Approche d'intégrité bout en bout pour les communications dans les systèmes embarqués critiques : application aux systèmes de commande de vol d'hélicoptères

Amira Zammali

THÈSE

En vue de l'obtention du

DOCTORAT DE L'UNIVERSITÉ DE TOULOUSE

Délivré par :

Université Toulouse 3 Paul Sabatier (UT3 Paul Sabatier)

Présentée et soutenue par :
Amira ZAMMALI

le mercredi 13 janvier 2016

Titre :

Approche d'intégrité bout en bout pour les communications dans les systèmes embarqués critiques : application aux systèmes de commande de vol d'hélicoptères

---

École doctorale et discipline ou spécialité :

EDSYS : Informatique et Systèmes Embarqués

Unité de recherche :

Laboratoire d'Analyse et d'Architecture des Systèmes (LAAS-CNRS)

Directeur/trice(s) de Thèse :

Agnan de BONNEVAL, Maître de conférences, Université Toulouse 3
Yves CROUZET, Chargé de Recherche CNRS, LAAS-CNRS

Jury :

Lorena ANGHEL, Professeur, INP de Grenoble, Phelma, Présidente
Simon COLLART-DUTILLEUL, Directeur de Recherche, IFSTTAR, Rapporteur
Pedro GIL VICENTE, Professeur, Universidad Politécnica de Valencia, Rapporteur
Jean-Paul BLANQUART, Ingénieur, Airbus Defence and Space, Examinateur
Pascal IZZO, Docteur-Ingénieur, Airbus Helicopters, Examinateur
Yann LABIT, Professeur, Université Toulouse 3, Examinateur

À la mémoire de ma mère…

Je te remercie maman pour tout l’amour que tu m’as donné
et pour toutes les bonnes valeurs que tu m’as apprises.
Tu resteras à jamais vivante dans mon cœur. Je t’aime.

AVANT-PROPOS

Les travaux présentés dans ce mémoire ont été menés au *Laboratoire d'Analyse et d'Architecture des Systèmes du Centre National de la Recherche Scientifique* (LAAS-CNRS) à Toulouse.

Je tiens à remercier Monsieur Jean Arlat, directeur du LAAS-CNRS durant ma thèse, de m'avoir permis d'accomplir mes travaux de thèse au sein de ce laboratoire.

Je remercie également Madame Karama Kanoun et Monsieur Mohamed Kaaniche, responsables successifs de l'équipe de recherche *Tolérance aux fautes et Sûreté de Fonctionnement informatique* (TSF), pour m'avoir permis de mener mes travaux de thèse dans cette équipe. Je les remercie chaleureusement d'avoir suivi mes travaux et pour tous leurs conseils précieux, et pour leurs relectures de mes articles. Un remerciement spécial pour Monsieur Mohamed Kaaniche pour sa relecture de mon manuscrit.

Je tiens à remercier mes encadrants, Agnan de Bonneval, maître de conférences à Université Toulouse III - Paul Sabatier, et Yves Crouzet, chargé de recherche au CNRS, pour m'avoir encadrée et soutenue tout au long de cette thèse. Je les remercie pour leurs conseils et leur disponibilité ainsi que toutes les discussions enrichissantes que nous avons eues tout au long de la thèse.

Je remercie l'école doctorale *Systèmes* de Toulouse de m'avoir permis d'obtenir un financement de contrat doctoral attribué par l'Université Toulouse III - Paul Sabatier.

Une mention particulière à l'équipe *Architecture des Systèmes de Commande de Vol d'Airbus Helicopters* pour m'avoir permis d'appliquer mes propositions sur le cas d'étude des futurs systèmes de commande de vol d'Airbus Helicopters. Je remercie particulièrement Monsieur Pascal Izzo qui m'a beaucoup aidée pour la compréhension du cas d'étude, merci pour tous les échanges enrichissants et les conseils précieux. Je remercie également Monsieur Cédric Landet (Atos, Externe Airbus Helicopters) pour m'avoir aidée pour mes expérimentations et pour ses réponses à mes très nombreuses questions concernant la plateforme d'expérimentation et les processeurs embarqués utilisés par Airbus Helicopters.

Je remercie Madame Lorena Anghel, Professeur à *l'Institut National Polytechnique de Grenoble*, qui m'a fait le très grand honneur de présider mon jury.

Je remercie également Monsieur Simon Collart-Dutilleul, Directeur de recherche à *l'IFSTTAR*, et Monsieur Pedro Gil Vicente, Professeur à *l'Université Polytechnique de Valence (Espagne)* pour avoir accepté la lourde tâche de rapporter mes travaux de thèse. Enfin, je remercie Monsieur Jean-Paul Blanquart, Ingénieur en sûreté de fonctionnement à *Airbus Defence and Space*, Monsieur Pascal Izzo, Docteur-Ingénieur à *Airbus Helicopters* et Monsieur Yann Labit, Professeur à *l'Université Toulouse III - Paul Sabatier*, pour avoir accepté d'examiner mes travaux.

Je remercie tous les permanents, doctorants, post-doctorants et stagiaires de l'équipe TSF que j'ai côtoyés pendant ma thèse. J'ai eu la chance de travailler avec des personnes formidables et très enrichissantes sur le plan scientifique et professionnel, ainsi que sur le plan personnel. Merci pour l'excellente ambiance de travail qui contribue à mener à bien les activités de recherche au sein de l'équipe.

Je remercie les secrétaires successives du thème *Informatique Critique du LAAS-CNRS*, Sonia Vierge et Caroline Malé, pour leur disponibilité et pour m'avoir aidée dans les différentes tâches administratives.

Une grande pensée à mes amis, ma famille et mon mari qui m'ont beaucoup soutenue et encouragée tout au long de la thèse.

TABLE DES MATIÈRES

TABLE DES MATIÈRES ..... **i**

LISTE DES FIGURES ..... **v**

LISTE DES TABLEAUX ..... **vii**

GLOSSAIRE ..... **viii**

INTRODUCTION **1**

CHAPITRE I CONTEXTE GÉNÉRAL ET CONCEPTS FONDAMENTAUX **5**

I.1 SYSTÈMES EMBARQUÉS CRITIQUES 6
I.1.1 Caractéristiques et exigences des systèmes embarqués critiques 6
I.1.2 Domaines d'application et exemple d'illustration 7
I.2 SÛRETÉ DE FONCTIONNEMENT : PRINCIPAUX CONCEPTS 9
I.2.1 Attributs 9
I.2.2 Entraves 10
I.2.2.1 Classification de base *10*
I.2.2.2 Corrélation et relativité des définitions de base *11*
I.2.3 Moyens 12
I.2.4 Sûreté de fonctionnement et certification 13
I.3 PROBLÉMATIQUE DE L'INTÉGRITÉ DES COMMUNICATIONS 15
I.3.1 Périmètre de l'intégrité des communications 15
I.3.1.1 Attributs d'une communication intègre *15*
I.3.1.2 Intégrité de bout en bout *16*
I.3.2 Corruption des communications : classification et causes d'erreurs 17
I.3.3 La tolérance aux fautes pour l'intégrité des communications 18
I.3.3.1 Redondance symétrique ou asymétrique (diversification) *19*
I.3.3.2 Granularité de la redondance *19*
I.3.3.3 Redondance statique (ou active), dynamique (ou passive) et hybride *19*
I.3.3.4 Redondance temporelle ou spatiale *20*
1) Redondance temporelle 20
2) Redondance spatiale 21
I.3.3.5 Exemples d'architectures redondantes *22*
I.3.3.6 Conclusions et redondances retenues pour l'intégrité des communications *23*
I.4 LES CODES DÉTECTEURS D'ERREURS POUR L'INTÉGRITÉ DES COMMUNICATIONS 24
I.4.1 Étude générique : description générale des codes détecteurs et des différentes familles de codes 24
I.4.2 Classification des codes détecteurs d'erreurs 25
I.4.2.1 Les codes en blocs *25*
1) Les codes en blocs linéaires/non linéaires 25
2) Les codes en blocs linéaires cycliques/non cycliques 26
I.4.2.2 Les codes concaténés *26*
I.4.2.3 Les codes convolutifs *26*
I.5 CONCLUSION 26

CHAPITRE II TRAVAUX CONNEXES ET SOLUTIONS POUR L'INTÉGRITÉ DES COMMUNICATIONS 27

II.1 PRINCIPAUX CODES DÉTECTEURS D'ERREURS POUR L'INTÉGRITÉ DES COMMUNICATIONS 28
II.1.1 Les codes CRC 28
II.1.2 Fletcher 29
II.1.3 Adler 29
II.1.4 WSC 29
II.1.5 XOR 30
II.1.6 Le complément à 1 30
II.1.7 Le complément à 2 30
II.1.8 Code de parité 30
II.2 L'INTÉGRITÉ DES COMMUNICATIONS DANS LES RÉSEAUX EMBARQUÉS 31
II.2.1 ARINC 429 31
II.2.2 MIL-STD-1553 31
II.2.3 AFDX 31
II.2.4 TTP/C 32
II.2.5 LIN 32
II.2.6 Flexray 32
II.2.7 CAN 33
II.3 SOLUTIONS BOUT EN BOUT POUR L'INTÉGRITÉ DES COMMUNICATIONS 34
II.3.1 Canal noir versus Canal blanc 34
II.3.2 PROFIsafe 35
II.3.3 FSoE 36
II.3.4 openSAFETY 36
II.4 TRAVAUX ACADÉMIQUES SUR L'INTÉGRITÉ DES COMMUNICATIONS 37
II.4.1 Approches basées sur les codes détecteurs d'erreurs 37
II.4.1.1 Utilisation des CRC (à différents niveaux) *37*
II.4.1.2 Approche d'intégrité évolutive basée sur les CRC *39*
II.4.1.3 Utilisation des sommes de contrôle arithmétiques *40*
II.4.1.4 Les codes flexibles *42*
II.4.2 Approches basées sur la redondance matérielle 42
II.4.3 Approches basées sur la diversification des données 43
II.4.3.1 Description générale de la diversification des données *43*
II.4.3.2 Diversification des données et intégrité des communications *44*
II.5 DISCUSSION ET ORIENTATION DE NOS TRAVAUX 45
II.6 CONCLUSION 46

CHAPITRE III APPROCHE D'INTÉGRITÉ BOUT EN BOUT BASÉE SUR LES CODES DÉTECTEURS D'ERREURS 47

III.1 PRINCIPE GÉNÉRAL DE L'APPROCHE D'INTÉGRITÉ BOUT EN BOUT 48
III.1.1 Approche d'intégrité bout en bout : présentation 48
III.1.2 Systèmes « contraignants » ou « très contraignants » : définitions 49
III.1.3 Les deux approches proposées : mono-code et multi-codes 49
III.1.4 Stratégie de sélection des codes 50
III.2 APPROCHE D'INTÉGRITÉ MONO-CODE 51
III.2.1 Limitations des approches existantes 51
III.2.2 Démarche de choix des codes détecteurs d'erreurs à évaluer 51
III.3 ÉVALUATIONS POUR L'APPROCHE MONO-CODE 53
III.3.1 Contexte général 53
III.3.2 Évaluation du pouvoir de détection des codes 54
III.3.2.1 Environnements d'évaluation *54*
1) Environnement logiciel basé sur Matlab-Simulink 54
2) Environnement logiciel en langage C 55
III.3.2.2 Scénarios mis en place *56*
1) Scénario n°1 : simulations Matlab avec génération exhaustive 56
2) Scénario n° 2 : simulations Matlab avec générations aléatoires 56
3) Scénario N°3 : simulations C avec générations aléatoires 57
III.3.2.3 Résultats préliminaires sur Matlab-Simulink *57*
III.3.2.4 Résultats étendus dans l'environnement d'évaluation en C *59*
1) Intervalles de confiance et choix du nombre des cas de test 59
2) Résultats des évaluations des codes de 8, 16 et 32 bits 61
3) Intervalles de confiance : calcul a posteriori 63

III.3.3 Évaluations du coût de calcul des codes présélectionnés 64
III.3.3.1 Problématique de l'évaluation des temps de calcul *64*
III.3.3.2 Environnement d'évaluation du temps de calcul *64*
1) L'outil « PyCRC » pour les codes CRC 65
2) L'outil « Universal CRC » pour les codes CRC 65
3) L'outil pour les codes Adler et Fletcher 66
III.3.3.3 Fonction de mesure des temps d'exécution, métriques et scénarios ciblés *66*
1) Fonctions clock() et times() 66
2) Fonction et stratégie de mesure 67
III.3.3.4 Évaluations sur un environnement matériel standard *69*
1) Environnement matériel et compilateur 69
2) Algorithmes implémentés et principes des scénarios évalués 69
3) Scénarios « sans favorisation du cache » 70
4) Scénarios « avec favorisation du cache » 72
III.3.3.5 Évaluations sur un environnement matériel embarqué (carte STM32) *75*
1) Description générale de la plateforme et de l'environnement 75
2) Mesures du temps 76
3) Évaluations : scénarios et résultats présentés 76
III.3.3.6 Comparaison des résultats sur environnements matériels standard et embarqué *78*
III.4 APPROCHE D'INTÉGRITÉ MULTI-CODES 79
III.4.1 Caractéristiques des systèmes ciblés 79
III.4.2 Problème du mode commun de défaillance 80
III.4.3 Approche multi-codes et complémentarité des codes détecteurs d'erreurs 81
III.5 ÉVALUATIONS POUR L'APPROCHE MULTI-CODES 83
III.5.1 Contexte général 83
III.5.2 Résultats des évaluations de la complémentarité des codes 84
III.5.2.1 Scénario n°1 : approche bi-codes combinant des codes 8 bits *84*
III.5.2.2 Scénario n°2 : approche tri-codes combinant des codes 8 bits *85*
III.5.2.3 Scénario n°3 : approche bi-codes combinant des codes 16 bits *85*
III.5.3 Conclusion sur l'approche multi-codes 86
III.6 CONCLUSION 87

CHAPITRE IV APPLICATION : FUTURS SYSTÈMES DE COMMANDE DE VOL D'AIRBUS HELICOPTERS 89

IV.1 SYSTÈME DE COMMANDE DE VOL : DESCRIPTION GÉNÉRALE 90
IV.1.1 Rôle du système de commande de vol 90
IV.1.2 Propriétés d'un système de commande de vol Hélicoptère 91
IV.2 CDV : ÉTUDE DE L'EXISTANT 92
IV.2.1 Historique des CDV : du mécanique au Fly By Wire (FBW) 92
IV.2.1.1 Le tout mécanique *92*
IV.2.1.2 Le concept de Fly By Wire *92*
IV.2.1.3 Les évolutions du Fly By Wire dans le monde avion *92*
IV.2.1.4 Les évolutions du Fly By Wire dans le monde hélicoptère *93*
IV.2.2 Architectures actuelles de référence des CDVE 94
IV.2.3 Architecture de base d'un CDVE hélicoptère 95
IV.2.4 Architecture du CDVE du NH90 : description et limites 95
IV.3 ARCHITECTURE DES FUTURS CDVE D'AIRBUS HELICOPTERS 97
IV.3.1 Description générale 97
IV.3.2 Politique d'intégrité des communications 100
IV.3.2.1 Cadre général *100*
IV.3.2.2 Processus de détection d'erreurs dans les communications internes *101*
IV.3.2.3 Processus de détection d'erreurs dans les communications externes *102*
IV.4 ENVIRONNEMENT D'EXPÉRIMENTATION 104
IV.4.1 Choix des processeurs 104
IV.4.1.1 Critères fonctionnels *104*
IV.4.1.2 Critères non-fonctionnels *105*
IV.4.1.3 Obsolescence des composants *105*
IV.4.1.4 Les processeurs candidats et retenus *105*
IV.4.1.5 Caractéristiques détaillées des TMS320C6657 et P2020 ..... *106*
IV.4.2 Cartes d'évaluation 107
IV.4.3 Description de la plateforme d'expérimentation 108
IV.5 CODE DÉTECTEUR BOUT EN BOUT POUR LES FUTURS CDV D'AIRBUS HELICOPTERS 109
IV.5.1 Hypothèses et méthodologie des expérimentations 109

IV.5.1.1 Codes détecteurs évalués *109*
IV.5.1.2 Métrique des expérimentations : le temps d'exécution *109*
IV.5.1.3 Algorithmes et langage d'implémentation *110*
IV.5.1.4 Démarche, Méthodologie pour les expérimentations *110*
IV.5.2 Évaluations sans benchmark pour le choix de la longueur des codes considérés 111
IV.5.2.1 Hypothèses *112*
IV.5.2.2 Constats *112*
IV.5.2.3 Interprétations et conclusions *113*
IV.5.2.4 Conclusion globale *114*
IV.5.3 Évaluations des temps d'exécution des codes 32 bits sans benchmarks 114
IV.5.4 Expérimentations avec benchmarks 116
IV.5.4.1 Benchmarks unitaires *117*
IV.5.4.2 Benchmark intermédiaire *119*
IV.5.4.3 Benchmark complet *120*
IV.5.5 Complémentarité entre le code détecteur applicatif retenu et le CRC HDLC 123
IV.6 CONCLUSION 124

CONCLUSIONS ET PERSPECTIVES 125

RÉFÉRENCES BIBLIOGRAPHIQUES 131

ANNEXE 1 : UTILISATION DE TABLES DE CORRESPONDANCE 139

ANNEXE 2 : UTILISATION DE PYCRC ET UNIVERSAL CRC POUR LA MESURE DU TEMPS DE CALCUL DES CODES CRC 141

II.1 Commandes utilisées au niveau de pycrc et Universal CRC 141
II.1.1 Cas de l'outil « PYCRC » *141*
II.1.2 Cas de l'outil « Universal CRC » *141*
II.2 Shells UNIX pour la génération automatique des expériences 142
II.2.1 Cas de l'outil « PYCRC » *142*
II.2.2 Cas de l'outil « Universal CRC » *144*
II.3 Programme de traitement des fichiers bruts 145

ANNEXE 3 : MODÈLES SIMULINK DE L'APPROCHE MULTI-CODES 147

LISTE DES FIGURES

Figure I.1 : Système de commande de vol avionique et les surfaces contrôlées d'un avion 8
Figure I.2 : Sûreté de fonctionnement : vue d'ensemble 9
Figure I.3 : Relation entre les trois types d'entraves de la sûreté de fonctionnement (chaîne de cause à effet).... 12
Figure I.4 : Contrôle d'intégrité de bout en bout 16
Figure I.5 : Classification des erreurs affectant les messages échangés 17
Figure I.6 : Différents types de redondance dans les systèmes embarqués critiques 20
Figure I.7 : Redondance temporelle et détection d'erreurs 20
Figure I.8 : Le principe de codage dans les codes détecteurs d'erreurs 24
Figure I.9 : Classification des codes détecteurs d'erreurs 25
Figure II.1 : Principe de la division polynômiale des codes CRC 28
Figure II.2 : Les concepts « canal noir/canal blanc » dans la norme IEC 61508 34
Figure II.3 : Approche d'intégrité évolutive 39
Figure II.4 : Complémentarité des codes CRC 40
Figure II.5 : Découpage de la trame en plusieurs zones 42
Figure II.6 : Principe général de réexpression de données 43
Figure II.7 : Mise en œuvre « Bloc de réessaie » de la diversification de données 44
Figure II.8 : Mise en œuvre « Programmation n-copies» de la diversification de données 44
Figure III.1 : Principe de l'approche d'intégrité bout en bout basée sur le concept du« canal noir » 48
Figure III.2 : Critères de sélection de codes pour les deux approches d'intégrité 50
Figure III.3 : Exemple d'un modèle d'évaluation Matlab-Simulink 55
Figure III.4 : Pud des codes de taille 8 bits (charges utiles de 8, 16 et 32 octets) 62
Figure III.5 : Pud des codes de taille 16 bits (charges utiles de 8, 16 et 32 octets) 62
Figure III.6 : Pud des codes de taille 32 bits (charges utiles de 8, 16 et 32 octets) 63
Figure III.7 : Fonction de mesure du temps de calcul 68
Figure III.8 : Temps de calcul (tick) « sans favorisation du cache » des codes 8 bits 70
Figure III.9 : Temps de calcul (tick) « sans favorisation du cache » des codes 16 bits 72
Figure III.10 : Temps de calcul (tick) « sans favorisation du cache » des codes 32 bits 72
Figure III.11 : Temps de calcul (tick) « avec favorisation du cache » des codes 8 bits 73
Figure III.12 : Temps de calcul (tick) « avec favorisation du cache » des codes 16 bits 73
Figure III.13 : Temps de calcul (tick) « avec favorisation du cache » des codes 32 bits 74

Figure III.14 : Temps de calcul « sans et avec favorisation du cache » et « avec favorisation du cache » pour les codes 32 bits avec 8 octets de charge utile....75
Figure III.15 : Carte d'évaluation STM3241G-EVAL du processeur STM32F417....77
Figure III.16 : Temps d'exécution sur STM32 avec deux options de compilation....77
Figure III.17 : Système à dynamique rapide VS système à dynamique lente....80
Figure III.18 : Problème du mode commun de défaillance dans l'approche mono-code....81
Figure III.19 : Principe général de l'approche multi-codes....81
Figure III.20 : Principe de complémentarité entre deux codes détecteurs d'erreurs....82
Figure III.21 : Taux de non détection de l'approche multi-codes, scénario de 2 codes de 8 bits....84
Figure III.22 : Taux de non détection de l'approche multi-codes, scénario de 3 codes de 8 bits....85
Figure III.23 : Taux de non détection de l'approche multi-codes, scénario de 2 codes de 16 bits....86
Figure IV.1 : Les quatre forces appliquées à un hélicoptère....90
Figure IV.2 : Les trois organes de commandes d'un hélicoptère....91
Figure IV.3 : CDV Hélicoptère conventionnelles Versus CDV Hélicoptère électriques (« Full FBW », NH90) .93
Figure IV.4 : L'architecture du système de commande de vol du NH 90....96
Figure IV.5 : Les liens de communication FCP vers FCP et FCP vers ACP....99
Figure IV.6 : Les liens de communication ACP vers ACP et ACP vers FCP....99
Figure IV.7 : Format de la trame EIA-485....100
Figure IV.8 : Étape 1 du processus interne de détection d'erreurs....101
Figure IV.9 : Étape 2 du processus interne de détection d'erreurs....101
Figure IV.10 : Étape 3 du processus interne de détection d'erreurs....102
Figure IV.11 : Étape 4 du processus interne de détection d'erreurs....102
Figure IV.12 : Étape 5 du processus interne de détection d'erreurs....102
Figure IV.13 : Détection d'erreurs dans les communications inter-calculateurs....103
Figure IV.14 : Détection d'erreurs dans les communications externes....103
Figure IV.15 : Photos des deux cartes d'évaluation....107
Figure IV.16 : Plateforme d'expérimentations (cas du P2020)....108
Figure IV.17 : WCMT sur P2020 selon la longueur du code....113
Figure IV.18 : WCMT sur TMS selon la longueur du code....113
Figure IV.19 : Temps de calcul sur P2020 des codes seuls (sans benchmark)....115
Figure IV.20 : Temps de calcul sur TMS320C6657 des codes seuls (sans benchmark)....115
Figure IV.21 : Comparatif des temps d'exécution des codes sur P2020 et TMS320C6657....116
Figure IV.22 : Surcoûts sur P2020 des codes intégrés dans les 4 benchmarks unitaires....118
Figure IV.23 : Temps de calcul sur P2020 des codes dans le benchmark intermédiaire....120
Figure IV.24 : Surcoûts sur P2020 des codes dans le benchmark complet (SansCRC 0%)....122

LISTE DES TABLEAUX

Tableau I.1 : Les niveaux SIL selon l'IEC61508 13

Tableau I.2 : Les niveaux de criticité dans le domaine avionique 14

Tableau I.3 : Correspondance des niveaux de criticité de différents domaines 14

Tableau III.1 : Pud des 3 codes, simulations exhaustives avec 8 bits de contrôle et 16 bits de charge utile 57

