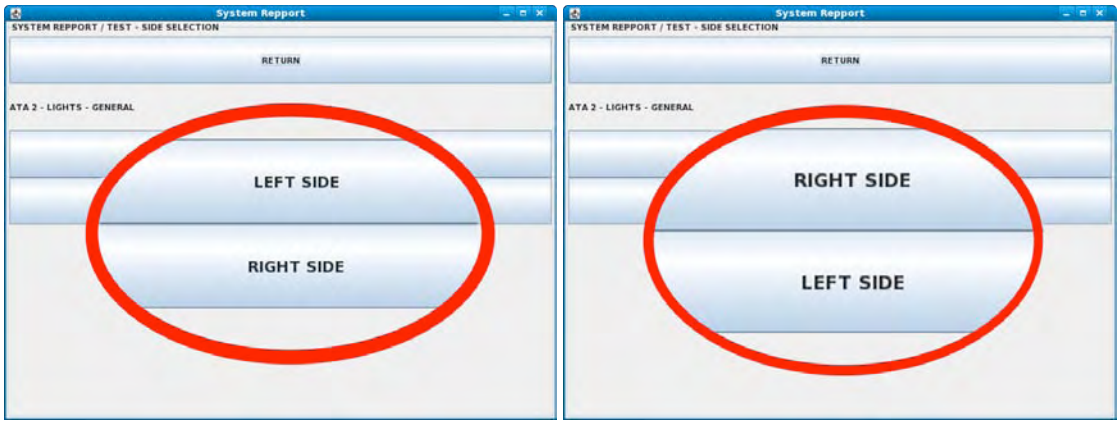


Figure C.2 - Modification des boutons

Ainsi, si le manipulateur effectue ce test, il vérifie que l'aile fonctionne bien, et peut ne pas voir réellement les équipements défaillants.

Si nous prenons le cas de l'autre JVM modifiée, celle qui efface les erreurs les plus graves, le manipulateur obtiendra :

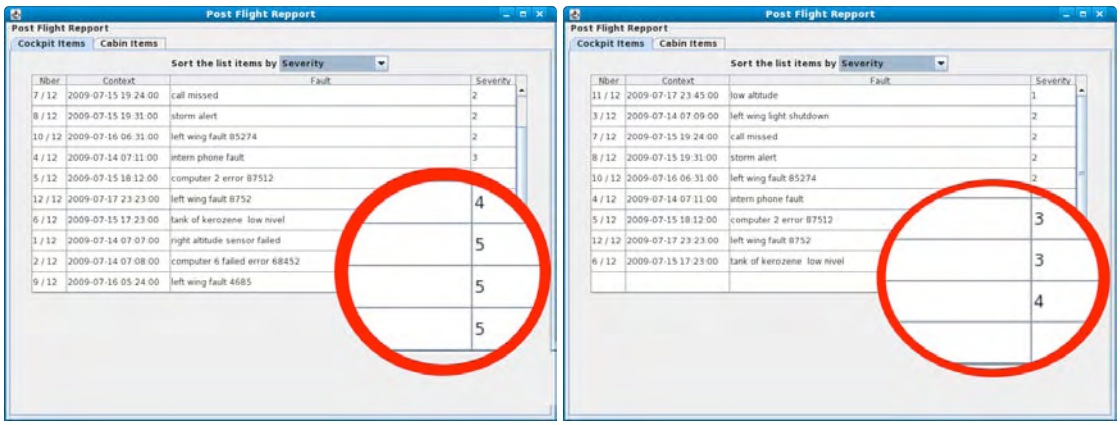


Figure C.3 - Liste d'erreurs modifiée, erreurs graves effacées

A gauche de la figure C.3, nous avons la liste originale, avec trois erreurs de niveau 5. A droite, nous avons la liste avec la JVM modifiée, nous voyons qu'après l'erreur de niveau 4, il n'y a rien, les erreurs les plus graves ont disparu. Si une attaque de ce type venait à se réaliser, l'agent de maintenance ne verrait pas les fautes les plus graves, et donc ne les résoudrait pas, ce qui pourrait avoir de graves conséquences.

Nous venons de voir comment nous pouvons provoquer de graves erreurs de fonctionnement de l'application si un attaquant arrivait à modifier la JVM. Faisons alors le test avec l'aspect et la duplication. La JVM de la machine sûre est certifié, et donc son fonctionnement ne peut pas être remis en cause. Modifions la JVM d'une des machines virtuelles avec celle inversant gauche/droite. Refaisons les mêmes actions que précédemment, affichons la liste des erreurs. Cette fois-ci la liste n'apparaît pas et un message d'erreur nous signale une incohérence :