Tableau III.2 : Pud des 3 codes, simulations aléatoires (8 bits de contrôle et de 1 à 112 bits de charge utile) 58

Tableau III.3 : Impact de la graine de la génération aléatoire sur le Pud des 3 codes 58

Tableau III.4 : Coefficients du niveau de confiance 60

Tableau III.5 : Calculs a priori des intervalles de confiance à 95 % et 99 % de confiance pour les simulations sur des codes de 32 bits 61

Tableau III.6 : Calcul a posteriori des intervalles de confiance à 99,9% de confiance pour les simulations sur des codes de taille 32 bits et 32 octets de données 63

Tableau III.7 : Temps d'exécution sur STM32 avec options de compilation o0 et o2 77

Tableau IV.1 : Principales caractéristiques des CDVE avion Airbus et Boeing 94

Tableau IV.2 : Récapitulatif des propriétés des deux processeurs P2020 et TMS320C6657 selon les critères de sélection 106

Tableau IV.3 : Récapitulatif des caractéristiques des cartes d'évaluation pour les processeurs P2020 et TMS320C6657 107

Tableau IV.4 : Principales étapes des expérimentations 111

Tableau IV.5 : Temps de calcul sur P2020 et TMS selon la longueur du code 112

Tableau IV.6 : Résultats d'expérimentation sur les 4 benchmarks unitaires 118

Tableau IV.7 : Temps de calcul sur P2020 des codes dans le benchmark intermédiaire 119

Tableau IV.8 : Temps de calcul sur P2020 des différents codes intégrés dans le benchmark complet 121

GLOSSAIRE

ACC : *Actuators Control Computer* (calculateur de commande des actionneurs)
ACP : *Actuators Control Processing* (programme de commande des actionneurs)
AESA : Agence européenne de la sécurité aérienne
AFCS : *Automatic Flight Control System* (système de commande de vol automatique)
AFDX : *Avionics Full DupleX switched Ethernet*
ASIL : *Automotive Safety Integrity Level* (niveau de sûreté du domaine automobile)
BER : *bit error rate* (taux d'erreur binaire)
CAN : *Controller Area Network*
CDV : Commandes De Vol
CDVE : Commandes De Vol Électrique
COM : voie COMmande des calculateurs Airbus
COTS : *Components Off The Shelf* (composants sur étagère**)**
CRC : *Cyclic Redundancy Check* (contrôle de redondance cyclique)
CSMA/CA : *Carrier Sense Multiple Access/Collision Avoidance*
DAL : *Design Assurance Level* (niveau de sûreté du domaine avionique)
FAA : *Federal Aviation Administration* (administration fédérale de l'aviation)
FBW : *Fly By Wire* (commandes de vol à transmission électrique)
FCAC : *Flight Control Analog Computer* (calculateur numérique de commande de vol)
FCC : *Flight Control Computer* (calculateur de commande de vol)
FCDC : *Flight Control Digital Computer* (calculateur numérique de commande de vol)
FCP : *Flight Control Processing* (programme de commande de vol)
FCS : *Frame Check Sequence* (séquence de contrôle de la trame)
FR : *Failure Rate* (taux de défaillance)
HD : *Hamming Distance* (distance de Hamming)
HDLC : *High-Level Data Link Control* (commande de liaison de données à haut niveau)
LIN : *Local Interconnect Network*
MON : voie MONnitoring des calculateurs Airbus
MTTF : *Mean Time To Failure* (temps moyen de bon fonctionnement avant défaillance)
PA : Pilote Automatique
PFCS : *Primary Flight Control System* (système primaire de commande de vol)
PRIM : *PRIMary computer* (calculateur primaire)
SEC : *SECondary computer* (calculateur secondaire)
SEC : Système Embarqué Critique
SIL : *Safety Integrity Level* (niveau de sûreté multi-domaines)
TDMA : *Time Division Multiple Access*
TMR : *Triple Modular Redundancy*
TTP : *Time Triggered Protocol*
WCET : *Worst Case Execution Time* (pire cas du temps d'exécution)
WCMT : *Worst Case Mesured Time* (pire cas du temps d'exécution mesuré)
WSC : *Weighted Sum Code*

INTRODUCTION

La sûreté de fonctionnement des systèmes embarqués critiques est depuis toujours un enjeu majeur qui préoccupe à la fois leurs constructeurs et leurs utilisateurs, et ce, pour plusieurs raisons. La première raison est que, dans ces systèmes, l'occurrence de certaines défaillances pourrait engendrer un évènement catastrophique avec des conséquences graves sur l'environnement ou même sur la vie humaine. La deuxième raison est que ces systèmes sont soumis à des exigences de sûreté de fonctionnement dictées ou recommandées par des standards et des normes de certification. Mais, dans un tel contexte, atteindre un haut niveau de sûreté de fonctionnement se heurte, d'une part, aux contraintes de ces systèmes en termes de poids, de volume et de consommation énergétique qui doivent être optimisés, et d'autre part, aux caractéristiques spécifiques à certains systèmes embarqués critiques, notamment, l'inaccessibilité du système pour la maintenance (par exemple, dans le domaine aérospatial), ou le fait que le système évolue dans un environnement sévère, ce qui multiplie le risque des causes de défaillances. Étant données toutes ces contraintes et entraves, atteindre le niveau requis en sûreté de fonctionnement est toujours un vrai défi à relever par les concepteurs des systèmes embarqués critiques. Ce défi devient encore plus important lorsque pour un système donné, on cherche à introduire une ou des nouvelles technologies, puisqu'étant nouvelles, elles sont par nature moins éprouvées, moins bien maîtrisées, demandant ainsi un travail supplémentaire pour pouvoir être intégrées.

Certains systèmes de « contrôle commande » font partie des systèmes embarqués critiques que nous ciblons. De manière générale, un système de contrôle commande est défini comme un système qui commande et contrôle (ou supervise) un autre système, comme par exemple une voiture, un avion, un satellite, un robot, etc. Dans tous les cas, ces systèmes sont composés de calculateurs, de capteurs et d'actionneurs, et, de plus en plus souvent, d'un réseau de communication. En effet, les systèmes de contrôle commande ne cessent de profiter des progrès technologiques au niveau des composants électroniques et informatiques, et ils adoptent ainsi des architectures de plus en plus distribuées par opposition aux anciennes architectures centralisées et ce, grâce à l'apparition des capteurs et des actionneurs intelligents qui permettent la distribution de l'intelligence du système (par exemple, la prise de décision en ce qui concerne le traitement des données). En effet, les dernières générations des capteurs et actionneurs intelligents intègrent des capacités importantes de stockage et de traitement, ainsi que des dispositifs évolués de communications numériques avec des débits de plus en plus élevés. Ces composants sont de plus en plus miniaturisés dans l'optique d'optimiser leur poids et leur consommation d'énergie, puisque ces deux aspects sont primordiaux dans le contexte des systèmes embarqués critiques. Pour illustrer ces propos, nous donnons l'exemple d'un système extrêmement critique, le système de commande de vol d'un avion (système qui gère la trajectoire de l'avion) dans lequel l'intelligence du système était centralisée dans les calculateurs, mais l'intégration des actionneurs et des capteurs intelligents dotés d'une électronique locale évoluée a permis la distribution d'une partie de cette intelligence vers ces capteurs et actionneurs. On peut également citer le domaine de l'automobile.

Cette évolution vers des architectures distribuées exige naturellement la mise en place de réseaux de communication adéquats pour les échanges des données. Au fil du temps, ces réseaux ont évolué de l'analogique au numérique. Les architectures distribuées basées sur des réseaux de communication numériques ont d'indéniables avantages, mais en contrepartie elles soulèvent une nouvelle problématique : quelles nouvelles méthodes mettre en place pour assurer l'intégrité des données désormais numériques ? Il est vrai que, dans beaucoup de domaines, cette mutation a déjà été largement réalisée, mais, contrairement aux idées reçues, ce n'est pas encore vrai dans tous les domaines. Dans tous les cas, cette problématique est toujours d'actualité sur de nombreux points qui restent encore à approfondir et à améliorer, comme en attestent nos travaux [Zammali *et al.* 2015b].

C'est dans ce cadre que nous nous intéressons dans ces travaux de thèse à l'intégrité des communications qui est un attribut particulier de la sûreté de fonctionnement. Pour assurer l'intégrité des communications, il existe deux approches différentes introduites par l'IEC61508 qui sont l'approche « ***canal blanc*** » et l'approche « ***canal noir*** ». L'approche « *canal blanc* » consiste à vérifier l'intégrité (par rapport aux exigences du système et/ou de la norme de certification) de toutes les couches (par référence aux sept couches OSI) et de tous les composants impliqués dans la transmission et le calcul des données. L'approche « *canal noir* », quant à elle, vise à vérifier l'intégrité de bout en bout, c'est-à-dire l'intégrité au niveau applicatif, en faisant abstraction de couches inférieures, ce qui implique que la politique et les mécanismes mis en place au niveau applicatif soient suffisants à eux seuls pour atteindre les exigences du système en intégrité des communications.

Étant donnés les avantages de l'approche « ***canal noir*** » qui vise à couvrir toutes les erreurs résiduelles et dont la certification nécessite moins de justifications (que l'approche « canal blanc »), nous proposons à travers nos travaux une approche d'intégrité des communications fondée sur le concept « canal noir », en vue de réaliser une surcouche d'intégrité bout en bout. Notre approche utilise les codes détecteurs d'erreurs, ce qui en soi n'est pas une nouveauté. Mais, les spécificités de notre approche sont de considérer le cas des erreurs à haute multiplicité (hypothèse rarement traitée) et d'étudier et comparer les performances de plusieurs codes détecteurs (***Fletcher, Adler et CRC***), à la fois en termes de pouvoir de détection et en termes de « coût de calcul », le tout, dans deux cadres différents : un cadre académique et une cadre industriel comme expliqué plus loin. Notre approche d'intégrité vise les conditions idéales suivantes : être peu coûteuse en temps de calcul, facile à mettre en place et à certifier. De l'approche d'intégrité globale découle deux approches différentes : une approche « ***mono-code*** » utilisant un seul et unique code détecteur pour assurer l'intégrité de tous les échanges, et une approche « ***multi-codes*** » utilisant plusieurs codes détecteurs complémentaires en termes de pouvoir de détection. L'objectif final étant, dans chacune des approches, de pouvoir évaluer les codes détecteurs et d'arriver à en sélectionner un ou plusieurs en fonction des critères : pouvoir de détection, coût de calcul, complémentarité.

Nous avions initialement envisagé les systèmes de commande de vol électriques (CDVE) d'avions civils comme cas d'étude, mais ce choix a changé en début de la troisième année de ces travaux de thèse lorsqu'une collaboration est née avec Airbus Helicopters autour de la problématique d'intégrité des communications pour les futures CDVE d'Airbus Helicopters. En effet, Airbus Helicopters envisage de passer à des architectures CDVE « tout numérique » pour ses futurs systèmes de commande de vol (l'architecture actuelle, telle que celle du NH90, est hybride numérique/analogique). D'où la nécessité de la mise en place d'une nouvelle politique d'intégrité. L'objectif de cette collaboration est la sélection d'un code détecteur pour assurer l'intégrité des communications bout en bout.

Nous débutons ce manuscrit par un **premier chapitre** dédié à la présentation du contexte général et des concepts fondamentaux en rapport avec l'intégrité des communications dans les systèmes embarqués critiques. Nous commençons par décrire les systèmes embarqués critiques, leurs caractéristiques et exigences. Puis nous présentons une description générique des concepts de base (attributs, entraves et moyens) de la sûreté de fonctionnement, et nous explicitons la problématique de certification, qui est un enjeu primordial dans un tel contexte. Nous déclinons alors ce vaste cadre générique en focalisant sur les réseaux de communication pour décrire le périmètre de nos travaux. Cela se traduit, d'une part, par la présentation de l'intégrité des communications, que nous cherchons à assurer, et d'autre part, d'une présentation du moyen de sûreté de fonctionnement que nous adoptons pour cela. Ce moyen est la tolérance aux fautes, dont un des fondements est le concept de redondance, et donc nous présentons de manière détaillée les différentes définitions et formes de la redondance (symétrique/asymétrique, statique/dynamique, temporelle/spatiale, matérielle/logicielle). Enfin, la dernière partie de ce chapitre est consacrée à une première introduction des principes de la détection d'erreurs, et d'une classification des différentes familles de codes détecteurs (en bloc, concaténés, convolutifs).

Le **deuxième chapitre** est consacré aux différents états de l'art concernant nos travaux. Nous présentons d'abord les principaux codes détecteurs communément utilisés pour assurer l'intégrité des communications (codes CRC, Adler, Fletcher, etc.). Puis nous détaillons un état de l'existant des principaux mécanismes d'intégrité déployés dans des protocoles des couches inférieurs dans les réseaux, tels que ARINC 429, MIL-STD-1553, AFDX, TTP/C, LIN, Flexray et CAN. Le troisième volet s'attache à l'intégrité bout en bout au niveau applicatif avec tout d'abord l'explication du concept de « *canal noir* » (présent dans l'IEC 61508), puis la présentation de quelques solutions (standardisées ou génériques) telles que PROFIsafe, FSoE et openSAFETY. Enfin, nous présentons une analyse de travaux académiques qui se situent dans un contexte et ont des hypothèses proches des nôtres. Nous positionnons alors nos travaux par rapport à tous ces éléments.

Le **troisième chapitre** présente notre approche d'intégrité qui comprend deux contributions consistant en deux approches différentes sur la base du même concept d'intégrité bout en bout. Nous commençons par l'analyse qui nous a amenés à définir deux classes de systèmes (nommés « très contraignants » et « contraignants ») et nous conduisant à définir nos deux approches « mono-code » « multi-codes ». La sélection de codes détecteurs pertinents pour les systèmes critiques embarqués étant au cœur de notre travail, nous présentons ensuite les critères (ou caractéristiques de ces codes) que nous évaluons : pour l'approche mono-code, il s'agit du pouvoir de détection intrinsèque d'un code donné et de son coût (ou temps) de calcul, à quoi s'ajoute, pour l'approche multi-codes, la « complémentarité » du pouvoir de détection entre les codes utilisés. Après avoir présélectionné les codes à évaluer (CRC, Adler et Fletcher), la première approche est alors longuement présentée (scénarios, évaluations, résultats et analyse), suivi d'une présentation de l'approche multi-codes, nécessairement plus courte puisque tous les éléments communs aux deux approches auront déjà été présentés, et que de plus, les évaluations ne portent que sur le troisième critère (la « complémentarité). Les deux environnements d'évaluations développés (sous Matlab-Simulink et en Langage C), ainsi que les deux environnements matériels utilisés pour les évaluations (un de type standard et un de type « embarqué » qui est une carte d'évaluation du processeur STM32) sont détaillés tout au long du chapitre.

Le **quatrième et dernier chapitre** présente l'application d'une partie de nos contributions (l'approche mono-code) sur un cas d'étude réel et industriel : les futurs systèmes de Commandes De Vol Électriques (CDVE) d'Airbus Helicopters. Cette contribution a un double objectif : confirmer et affiner dans un environnement industriel nos conclusions du chapitre III, mais également apporter un cadre académique à une réalité industrielle, puisque les évaluations réalisées doivent permettre de déterminer si les codes détecteurs étudiés répondent aux exigences des futurs CDVE d'Airbus Helicopters. Ce chapitre comprend deux grands volets. Le premier s'attache à la description du système ciblé, de ses communications, et de l'environnement utilisé pour nos évaluations de codes détecteur. Après une introduction sur le rôle générique d'un système de commande de vol et sur ses caractéristiques plus spécifiques dans le cas des hélicoptères, nous résumons l'historique de l'évolution du « Fly By Wire » dans les systèmes de CDVE, en montrant le parallèle et les différences entre les mondes avion et hélicoptère. Puis, par étape, nous présentons le principe de fonctionnement de l'architecture actuelle de CDVE de l'hélicoptère NH90, suivi de la description de l'architecture en cours de finalisation pour le futur. Cette dernière description est focalisée sur les communications, pour permettre ensuite la présentation très détaillée de la politique d'intégrité du processus de détection dans les communications dans, et entre, les calculateurs de CDVE. Enfin, ce volet s'achève avec la présentation de l'environnement utilisé pour les évaluations : les deux processeurs (TMS320C6657 et P2020) et leurs cartes d'évaluation associées. Tout le contexte du cas d'étude et du cadre de nos évaluations étant présentés, le deuxième volet est alors entièrement consacré à nos expérimentations (et à leurs résultats), dont le but est d'évaluer un ensemble de codes détecteurs, avec cette fois (en complément du chapitre III) une mise en regard des résultats avec le cadre des exigences réelles des futurs CDVE d'Airbus Helicopters. Cette mise en regard doit aboutir au choix d'un de ces codes pour assurer l'intégrité des communications de bout en bout dans les futurs CDVE d'Airbus Helicopters.

Nous finissons le manuscrit par une **conclusion** générale sur l'ensemble des travaux et des contributions apportées, avec en particulier une discussion sur les approches proposées. Cette conclusion comporte également un ensemble de perspectives sur les différentes pistes à court et moyen termes et les différents axes qui restent encore à explorer pour apporter des améliorations et des extensions à nos propositions, voire à d'autres travaux.

Chapitre I

CONTEXTE GÉNÉRAL ET CONCEPTS FONDAMENTAUX

Les Systèmes Embarqués Critiques (SEC) sont de nos jours de plus en plus distribués s'appuyant nécessairement sur des réseaux de communication qui peuvent être très complexes. En effet, ces systèmes ont profité des progrès technologiques au niveau des composants électroniques qui fait qu'il existe maintenant des capteurs et des actionneurs intelligents capables de prendre des décisions en ce qui concerne le traitement des données (à travers un système de vote, par exemple) ce qui permet la distribution de l'intelligence du système (comme par exemple des nouvelles générations des systèmes de contrôle commande). Cette distribution de l'intelligence se traduit nécessairement par des architectures distribuées et basées sur des réseaux de communication numériques. Dans un tel contexte, assurer l'intégrité des communications est crucial afin d'assurer le bon fonctionnement du système. Ainsi, nous nous intéressons particulièrement dans ces travaux de thèse à la problématique de l'intégrité des communications. Nous introduisons dans ce chapitre les concepts de base permettant de définir le périmètre de nos travaux et qui sont nécessaires pour la compréhension de la problématique, des objectifs et des contributions de la thèse. Dans la première section, nous commençons par définir les systèmes embarqués critiques en introduisant leurs caractéristiques, contraintes et exigences. Puis, nous citons quelques domaines d'application pour illustrer le large spectre couvert par les SEC, avant d'introduire très brièvement un exemple lié directement à nos travaux : un système de commande avionique (cet exemple sera repris et détaillé dans le chapitre IV). Puisque assurer, pour les SEC, la sûreté de fonctionnement est un enjeu primordial, nous consacrons la deuxième section à une présentation générique des concepts de base de la sûreté de fonctionnement qui peut se caractériser via ses attributs, ses entraves et ses moyens. La dernière partie de cette section aborde le lien très fort entre la sûreté de fonctionnement et les normes de certification. Dans la troisième section, nous déclinons cette étude générique dans le cadre des réseaux de communication, avec pour but de décrire les aspects de la sûreté de fonctionnement concernés par nos travaux. En premier lieu, il s'agit d'un attribut particulier qui est l'intégrité des communications, avec ses différentes déclinaisons, ainsi que le principe d'intégrité « de bout en bout ». En second lieu, il s'agit de décrire et classifier les erreurs qui peuvent affecter les données et d'énoncer les types d'erreurs ciblés par ces travaux. Enfin, il s'agit du principal moyen de sûreté de fonctionnement qui sera mis en œuvre, qui est la tolérance aux fautes, avec notamment une description détaillée des différentes formes de redondance, qui sont les mécanismes de base de la tolérance aux fautes. Ces définitions étant données, la section suivante s'attache à présenter les codes détecteurs sur lesquels se basent nos contributions.

I.1 SYSTÈMES EMBARQUÉS CRITIQUES

Les systèmes ciblés par nos travaux sont les systèmes embarqués critiques. Nous pouvons commencer par noter que ces deux propriétés « embarqués » et « critiques » ne sont pas nécessairement corrélées : cela veut dire qu'un système embarqué peut ne pas être critique et vice-versa.

La propriété « **embarqué** » se traduit par le fait que le système en question est enfoui dans un autre système qui lui-même assure une fonction ou un service plus global. Nous pouvons trouver des systèmes enfouis (embarqués) dans des systèmes plus grands en taille et complexité, comme les voitures ou les avions, mais cela peut aussi être le cas des systèmes que nous utilisons dans notre vie quotidienne, à savoir le téléphone portable ou encore la machine à laver. Pour faire fonctionner ces systèmes, des systèmes y sont embarqués dans l'objectif d'exécuter des tâches précises.

La deuxième propriété des systèmes ciblés est la propriété « **critique** ». Un système est dit « critique » si une défaillance d'un de ses composants peut engendrer des dégâts matériels importants ou/et des conséquences graves sur l'environnement ou sur la vie humaine. Par exemple, un avion est un système critique du fait qu'une défaillance pourrait, dans certains cas, mettre en danger la vie des passagers et de l'équipage. Cette propriété écarte tous les systèmes embarqués non critiques du périmètre de nos travaux.

Étant données ces deux propriétés, les systèmes embarqués critiques ont des caractéristiques et des exigences particulières que nous allons définir dans ce qui suit.

I.1.1 Caractéristiques et exigences des systèmes embarqués critiques

Les SEC sont généralement des systèmes électroniques et informatiques « autonomes » et « réduits » (en poids, volume, puissances de calcul, etc.). Ainsi, la conception de tels systèmes diffère de celle des systèmes électroniques et informatiques standards par le fait, d'une part, que la composante logicielle et la composante matérielle sont intimement liées (utilisant principalement des micro-processeurs ou des microcontrôleurs), et d'autre part, par le fait de devoir tenir compte d'un ensemble d'exigences et de restrictions. Puisqu'ils sont conçus pour être enfouis dans d'autres systèmes et afin de réduire leurs coûts, tous les systèmes embarqués critiques se caractérisent selon [Armoush 2010] par :

- un poids et un volume réduits,
- des capacités mémoire et puissances de calcul réduites,
- une consommation énergétique réduite.

Certains SEC se caractérisent par :

- un environnement sévère avec des radiations, des vibrations, des variations de températures, etc., ce qui augmente les sources et les risques de défaillances,
- des possibilités limitées d'intervention et de réaction en cas de défaillance, par manque « d'accessibilité » au système (par exemple les SEC dans le domaine spatial).

Ainsi, étant données les caractéristiques communes à tous les SEC et celles spécifiques à certains, la conception des systèmes embarqués critiques doit répondre aux exigences suivantes :

- faible consommation d'énergie : des mécanismes d'optimisation de la consommation d'énergie doivent être déployés (par exemple en optimisant les temps d'exécution),
- haut niveau de performance : des mécanismes d'amélioration de la réactivité du système (afin de répondre par exemple aux contraintes temps réel) doivent être mis en place,
- haut niveau de sûreté de fonctionnement (définie à la section I.2) : cela veut dire que le système doit remplir ses fonctionnalités en satisfaisant les niveaux exigés des différents attributs de sûreté de fonctionnement ciblés, par exemple la disponibilité, l'intégrité (définis dans la section I.2), etc.

À noter que les caractéristiques et les exigences des systèmes embarqués critiques sont parfois contradictoires. Par exemple, le haut niveau de sûreté de fonctionnement requis est plus coûteux à atteindre étant donné un environnement sévère, qui augmente les sources de défaillances et demande donc la mise en place de davantage de mécanismes de tolérance aux fautes, ce qui demande davantage de ressources, alors même que les ressources sont limitées. Ainsi, satisfaire les exigences des SEC est un important défi à relever pour ses concepteurs.

I.1.2 Domaines d'application et exemple d'illustration

Avec les progrès en microélectronique et en informatique embarqués, le spectre des systèmes embarqués ne cesse de s'élargir, ces systèmes peuvent être de petite taille (ex : les montres intelligentes) ou de grande taille (ex : un avion) et ils peuvent être produits en quelques unités (ex : un robot médical) ou en plusieurs millions d'unités (ex : voitures). Les SEC peuvent avoir différents niveaux de contraintes de conception (en termes de consommation d'énergie, contraintes temporelles, etc.) ce qui fait qu'il existe des SEC plus contraignants que d'autres (cela dépend aussi du domaine). Le niveau de criticité (qui s'explique par le niveau de sûreté de fonctionnement ciblé) diffère aussi d'un système à un autre selon le domaine et le rôle du système. Les systèmes embarqués critiques sont déployés dans plusieurs domaines, comme le domaine avionique (civil ou militaire), automobile, ferroviaire, nucléaire, spatial, médical, etc.

Nos travaux de thèse ciblent les SEC en général mais se réfèrent plus particulièrement aux systèmes de commande de vol avioniques. Pour cela, nous les considérons comme exemple pour illustrer notre description des SEC. Le système de commande de vol est le système qui contrôle la trajectoire de l'aéronef (avion, hélicoptère ou autre). Il est composé de l'ensemble de calculateurs, du système de communication (particulièrement pour les architectures « Fly By Wire »), et des capteurs et actionneurs servant à assurer le vol de l'appareil dans les trois phases : décollage, vol et atterrissage. Le système de commande de vol contrôle les gouvernes (voir Figure I.1) en envoyant des ordres, déterminés à partir de la consigne du pilote et des informations issues des capteurs, aux surfaces contrôlées (les actionneurs) manuellement par le pilote ou automatiquement par l'autopilote.

Le système de commande de vol est un système embarqué critique ayant de fortes contraintes et exigences. Une défaillance d'un composant de ce système pourrait engendrer la défaillance du système ce qui peut nuire à la vie de l'équipement et des passagers au bord de l'avion. La conception de ce système est soumise à des fortes contraintes en sûreté de fonctionnement, en relation étroite avec les aspects de certification (le taux de défaillance doit être inférieur à 10^{-9} par heure de vol). Avec les progrès technologiques au niveau des composants et l'apparition des capteurs et actionneurs de plus en plus intelligents (dotés de capacités de mémorisation et de traitement de données), les systèmes de commande de vol ont évolué vers des architectures de plus en plus distribuées où l'intelligence du système, qui était jusque-là centralisée au niveau des calculateurs, est devenue de plus en plus répartie [Sghairi 2010]. Ces évolutions vers des architectures distribuées augmentent la complexité de ces systèmes.

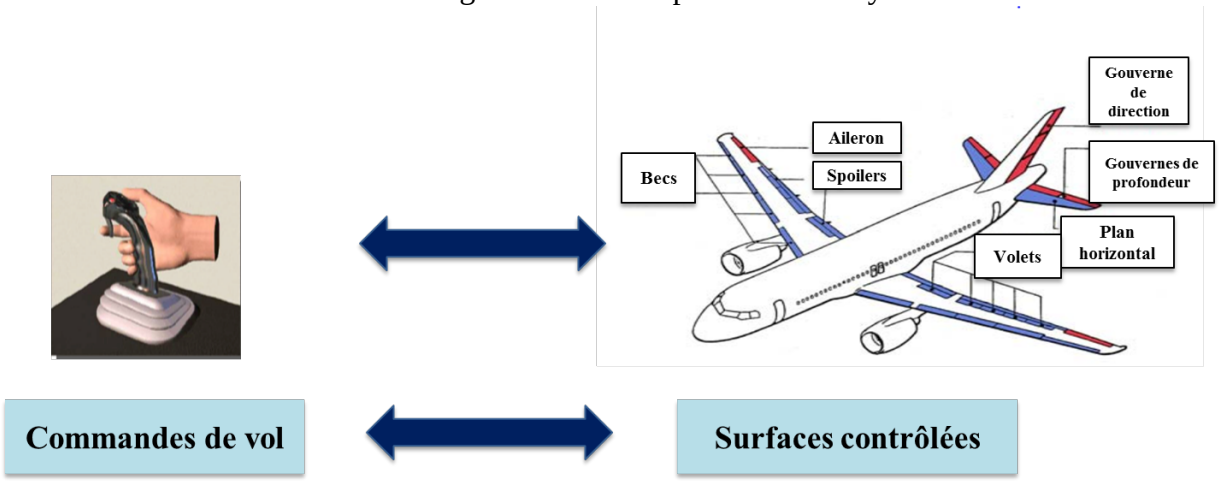


Figure I.1 : Système de commande de vol avionique et les surfaces contrôlées d'un avion

I.2 SÛRETÉ DE FONCTIONNEMENT : PRINCIPAUX CONCEPTS

Comme nous l'avons dit précédemment, les systèmes embarqués critiques requièrent un niveau relativement élevé de sûreté de fonctionnement (particulièrement les systèmes avec un très haut niveau de criticité comme le système de commande de vol avionique). La sûreté de fonctionnement d'un système se définit [Laprie *et al.* 1996] comme *« 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 que perçu ou requis par ses utilisateurs »*. La sûreté de fonctionnement est une propriété générique (voir Figure I.2) qui se caractérise d'abord par différents attributs que nous décrirons dans cette section, tout comme nous décrirons les entraves qui empêchent un système d'être sûr et enfin les moyens pour assurer le niveau requis en sûreté de fonctionnement. Enfin, nous aborderons les liens étroits entre la sûreté de fonctionnement et la problématique de certification.

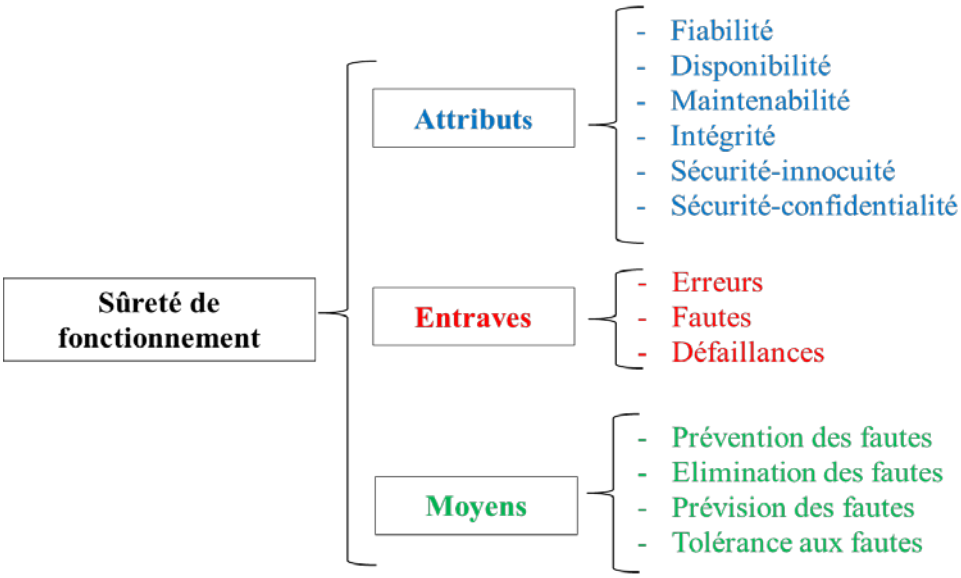


Figure I.2 : Sûreté de fonctionnement : vue d'ensemble

I.2.1 Attributs

Les attributs de la sûreté de fonctionnement sont un ensemble de critères ou propriétés que le système doit satisfaire pour qu'il puisse être considéré comme un système sûr. Ces attributs dépendent du système en question et de ses fonctionnalités. Par exemple, dans le secteur bancaire, la confidentialité de données est l'attribut primordial. Ces attributs dépendent également des objectifs visés en termes de niveau de sûreté de fonctionnement. Ainsi, pour un système donné, on peut, soit seulement chercher à évaluer (mesurer) le niveau atteint par les attributs, soit œuvrer pour que les attributs atteignent un niveau donné.

Plus généralement, les attributs usuels de la sûreté de fonctionnement sont [Laprie *et al.* 1996], [Knight 2012] et [Storey 1996] :

- **La fiabilité** : c'est la continuité du bon fonctionnement (délivrance d'un service correct) du système. Des exemples de métriques de fiabilité sont le FR (Failure Rate) qui est le taux des défaillances qui signifie la fréquence d'occurrence des défaillances et le MTTF (Mean Time To Failure) qui est le temps moyen jusqu'à l'occurrence d'une défaillance.

- **La disponibilité** : c'est l'aptitude à être prêt à être utilisé et à délivrer un service correct au moment où l'utilisateur le demande. Nous pouvons mesurer la disponibilité du système en calculant la proportion du temps de bon fonctionnement (service correct) sur le temps total d'exécution du système (alternance service correct-service incorrect).
- **La maintenabilité** : cela signifie la possibilité de faire des réparations ou des évolutions sur le système.
- **L'intégrité** : cela se traduit par la non-occurrence d'altérations inappropriées (changement) de l'information.
- **La sécurité-innocuité (Safety)** : cela traduit l'absence des conséquences catastrophiques sur l'environnement et/ou les personnes.
- **La sécurité-confidentialité (Security)** : cela signifie l'absence de divulgations (accès ou utilisation) non autorisées de l'information.

Il existe d'autres attributs, résultant notamment des combinaisons des attributs précédents. Généralement, pour un système donné, non seulement on ne cible qu'un sous-ensemble des attributs, et bien souvent, on cherche un compromis entre les attributs qui ne sont pas toujours tous compatibles entre eux, et qui ont également un coût.

Dans le cadre de ces travaux de thèse, nous nous focalisons sur **l'intégrité** qui peut être considérée comme un prérequis pour la fiabilité et la sécurité-innocuité. Nous focalisons plus spécifiquement sur **l'intégrité des communications** (des données échangées) dans les systèmes embarqués critiques, que nous déclinerons plus en détail dans la section I.3.

I.2.2 Entraves

Les entraves de la sûreté de fonctionnement sont l'ensemble de phénomènes qui pourraient empêcher le système d'atteindre les niveaux ciblés des attributs décrits précédemment. Ce sont les phénomènes susceptibles de provoquer un comportement « non correct » du système voire sa destruction. Trois types d'entraves de la sûreté de fonctionnement sont distingués selon [Laprie *et al.* 1996] : ce sont les défaillances, les erreurs et les fautes. La section suivante décrit ces trois entraves.

I.2.2.1 Classification de base

1) **Défaillance :** une défaillance survient quand le service délivré par le système dévie de l'accomplissement de la fonction à laquelle il est destiné. Selon la façon selon laquelle le système défaille, on peut distinguer des « modes de défaillance » différents. Le « mode de défaillance » peut être caractérisé selon quatre points de vue [Laprie *et al.* 1996].
   - Selon **le domaine de défaillance**, on distingue deux modes :
     - les défaillances **en valeur** : la valeur du service délivré par le système ne permet plus l'accomplissement de la fonction du système,
     - les défaillances **temporelles** : la valeur du service est délivrée trop tôt, trop tard ou n'est pas délivré du tout.
   - Selon la **perception des défaillances** par les utilisateurs, on distingue deux modes :
     - les défaillances **cohérentes** si tous les utilisateurs du système ont la même perception des défaillances,

  - les défaillances **incohérentes** si les utilisateurs ont des perceptions différentes des défaillances.
- Selon le **degré de sévérité des conséquences**, on distingue plusieurs modes allant des défaillances « **bénignes** » (la perte consécutive à la défaillance est du même ordre de grandeur que le bénéfice du service correct) ou « **catastrophiques** » (la perte est sans commune mesure avec le bénéfice).
- Selon la **dépendance entre les défaillances**, on appelle **mode commun de défaillance** le phénomène où les défaillances sont dépendantes ayant la même cause directe ou indirecte et conduisant au même mode (type, nature) de défaillance.

2) **Erreurs :** l'erreur est un état du système (ou d'une partie du système) qui est « susceptible » d'entraîner une défaillance. Le fait que l'erreur conduise effectivement ou non à une défaillance dépend de plusieurs facteurs qui sont principalement :
   - la **composition du système** : la présence de redondance (détaillée plus tard) pourrait empêcher l'occurrence d'une défaillance en dépit de l'erreur,
   - l'**état (activité) du système** : une erreur peut être éliminée avant de conduire à une défaillance,
   - la **perception de la défaillance par l'utilisateur** : une défaillance peut être inacceptable pour un utilisateur et tolérable pour un autre.

3) **Fautes :** la faute est la cause (adjugée ou supposée) d'une erreur. Il y a plusieurs points de vue pour classifier les fautes :
   - **Selon la cause** de l'erreur, nous pouvons distinguer deux types de fautes :
     - une faute **physique** quand elle est due à un phénomène physique,
     - une faute **humaine** quand elle est due à l'imperfection humaine.
   - **Selon sa nature**, la faute peut être :
     - une faute **accidentelle** c'est-à-dire non volontaire,
     - une faute **intentionnelle non malveillante** quand elle est commise volontairement mais pas dans l'objectif de nuire au bon fonctionnement du système,
     - une faute **intentionnelle malveillante** si elle est commise d'une façon volontaire dans l'objectif de nuire au bon fonctionnement du système (exemple : un pirate informatique).
   - Selon **la phase de sa création ou de son occurrence**, la faute peut être soit :
     - une faute **de développement** si elle est commise au cours du développement du système ou de modifications lors de sa maintenance,
     - une faute **opérationnelle** si elle apparaît durant l'exploitation du système.
   - Du point de vue de la **frontière du système**, nous pouvons distinguer deux classes de fautes :
     - les fautes **internes** si elles viennent du système lui-même,
     - les fautes **externes** si elles viennent de l'interaction du système avec son environnement.
   - Finalement, selon leur **persistance**, les fautes peuvent être :
     - **permanentes**,
     - **temporaires** (présentes sur des durées limitées).

I.2.2.2 Corrélation et relativité des définitions de base

Comme décrit sur la Figure I.3, les trois types d'entraves de la sûreté de fonctionnement sont corrélés. En effet, une faute engendre une erreur qui à son tour pourrait déclencher une défaillance. Pour illustrer la chaîne de causalité de ces trois entraves, nous donnons un

exemple simple d'un logiciel contenant une faute de développement (par exemple, un algorithme de calcul faux), qui pourrait engendrer des résultats de calcul erronés (c'est l'erreur) et qui pourraient engendrer une défaillance du système (la défaillance).

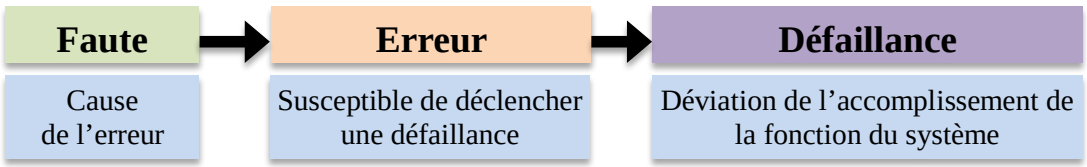


Figure I.3 : Relation entre les trois types d'entraves de la sûreté de fonctionnement (chaîne de cause à effet)

À noter que le système peut comprendre des fautes ou des erreurs sans conduire à des défaillances si les erreurs ou les fautes sont silencieuses (n'engendrant pas de défaillance) ou simplement non déclenchées. Et ces définitions sont « relatives » : une défaillance d'un système peut être une faute dans le système qui utiliserait le « service défaillant ».

I.2.3 Moyens

Les moyens de la sûreté de fonctionnement sont l'ensemble des méthodes et techniques qui, d'une part, permettent d'analyser (évaluer) le niveau de sûreté de fonctionnent d'un système, et d'autre part, sont à mettre en œuvre pour qu'un système atteigne un niveau donné de sûreté de fonctionnement en réduisant les effets des entraves. On distingue quatre classes de moyens [Laprie *et al.* 1996] et [Knight 2012] :

- **La prévention des fautes** : son objectif est d'empêcher l'occurrence ou l'introduction de fautes. La prévention des fautes se fait avant la mise en service du système, dès la phase de spécification et relève pour beaucoup de l'ingénierie système.
- **L'élimination des fautes** : elle vise à réduire la « présence » de fautes dans un système. Ce moyen est fondé sur la vérification de la satisfaction, par le système, de certains types de propriétés. En cas de non satisfaction, il faut localiser les fautes dans le système afin de réduire leur nombre et sévérité. Bien souvent (mais pas seulement), l'élimination des fautes se fait pendant la phase de validation (tests) ou les premières utilisations du système (ex. : version béta d'une application, essais en vol d'un avion).
- **La prévision des fautes** : à défaut de pouvoir les éviter ou les éliminer, il faut admettre l'occurrence éventuelle de fautes quand le système est en service en s'assurant que leur impact sur le service reste acceptable. La prévision des fautes se fonde sur des analyses du comportement du système (analyses des défaillances des composants et de leurs conséquences) pour estimer et évaluer la présence de fautes et donc à pouvoir prendre une décision si le système répond aux exigences requises en termes de sûreté de fonctionnement.
- **La tolérance aux fautes** : toujours fondée sur l'idée d'admettre l'occurrence éventuelle de fautes, et pour répondre aux exigences du système en termes de sûreté de fonctionnement, la tolérance aux fautes consiste généralement (mais pas toujours, certains systèmes étant intrinsèquement tolérants aux fautes) à mettre en place dans le système, des mécanismes supplémentaires particuliers, notamment la redondance (de « composants » au sens large) pour permettre au système de continuer à fonctionner normalement en dépit des fautes.

Ces quatre types de moyens s'utilisent bien souvent de manière combinée (et non pas de manière isolée) pour accroitre le niveau de sûreté de fonctionnement (ou de confiance) d'un système. Certaines techniques et méthodes sont communes à plusieurs types de moyens (en

particulier à l'élimination et la prévision des fautes), mais ne sont pas employées dans les mêmes buts selon le type de moyen. Nous nous intéressons dans ces travaux à la tolérance aux fautes pour assurer l'intégrité des communications dans les systèmes embarqués critiques, la section I.3.3 décrivant cela plus en détail.

I.2.4 Sûreté de fonctionnement et certification

Dans le contexte des systèmes critiques et particulièrement dans certains domaines, à savoir le domaine avionique, automobile, nucléaire, spatial ou ferroviaire, le niveau requis en sûreté de fonctionnement est soumis à des règlements dictés ou recommandés par des standards de certification.

Plus précisément, il y a des domaines qui ont leurs propres standards de certification, alors que pour les autres, ils ont le choix d'adhérer ou non aux recommandations des standards de certification génériques. Par exemple, un standard générique qui considère tous les systèmes électroniques programmables est l'IEC61508 [IEC61508 2010]. Ce standard vise à faire appliquer des fonctions de sûreté qui réduisent les risques de défaillance. Il considère à la fois, la probabilité (le niveau d'occurrence) de l'évènement redouté et la sévérité de ses conséquences afin de déterminer le niveau SIL (Safety Integrity Level) requis. Il existe cinq niveaux SIL (voir Tableau I.1), chaque niveau correspondant à un taux d'occurrence de défaillance par heure permis par la norme.

Catégorie	SIL	Conséquences
Catastrophique	4	Perte de plusieurs vies
Critique	3	Perte d'une seule vie
Marginal	2	Blessures majeures
Négligeable	1	Blessures mineures et dommage matériels
Sans conséquences	0	Pas de dégât à part le non satisfaction de l'utilisateur

Tableau I.1 : Les niveaux SIL selon l'IEC61508

Un niveau SIL sert de guide dans les choix architecturaux comme par exemple l'obligation d'avoir des architectures redondantes pour certains niveaux SIL. De l'IEC61508 ont été dérivés d'autres standards spécifiques comme l'ISO26262 [ISO26262 2011] pour l'automobile, l'EN50128 [EN50128 2011] pour le ferroviaire. Mais pour ces domaines, l'adhésion aux recommandations des normes spécifiques est optionnelle (pas imposée légalement) mais il faut démontrer dans le cas échéant que la méthodologie utilisée est aussi bonne que celle proposée par le standard.

Pour l'avionique, les standards de certification sont développés indépendamment de l'IEC61508 ; nous pouvons citer l'ARP 4754/ED-79 [ARP4754A 2010] « Guidelines For Development of Civil Aircraft and Systems », l'ED-12/DO-178 (A en 1985, B en 1992, C en 2011) « Software Considerations in Airborne Systems and Equipement Certification ». Ces standards définissent leurs propres niveaux de criticité (voir Tableau I.2).

Pour l'avionique, ces standards de certification ont même une autorité légale (autorité de certification) qui autorise ou non, au final, la mise en service du système en évaluant son niveau de sûreté. À titre d'exemple d'autorité de certification, on peut citer la FAA (Federal Aviation Administration) pour l'aviation civile aux États-Unis, et l'AESA (Agence Européenne de la Sécurité Aérienne) au niveau européen.

Dénomination des défaillances	**Taux d'occurrence de défaillance par heure**		**Conséquences des défaillances**
Mineure	Probable	$> 10^{-5}$	Réduction non significative de la sécurité de l'avion pouvant inclure une légère réduction des marges de sécurité ou des fonctions, un léger accroissement de la charge de travail, quelques inconvénients pour les passagers.
Majeure	Rare	Comprise entre 10^{-5} et 10^{-7}	Réduction significative des marges de sécurité ou des fonctions, ou augmentation significative de la charge de travail de l'équipage, ou inconfort des occupants avec possibilité de blessures.
Dangereuse	Extrêmement rare	Comprise entre 10^{-7} et 10^{-9}	Grande réduction des marges de sécurité ou des fonctions, ou détresse physique ou grande surcharge de travail, ou blessure fatale ou sérieuse pour un relativement petit nombre de passagers.
Catastrophique	Extrêmement improbable	$< 10^{-9}$	Empêche la continuité de la sécurité à l'atterrissage ou en vol: l'avion, les passagers et l'équipage sont perdus.

Tableau I.2 : Les niveaux de criticité dans le domaine avionique

Dans [Verhulst *et al*. 2013], [Baufreton *et al.* 2010] et [Baufreton *et al*. 2011], les auteurs proposent de faire une correspondance entre les différents standards (voir Tableau I.3 extrait de [Verhust *et al.* 2013]) des différents domaines. À chaque niveau de criticité correspond un taux d'occurrence de défaillance autorisé ou recommandé. Par exemple, pour le niveau « catastrophique » de l'avionique, le taux d'occurrence de défaillance ne doit pas dépasser 10^{-9} défaillances par heure de vol, ce qui est notamment le cas des systèmes de commande de vol.

Après avoir décrit les principaux concepts de la sûreté de fonctionnement, à savoir ses attributs, ses entraves et ses moyens, et après avoir décrit la problématique de la certification en relation avec la sûreté de fonctionnement qui est une problématique cruciale pour les systèmes critiques, nous allons décliner notre étude pour analyser avec plus de détails la problématique d'intégrité des communications qui est le cœur de ces travaux de thèse.

Domaine	**Niveaux de criticité spécifiques aux différents domaines**				
Général (IEC61508)	SIL 0	SIL 1	SIL 2	SIL 3	SIL 4
Automobile (ISO26262)	ASIL A	ASIL B	ASIL C	ASIL D	
Aviation (Do178C)	DAL E	DAL D	DAL C	DAL B	DAL A
Ferroviaire (EN50128)	SIL 0	SIL 1	SIL 2	SIL 3	SIL 4

Tableau I.3 : Correspondance des niveaux de criticité de différents domaines

I.3 PROBLÉMATIQUE DE L'INTÉGRITÉ DES COMMUNICATIONS

Nous décrivons dans cette partie la problématique de l'intégrité des communications sur laquelle s'axent nos travaux de thèse. L'intégrité contient différents aspects. Par exemple, dans [Storey 1996], l'auteur fait la distinction entre *l'intégrité du système* et *l'intégrité des données*. *L'intégrité du système* est définie comme la capacité du système à détecter les défaillances de son fonctionnement et d'informer (notifier) un opérateur humain, alors que *l'intégrité des données* est définie comme la capacité du système à détecter voire corriger les données erronées (stockées dans une base de données endommagée ou échangées dans un réseau présentant des sources de corruption de données). Plus largement, l'intégrité des données est un point crucial pour l'intégrité globale d'un système. Par ailleurs, les systèmes embarqués critiques que nous ciblons sont basées sur des architectures distribuées reposant sur des réseaux de communication numériques complexes (avec des composants intelligents comme le cas des actionneurs et des capteurs intelligents dans les systèmes de contrôle commande actuels) ce qui soulève la problématique de corruption des données échangées et rend nécessaire le fait d'assurer l'intégrité des communications.