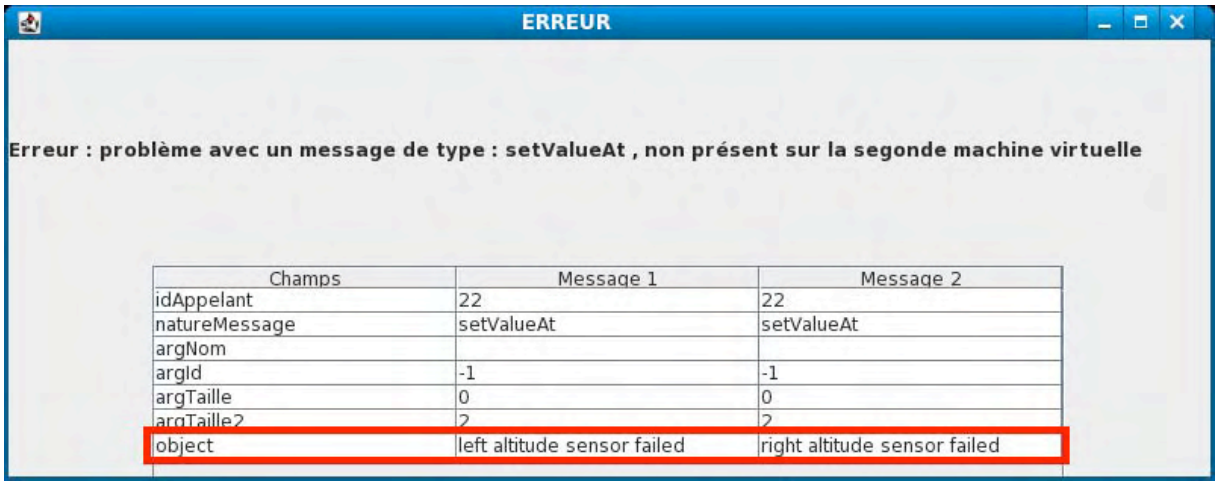


Figure C.4 - Message d'erreur lors de l'affichage de la liste d'erreurs

Nous voyons qu'un message informe d'une défaillance à gauche et l'autre à droite pour la même case du tableau (tous les autres paramètres sont égaux). L'erreur a bien été détectée, et le programme s'est arrêté, comme le montre la figure C.4.

De même si nous avions voulu lancer un test sur une aile, lors de l'affichage des boutons de test, une autre erreur aurait été détectée :

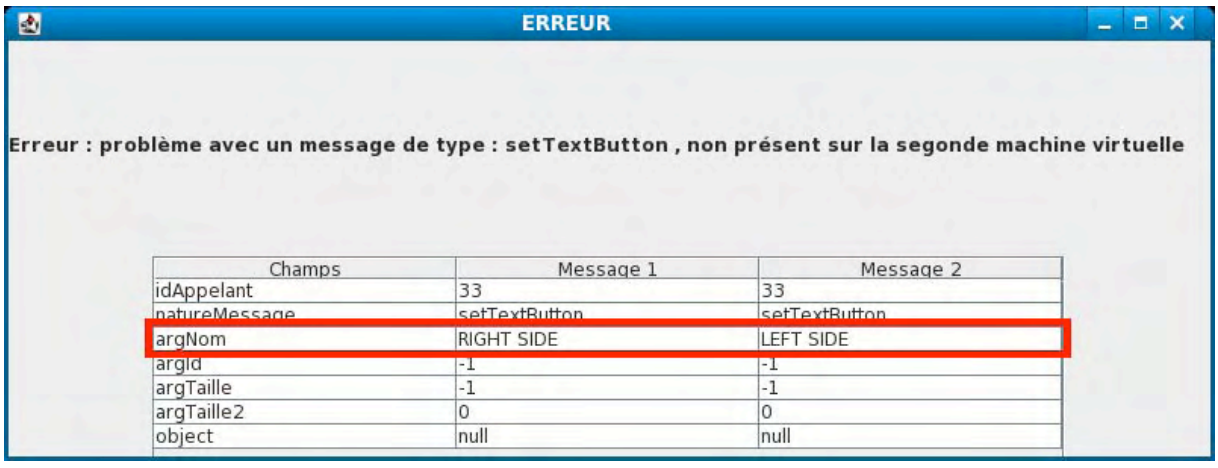


Figure C.5 - Message d'erreur lors de l'affichage des boutons de tests

Là aussi nous voyons qu'il y a un problème entre les deux messages, un voulant nommer un bouton avec gauche, et l'autre avec droite. Là aussi, le programme s'arrête de lui-même.

Dans l'autre cas, si la JVM avait été modifiée afin de ne pas afficher les erreurs les plus graves, une erreur se serait déclenchée :