Nos travaux visent plus spécifiquement l'intégrité des données échangées, via un réseau de communications numériques entre les différents nœuds d'un système. Nous définissons l'intégrité des communications par la capacité d'un système à détecter et recouvrir les erreurs que peuvent contenir les messages échangés. Nous visons plus particulièrement les erreurs issues de fautes accidentelles et non les erreurs issues de fautes malveillantes, c'est-à-dire nous proposons une solution de sécurité-innocuité (« safety ») et non pas de sécurité-confidentialité (« security »).

I.3.1 Périmètre de l'intégrité des communications

Nous décrivons dans cette section le périmètre que nous considérons pour l'intégrité des communications. Nous commençons par présenter les attributs d'une communication intègre. Nous décrivons ensuite le principe de l'intégrité « de bout en bout ». Finalement, nous dressons une classification des erreurs en présentant celles que ciblent nos travaux.

I.3.1.1 Attributs d'une communication intègre

Pour qu'une communication soit intègre, elle doit satisfaire les attributs suivants :

- **Source et destination intègres** : cela veut dire que le message est émis par l'émetteur qui est censé l'émettre et est reçu par le récepteur qui est censé le recevoir. Cela impose d'ajouter des informations d'identification des émetteurs et destinataires.
- **Fraîcheur des données** : cela impose que le message soit transmis dans un délai acceptable en rapport avec la contrainte « temps réel » du système en question. Par conséquent, le message émis contient des données « fraîches » et non obsolètes reflétant l'état du système. Au final, cela impose d'ajouter des informations de type temporel.
- **Ordre correct** : cela veut dire que les messages sont reçus dans le bon ordre et que donc la séquence des messages reçus respecte le même ordre qu'à l'émission, ou que le récepteur soit capable de réordonner les messages reçus. Cela impose d'ajouter des informations de numérotation des messages.

- **Non altération/corruption de données** : cela veut dire que les données contenues dans le message reçu sont exactement les mêmes que dans le message émis. Cela impose d'ajouter des informations et/ou des mécanismes permettant de « comparer » l'émis et le reçu.
- **Non perte des données** : cela veut dire que chaque message émis est non perdu et reçu par son destinataire.

On constate que pour la plupart de ces caractéristiques, il est nécessaire d'ajouter des informations et/ou des mécanismes supplémentaires, par rapport aux données réellement utiles à transmettre. L'ajout de ces données supplémentaires peut se faire à l'intérieur même des messages, mais également via d'autres mécanismes dont le développement n'est pas l'objet dans cette section.

Si l'on prend le terme « données » au sens très large (données utiles et supplémentaires), on peut considérer que quasiment toutes les caractéristiques d'intégrité des communications sont liées (voire sous-entendues) à la caractéristique « non altération de données ».
C'est pour cette raison que, dans ces travaux de thèse, nous nous focalisons sur cet attribut « non altération de données ».

I.3.1.2 Intégrité de bout en bout

Pour assurer l'intégrité des communications, on déploie usuellement un ensemble de mécanismes pour la détection des erreurs. Ces mécanismes sont généralement mis en place dans les couches basses puis dans une moindre mesure dans les couches plus hautes (intermédiaires). Dans ces travaux de thèse, nous nous intéressons plutôt au contrôle d'intégrité de bout en bout à mettre en œuvre dans les couches hautes (notamment la couche applicative) (voir Figure I.4). Le contrôle de bout en bout permet de faire abstraction du matériel réseau réellement utilisé dans les couches inférieures et qui pourrait être plus ou moins performant en termes de détection des erreurs. Dans le cas d'un système critique qui est la cible de nos travaux, il est de plus important pour l'industriel de montrer aux organismes de certification que la sécurité repose sur un mécanisme de haut niveau facilement compréhensible et qui inspire confiance.

En effet, les travaux de [Fuchs 1996] affirment que la couverture des fautes ne peut atteindre 100% qu'avec l'ajout d'un mécanisme de détection d'erreurs au niveau bout en bout. Ceci est confirmé aussi dans les travaux de [Arlat *et al.* 2003] qui prouvent que l'ajout d'un contrôle bout en bout permet d'atteindre un niveau très élevé de couverture de fautes.

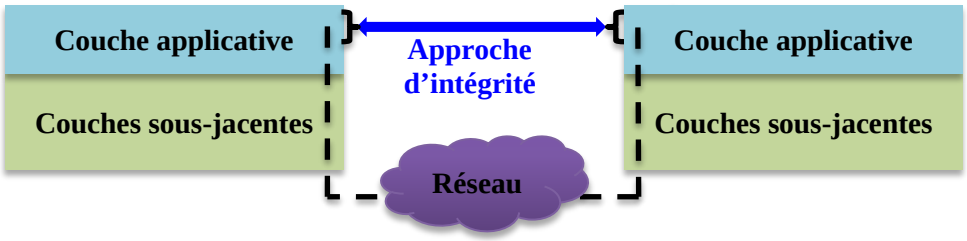


Figure I.4 : Contrôle d'intégrité de bout en bout

Maintenant que nous avons décrit le cadre de la problématique de l'intégrité des communications, la section suivante s'attache à présenter les moyens habituels pour répondre à cette problématique.

I.3.2 Corruption des communications : classification et causes d'erreurs

Dans la littérature, nous trouvons des propositions de classification des fautes et de classification des défaillances (cf. section I.2.2.1), mais très peu de classification des erreurs particulièrement les erreurs dans le contexte des communications. Nous proposons donc ici une classification des erreurs, uniquement dans le cadre des communications numériques, et qui reprend une partie de la terminologie de la classification des fautes et des appellations d'erreurs communes. Selon l'ensemble ci-dessous de 3 critères, nous distinguons différents types d'erreurs représentés sur la Figure I.5:

- **La taille ou multiplicité de l'erreur** : il s'agit du nombre de bits affectés. Si un seul bit est affecté (« bit flip »), l'erreur est dite **simple**. Si plusieurs bits sont affectés, l'erreur est dite **multiple**. Nous distinguerons toutefois les erreurs à multiplicité faible (ex. : un « *burst* » des erreurs dont la multiplicité n'est bornée que par la longueur de la trame dont tous les bits peuvent être affectés. Dans nos travaux, c'est ce dernier type d'erreur que nous considérerons et noterons simplement « erreurs multiples ».
- **La durée de vie de l'erreur** : il s'agit de la persistance de l'erreur. Si une fois apparue, l'erreur est irréversible et persistante, elle est dite **permanente**. Si elle apparaît sur un laps de temps fini et disparaît après, l'erreur est dite **transitoire**. Si l'erreur est sporadique ou occasionnelle (déclenchée par un évènement particulier), elle est dite **intermittente**.
- **Le mode de distribution (spatiale) de l'erreur** : il s'agit de l'existence ou non de corrélation entre les bits erronés. Si les bits sont affectés d'une façon aléatoire alors l'erreur est dite **aléatoire**. Si l'erreur affecte une partie du message (du bit de position *i* jusqu'au bit de position *f*, les bornes sont affectées et les bits entre i et f ne sont pas nécessairement affectés), elle est dite **en *rafales*** aussi appelées *en paquet ou « burst »*.

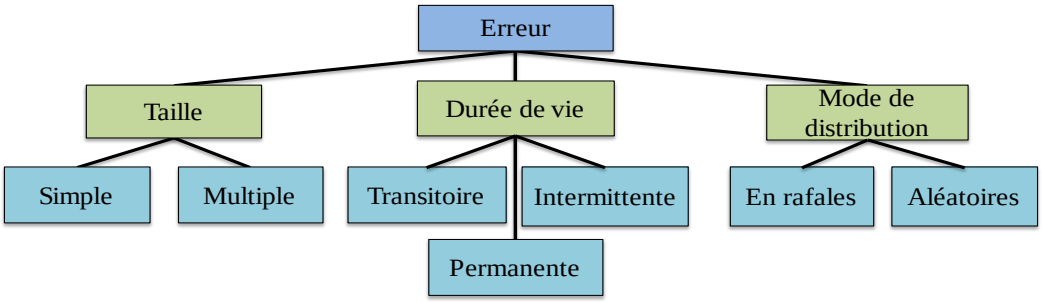


Figure I.5 : Classification des erreurs affectant les messages échangés

À noter que ces différents types d'erreurs sont indépendants les uns des autres vu l'indépendance des critères (taille, durée de vie, etc.). De ce fait, par exemple, une erreur peut être à la fois permanente et multiple.

Pour étudier les causes des erreurs, nous retiendrons quatre catégories principales de fautes qui se distinguent entre fautes internes ou fautes externes affectant le médium de communication ou le contrôleur de réseau :

- Le **bruit thermique** (lié à l'agitation de la matière) est la principale source de **faute interne** affectant le médium, et dans une moindre mesure, les équipements réseaux. Il peut provoquer des **erreurs simples** (inversions de bits, bit-flip) pour des trames de taille réduite ou des **erreurs multiples aléatoires** (sans corrélation entre les bits affectés) pour des trames de taille relativement grande (cela dépend du taux d'erreur binaire BER ou *Bit Error Ratio*). Ces erreurs sont **transitoires**, leurs durées de vie sont finies et relativement courtes.

- Les **perturbations électromagnétiques** (liées à l'environnement) sont la principale source de **fautes externes** affectant le médium, et dans une moindre mesure, les équipements réseaux. De par la nature du phénomène, les inversions de bits au niveau d'une trame concernent des bits « voisins ». Le modèle d'erreur associé est celui des **erreurs en rafales (burst)**. Ces erreurs sont **intermittentes** vu qu'elles sont déclenchées par un évènement particulier.
- Les **radiations (particules énergétiques)** auxquelles peut être soumis un circuit intégré sont les principales sources de **fautes externes** qui peuvent affecter la partie circuit logique d'un contrôleur (ou autre équipement réseau), que cela soit au niveau des zones mémoire, des registres ou tout simplement d'une bascule. Ces perturbations sont appelées des évènements singuliers (SEE : Single Event Effects) dont les plus connus sont les SEU (Single Event Upsets) qui provoquent des inversions du contenu (bit-flip) de cellules mémoires et registres et bascules et les SEL (Single Event Latchups) qui donnent lieu à des courts-circuits qui peuvent conduire à la destruction du circuit par effet thermique. Les SEU se traduisent, dans une grande proportion, par des erreurs simples transitoires au niveau de la mémoire et se traduisent donc par des **erreurs simples transitoires** au niveau d'une trame. Lorsque le SEU affecte un registre ou une bascule d'un circuit, les erreurs induites peuvent être **permanentes** et, comme cet évènement peut affecter le comportement du circuit logique, il peut se traduire par des **erreurs multiples** (un grand nombre de bits affectés) et **permanentes** au niveau de la trame, et peut être la même dans toutes les trames.
- Les **défaillances de transistors ou de connexions** au sein d'un circuit intégré sont les principales sources de **fautes internes** à un contrôleur réseau. Contrairement au cas des SEU traité précédemment, les fautes sont ici des fautes permanentes. En fonction de la partie affectée dans le circuit**,** l'erreur induite au niveau de la trame peut être une **erreur simple ou multiple** et **permanente.**

Dans ces travaux de thèse, les erreurs que nous considérons sont celles qui résultent de fautes physiques qui affectent le système de communication tant au niveau du médium de communication qu'au niveau du contrôleur de communication (au sens large), nous ciblons les **erreurs multiples**, qu'elles soient transitoires, intermittentes ou permanentes et quel que soit leur mode de distribution (aléatoire ou en rafale). Le fait de cibler les erreurs multiples couvre les erreurs simples qui seront implicitement couvertes par notre approche.

I.3.3 La tolérance aux fautes pour l'intégrité des communications

Comme décrit dans la section I.2.3, les moyens pour assurer la sûreté de fonctionnement des systèmes sont : la prévention des fautes, l'élimination des fautes, la prévision des fautes et la tolérance aux fautes. Pour les systèmes embarqués critiques qui sont la cible de nos travaux, la tolérance aux fautes, qui consiste à accepter la possibilité de l'occurrence éventuelle des fautes et à faire en sorte que le système puisse continuer à fonctionner en dépit des fautes, est un moyen incontournable. En effet, au-delà d'une certaine complexité du système, il devient impossible de reproduire tous les scénarios de défaillance pour pouvoir éliminer ou prévenir toutes les fautes. Il subsiste donc des risques de présence ou d'apparition des fautes, ce qui pour un système embarqué est encore plus problématique de part ses caractéristiques.
Comme défini par [Laprie *et al.* 1996], la tolérance aux fautes vise à éviter les défaillances en dépit des fautes présentes, et se base essentiellement sur la redondance. La redondance consiste à créer des copies multiples d'un composant (matériel, logiciel, données, etc.) ou d'une exécution de sorte que le duplicata assure la même fonction, le même service ou le même rôle que le composant (ou l'exécution) original.

Nous définissons ici la notion de redondance en introduisant dans un premier temps les différents types de redondance déployés dans les systèmes tolérants aux fautes (et les architectures classiques qui en découlent) pour assurer la sûreté de fonctionnement. Nous ciblons dans un deuxième temps les types de redondance que nous retenons ou que nous allons adapter pour assurer l'intégrité des communications.

I.3.3.1 Redondance symétrique ou asymétrique (diversification)

En nous inspirant des concepts définis dans la littérature et dans les normes [ISO26262 2001], nous distinguons deux grandes classes de redondance. La première classe est celle de la redondance ***symétrique*** qui consiste à dupliquer un composant de sorte que la copie soit à l'identique du composant original (ex. : deux processeurs ou deux compilateurs identiques). La deuxième classe est dite ***asymétrique***, elle consiste plutôt à avoir une copie différente du composant (ex. : deux processeurs différents avec des conceptions ou des constructeurs différents). On parle alors de « ***diversification*** ». Une difficulté de cette solution est d'arriver à trouver ou à produire une copie qui soit à la fois différente du composant orignal, tout en assurant les mêmes fonctionnalités que celles du composant original. Cela demande également un travail supplémentaire d'étude de la sûreté de fonctionnement pour le second composant du fait de ses différences. Mais, en contrepartie, l'avantage principal de la redondance asymétrique est qu'elle réduit le risque du **mode commun de défaillance** (défini dans la section I.2.2.1). Cet avantage prime sur les difficultés et les coûts de cette solution.

I.3.3.2 Granularité de la redondance

Selon la granularité de la redondance, dans [Knight 2012] l'auteur distingue deux familles de redondance qui sont : la ***redondance à grands composants*** et la ***redondance à petits composants***. Par exemple, dans le cas d'une mémoire et de redondance d'information, si les bits de parité sont utilisés comme technique de redondance, la redondance à petits composants consiste dans ce cas à ajouter un bit de parité pour chaque mot de la mémoire, alors que la redondance à grands composants consiste à ajouter un seul bit de parité pour la totalité de la mémoire. L'avantage de la redondance à petits composants est que dans le cas de détection de fautes (ou erreurs ou de défaillance), la zone erronée étant de plus petite taille (ex. : mots mémoire), la continuité de service du système global à moins de risque d'être compromise que dans le cas d'une redondance à grands composants (ex. : toute la mémoire).

I.3.3.3 Redondance statique (ou active), dynamique (ou passive) et hybride

La **redondance est dite active** [Dubrova 2013] ou **statique** [Knight 2012] quand les composants principaux et dupliqués fonctionnent simultanément dans l'objectif de fournir exactement le même service. Un exemple de redondance statique est d'avoir deux calculateurs qui fournissent la même donnée, ce qui permet de comparer les résultats et détecter une anomalie si les deux calculateurs ne fournissent pas le même résultat. La redondance active peut aussi être déployée pour assurer la disponibilité du système.

La **redondance est dite passive** [Dubrova 2013] ou **dynamique** [Knight 2012] si la mise en fonctionnement d'au moins une partie du composant (ou des composants) dupliqué est initiée par un évènement particulier comme la défaillance du composant principal. Un exemple de redondance dynamique est la duplication des calculateurs en gardant les calculateurs redondants hors tension jusqu'à la défaillance du calculateur principal. L'avantage de la redondance dynamique est qu'elle est moins coûteuse consommation d'énergie. De plus, pour des composants physiques, elle entraîne moins d'usure des composants redondants. L'inconvénient est que le niveau de disponibilité de ces composants doit être plus élevé.

La **redondance hybride** [Essamé 1998] combine les avantages de la redondance statique et dynamique où une partie est redondée d'une façon active et l'autre partie est redondée d'une façon passive. La redondance hybride se base sur les techniques de détection et recouvrement d'erreurs qui permettent de reconfigurer le système dans le cas d'occurrence d'une faute. Elle se base sur la technique de masquage des fautes pour prévenir la production de résultats incorrects.

I.3.3.4 Redondance temporelle ou spatiale

Selon [Dubrova 2013], deux familles de redondance sont utilisées dans les systèmes tolérants aux fautes : la famille de ***redondance spatiale*** et la famille de ***redondance temporelle***. Dans la famille de redondance spatiale, trois classes de redondances peuvent être distinguées : la redondance matérielle qui peut être active, passive ou hybride, la redondance logicielle qui peut être à version unique ou multi-versions et la redondance d'information. La Figure I.6 résume les différents types de redondance distingués par [Dubrova 2013].

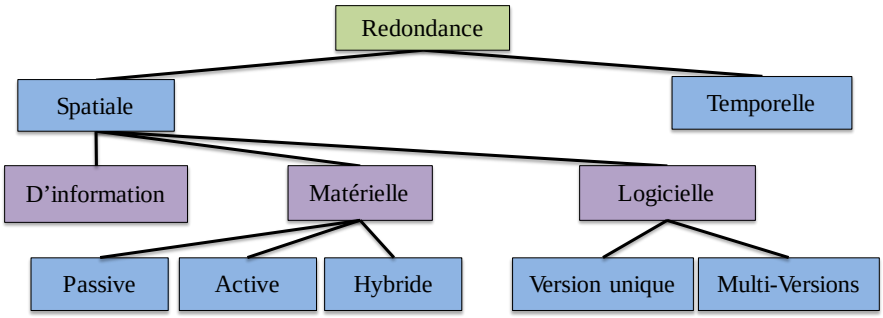


Figure I.6 : Différents types de redondance dans les systèmes embarqués critiques

1) Redondance temporelle

La redondance temporelle est définie par le fait qu'un même composant exécute la même tâche (ex. : calcul ou transmission d'un message) plusieurs fois dans le temps [Dorow 2003] [Dubrova 2013]. Dans les systèmes critiques, la redondance temporelle est couramment déployée dans l'objectif d'assurer l'intégrité des communications [Marques *et al.* 2012]. Elle consiste à envoyer à des instants distincts (t_0, t_1, ...) plusieurs copies du même message (D) comme le montre la Figure I.7. La transmission répétitive peut être systématique ou déclenchée par un évènement particulier (par sollicitation) comme notamment la détection d'une copie erronée d'un message. Comme le montre la figure I.7, où *D* représente un message, la redondance temporelle (systématique dans ce cas de figure) est un des moyens qui permet de détecter, masquer et recouvrir les fautes (ou erreurs) à l'aide d'un système de vote par exemple [Dubrova 2013].

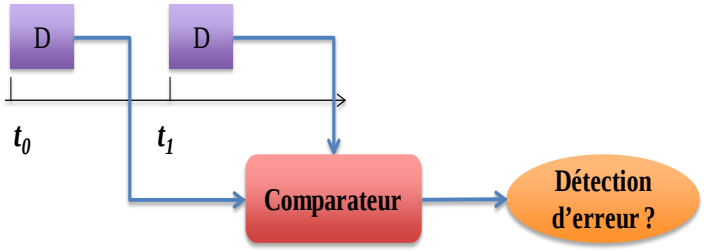


Figure I.7 : Redondance temporelle et détection d'erreurs

2) Redondance spatiale

La redondance spatiale consiste à déployer en local ou d'une façon distribuée plusieurs copies d'un même composant [Dorow 2003]. Ce composant peut être : *matériel* (par exemple, duplication des calculateurs produisant la même donnée), ou *logiciel* (par exemple, duplication d'un logiciel de calcul de commande de vol), ou encore « de données » (par exemple, duplication d'une base de donnée). L'objectif de la redondance spatiale est que dans le cas où une ou plusieurs copies de ce composant est défaillante (ne fonctionne pas correctement ou ne fonctionne pas du tout), il y aura toujours une copie du composant qui fonctionne correctement, ce qui augmente la sûreté du système en question. Dans le contexte des communications, la redondance spatiale consiste par exemple à dupliquer les canaux de communication ou les nœuds intermédiaires (les relais). Ce type de redondance permet de gérer les défaillances et de tolérer les pertes de composants (ou de données).

La redondance spatiale peut être matérielle, logicielle ou d'information. Détaillons maintenant ces 3 formes de redondances spatiales.

a) Redondance matérielle

La redondance matérielle consiste à déployer un nombre *N* (*N>=2*) de copies physiques d'un composant matériel (capteur, bus de communication, …). La redondance matérielle permet de maintenir la sûreté de fonctionnement (notamment la disponibilité) du système dans le cas de la défaillance d'un composant matériel. L'inconvénient de la redondance matérielle est qu'elle est coûteuse (en temps de fabrication, en coût matériel), augmente le poids et la taille du système ce qui n'est pas approprié aux systèmes contraignants comme les systèmes embarqués critiques. La redondance matérielle peut être active, passive ou hybride (décrites précédemment).

b) Redondance logicielle

La redondance logicielle [Dubrova 2013] permet d'assurer la sûreté de fonctionnement du système dans le cas de la défaillance d'un composant logiciel. Elle pourrait entraîner une augmentation des temps de conception, de test. La redondance logicielle peut être soit à version unique ou multi-versions.

La **redondance logicielle à version unique** est basée sur une version unique du logiciel auquel on ajoute des composants ou des fonctionnalités permettant, par différentes techniques, la détection, le recouvrement et la non propagation des erreurs.

La **redondance logicielle multi-versions** est basée sur un ensemble de versions différentes du même logiciel. Les différences entre les versions sont introduites par le biais de un ou plusieurs mécanismes de diversification suivants : différentes équipes de développement, différents langages de programmation, différents compilateurs, ou autres techniques de « diversification de conception ». Cette diversification vise à réduire le risque de mode commun de défaillance. À noter qu'on peut aussi redonder un logiciel à l'identique, mais cela n'a d'intérêt qu'en combinaison avec d'autres formes de redondance matérielles. Il existe différentes techniques de tolérance aux fautes logicielles basées sur la diversification à savoir N-version, Bloc de recouvrement, N-autotestable etc.

c) Redondance d'information (ou de données ou en valeur)

La redondance en valeur consiste à « ajouter » une partie « extra données » aux données à stocker ou à envoyer (dites « données utiles ») [Dorow 2003]. Ces extra données peuvent être regroupées dans un bloc à part des données, ou insérées/entrelacées dans les données. Dans le contexte des communications et au niveau binaire, ces « extra données » sont souvent appelées *bits de contrôle* et servent a minima à détecter des erreurs, et au mieux, à corriger. Les bits de contrôle sont générés par des algorithmes comme les codes détecteurs ou correcteurs d'erreurs, comme par exemple les codes de parité, les codes CRC (Cyclic Redundancy Code), les codes de Hamming. On peut aussi citer les fonctions de hachage ou les fonctions cryptographiques.

I.3.3.5 Exemples d'architectures redondantes

Présentons maintenant quelques architectures redondantes qui découlent des différents principes de redondance décrits précédemment. Il n'est bien sûr pas question, ni possible de décrire toutes les architectures existantes. On peut déjà citer la synthèse faite dans [Knight 2012], qui présente quatre architectures redondantes : la redondance double, la redondance double commutée, la redondance N-modulaire et la redondance hybride.

L'architecture à redondance double consiste à la duplication à l'identique d'un composant et l'utilisation d'un comparateur qui détecte une erreur si les deux composants fournissent des sorties différentes.

Tout comme l'architecture double, **l'architecture double commutée** consiste en la duplication d'un composant. Toutefois, les deux composants ne fonctionnent pas en parallèle mais il existe un commutateur qui bascule sur le deuxième composant quand une erreur est détectée au niveau du premier composant. Pour permettre la détection d'erreurs, des techniques de détection supplémentaires sont ajoutées au premier composant à savoir, notamment, la parité, les codes détecteurs, les watchdogs et des tests de vraisemblance.

L'architecture N-modulaire est basée sur le principe de la redondance double, mais cette fois en nombre N (avec $N>2$) copies. Par rapport à l'architecture double, cette architecture apporte la possibilité de masquer l'erreur par compensation, et ce, grâce à un module de vote qui permet de considérer la sortie majoritaire comme correcte. Il n'y a donc pas de réelle détection d'erreur. Ce type d'architecture est utilisé dans les systèmes qui requièrent un haut niveau de fiabilité comme, par exemple, dans les calculateurs des systèmes de commande de vol chez certains avionneurs. On peut notamment mentionner l'architecture TRIPLEX utilisée par Boeing [Yeh 1998].

L'architecture hybride est une architecture *N-modulaire* reposant sur la redondance dynamique. En effet, les N copies du composant à redonder ne fonctionnent pas en parallèle mais il y a un sous-ensemble de K copies ($2< K <N$) qui ne sera activé que lorsque le module de vote détecte une erreur (ou faute de dégradation). Cette architecture est utilisée dans les systèmes critiques à longue durée de fonctionnement comme dans le domaine de l'espace.

Parmi les autres variantes ou déclinaisons possibles d'architectures redondantes, citons également l'architecture duplex de type COM/MON (Commande/Moniteur) utilisée dans les systèmes de commandes de vol de AIRBUS. Cette solution se situe entre l'architecture à redondance double et celle à redondance double commutée. Un calculateur est composé d'une partie COM et d'une partie MON, actives toutes les deux pour calculer la même chose, mais de manière différente, l'un « surveillant » l'autre par comparaison à la sortie des deux parties.

I.3.3.6 Conclusions et redondances retenues pour l'intégrité des communications

Ce que nous avons décrit précédemment sont les différentes classes de redondance déployées dans les systèmes tolérants aux fautes dont l'objectif n'est pas forcément d'assurer l'intégrité des communications. Ces différentes formes de redondances sont rarement employées seules, mais généralement de manière combinée. Déjà, la plupart des formes peuvent se décliner par rapport à la redondance statique ou dynamique. Par exemple, des messages peuvent être répétés (redondance temporelle) via des canaux différents (redondance spatiale matérielle). Autre exemple, un logiciel multi-versions sur des processeurs diversifiés. Plus le nombre de formes combinées est grand, plus le niveau tolérance obtenu est important, mais le coût de la solution est potentiellement plus élevé. Il faut donc toujours raisonner en termes de compromis, mais surtout éviter la « sur-redondance ». À noter que le fait de combiner différentes formes de redondances s'apparente aussi à ce qui est appelé de la diversification.
Puisque la cible dans ces travaux est l'intégrité des communications, nous allons nous inspirer de ces redondances pour en retenir celles qui peuvent être déployées (ou exploitées si elles existent déjà et auxquelles on peut associer d'autres mécanismes de détection d'erreurs) pour le contrôle d'intégrité des communications. Ces redondances sont : la redondance temporelle, la redondance spatiale particulièrement la redondance d'information.

La section suivante décrit une technique de redondance d'information qui est les codes détecteurs d'erreurs sur lesquels se basent les contributions de la thèse.

I.4 LES CODES DÉTECTEURS D'ERREURS POUR L'INTÉGRITÉ DES COMMUNICATIONS

Après avoir décrit les différents mécanismes de tolérance aux fautes en présentant les différentes formes de redondance, nous déclinons notre analyse pour nous focaliser dans cette section sur la détection d'erreurs. La détection d'erreurs est une technique souvent associée à des mécanismes visant le rétablissement du système par le traitement d'erreurs et de fautes associées. Ces mécanismes sont le recouvrement, la reprise, la retransmission ou l'utilisation de la dernière donnée correcte (non erronée) reçue dans le cas des communications. Mes travaux de thèse cible uniquement la détection et non la prise en compte de la détection, cet aspect ne sera donc pas développé davantage dans ce manuscrit. Nous nous focalisons particulièrement sur les codes détecteurs d'erreurs du fait que notre but est de détecter et non pas de corriger… mais aussi le contexte des systèmes embarqués critiques nous impose l'utilisation des mécanismes avec des coûts relativement acceptables (en temps de calcul, etc.) ce qui n'est pas le cas des mécanismes de correction comme les codes correcteurs d'erreurs.

I.4.1 Étude générique : description générale des codes détecteurs et des différentes familles de codes

Même si dans la suite de nos travaux nous n'utiliserons pas des codes correcteurs, nous allons en parler dans cette section qui se veut générique sur les principes de bases des codes. Il est important de rappeler que tout code correcteur est un code détecteur et la réciproque n'est pas nécessairement vraie : il y a des codes détecteurs qui ne permettent pas la correction de l'erreur (par exemple les codes de parité). Un code détecteur (ou correcteur) d'erreurs est un « code » (une transformation, ou plus précisément une application injective) qui permet de générer à partir d'une suite de données initiales de taille *k bits* une nouvelle suite de données de taille $n = k + r$ *bits (voir* Figure I.8*)*. Les *r* bits générés sont une « image », une « signature » des données initiales. Ils constituent une redondance de données, de type redondance d'information.

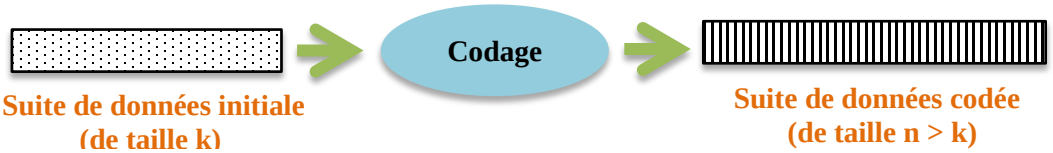


Figure I.8 : Le principe de codage dans les codes détecteurs d'erreurs

Précisons qu'un code est dit **systématique** si les *k* premiers bits de la suite de données générée par le code sont les k bits initiaux de la suite de données initiale.

De plus, un code se caractérise par son rendement *R* qui est égal au rapport de la taille de la suite de données initiale sur la taille de la suite de données codée : $R=k/n = k/(k + r)$.

Deux autres caractéristiques fondamentales des codes détecteurs ou correcteurs d'erreurs sont la **distance de Hamming** et le **poids de Hamming**.

- La **distance de Hamming** d'un code est la distance minimale entre tous les mots de code (les données générées par ce code), sachant que la distance de Hamming entre deux mots de code est le nombre de bits différents entre ces deux mots.
- Le **poids de Hamming** d'un mot est le nombre de bits à 1 qu'il contient.

Par exemple, le poids de Hamming de la suite binaire (010101) est égal à 3 et la distance de Hamming du couple de suites binaires (010101, 010011) est égale à 2

Un code se définit par son pouvoir de détection et, de correction dans le cas des codes correcteurs. Un code est capable de détecter toutes les erreurs de taille *e* (les erreurs qui inversent *e* bits) si sa distance de Hamming est supérieure à *1+e*. Un code est capable de corriger une erreur de taille *e* si sa distance de Hamming est supérieure à *1+2e*.

I.4.2 Classification des codes détecteurs d'erreurs

La Figure I.9 présente une classification des codes détecteurs d'erreurs inspirée de [Jaber 2009]. Les codes détecteurs d'erreur peuvent être répartis en trois grandes classes : les codes en blocs, les codes concaténés et les codes convolutifs. Nous détaillons ces différentes classes dans ce qui suit. Nos travaux de thèse s'appuient essentiellement sur les codes en blocs, qui seront donc les plus détaillés ici.

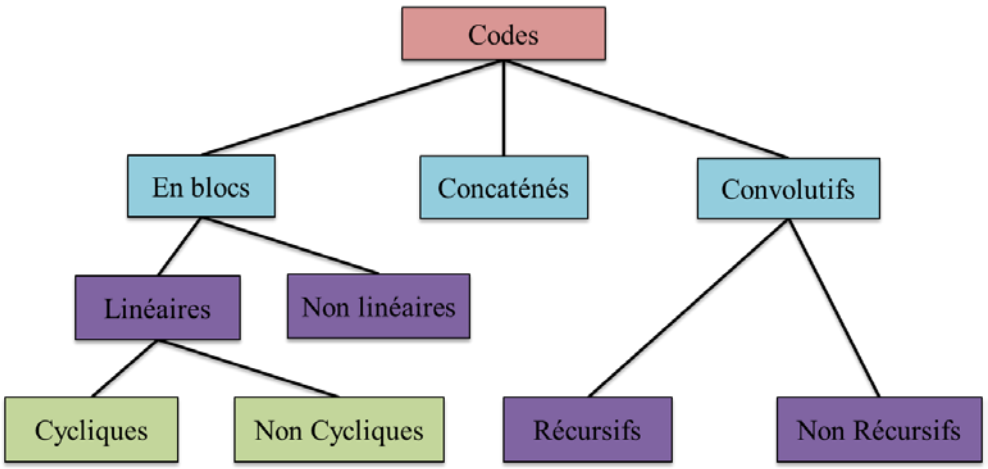


Figure I.9 : Classification des codes détecteurs d'erreurs

I.4.2.1 Les codes en blocs

Le codage en blocs consiste à décomposer la suite de données à coder en plusieurs blocs de la même taille *k* puis à transformer chaque bloc en un mot de code de taille $n = k + r$ en suivant une loi linéaire.

Pour un code en blocs, le codage et le décodage d'un bloc de données avec ce code dépendent uniquement du bloc en question et non pas des autres blocs de données constituant la suite de données à protéger. Les codes en blocs peuvent se classifier en deux sous-classes ; les codes en blocs linéaires et les codes en blocs non linéaires. Nous décrivons ces deux classes dans ce qui suit.

1) Les codes en blocs linéaires/non linéaires

Le codage dans le cas des codes en blocs linéaires est caractérisé par sa **matrice génératrice** telle que : $n = k.G$. Il est également caractérisé par la matrice *H* telle que $G.T(H) = 0$ où $T(H)$ est la matrice transposée de *H*. Pour le décodage, il faut calculer *n'*, le syndrome de la suite de données reçue, si $n'.T(H)$ est égal à 0, alors aucune erreur est détectée. Les codes en blocs qui n'ont pas ces propriétés sont des codes en blocs non linéaires.

2) Les codes en blocs linéaires cycliques/non cycliques

Un code cyclique est caractérisé par son polynôme générateur qui est unique et de degré r, r étant la longueur de la suite de données à ajouter à la suite de données initiales pour former la suite codée. Dans le cas d'un code linéaire cyclique, toute permutation circulaire d'une suite de données codées donne une autre suite de données codées. Chaque suite de données codées générée par un code cyclique est identifiée à un polynôme. Les codes en blocs linéaires qui n'ont pas ces propriétés sont des codes linéaires non cycliques. La famille des codes cycliques est généralement utilisée pour sa bonne capacité de détection, et de correction.

I.4.2.2 Les codes concaténés

La concaténation des codes peut se faire en parallèle, en série ou encore en combinant les deux ce qui est appelé la concaténation hybride. La famille des codes concaténés a été inventée dans l'objectif de minimiser la complexité du système de codage/décodage en maintenant un pouvoir de détection équivalent aux codes simples (non concaténés).

I.4.2.3 Les codes convolutifs

Ils sont appelés aussi les codes en treillis. Contrairement aux codes en blocs, le codage et le décodage d'un bloc de données dans le cas des codes convolutifs dépendent des autres blocs de données constituant la suite de données à protéger. Ils dépendent généralement des blocs de données précédemment codés. Donc, le codeur et le décodeur convolutifs disposent d'une capacité de mémorisation. Les codes convolutifs s'appliquent sur une séquence infinie de données et génèrent une séquence infinie de données codées. Ils sont très utilisés dans les systèmes de télécommunications fixes et mobiles. Précisons qu'il existe des codes convolutifs récursifs et non récursifs. Les codes convolutifs récursifs se caractérisent par une récursivité dans le processus de codage (en appliquant une boucle de retour au codeur), un exemple d'un code convolutif récursif est le RSC (Recursive Systematic Code).

I.5 CONCLUSION

Dans ce chapitre, nous avons décrit le contexte général de nos travaux de thèse en présentant tout d'abord les systèmes ciblés qui sont les systèmes embarqués critiques. Nous nous sommes focalisés par la suite sur les concepts de la sûreté de fonctionnement sur lesquels sont axés ces travaux en introduisant ses attributs, entraves et moyens. Parmi les attributs de la sûreté de fonctionnement, nous ciblons en particulier l'intégrité des communications. Nous avons détaillé cette problématique avec un focus sur le concept de la tolérance aux fautes particulièrement la redondance dont nous avons détaillé les principaux types en précisant ceux qui sont retenus dans nos travaux pour traiter la problématique de l'intégrité des communications. Finalement, nous avons détaillé les codes détecteurs sur lesquels s'axent nos travaux. Après avoir décrit les concepts fondamentaux, le chapitre suivant s'attache à faire une étude de l'existant en décrivant les différentes solutions et approches existantes pour assurer l'intégrité des communications. Nous présentons par la suite les principales orientations et contributions de nos travaux.

Chapitre II

TRAVAUX CONNEXES ET SOLUTIONS POUR L'INTÉGRITÉ DES COMMUNICATIONS

Après la présentation dans le premier chapitre du contexte général des travaux de thèse et des concepts fondamentaux, ce deuxième chapitre a pour but de présenter un état de l'existant ciblant les principaux travaux connexes proposant des mécanismes ou des approches pour assurer l'intégrité des communications dans un contexte proche du notre, celui des systèmes embarqués critiques. Étant donné que ces travaux sont généralement basés sur les codes détecteurs d'erreurs, nous commençons par présenter les principaux codes communément utilisés pour assurer l'intégrité des communications. Cela servira de référence pour les sections suivantes. Ces dernières vont décrire les différentes approches d'intégrité existantes en commençant par les différents mécanismes d'intégrité mis en œuvre dans les principaux réseaux embarqués, ces mécanismes sont déployés dans les couches basses. Nous présentons par la suite quelques solutions industrielles de surcouches pour l'intégrité des communications, ces surcouches sont basées sur le concept du canal noir et sont mises en œuvre dans les couches hautes, notamment la couche applicative. Puis, nous focalisons sur les travaux académiques sur l'intégrité des communications en décrivant les différentes approches d'intégrité des communications proposées. Nous discutons finalement toutes ces approches et solutions en positionnant nos travaux de thèse par rapport aux approches existantes et en présentant leur orientation.

II.1 PRINCIPAUX CODES DÉTECTEURS D'ERREURS POUR L'INTÉGRITÉ DES COMMUNICATIONS

Dans cette section, nous décrivons les principaux codes détecteurs d'erreurs utilisés dans les solutions et travaux connexes sur l'intégrité des communications. Ces principaux codes sont CRC, Fletcher, Adler, WSC, XOR, le complément à 1, le complément à 2 et le code de parité.

II.1.1 Les codes CRC

Les codes **CRC** font partie de la famille des codes en blocs linéaires cycliques (voir chapitre I, section I.4.2.1). Le codage CRC utilise le principe de la division polynomiale. En effet, cela consiste à diviser la suite initiale de données par le polynôme générateur. Le reste de cette division polynomiale donne la suite de bits de contrôle (aussi couramment appelée « le CRC ») à ajouter à la suite initiale de données pour former la suite codée de données à envoyer. La Figure II.1 illustre cette division polynomiale avec un polynôme générateur $G = x^3 + x + 1$ (la suite « 1 0 1 1 ») et une suite de données initiale égale à « 1 1 0 0 1 0 1 ». Le CRC à ajouter à cette suite de données initiale est égal à « 0 1 0 » qui est le reste de la division polynômiale de la suite initiale par le polynôme générateur.

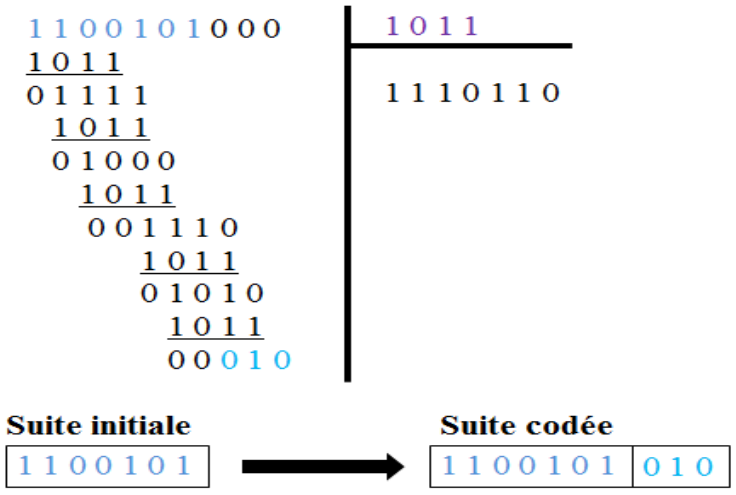


Figure II.1 : Principe de la division polynômiale des codes CRC

À la réception, la suite de données reçue (utiles + CRC) est divisée par le polynôme générateur, une erreur est détectée si le reste est non nul. À noter que cette description (ainsi que celles des autres codes) est relative au contexte des communications. Ces codes détecteurs peuvent être utilisés aussi pour l'intégrité des fichiers, bases de données, etc.

Pour un même nombre de bits de contrôle *r*, on peut avoir plusieurs CRCs, du fait qu'il existe plusieurs polynômes de même degré et chacun peut être utilisé comme polynôme générateur d'un code CRC. Quel que soit le polynôme générateur, un CRC détecte toutes les erreurs qui inversent un seul bit (erreurs simples) et toutes les erreurs en rafale avec une taille inférieure ou égale au degré de son polynôme générateur. Un CRC détecte toutes les erreurs doubles si son polynôme générateur *G(x)* a un facteur irréductible de trois termes, et toutes les erreurs impaires si *G(x)* a un facteur du type *(x + 1)*.

Nous présentons maintenant une famille particulière des codes CRC qui sont les codes **Fast CRC**. Ce sont des codes CRC avec un polynôme générateur particulier [Nguyen 2009]. Pour produire un nombre *r* de bits de contrôle, un Fast CRC ne peut utiliser qu'un unique polynôme générateur : ce polynôme est $G(x) = x^r + x^2 + x + 1$.

Par rapport aux codes CRC de base, les Fast CRC annoncent une complexité algorithmique plus faible et promettent donc des temps de calcul plus faibles. On peut cependant noter, qu'il est difficile de trouver des réalisations prouvant ces avantages annoncés.

II.1.2 Fletcher

Le code **Fletcher** est un code en bloc linéaire. C'est une somme de contrôle arithmétique (se basant sur des calculs arithmétiques) qui consiste à partager le bloc de bits de contrôle *C* de taille *r* en deux blocs *C1* et *C2* de taille *r/2* [Fletcher 1982]. *C*1 et *C2* sont initialisés à 0. Pour *C1*, on additionne itérativement octet par octet la suite des bits à protéger. À chaque itération, on affecte à *C1* la valeur de C1 modulo $2^{r/2}-1$. À chaque itération, on additionne à C2 la valeur de *C2* modulo $2^{r/2}-1$. La valeur de *C* est obtenue en mettant la valeur finale de *C2* dans les *r/2* bits de poids forts de C et la valeur finale de *C1* dans les *r/2* bits de poids faibles de C. La veleur C est à ajouter à la suite des bits initiale à envoyer ; à la réception, la somme de contrôle Fletcher de la suite de données utiles reçue est calculée et comparée à la somme de contrôle reçue. Si ces deux sommes sont différentes, une erreur est détectée. Pour un nombre donné *r* de bits de contrôle, on ne peut avoir qu'un seul code Fletcher.

II.1.3 Adler

Le code **Adler** est une variante du code Fletcher [Deutsch & Gailly 1996]. Tout comme le code Fletcher, il s'agit d'une somme de contrôle arithmétique qui consiste à diviser le bloc de bits de contrôle *C* de taille *r* en deux blocs *C1* et *C2* chacun de taille *r/2*. Mais cette fois *C1* est initialisé à 1, puis, comme pour Fletcher, on lui ajoute octet par octet la suite de bits à protéger. La différence est qu'à chaque itération de calcul, C1 n'est pas affecté par sa valeur modulo $2^{r/2}$, *mais* sa valeur modulo le **plus grand nombre premier** inférieur à $2^{r/2}$. *C2* est initialisé à 0, on lui ajoute la somme des valeurs successives de *C1*, et à chaque itération, à *C2* est affectée sa valeur modulo le plus grand nombre premier inférieur à $2^{r/2}$. Pour le décodage, la somme de contrôle Adler de la suite de données utiles reçue est calculée puis le résultat est comparé à la somme de contrôle reçue. Si ces deux sommes sont différentes, une erreur est détectée.

Tout comme pour Fletcher et Fast CRC, on ne peut avoir qu'un seul code Adler pour un nombre donné *r* de bits de contrôle.

II.1.4 WSC

Les codes Weighted Sum Codes WSC [McAuley 1994] sont basés à la fois sur un calcul polynomial et un calcul arithmétique. Les bits de données sont divisés en *n* blocs *Qi* comportant au maximum *2k/r* bits (*k* est le nombre maximum de bits que peut protéger le code et *r* est le nombre de bits de contrôle). Les bits de contrôle comportent deux blocs P_1 et P_0 à ajouter au message à envoyer dans cet ordre (P_1 puis P_0). Ils sont calculés en suivant les deux équations P_1 et P_0 :

$$P_1 = \sum_{i=0}^{n-1} W_i \otimes Q_i mod M = (W_{n-1} \otimes Q_{n-1}) \oplus \dots \oplus (W_0 \otimes Q_0)\ mod\ M$$
$$P_0 = P_1 \oplus \sum_{i=0}^{n-1} Q_i = (P_1 \otimes Q_{n-1}) \oplus Q_0$$

Sachant que *M* est un polynôme primitif de degré *r/2*[4]. Le récepteur recalcule P_1 et P_0 en utilisant les mêmes poids et le même polynôme générateur. Si les blocs calculés ne sont pas identiques à ceux qui sont reçus, une erreur est détectée.

II.1.5 XOR

Le calcul de la somme arithmétique de contrôle XOR se fait en sommant avec l'opération arithmétique « ou exclusif » (XOR) les mots de taille *m* formant la suite de bits de données initiales. À la fin du calcul, le résultat est la somme de contrôle XOR à ajouter au message. À la réception, on calcule la somme arithmétique XOR de la suite de données utiles reçue et on la compare à la somme XOR reçue, si elles sont différentes, une erreur est détectée.

II.1.6 Le complément à 1

Les bits de contrôle sont générés en additionnant les mots de *r* bits (*r* étant la taille de la somme de contrôle) composant la suite de bits de données à envoyer et en ajoutant le "report" au bit ayant le poids minimum, le résultat est le complément à 1. À la réception, on calcule la somme arithmétique complément à 1 de la suite de données utiles reçue et on la compare à la somme complément à 1 reçue, si elles sont différentes, une erreur est détectée. Une implémentation de ce code est la somme de contrôle d'Internet (Internet Checksum).