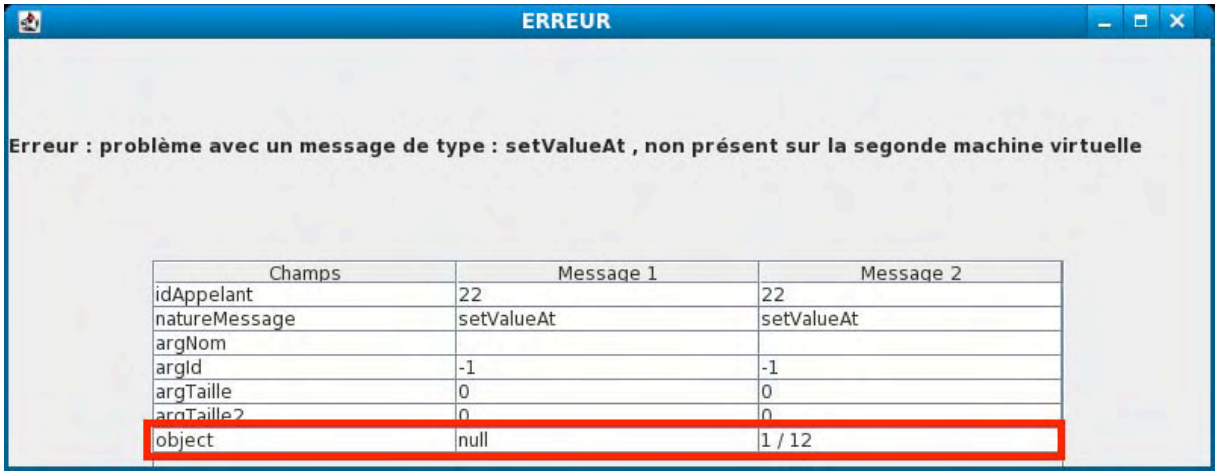


Figure C.6- Message d'erreur lors de l'affichage de la liste d'erreurs

Dans le message de la figure C.6, on voit qu'un message veut commencer à écrire dans le tableau (Message 2) en écrivant « 1/12 », et l'autre nom (Message 1 paramètre objet à *null*). Ce qui se passe, c'est que la machine non corrompue veut afficher la première ligne, alors que la machine corrompue non. En effet, la première ligne correspond à un degré de sévérité égal à 5, et la machine corrompue n'affiche pas ces lignes-là.

À chaque fois qu'une erreur se produit, l'opérateur est averti, et donc aucune action erronée n'atteint l'avion. Nous voyons donc que la méthode de protection développée ici est efficace face à ce type d'erreur.

*****

Nous proposons de montrer dans ce qui suit comment nous avons procédé pour modifier la JVM.

Dans la JVM, un fichier nommé *rt.jar* contient les différents packages que nous utilisons lors de la phase de développement, tels que : *javax.swing*, *java.io*, etc. Ce fichier contient l'ensemble des *bytecodes* associés. Mais il est possible de voir le code source de ces fichiers. Par exemple, lorsque nous utilisons un *ObjectOutputStream* pour envoyer un objet sur le réseau grâce à la fonction *writeObject(obj)*. Lorsqu'on fait un « *open declaration* » dans l'environnement de développement intégré Eclipse sur cette fonction, le fichier *ObjectOutputStream.class* situé dans rt.jar s'ouvre sur cette fonction :

```
public final void writeObject(Object obj) throws IOException {
        if (enableOverride) {
            writeObjectOverride(obj);
            return;
        }
        try {
            writeObject0(obj, false);
        } catch (IOException ex) {
            if (depth == 0) {
                writeFatalException(ex);
            }
            throw ex;
        }
}
```

Nous recopions tout le fichier dans un nouveau fichier que nous nommerons *ObjectOutputStream.java*. Puis nous modifions cette fonction. Pour envoyer des commandes de test à l'avion, nous envoyons simplement des strings à l'avion contenant les instructions. A ce niveau aussi il est possible de changer droite en gauche et inversement, mais cette fois au niveau de l'envoie d'instruction. Voilà comment il faudrait réaliser la modification :

```
public final void writeObject(Object obj) throws IOException {
        System.out.println("On fait un write Objects");
        String compare = "test";
        //On test si l'objet est un string
        if(obj.getClass().equals(compare.getClass())){
                System.out.println("C'est un string");
                String message = (String) obj;
                if(message.contains("Droite")){
                        String newMessage = message.replaceAll("Droite", "Gauche");
                        System.out.println("Droite remplacer : " + newMessage);
                        obj = newMessage;
                } else {
                        String newMessage = message.replaceAll("Gauche", "Droite");
                        System.out.println("Gauche remplacer : " + newMessage);
                        obj = newMessage;
                }
        } else System.out.println("C'est pas un string");
        if (enableOverride) {
                writeObjectOverride(obj);
        return;
        }
        try {
                writeObject0(obj, false);
        } catch (IOException ex) {
                if (depth == 0) {
                        writeFatalException(ex);
                }
                throw ex;
        }
}
```

Une fois cette étape réalisée, il faut le compiler (grâce à la commande *javac*). Ensuite il faut chercher où se situe la JRE (*Java Runtime Environment*) qui est utilisé comme JVM. Dans les fichiers de la JRE se trouve rt.jar. Il faut de décompresser (avec la commande *unzip*), puis chercher les fichiers compiler correspondant a *ObjectOutputStream*, les supprimer, et enfin les remplacer par les fichiers obtenus à la compilation. Il ne reste plus qu'à re-compresser (avec la commande *zip*), et remplacer l'ancien rt.jar par le nouveau ainsi obtenu.

Nous pouvons maintenant lancer notre application. Si l'on n'y prête pas attention, nous ne remarquerons aucune différence. Mais on se rend compte que lorsqu'on lance un test sur l'aile gauche, l'aile droite va s'afficher sur l'avion – et inversement -, ce qui est incohérent. Et ce qui prouve que la modification de la JVM a fonctionné.