II.1.7 Le complément à 2

Les bits de contrôle sont générés en additionnant les mots de *r* bits (*r* étant la taille de la somme de contrôle) composant la suite de bits de données à envoyer et en ajoutant le "report" au bit ayant le poids minimum (comme pour le complément à 1), on ajoute 1 au résultat précédent. Le résultat final est le complément à 2. À la réception, on calcule la somme arithmétique Complément à 2 de la suite de données utiles reçue et on la compare à la somme Complément à 2 reçue, si elles sont différentes, une erreur est détectée.

II.1.8 Code de parité

Le code de parité consiste à ajouter un bit de contrôle. Ce bit appelé « bit de parité » est utilisé pour vérifier la parité paire ou impaire (pour la parité impaire, le bit de parité doit être égal à 0 si le nombre de bits à 1 de la suite de données initiale est un nombre impair). A la réception, le bit de parité de la suite de données utiles reçue est calculé et comparé au bit de parité reçu, s'ils sont différents, une erreur est détectée.

II.2 L'INTÉGRITÉ DES COMMUNICATIONS DANS LES RÉSEAUX EMBARQUÉS

Pour assurer l'intégrité des communications, on peut observer aujourd'hui un ensemble de techniques et de mécanismes de détection ou de masquage d'erreurs (rappelons que le masquage sert à empêcher la propagation et l'effet de l'erreur sans la détecter, par exemple en transmettant le message n fois et en utilisant un système de vote) qui est déployé dans les réseaux embarqués. Puisqu'ils sont communément utilisés dans les systèmes embarqués critiques, nous allons décrire les mécanismes d'intégrité déployés dans les protocoles suivants : ARINC 429, MIL-STD-1553, AFDX, TTP/C, LIN, Flexray et CAN. Le focus de cette présentation porte sur ces mécanismes d'intégrité (on ne vise pas une description exhaustive de toutes les caractéristiques de ces protocoles), et il est limité aux solutions mises en place dans les domaines que ciblent nos travaux.

II.2.1 ARINC 429

L'ARINC 429 est un bus de données utilisé particulièrement dans le domaine avionique. Nous pouvons le retrouver, par exemple, dans les Airbus A310/A320 et A330/340, les Boeing (du 727 au 767) et dans les hélicoptères Bell. C'est un bus série simplex avec des canaux de communication permettant de transmettre la donnée dans un unique sens à la fois. ARINC 429 est basé sur des communications de bout en bout. La taille des messages échangés est égale à 32 bits dont 19 bits de charge utile. Pour assurer l'intégrité des communications, l'ARINC 429 implémente une redondance d'information (voir chapitre I, section I.3.3.4) consistant en un code de parité pour protéger toute la trame.

II.2.2 MIL-STD-1553

Le bus MIL-STD-1553 (« Aircraft Internal Time Division Command/Response Multiplex Databus ») est un bus de communication qui a été développé par l'armée américaine pour l'avion militaire F-16 en 1973. Il est communément utilisé dans le domaine avionique militaire ainsi que le domaine spatial [Fuchs 2012]. Les messages échangés contiennent 20 bits dont 16 bits de charge utile. Le MIL-STD-1553 implémente les mécanismes de détection d'erreurs suivants :

- une redondance d'information consistant en un bit de parité,
- une redondance matérielle basée sur la duplication des canaux de communication.

II.2.3 AFDX

Le bus AFDX (Avionics Full DupleX switched Ethernet) est une version de l'Ethernet plus fiable et plus redondante développée et installée dans l'Airbus 380. C'est un bus « Full duplex » redondant normalisé par la partie 7 de la norme ARINC 664. La taille de trame AFDX peut atteindre 1500 octets.

Pour assurer l'intégrité des communications, l'AFDX est basé sur :

- une redondance d'information consistant en un CRC 32 bits protégeant toute la trame,
- une redondance matérielle basée sur la duplication des canaux de communication, ce qui permet l'envoi de deux copies de la même donnée ; une erreur est détectée si les deux copies reçues sont différentes.

II.2.4 TTP/C

TTP (« Time Triggered Protocol TTP ») est basé sur une topologie en étoile duplex avec un contrôle d'accès au média de type TDMA (« Time Division Multiple Access »). Pour assurer l'intégrité des communications, TTP implémente :

- une redondance d'information consistant en un CRC-16 pour renforcer l'intégrité de toute la trame,
- une redondance matérielle dynamique qui sert à dupliquer les nœuds, les nœuds dupliqués ne seront mis en fonctionnement que si les nœuds principaux défaillent.

II.2.5 LIN

Le bus LIN (« Local Interconnect Network ») est un réseau embarqué principalement utilisé dans l'industrie automobile. C'est un réseau à faible coût parfois utilisé comme sous réseau du bus CAN. Il est basé sur la norme ISO 9141. Le bus LIN est basé sur une topologie de bus en série et caractérisé par un coût et une complexité faibles. Les messages échangés peuvent contenir jusqu'à 8 octets de charge utile (64 bits).

Pour assurer l'intégrité des communications, le bus LIN implémente comme mécanismes :

- une redondance d'information consistant en une somme de contrôle arithmétique de taille 8 bits [Rahmani et al. 2007], deux bits de parité pour protéger exclusivement le champ d'identificateur et un CRC protégeant toute la trame,
- un dispositif de contrôle : chaque émetteur utilise un dispositif de contrôle pour comparer le signal envoyé et le signal observé sur le canal de communication.

II.2.6 Flexray

Flexray est un protocole de bus de données utilisé dans l'automobile qui peut être basé sur une topologie en bus, en étoile ou hybride. FlexRay est un ensemble de standards ISO de 17458-1 à 17458-5. Les messages échangés dans Flexray peuvent contenir jusqu'à 254 octets de charge utile. Une trame Flexray est composée de 5 octets d'entête, d'un champ de données utiles et d'un champ de contrôle. Flexray présente moins de flexibilité que le CAN [Paulitsch & Hall 2008]. Les principaux mécanismes implémentés par FlexRay pour assurer l'intégrité des communications sont les suivants :

- une redondance d'information consistant en deux CRC différents utilisés pour chaque trame. Un CRC-11 est utilisé pour protéger l'entête, et un CRC-24 est utilisé pour protéger à la fois l'entête et les données utiles ;
- une redondance matérielle qui est la duplication des canaux de communication : les données sont envoyées simultanément via deux canaux de communication, une erreur est détectée si les deux copies reçues sont différentes ;
- un Gardien de bus : c'est un gardien de bus local qui a été conçu pour renforcer le contrôle temporel et qui a évolué pour une mise en œuvre compliquée comportant les mêmes capacités qu'un contrôleur de communication et ayant de grandes performances en détection d'erreurs.

II.2.7 CAN

Ce bus est communément utilisé dans l'industrie et plus particulièrement dans l'automobile. Il est normalisé selon la norme ISO 11898. C'est un bus de diffusion basé sur la technique CSMA-CA. Les messages échangés via le bus CAN peuvent contenir jusqu'à 64 bits de charge utile. CAN permet la retransmission automatique des trames mais ne propose pas la redondance des canaux de communication. Le bus CAN implémente un ensemble de mécanismes et de mesures pour la détection d'erreurs [Paret 2007], dont les principaux ci-sont :

- Une redondance d'information consistant en un CRC : un CRC-15 dont le polynôme générateur est $G(x) = x^{15}+x^{14}+x^{10}+x^{8}+x^{7}+x^{4}+x^{3}+1$.
- Un « Bit stuffing » : le CAN utilise une technique de bourrage qui consiste à ajouter un bit à 0 (respectivement un bit à 1) après chaque séquence de 5 bits à 1 (respectivement 5 bits à 0) consécutifs. Ainsi, une erreur est détectée si le message reçu contient une séquence de 6 bits similaires consécutifs. C'est une autre forme de redondance d'information.
- Un « Bus Monitoring » : l'émetteur vérifie si le signal qu'il désire envoyer correspond au signal observé sur le canal physique. S'il est différent, l'émetteur envoie une trame d'erreur. Ainsi à la destination, l'erreur est détectée.
- Un « Message Frame Check »: pour détecter les trames corrompues, le CAN procède aussi par la vérification de la structure de la trame pour voir si elle est conforme au standard et détecter la corruption de données (par exemple, vérifier si le premier bit reçu est un bit dominant).

En conclusion de cette présentation des principaux réseaux embarqués (sur lesquels s'appuient plusieurs systèmes embarqués critiques) et leurs principaux mécanismes d'intégrité des communications, nous pouvons constater que les réseaux embarqués sont sensibles à l'intégrité des données en implémentant différents mécanismes pour détecter ou pour masquer les erreurs, allant des plus basiques (bit de parité) jusqu'à une panoplie de mécanismes complexes comme pour le CAN. Cependant, ces mécanismes de détection seuls ne sont pas suffisants pour atteindre un bon niveau de couverture, car ils sont limités aux couches basses des réseaux de communication. Dans le contexte des systèmes ayant de fortes exigences en sûreté de fonctionnement en général et en intégrité des communications en particulier, il est primordial d'implémenter des mécanismes d'intégrité dans les couches hautes comme, par exemple, une approche d'intégrité bout en bout mise en œuvre dans la couche applicative. Dans ce qui suit, nous présentons quelques solutions existantes proposant des approches d'intégrité bout en bout dans l'objectif d'assurer l'intégrité des communications indépendamment des couches basses. Par conséquent, ces approches viennent compléter les mécanismes d'intégrité déployés dans les réseaux/protocoles des couches basses.

II.3 SOLUTIONS BOUT EN BOUT POUR L'INTÉGRITÉ DES COMMUNICATIONS

Dans cette section, nous explorons quelques solutions standardisées ou des solutions génériques (sur étagère, Component On The Shelf COTS) pour la fiabilité des communications en général et couvrant ainsi l'intégrité. En effet, les standards comme l'IEC 61508 se sont intéressés à cette problématique. Ces standards ciblent toutes les entraves à la fiabilité des communications comme la corruption, la perte, etc. Ils préconisent un ensemble de mesures et règles pour la fiabilité (y compris l'intégrité) des communications comme l'utilisation des codes détecteurs d'erreurs. Deux approches de fiabilité des communications sont préconisées par les standards : une approche basée sur le concept « *canal noir* » (« *black channel* ») qui s'inspire du contrôle d'intégrité bout en bout (voir chapitre I, section I.3.1.2) et une autre basée sur le concept « *canal blanc* » (« *white channel* »). Dans ce qui suit, nous allons tout d'abord détailler ces deux concepts, puis nous décrirons quelques solutions standardisées adoptant le concept canal noir (intégrité bout en bout) pour l'intégrité des communications : FSoE, OpenSafety et PROFIsafe.

II.3.1 Canal noir versus Canal blanc

Le concept du « canal noir » utilisé pour assurer des communications intègres et sûres hérite son nom du concept « boîte noire » utilisé dans la conception logicielle. Ces deux concepts sont tous les deux basés sur le fait de ne pas voir ce qui est dans la « boîte » (qui sont les couches sous-jacentes dans le contexte des communications). Dans le contexte des communications, le canal blanc consiste à implémenter des mesures de sûreté sur tous les composants et ce à partir de la couche la plus basse (couche physique). Cette approche est coûteuse puisqu'elle implique/nécessite la certification de tous les composants dans toutes les couches. A l'opposé, les approches basées sur le concept du canal noir ne nécessitent pas de certifier tous les composants. Le fondement de ces approches est de considérer le réseau sous-jacent et les communications comme des boîtes noires, et de préconiser d'ajouter une couche de fiabilité (« Safety layer ») au niveau applicatif. Le concept canal noir dans le contexte de l'intégrité des communications est à rapprocher du contrôle d'intégrité bout en bout.

Le concept de canal noir/blanc a été introduit dans l'édition 2 de la norme IEC 61508 (voir chapitre I, section I.2.4) parue en avril 2010. La Figure II.2 décrit le concept canal noir/canal blanc pour la norme IEC 61508 (la source de la figure est la norme IEC 61508-2).

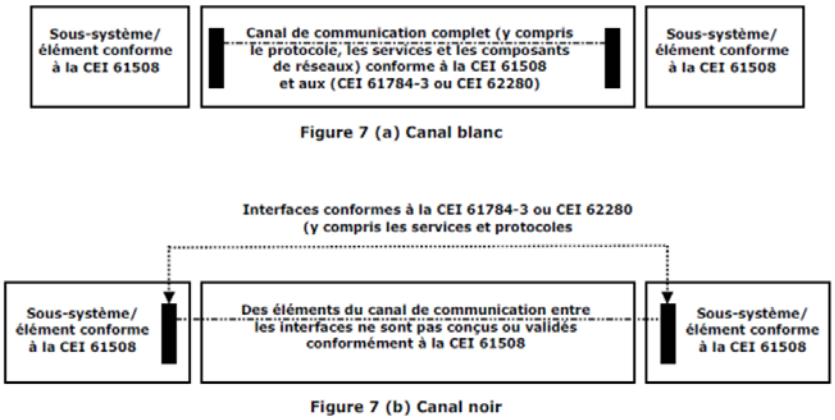


Figure II.2 : Les concepts « canal noir/canal blanc » dans la norme IEC 61508

Un exemple de couche de fiabilité basée sur le concept du canal noir est celle proposée par l'IEC 62280-1 pour la signalisation ferroviaire. Les avantages du concept canal noir sont :

- une implémentation plus facile et plus rapide que les approches « canal blanc »,
- l'indépendance par rapport aux réseaux sous-jacents : le canal noir fait abstraction du matériel réseau réellement utilisé,
- un coût réduit,
- la possibilité de réutiliser les mécanismes/protocoles/réseaux déployés dans les couches basses et renforcer l'intégrité des communications sans modifier les couches basses,
- la facilité de certification puisque c'est seulement la couche d'intégrité applicative qui est concernée par la certification : dans le cas d'un système critique soumis à une certification, il est plus important pour l'industriel de montrer que la sûreté du système repose sur un mécanisme de haut niveau facilement compréhensible et crédible (par rapport aux normes de certification)

Après avoir défini le concept du canal noir, dans ce qui suit, nous allons décrire quelques solutions se basant sur ce concept.

II.3.2 PROFIsafe

Basée sur les bus de terrain PROFIBUS ou PROFINET, PROFIsafe[1] est une technologie pour les communications sûres s'appuyant sur le concept canal noir préconisant une couche de sûreté située au niveau applicatif. PROFIsafe peut être utilisée pour les communications critiques de niveau SIL3 (voir chapitre I, section I.2.4). PROFIsafe a été standardisée par l'IEC 61784-3-3.
Pour l'intégrité des communications, PROFIsafe s'appuie sur :

- un code CRC applicatif,
- un contrôleur de séquencement des messages.

L'article [Akerberg *et al.* 2010] illustre l'utilisation de la technologie PROFIsafe pour la mise en œuvre d'une couche de sûreté dans le cas de communications sans fil utilisant la technologie WirelessHART basée sur le protocole HART, qui est une norme de communication internationale de communication ouverte sans fil pour l'industrie des processus.

Dans l'article [Akerberg & M. Bjorkman 2009], les auteurs reviennent sur la technologie PROFIsafe, mais cette fois-ci ils s'intéressent aux aspects sécurité-confidentialité (security) de la solution. Ils montrent que les trames peuvent être attaquées de manière malveillante et proposent des mesures permettant d'assurer une intégrité et une authentification des trames vis-à-vis de fautes malicieuses.

[1] http://www.profibus.fr/technologies/profisafe

II.3.3 FSoE

FsoE est une technologie développée au-dessus de EtherCAT[2] selon la norme IEC 61508 et standardisée par l'IEC 61784-3. C'est un bus de terrain orienté temps réel qui convient aux applications critiques avec un niveau d'intégrité allant jusqu'au SIL 3. La surcouche permettant d'assurer des propriétés de sécurité-innocuité des communications (« safety layer ») appelée « Fail Safe over EtherCAT » préconise que les trames de données standards et les trames de données critiques sont transférées via le même canal de communication. Mais, les trames des données critiques contiennent les données utiles critiques et un champ de contrôle applicatif (généré par un code détecteur applicatif). Plus précisément, la séquence de données critiques à protéger est découpée en fragment de deux octets et à chaque fragment est appliqué un CRC applicatif de 16 bits. Pour en savoir plus, il est possible de se reporter à une description[3] et une présentation[4] disponibles sur le site de EtherCAT.

L'article [Liu & Song 2012] est une petite illustration de l'utilisation de la solution FSoE au cas du contrôle d'un servomoteur.

II.3.4 openSAFETY

openSAFETY[5] est un protocole de communication certifié par l'IEC 61508 et selon les exigences SIL 3. openSAFETY peut être adopté pour tous les protocoles de communication sous-jacents. openSAFETY est indépendant des technologies sous-jacentes, matérielles, logicielles et réseaux ; il est totalement ouvert, d'un point de vue technique et légal. Pour l'intégrité des communications, la séquence de données utiles de chaque trame est dupliquée, chaque dupliqua est protégé par un CRC. openSAFETY met aussi en place une surveillance temporelle des données (voir la structure de la trame[6] sur le site de openSAFETY).

L'article [Soury *et al.* 2015] présente, pour un système de levage, le cas de la transition d'une solution à base de composants électromécaniques vers un système de commande/contrôle en réseau. Les auteurs utilisent la solution openSafety au niveau du réseau pour être compatible avec les exigences SIL3 de la norme IEC 61508. Il s'agit de travaux développés dans le cadre du projet ADN4SE (Atelier de Développement et Noyau Pour Système Embarqué) financé dans le cadre du programme BGLE (Briques Génériques du Logiciel Embarqué) des investissements d'avenir.

Ces différentes solutions de surcouches visant à assurer l'intégrité des communications indépendamment des couches basses profitent des divers avantages du concept du canal noir. Ce sont des solutions appropriées aux systèmes embarqués critiques où la certification est un enjeu crucial et où prouver la sûreté de fonctionnement de tous les composants de toutes les couches est une tâche très complexe. Dans ces travaux de thèse, nous allons alors nous inspirer de ces solutions et nous allons proposer une approche d'intégrité bout en bout basée sur le concept canal noir.

---

[2] https://www.ethercat.org/
[3] https://www.ethercat.org/download/documents/pcc0107_safety_over_ethercat_e.pdf
4 http://www.ethercat.org/pdf/english/Safety_over_EtherCAT_Overview.pdf
5 http://www.open-safety.org/
6 http://www.open-safety.org/index.php?id=23&L=olrvrriytb

II.4 TRAVAUX ACADÉMIQUES SUR L'INTÉGRITÉ DES COMMUNICATIONS

Dans cette section, nous présentons les différents travaux académiques sur l'intégrité des communications. Ces travaux se situent dans un contexte et ont des hypothèses proches des nôtres. Parmi les approches d'intégrité proposées dans la littérature, nous retenons trois grandes familles qui sont les approches basées sur les codes détecteurs d'erreurs, les approches basées sur les architectures redondantes et les approches basées sur la diversification des données.

II.4.1 Approches basées sur les codes détecteurs d'erreurs

II.4.1.1 Utilisation des CRC (à différents niveaux)

Pour assurer l'intégrité des communications dans les systèmes embarqués critiques, beaucoup de travaux sont basés sur l'utilisation des codes détecteurs d'erreurs. Le code détecteur d'erreurs le plus communément utilisé dans les travaux connexes aux nôtres est le CRC. Habituellement, les concepteurs choisissent des CRC conventionnels déjà adoptés par des protocoles, normes ou standards (par exemple, le CRC-32 utilisé dans l'IEEE 802.3). Pourquoi une telle pratique ? Parce qu'il faut reconnaître que la recherche d'un CRC optimal exige un très long temps de recherche (il faut tester un espace d'état de 2^k polynômes générateurs pour trouver le meilleur CRC de degré k). Ce temps de recherche est souvent trop coûteux, voire irréaliste. C'est ce qui a incité beaucoup de concepteurs à se contenter d'utiliser les CRC standardisés, donc éprouvés, mais pas nécessairement les plus performants en termes, par exemple, de temps de calcul ou de capacités de détection.

En résumé, les CRC standardisés étaient souvent sélectionnés parmi un espace réduit de polynômes générateurs et ne sont pas forcément optimaux.

Pour accélérer la recherche et ne pas être limité à un ensemble particulier de polynômes générateurs, les travaux de [Wagner 1986] se sont basés sur une propriété mathématique des codes CRC. Il s'agissait de la linéarité du codage CRC, traduite par :

CRC (A xor B) = CRC (A) xor CRC (B)

Cette propriété accélère le temps d'évaluation des performances des CRC du fait qu'elle permet de stocker une partie du résultat dans des tables. En effet, si on suppose que les données A sont soumises au vecteur d'erreur *B*, c'est-à-dire A' les données reçues et erronées sont telles que A' = *A xor B alors CRC (A') = CRC (A xor B) = CRC (A) xor CRC (B).*

Plus tard, pour chercher les polynômes optimaux, les auteurs [Wolf & Blakeney 1988] ont utilisé une technique analytique permettant une évaluation exacte de la probabilité de non détection d'erreur par des polynômes CRC. Cette technique a permis de proposer un nouveau CRC avec 16 bits de contrôle appelé le CRC-16Q ayant des meilleures performances (meilleure distance de Hamming) que celles des CRC 16 bits standardisés.

Enfin, les travaux de [Chun & Wolf 1994] ont proposé une méthode matérielle de recherche des CRC optimaux. Ils se sont limités aux polynômes divisibles par *(x+1)* et ont confirmé que le CRC-16Q est le meilleur CRC en termes de pouvoir de détection parmi cet ensemble particulier de polynômes générateurs.

Ce que nous remarquons, c'est que ces travaux se sont limités à des ensembles particuliers de polynômes générateurs, par exemple la famille des polynômes divisibles par *(x+1),* et ce, dans l'objectif d'éviter l'explosion combinatoire de l'espace de la recherche, mais également dans l'objectif de garantir certaines propriétés, à savoir, par exemple, la capacité de détecter toutes les erreurs impaires dans le cas des polynômes divisibles par *(x+1).*

Ces approches visant à accélérer le temps de recherche de polynômes générateurs n'ont donc pas permis d'explorer tous les polynômes générateurs pour une taille donnée de bits de contrôle, et encore moins pour toutes les tailles. L'approche dominante était de prendre un CRC standardisé, comme décrit par les auteurs de [Press *et al.* 2002], la sélection d'un CRC était considérée comme une histoire de « sagesse conventionnelle ».

La question de fond qui en est ressortie, est de savoir si ces polynômes standardisés sont vraiment optimaux. De nombreux travaux ont alors tenté d'apporter des réponses à cette question.

Ainsi, dans les travaux de [Chakravarty 2001], l'auteur confirme que certains CRC standardisés sont « sous-optimaux ». Il explique ce choix sous-optimal dans les CRC standards par le fait que ceux-ci ont été choisis initialement pour des messages longs ce qui n'est pas le cas des messages courts échangés dans toute une partie des réseaux embarqués. Cependant, limitée par le besoin d'une puissance de calcul très importante, la recherche des CRC optimaux s'est d'abord limitée au début à la recherche de polynômes divisibles par *(x+1).*

Dans [Chakravarty 2001] et [Koopman & Chakravarty 2004], les auteurs confirment que les polynômes générateurs standards ne sont pas optimaux en termes de pouvoir de détection. En explorant, pour des nombres de bits de contrôle allant de 3 à 16 bits et pour des tailles de données allant de 8 à 2048 bits, et en testant tous les polynômes générateurs correspondants, les auteurs découvrent qu'il existe des polynômes non standardisés ayant un meilleur pouvoir de détection. Ils proposent par exemple un CRC 12 bits qui est plus performant que le CRC CITT 16 bits utilisé communément dans les réseaux embarqués, typiquement pour protéger des messages de taille 64 bits [Chakravarty 2001].

Pour un nombre de bits de contrôle de 32 bits, les travaux de [Koopman 2002] montrent que les CRC-32 conventionnellement utilisés ne sont pas optimaux. La métrique utilisée dans ces travaux pour évaluer les performances des CRC est la Distance de Hamming (DH, voir chapitre I, section I.4.1).

Les travaux de [Koopman 2002] proposent un CRC-32 ayant le polynôme générateur suivant :

$$G(x) = x^{32}+x^{30}+x^{29}+x^{28}+x^{26}+x^{20}+x^{19}+x^{17}+x^{16}+x^{15}+x^{11}+x^{10}+x^{7}+x^{6}+x^{4}+x^{2}+x+1)$$
$$= (x+1)(x^{3}+x^{2}+1)(x^{28}+x^{22}+x^{20}+x^{19}+x^{16}+x^{14}+x^{12}+x^{9}+x^{8}+x^{6}+1).$$

Ce polynôme atteint une distance de Hamming de 6 jusqu'à une taille de données égale à 16360 bits et une DH de 4 jusqu'à une taille de données égale à 1114663 bits. Ce CRC dépasse en performance le CRC-32 utilisé dans l'Ethernet surtout pour les données de taille inférieure à 16360 bits (ce qui est le cas dans les systèmes embarqués basés sur des messages de petite taille).

Dans [Koopman & Chakravarty 2004], il a également été prouvé que le choix des polynômes générateurs dépend intimement de la taille des données à protéger et que donc un polynôme peut être performant pour une taille particulière de données et non performant pour d'autres tailles.

Dans [Powell 2001], l'auteur propose une approche d'intégrité basée aussi sur les CRC, la spécificité de cette approche est que les CRC font appel à des clés (« Keyed CRC »). En effet, chaque canal de communication donne une clé différente à chacun des autres canaux de communication. Deux aspects d'intégrité sont ciblés par cette approche, l'intégrité de la source et de la destination et le non altération de données (voir chapitre I, section I.3.1.1).

II.4.1.2 Approche d'intégrité évolutive basée sur les CRC

Présentons maintenant d'autres travaux sur les CRC [Youssef 2005] et [Youssef *et al.* 2006], développés par notre équipe de recherche et qui sont à l'origine des travaux actuels de cette thèse. Ces travaux proposent une approche d'intégrité évolutive utilisant les codes CRC et visant les communications dans un système de contrôle commande avionique. Ces travaux raisonnent sur un lot de messages et non pas sur un message unique comme le font tous les autres travaux présentés, ceci du fait que dans le système considéré plusieurs copies du même message sont envoyées selon une redondance temporelle. L'idée était d'utiliser un code CRC différent dans chaque cycle d'envoi de messages, de sorte que les codes CRC utilisés soient complémentaires entre eux pour augmenter le pouvoir de détection résultant en raisonnant sur un lot de messages et non plus sur un message unique à chaque fois, et éviter ainsi le mode commun de défaillance. La Figure II.3 décrit l'approche avec trois CRC complémentaires CRC1, CRC2 et CRC3 dans le contexte d'une erreur répétitive et non détectée par CRC1. Le fait de diversifier les CRC augmente le pouvoir de détection global de l'approche.

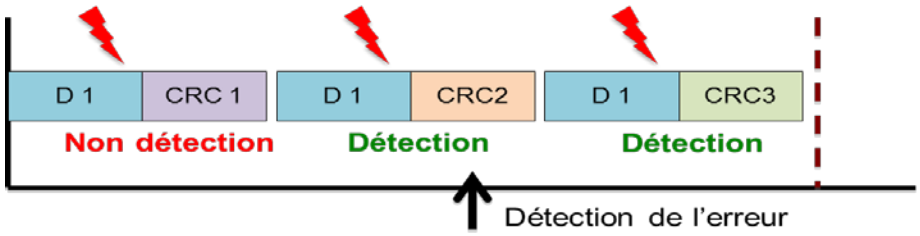


Figure II.3 : Approche d'intégrité évolutive

La stratégie de choix de cet ensemble des polynômes CRC consiste à choisir des polynômes générateurs ayant un minimum de facteurs en commun. En effet, en se basant sur la propriété mathématique qui dit que le CRC ne détecte pas les erreurs multiples de son polynôme générateur (dans ce cas l'erreur est multiple du polynôme générateur de CRC1), les polynômes candidats sont factorisés en polynômes irréductibles et sélectionnés de telle sorte qu'ils partagent un minimum de facteurs en commun (dans ce cas, l'erreur ne peut pas être multiple de tous les polynômes générateurs des CRC complémentaires retenus) pour réduire le risque du mode commun de défaillance.

La Figure II.4 décrit la sélection des CRC complémentaires. Les deux polynômes générateurs ayant des facteurs en commun avec G1 (le polynôme qui a P3 comme facteur commun avec G1 et le polynôme qui a P2 et P3 comme facteurs communs avec G1) sont considérés comme des polynômes non complémentaires à G1.

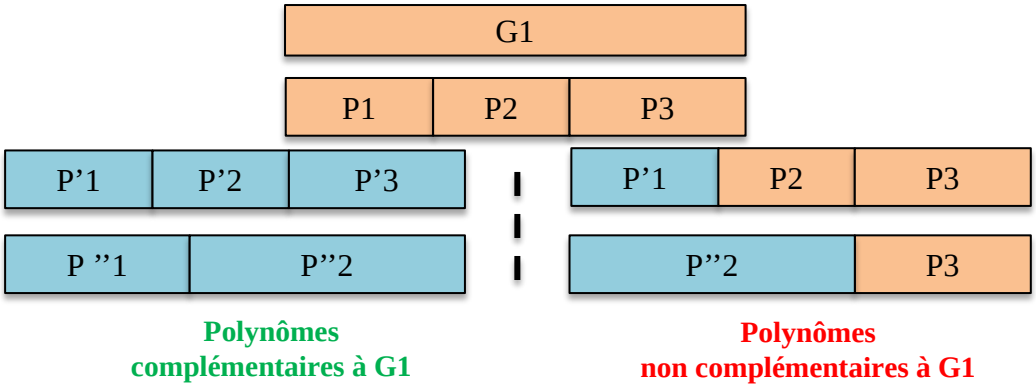


Figure II.4 : Complémentarité des codes CRC

II.4.1.3 Utilisation des sommes de contrôle arithmétiques

Du fait que les CRC se caractérisent par un temps d'exécution relativement long, certains travaux proposent des approches d'intégrité utilisant des sommes de contrôle arithmétiques au lieu des CRC. L'idée est de trouver un compromis entre le coût en temps d'exécution et l'efficacité en pouvoir de détection pour satisfaire les exigences et les contraintes de certains systèmes et particulièrement les systèmes embarqués critiques.

En étudiant les travaux connexes, nous avons remarqué que les sommes de contrôle communément utilisées dans les systèmes embarqués avec des ressources limitées sont :

- le « XOR »,
- l'addition complément à 1,
- l'addition complément à 2.

Ces sommes de contrôle sont simples et faciles à mettre en œuvre. Selon [Plummer 1978], le XOR a le coût le plus faible en temps d'exécution. Le complément à 2 a un meilleur pouvoir de détection que le XOR pour un coût en temps d'exécution similaire. Le complément à 1 est légèrement plus coûteux que les deux dernières sommes de contrôle ; il est utilisé par le protocole TCP (« Transmission Control Protocol »). Ainsi, ces trois sommes de contrôle ne nécessitent pas d'importantes puissances de calcul, mais ont des pouvoirs de détection relativement faibles ce qui n'est pas convenable pour le niveau de sûreté exigé par les systèmes critiques.

Pour résoudre ce problème de faible pouvoir de détection, tout en prenant en compte la contrainte de temps de calcul de certains systèmes, deux sommes de contrôle arithmétiques ont été proposées :

- la somme de contrôle Fletcher,
- et plus tard, en guise d'amélioration de Fletcher, a été proposée la somme de contrôle Adler.

Cependant, Fletcher et Adler restent plus coûteux en temps d'exécution comparés aux sommes de contrôle décrites précédemment. Ainsi, dans l'objectif d'optimiser ce temps d'exécution, des propositions d'optimisation de l'implémentation du Fletcher ont été décrites par [Nakassis 1988], [Sklower 1989] et [Kodis 1992]. Nous pouvons appliquer ces optimisations à Adler.

Dans [Maxino 2006a], [Maxino 2006b] et [Maxino *et al.* 2009] est décrite une évaluation des performances des codes Adler, Fletcher, complément à 1 et CRCs pour des nombres de bits de contrôle 8, 16 et 32 bits, et pour une taille de message allant jusqu'à 2560 bits. Les auteurs confirment que les performances des sommes de contrôle dépendent de plusieurs facteurs à savoir :

- l'algorithme utilisé,
- le nombre de bits de contrôle,
- la taille de la donnée à protéger et son type.

Ils en ont déduit les constats suivants :

- Les CRC dépassent toujours les sommes de contrôle en pouvoir de détection, mais avec des temps d'exécution beaucoup plus longs.
- Adler et Fletcher
  - dépassent le complément à 1 en pouvoir de détection, mais toujours avec un coût plus important en temps d'exécution,
  - ont des coûts similaires avec un pouvoir de détection légèrement supérieur pour Fletcher.
- Pour les erreurs en rafales, le complément à 2 et le CRC sont meilleurs que le complément à 1, Fletcher et Adler.
- Pour les autres types d'erreurs, un « bon » CRC présente le meilleur choix, il augmente la distance de Hamming au moins de 1 par rapport aux autres sommes de contrôle, mais il présente quand même un facteur de 2 à 4 en temps de calcul.
- Pour les systèmes à fortes contraintes en temps de calcul, Fletcher est le meilleur choix. Il a un coût de calcul plus faible qu'Adler et meilleur en pouvoir de détection.
- Pour les systèmes ne pouvant pas adopter Fletcher ou Adler pour leurs coûts de calcul, le complément à 1 peut être utilisé ou substitué par le complément à 2. Le XOR ne doit pas être utilisé, parce qu'avec un même coût de calcul, il a un pouvoir de détection plus faible que le complément à 1 et le complément à 2.

Dans les travaux [Partridge *et al.* 1995], [Stone *et al.* 1998] et [Stone & Partridge 2000], les auteurs comparent les performances de l'addition complément à 1, Fletcher et CRC. Leurs constats sont que CRC dépasse toujours les sommes de contrôle en pouvoir de détection.

Enfin, signalons un autre code, le Weighted Sum Code (WSC) qui a été proposé par [McAuley 1994]. Il a été conçu comme une alternative aux codes Fletcher et CRC. Dans [Feldmeier 1995], l'auteur compare la performance en détection d'erreurs de WSC à celles du code complément à 1, XOR, code de parité, Fletcher et CRC. Il confirme que WSC a une performance équivalente à celle des codes CRC et un coût de calcul équivalent à celui de Fletcher. WSC n'a été adopté à ce jour par aucune application industrielle et donc son utilisation manque de maturité, donc nous ne retenons pas ce type de code pour notre approche.

II.4.1.4 Les codes flexibles

Les codes flexibles constituent une autre approche d'intégrité basée sur les codes détecteurs et correcteurs, et a été présentée dans [Saiz-Adalib *et al.* 2013]. En effet, les auteurs considèrent que dans un message, les données ne sont pas toutes soumises au même taux de bit erronés (BER) selon leur position dans le message (le message est alors vu avec différentes « zones » de données). De ce fait, les exigences d'intégrité ne sont pas forcément les mêmes selon les zones. Les auteurs s'inspirent de travaux existants proposant des codes de contrôle d'erreur différents en préconisant de diviser la trame en deux zones et d'appliquer à chaque zone un code détecteur (ou correcteur selon les exigences) différent. Cependant, les auteurs de [Saiz-Adalid *et al.* 2013] estiment que cette proposition est rigide du fait qu'elle propose de diviser la trame que par deux. Ils proposent alors une approche plus flexible qui permet de diviser la trame en n'importe quel nombre *k* de zones pas nécessairement de la même taille et d'attribuer à chaque zone un code de protection contre les erreurs adapté aux exigences en intégrité de la zone considérée.

La figure II.5 décrit ce concept de découpage de la trame en différentes zones. La taille différente des champs de contrôle est un exemple d'illustration d'adaptation du code détecteur aux exigences de la zone associée : dans le cas de la Figure II.5, on considère que la zone trois est la plus critique et la zone 2 est la moins critique.

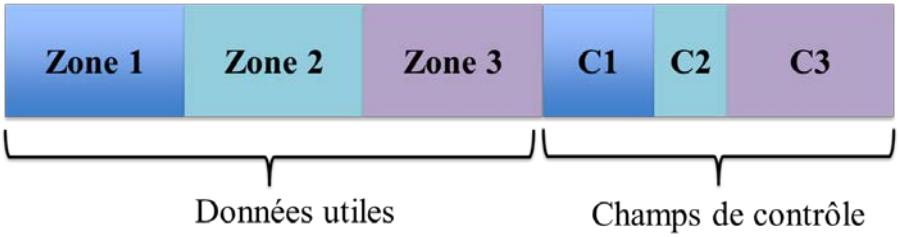


Figure II.5 : Découpage de la trame en plusieurs zones

II.4.2 Approches basées sur la redondance matérielle

Une autre grande famille d'approches de détection d'erreurs basée aussi sur la redondance est celle consistant à dupliquer les canaux de communication, cela veut dire envoyer k copies du message sur différents canaux de communication et les comparer à la réception pour détecter des éventuelles erreurs.

Il existe plusieurs stratégies de détection, la plus commune étant l'utilisation d'une unité de vote qui sert à considérer comme correcte la copie ayant le vote majoritaire (la copie reçue par la majorité des destinataires).

Dans ce contexte, un exemple type est donné dans [Paulitsch & Hall 2007] qui propose une solution architecturale dont l'objectif est d'assurer la disponibilité et l'intégrité des messages échangés. Dans cette approche, chaque nœud reçoit une copie du message de ses voisins et des voisins de ses voisins. Pour détecter les erreurs, une stratégie de vote est mise en œuvre pour comparer les différentes copies reçues.

II.4.3 Approches basées sur la diversification des données

II.4.3.1 Description générale de la diversification des données

Utilisée généralement pour la sûreté de fonctionnement des logiciels dans l'objectif de tolérer des fautes de conception (surtout le mode commun de défaillance, voir chapitre I, section I.2.2.1), l'idée de la diversification de données est venue de l'observation que les défaillances logicielles peuvent être dépendantes des données en entrée et donc un programme défaille généralement si les données en entrée ont des valeurs particulières [Hiller 1998], [Ammann & Knight 1988] et [Dubrova 2013]. L'ensemble de ces données en entrée qui engendre la défaillance du logiciel est appelé « domaine de défaillance » du logiciel.

L'idée de base de la diversification de données est alors fondée sur le fait que si les valeurs des données en entrée sont modifiées (ou ré-exprimées, voir Figure II.6, F est la fonction de réexpression et D' est la suite de données ré-exprimées à partir de D, le risque d'activer la même défaillance serait réduit. C'est pour cela qu'il a été proposé de diversifier (ré-exprimer) les données en entrée.

Comme décrit dans [Ammann & Knight 1988], la performance de l'approche basée sur la diversification de données dépend de la capacité de la fonction de réexpression à produire des données en dehors du domaine de défaillance, pour (ou à partir de) données appartenant au domaine de défaillance.

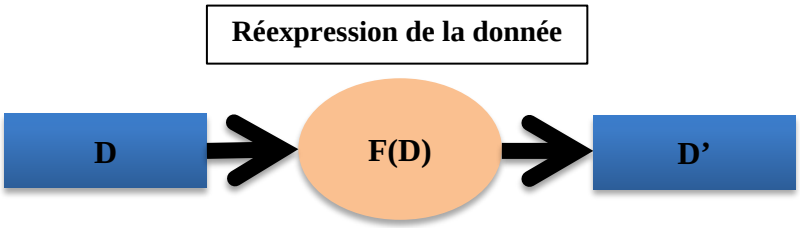


Figure II.6 : Principe général de réexpression de données

D'après [Dubrova 2013] et dans le contexte de sûreté de fonctionnement logicielle, il existe trois techniques de diversification de données :

- Réexpression des données en entrée sans ajustement ultérieur.
- Réexpression des données en entrée avec ajustement ultérieur : le résultat doit être ajusté ultérieurement selon un ensemble de règles. C'est une procédure de restitution qui se base généralement sur les fonctions inverses de celles de réexpression des données pour restituer la suite initiale de données.
- Réexpression des données avec décomposition et recomposition des données : dans ce cas, l'ensemble de données en entrée est décomposé en sous-ensembles. La recomposition sert à restituer la suite initiale de données.

D'après [Hiller 1998], et toujours dans le contexte de sûreté de fonctionnement logicielle, deux autres approches de mise en œuvre de la diversification de données sont proposées (nous avons adapté les définitions originales pour les rendre plus génériques) :

- « **Bloc de réessaie** » (voir Figure II.7 **:** il s'agit d'utiliser d'abord des données initiales sans réexpression. Si le résultat est accepté alors pas de défaillance signalée. Sinon, tant que le délai n'est pas expiré (cela concerne surtout les applications temps réel), les données sont ré-exprimées et toute la procédure est refaite jusqu'à l'obtention d'un résultat correct ou l'expiration du délai.

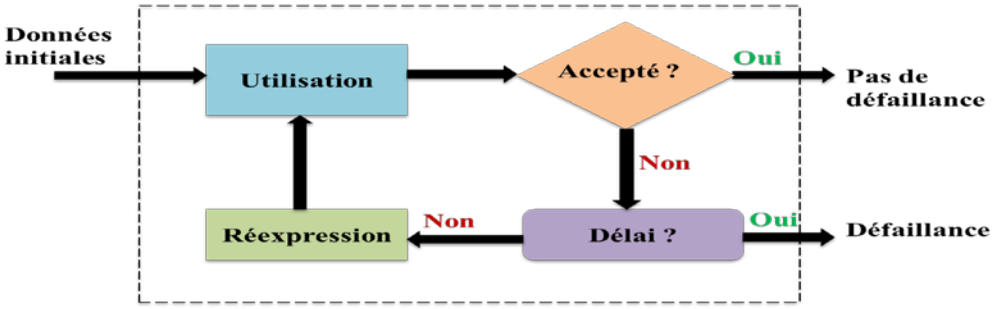


Figure II.7 : Mise en œuvre « Bloc de réessaie » de la diversification de données

- « **Programmation n-copies** » (voir Figure II.8 : il s'agit de ré-exprimer les données initiales pour en avoir *n* exemplaires différents. Les résultats d'exécution de ces différents exemplaires sont soumis à un vote. S'il y a un accord (un résultat majoritaire), le résultat voté est considéré comme correct et il n'y pas de défaillance signalée. Sinon, une défaillance est détectée.

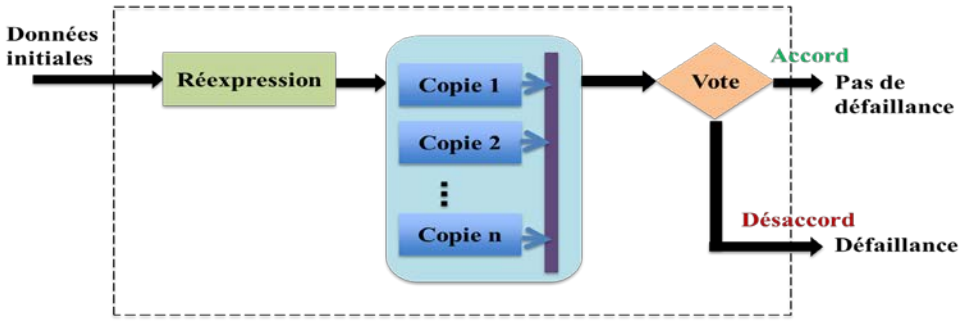


Figure II.8 : Mise en œuvre « Programmation n-copies» de la diversification de données

II.4.3.2 Diversification des données et intégrité des communications

Comme nous l'avons décrit dans la section précédente, le concept de diversification de données est utilisé généralement dans l'objectif d'assurer la sécurité du logiciel [Nguyen-Tuong *et al.* 2008]. Cependant, dans les travaux de [Ammann & Knight 1988], la diversification des données est utilisée pour l'intégrité des données stockées. Dans leur article, les auteurs définissent deux propriétés que doit vérifier toute fonction de réexpression :

- La propriété « *inverse* » : cette propriété consiste au fait que chaque fonction de réexpression a une fonction inverse. Cela permet de restituer la donnée originale. Cette propriété réduit la probabilité de perdre la sémantique de la donnée réellement envoyée et permet de détecter les erreurs si les données restituées sont différentes des données reçues.
- La propriété « *incohérence* » : elle signifie que si on applique deux fonctions de réexpression différentes sur la même copie de données, on aura deux résultats différents. Cela permet la détection systématique des erreurs dans le cas où deux copies reçues sont similaires après avoir été ré-exprimées à l'aide de deux fonctions différentes.

Cette approche peut être adoptée pour assurer l'intégrité des communications. Cependant, il faut l'associer à une stratégie de détection ou de masquage d'erreur (via un système de vote massif par exemple).

II.5 DISCUSSION ET ORIENTATION DE NOS TRAVAUX

Des travaux connexes et solutions existantes pour l'intégrité des communications, nous avons pu identifier certaines limitations à savoir l'insuffisance des mécanismes d'intégrité implémentés dans les réseaux embarqués. Pour ceci, nous allons proposer une approche d'intégrité bout en bout, une surcouche à implémenter dans la couche applicative. Notre solution sera basée sur le concept « canal noir » introduit par l'IEC61508, comme le cas des solutions présentées dans ce chapitre (PROFIsafe par exemple) qui s'inspirent du concept d'intégrité bout en bout. De part la maturité de l'utilisation des codes détecteurs d'erreurs dans les travaux connexes et le retour d'expériences qui prouve leurs performances probantes, nous les avons retenus pour notre approche d'intégrité. Par rapport aux travaux existants qui ciblent généralement des erreurs de multiplicité faible, une hypothèse fondamentale pour notre travail est le fait que nous considérons que tous les bits de la trame peuvent être erronés.

Un ensemble de leçons apprises des travaux connexes à prendre en considération par notre approche d'intégrité à savoir le fait que les hypothèses et la spécificité du système étudié ont un impact significatif sur la politique de détection d'erreurs à adopter.

Les auteurs de [Paulitsch *et al.* 2005] confirment que la probabilité de détection d'erreurs des codes CRCs dépend de beaucoup de facteurs à savoir :

- le pouvoir intrinsèque du code utilisé qui dépend notamment du polynôme générateur dans le cas des CRC (dans le cas des erreurs à multiplicité faible)
- la probabilité d'occurrence d'erreurs affectant les données
- la taille des données à protéger

En outre, ils montrent que, dans beaucoup de travaux sur la détection d'erreurs dans les systèmes embarqués critiques, les hypothèses prises ne reflètent pas toujours les vraies exigences du système considéré. Ces hypothèses sont :

- des nœuds intermédiaires passifs
- des canaux de communication sans mémoire
- des erreurs aléatoires et indépendantes

Ces hypothèses qui ne sont pas nécessairement réalistes sont susceptibles d'engendrer une politique de détection d'erreurs non efficace ou du moins non optimale. Ainsi, les hypothèses doivent être raisonnables et refléter les vraies contraintes et exigences du système pour pouvoir concevoir une politique de détection d'erreurs efficace et optimale.

Pour les codes détecteurs d'erreurs, selon [Paulitsch *et al.* 2005], la borne supérieure du taux de non détection d'erreurs pour un code de taille k est égale à 2^{-k} si la distribution d'erreur est considérée comme uniforme (il n'y a pas des zones de la trame qui ont une probabilité d'occurrence d'erreurs plus grande que celle des autres).

Les codes CRC sont communément utilisés, leur pouvoir de détection est probant (des évaluations expérimentales où les erreurs à faible multiplicité ont été ciblées) mais leur coût de calcul est un frein à leur utilisation dans les systèmes embarqués critiques. Cependant, il existe des pistes d'optimisation des coûts de calcul des CRC, et il existe certaines sommes de contrôle arithmétiques qui peuvent être une bonne alternative aux CRC. Nous allons creuser ces différentes possibilités et ces pistes d'optimisation des coûts de calcul des CRC ainsi qu'explorer les avantages d'utiliser des codes relativement légers en temps de calcul comme les sommes de contrôle arithmétiques.

II.6 CONCLUSION

Dans ce chapitre, nous avons décrit l'état de l'art spécifique de ces travaux de thèse en présentant tout d'abord les codes détecteurs d'erreurs communément utilisés par les approches d'intégrité. Puis, nous avons décrit les mécanismes d'intégrité déployés dans les réseaux embarqués, ces mécanismes sont plutôt déployés dans les couches basses. Nous avons présenté aussi quelques solutions industrielles proposant des approches d'intégrité basées sur le concept du canal noir et à implémenter dans la couche applicative. Nous avons fini notre analyse des travaux connexes par présenter les approches d'intégrité proposées par les travaux académiques. Le chapitre suivant est dédié à la présentation de l'approche d'intégrité que nous proposons dans ces travaux de thèse

Chapitre III

APPROCHE D'INTÉGRITÉ BOUT EN BOUT BASÉE SUR LES CODES DÉTECTEURS D'ERREURS

Dans ce chapitre, nous allons présenter les principales contributions de ces travaux de thèse. Nous proposons une approche d'intégrité bout en bout des communications basée sur les codes détecteurs d'erreurs. Elle s'appuie sur le concept « canal noir » introduit par l'IEC61508. Selon les caractéristiques du système auquel elle s'applique, notre approche se décline en deux approches différentes : une approche « mono-code » pour les systèmes à faible niveau de redondance que nous appelons systèmes « très contraignants » et une approche « multi-codes » pour les systèmes ayant un niveau de redondance plus important que nous appelons systèmes « contraignants ».

Nous commençons par présenter le principe général de cette approche d'intégrité bout en bout, en expliquant d'abord les motivations et les raisons qui nous ont conduit à nous baser sur l'intégrité bout en bout et le concept canal noir pour notre approche, en définissant ensuite les notions de systèmes « contraignants » et « très contraignants », et en précisant enfin les grandes lignes des approches « mono-code » et « multi-codes » qui en découlent. L'approche « mono-code » fait alors l'objet de la deuxième et la troisième section de ce chapitre. La deuxième section décrit de manière détaillée l'approche. La troisième section présente les expérimentations réalisées et des résultats obtenus, à la fois sur le pouvoir de détection des codes détecteurs, et sur le temps de calcul de ces codes. Les deux sections suivantes sont dédiées à la présentation de l'approche « multi-codes ». La quatrième section focalise sur la présentation de cette approche en détaillant les caractéristiques des systèmes ciblés, le problème du mode commun de défaillance qui a incité à la proposition de cette approche et en caractérisant enfin la notion de « complémentarité » des codes détecteurs sur laquelle se base l'approche.

Puisque les deux premiers critères pouvoir de détection intrinsèque de chaque code et son coût de calcul sont communs aux deux approches, la cinquième section présente les expérimentations réalisées et les résultats obtenus qui ciblent uniquement la complémentarité des codes détecteurs afin d'évaluer et valider l'approche « multi-codes ».

III.1 PRINCIPE GÉNÉRAL DE L'APPROCHE D'INTÉGRITÉ BOUT EN BOUT

III.1.1 Approche d'intégrité bout en bout : présentation

Notre objectif est de définir une approche d'intégrité indépendante des technologies utilisées dans les différentes couches d'un système de communication (par référence aux sept couches du modèle OSI). En nous inspirant des solutions d'intégrité bout en bout basées sur le concept du « canal noir » (voir chapitre II, section II.3.1), nous proposons une approche d'intégrité bout en bout basée sur le concept du « canal noir » introduit par l'IEC61508 ; elle est mise en œuvre au niveau applicatif et fait abstraction des technologies des couches inférieures. Ainsi, notre solution peut être adoptée, peu importe, notamment, que le système soit basé sur Ethernet, CAN, ARINC ou une autre technologie. La Figure III.1 résume ce principe.

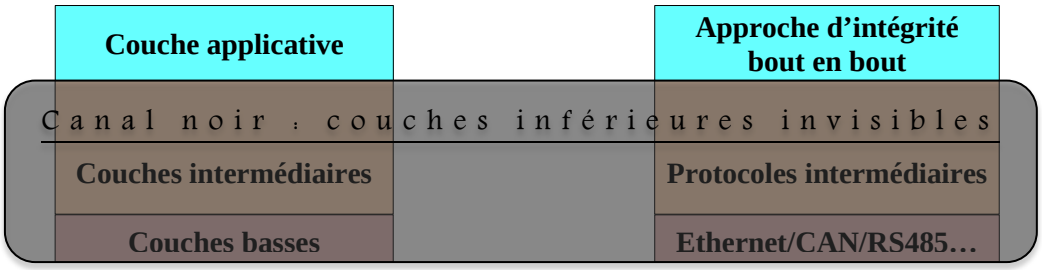


Figure III.1 : Principe de l'approche d'intégrité bout en bout basée sur le concept du« canal noir »

Nous nous reposons sur le concept bout en bout pour deux raisons :

1) La première raison est la certificabilité des approches bout en bout, qui est un atout puisque nous ciblons les systèmes embarqués critiques où la certification est cruciale. En effet, rappelons que pour certifier une telle approche, il n'est pas nécessaire de certifier (vérifier) tous les composants de toutes les couches, mais en contrepartie, il faut prouver que l'approche d'intégrité bout en bout permet de répondre au niveau de sureté requis. Cela est d'autant plus un avantage que les caractéristiques (en sûreté de fonctionnement) des différents composants des couches inférieures ne sont pas toujours fournies par les fabricants.
2) La deuxième raison est qu'en implémentant notre approche au niveau applicatif, elle vient ainsi compléter les mécanismes d'intégrité déployés dans les couches inférieures, ce qui pourrait permettre de couvrir les erreurs résiduelles des couches inférieures.

Enfin, rappelons que pour assurer l'intégrité des communications dans les systèmes embarqués critiques, nous avons étudié les différents mécanismes possibles, à savoir, les codes détecteurs, les codes correcteurs, les fonctions de hachage et la cryptographie. Nous avons retenu les codes détecteurs, d'une part, parce que leur efficacité a été prouvée via de nombreux travaux sur l'intégrité et donc il existe une certaine maturité dans leur utilisation ; cette maturité est d'autant plus importante quand il s'agit de certification qui est un enjeu primordial dans les systèmes que nous ciblons. D'autre part, étant données les contraintes des systèmes ciblés, nous cherchons à éviter l'utilisation de mécanismes lourds en temps de calcul. Les codes détecteurs d'erreurs présentent une bonne efficacité en capacité de détection d'erreurs (sous réserve de bien choisir le code), un temps de calcul beaucoup moins coûteux que les autres mécanismes et enfin une facilité de mise en place et de certification.

III.1.2 Systèmes « contraignants » ou « très contraignants » : définitions

Cette section introduit une classification des systèmes embarqués critiques que nous ciblons, une classification qui nous a conduits à décliner notre approche d'intégrité des communications générique en deux approches différentes. En effet, ces systèmes, bien que partageant un grand ensemble de caractéristiques, peuvent avoir différents niveaux de redondance résultant des différents niveaux de contraintes. Nous considérons que plus le niveau de redondance est important, moins les systèmes sont « contraignants », parce qu'ils sont davantage tolérants à l'occurrence de certains scénarios de défaillance. On distinguera donc par la suite deux classes de systèmes : les systèmes « **contraignants** » et les systèmes « **très contraignants** ». Cette notion de « contraignant » est à comprendre relativement aux mécanismes d'intégrité des communications à mettre en place, c'est-à-dire en fait, que plus il y a déjà certaines formes de redondance dans le système, plus cela offre d'opportunités de gérer l'intégrité.

Les systèmes que nous considérons « **très contraignants** » sont ceux où une information à transmettre ne donne lieu qu'à un seul et unique message : un seul exemplaire du même message est alors transmis. Dans ce cas, l'évènement redouté est la non détection d'erreur sur cet unique exemplaire. Dit autrement, si une erreur affecte cet unique exemplaire, et que cette erreur n'est pas détectée, alors, il n'y aucune autre opportunité de la détecter. Par conséquent, l'objectif d'intégrité doit nécessairement porter sur chaque message, pris individuellement.

Les systèmes que nous considérons « **contraignants** » sont ceux possédant des formes de redondance (par exemple une redondance spatiale ou temporelle) qui permettent qu'une information à transmettre donne lieu à plusieurs messages : plusieurs exemplaires du même message sont transmis. Et dans ce cas, l'évènement redouté est la non détection d'une erreur sur tous les exemplaires (si on considère le pire cas où tous les exemplaires sont affectés par l'erreur). Par conséquent, l'objectif d'intégrité peut porter sur un lot de messages : typiquement les différents exemplaires d'un même message.

On constate donc, que selon la classe du système, l'intégrité est à considérer différemment. D'où la pertinence de proposer deux approches différentes.

III.1.3 Les deux approches proposées : mono-code et multi-codes

Suite aux constats de la section précédente, nous proposons deux approches d'intégrité bout en bout différentes.

- Pour les systèmes **« très contraignants »**, nous optons pour une **approche d'intégrité « mono-code »,** c'est-à-dire utilisant un seul et même code détecteur pour protéger chacun des messages échangés.
- Pour les systèmes **« contraignants »**, qui rendent possible de tolérer la non détection d'erreur sur plusieurs messages, nous optons pour une approche « **multi-codes** », c'est-à-dire que, pour deux (ou plusieurs) messages portant la même information, on utilise deux (ou plusieurs) codes détecteurs différents. L'objectif est qu'au moins un de ces codes sera apte à détecter l'erreur. Pour cela, les différents codes doivent être « complémentaires » en termes de capacité de détection. Dit autrement, de manière simple, deux codes sont dits complémentaires, quand le deuxième code arrive à détecter des erreurs (dans le cas idéal toutes les erreurs) que le premier code n'arrive pas à détecter. La complémentarité sera définie plus précisément dans la section III.4.3.

III.1.4 Stratégie de sélection des codes

Dans les deux approches mono-code et multi-codes, l'objectif commun est d'arriver à assurer l'intégrité des communications tout en répondant aux besoins et exigences du système ciblé, et ceci tout en respectant ses contraintes. Nous rappelons qu'un haut niveau d'intégrité des communications est requis étant données des contraintes temporelles strictes et des ressources restreintes. Pour atteindre cet objectif, il faut choisir le ou les « bons » codes, ce qui nécessite de mettre en place une stratégie de sélection des codes. Cette stratégie de sélection consiste pour chaque approche à un compromis entre un ensemble de critères comme le montre la Figure III.2.

Pour l'approche mono-code, les critères sont : le pouvoir de détection intrinsèque d'un code donné, et son coût de calcul en temps d'exécution. La stratégie de sélection consiste à étudier pour chaque code candidat, si son coût de calcul et son pouvoir de détection intrinsèque répondent aux objectifs fixés vis-à-vis des contraintes et exigences du système.
Pour l'approche multi-codes, aux deux critères précédents s'ajoute un troisième critère qui est la « complémentarité » entre les codes candidats. La stratégie de sélection devient alors plus complexe, puisque l'étude des deux premiers critères pour chaque code candidat (comme pour l'approche mono-code) est maintenant à combiner avec l'étude du troisième critère, qui s'applique à un ensemble de codes candidats.

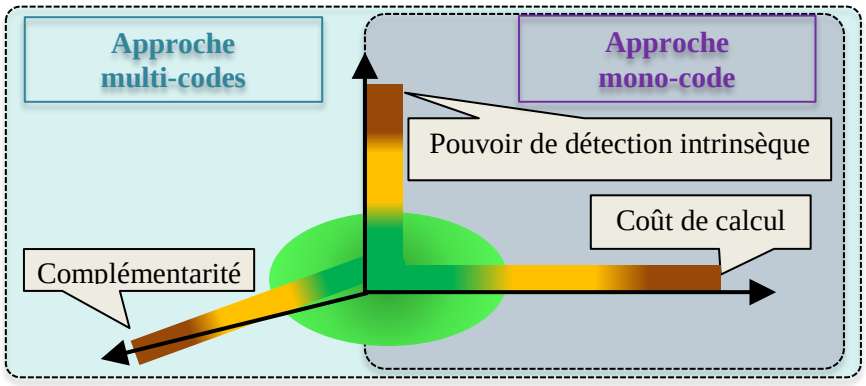


Figure III.2 : Critères de sélection de codes pour les deux approches d'intégrité

Précisons que chacun des trois critères se décompose encore en d'autres critères plus élémentaires qui seront présentés dans les sections suivantes. Par exemple : le critère de coût de calcul fera intervenir comme critères élémentaires les algorithmes d'implémentation des codes, mais également les caractéristiques matérielles des processeurs qui vont exécuter ces codes.

Pour conclure, vu le chevauchement des critères de sélection des deux approches, nous avons choisi de présenter par la suite, d'abord l'approche mono-code avec une analyse des performances des codes détecteurs candidats selon les deux premiers critères (pouvoir de détection intrinsèque et temps de calcul). Puis, pour l'approche multi-codes, nous focalisons surtout sur la présentation de l'analyse de la complémentarité de ces codes et de l'efficacité de l'utilisation des codes complémentaires dans l'augmentation du pouvoir de détection global de l'approche.

III.2 APPROCHE D'INTÉGRITÉ MONO-CODE

L'approche mono-code, comme nous l'avons déjà défini, repose sur le principe de l'utilisation d'un seul et unique code détecteur d'erreurs pour assurer l'intégrité de tous les messages échangés. Ce principe ne présente pas une contribution en soi mais l'analyse des travaux existants basés sur ce principe nous a conduits à identifier un ensemble de limitations que nous visons à améliorer dans notre approche. Cette section présente alors, dans un premier temps, les limitations des travaux existants, puis notre démarche de choix des codes détecteurs d'erreurs à évaluer.

III.2.1 Limitations des approches existantes

Le principe du mono-code est classiquement adopté pour assurer l'intégrité des communications et les codes les plus utilisés sont les codes CRC. L'étude des solutions existantes montre, qu'aujourd'hui encore, ces solutions présentent des limitations. C'est à partir de ce constat que nos contributions visent à apporter des améliorations dans l'utilisation de ce principe mono-code. Les principales limitations sont :

1) Le besoin de dimensionner au plus juste : pour augmenter le pouvoir de détection, les solutions existantes ont souvent tendance à utiliser des codes détecteurs d'erreurs surdimensionnés en augmentant la taille des codes adoptés. Or, cette solution est très coûteuse ce qui n'est pas pertinent dans le contexte des systèmes embarqués critiques.
2) Le besoin de reconsidérer les codes dans de nouvelles hypothèses d'erreur : les travaux existants sur les codes visent majoritairement des erreurs avec une faible multiplicité alors que pour nous la multiplicité de l'erreur n'est pas bornée, cela veut dire que tous les bits du message peuvent être erronés.
3) Besoin d'une caractérisation des temps de calcul des codes : pratiquement pas de travaux caractérisant le temps de calcul de manière précise.

III.2.2 Démarche de choix des codes détecteurs d'erreurs à évaluer

L'approche mono-code demande au préalable d'identifier (de « présélectionner ») parmi tous les codes détecteurs existants ceux qui seraient a priori des candidats potentiels pour permettre des améliorations (par référence aux limitations des approches existantes). Nous présentons donc dans cette section notre démarche pour déterminer ces candidats, avec en conclusion, les codes effectivement retenus.

Notre présélection est fondée sur une analyse théorique (sans lancer des expérimentations) des codes détecteurs mentionnés dans des travaux connexes (présentés dans le Chapitre II, section II.4) et partageant avec nous les mêmes caractéristiques des systèmes ciblés. Et pour chaque code, cette analyse porte sur les deux critères du pouvoir de détection et du temps de calcul.

Les codes CRC font partie des codes détecteurs les plus performants en termes de pouvoir de détection. Mais, les codes CRC font aussi partie des codes détecteurs les plus coûteux en termes de temps de calcul. De ce fait, ces codes pourraient ne pas être adéquats pour certains systèmes embarqués critiques. Néanmoins, nous avons exploré des travaux proposant des pistes pour alléger le coût de calcul des codes CRC en adoptant des implémentations basées sur des tables de correspondance. Pour cette raison, nous n'avons pas écarté les codes CRC de notre liste des codes « bons candidats ».

N'ayant pas trouvé des comparaisons chiffrées qui prouvent l'apport de l'utilisation des tables de correspondance, nous avons décidé d'explorer cette piste pour voir si elle permettait réellement d'alléger le temps de calcul des codes CRC.

Comme alternative aux CRC, on trouve les sommes de contrôle arithmétiques telles qu'Adler et Fletcher. Ces derniers ont de « bonnes » capacités de détection avec des coûts acceptables en termes de temps de calcul. Les codes Adler et Fletcher pourraient être alors de « bons candidats » pour notre approche d'intégrité. Comme pour les codes CRC, nous avons aussi cherché des pistes d'optimisation pour optimiser les temps de calcul d'Adler et Fletcher.

Les autres codes identifiés dans les solutions existantes n'ont pas été présélectionnés, car jugé trop mauvais sur un des deux critères.

Au final, nous avons retenu les codes Adler, Fletcher et CRC comme candidats à l'évaluation. Il reste à voir s'ils sont réellement conformes aux contraintes et exigences de nos systèmes cibles. Précisons que, pour l'évaluation du temps d'exécution, nous avons retenu de multiples implémentations pour chaque type de code, puisque l'implémentation a un impact sur le temps de calcul, ce qui n'est pas le cas pour le pouvoir de détection, pour lequel nous n'avons retenu qu'une seule implémentation pour chaque code.

III.3 ÉVALUATIONS POUR L'APPROCHE MONO-CODE

III.3.1 Contexte général

Cette section décrit en détail nos expérimentations et résultats concernant les évaluations des performances, tant sur le temps de calcul que sur le pouvoir de détection, des codes présélectionnés (Adler, Fletcher et CRC) pour l'approche d'intégrité mono-code. Ces deux critères étant différents, le plan de notre description n'est pas exactement le même pour les deux types d'évaluation.

D'une part, nous avons utilisé deux environnements logiciels d'évaluation : des évaluations préliminaires sur le pouvoir de détection ont été faites avec l'outil Matlab-Simulink, puis, face aux limitations de cet environnement, des évaluations beaucoup plus poussées ont été faites dans un environnement développé en langage C, qui sera ensuite également l'environnement principal utilisé pour évaluer le temps de calcul.

D'autre part, l'environnement matériel (ex : le processeur et les mémoires) n'a pas d'impact sur le critère de pouvoir de détection, alors qu'il impacte le critère de temps de calcul. De ce fait, pour ce dernier critère, nous avons mené des évaluations dans deux environnements matériels différents. Nous avons d'abord travaillé dans un environnement standard (un ordinateur de bureau), et nous avons complété nos travaux par des expérimentations dans un environnement embarqué : une carte d'évaluation pour un processeur STM32.

Pour le pouvoir de détection, la métrique sera le taux de non détection de messages erronés (appelé ***Pud***). Pour l'évaluation du temps de calcul, c'est plus complexe, car se pose la question de savoir quels sont les moyens disponibles pour faire des mesures de temps (nombre de cycles, fréquence d'horloge du processeur ou de la carte d'évaluation, etc.), et donc, quelle est la réelle nature du temps mesuré.

Enfin, comme dans toute évaluation, se posent les questions sur les scénarios de test : quelles entrées, quel nombre et quelle durée des tests. Nous avons d'abord utilisé des générations exhaustives (des entrées), pour lesquelles il est facile de calculer le nombre de cas de test avec des contraintes sur la taille des trames et des codes à tester. Pour cela, nous avons également beaucoup plus largement utilisé des générations aléatoires, pour lesquelles nous avons même effectué, lorsque cela était nécessaire, des calculs d'intervalles de confiance sur les résultats.

Pour chaque type d'évaluation, nous présentons les environnements d'évaluation, les métriques utilisées, les scénarios d'évaluation mis en place (avec leurs hypothèses), et une sélection de résultats. Nous présentons d'abord les évaluations sur le pouvoir de détection (section III.3.2), puis les évaluations sur le temps d'exécution (section III.3.3). Dans le détail, pour le pouvoir de détection, nous introduisons les deux environnements d'évaluations développés sous Matlab-Simulink et en langage C, puis les trois types de scénarios de test mis en place), puis les résultats préliminaires obtenus dans le premier environnement, suivis des résultats étendus dans le deuxième environnement, dans lequel sera également présentée l'utilisation des intervalles de confiances. Pour le temps d'exécution, nous commençons par une présentation beaucoup plus poussée de l'environnement de simulation en C, essentiellement en introduisant les outils retenus *PyCRC* et *Universal CRC*. Puis, les moyens de mesurer le temps sont discutés, avant de présenter les résultats obtenus dans un environnement matériel de type « ordinateur de bureau », puis ceux obtenus sur une plateforme embarquée STM32.

III.3.2 Évaluation du pouvoir de détection des codes

Pour évaluer le pouvoir de détection des codes présélectionnés, nous avons développé deux environnements d'évaluation logiciels différents. Un environnement basé sur l'outil Matlab-Simulink dans un premier temps de par l'expérience acquise avec cet outil. Puis, face à certaines limites de cet outil, nous sommes passés dans un deuxième temps à un environnement d'évaluation en C. Dans cette section, nous commençons par décrire ces environnements d'évaluation, les métriques et les scénarios considérés. Nous exposons par la suite les résultats : d'abord des résultats préliminaires obtenus avec Matlab-Simulink, puis, les résultats étendus obtenus grâce à l'environnement développé en C.

Rappelons que, pour toutes les évaluations sur le pouvoir de détections, le modèle d'erreurs considéré est celui d'**erreurs multiples** (cf. section III.2.1 - limites des approches existantes).

Par ailleurs, la métrique utilisée pour mesurer les pouvoirs de détection des différents codes, sera le **taux de non détection** (noté **Pud**) qui est le rapport entre le nombre des messages erronés non détectés et le nombre total des messages erronés testés. Il est important de préciser, que pour des raisons de temps de simulation, tous les messages générés dans toutes nos expérimentations sont erronés, ce qui fait que le nombre total de messages erronés est égal au nombre total des messages testés.

III.3.2.1 Environnements d'évaluation

Nous avons donc utilisé deux environnements : un premier basé sur l'outil Matlab-Simulink, et par la suite, un second environnement écrit en langage C, que nous avons développé à partir de plusieurs outils libres (tels que *PyCRC* et *Universal CRC*). Pour l'environnement basé sur Matlab-Simulink, nous en présentons ici les détails, puisque cet environnement ne sera utilisé que pour les évaluations sur le pouvoir de détection. Par contre, pour l'environnement en C, ses détails seront donnés plus loin, au niveau des évaluations cette fois du temps de calcul (voir section III.3.3.2), puisque c'est pour ce second critère que ces détails sont les plus importants. De cet environnement, nous ne présentons donc ici que le strict nécessaire à la compréhension des évaluations faites pour le pouvoir de détection.

1) Environnement logiciel basé sur Matlab-Simulink

Nous avons mené nos évaluations dans un premier temps avec l'outil **Matlab-Simulink**, qui est un outil de calcul scientifique de la société MathWorks. Ce choix a été motivé par le fait que des travaux antérieurs nous avaient permis d'acquérir une bonne expérience dans l'utilisation et les capacités de cet outil pour la simulation de communications et plus particulièrement pour l'évaluation du pouvoir de détection des codes CRC. En effet, la partie Matlab offre de nombreuses fonctionnalités qui nous permettent de modéliser assez simplement la génération de trames de communication directement à un niveau binaire, la simulation d'erreurs et l'évaluation des pouvoirs de détection des codes détecteurs. Quant à Simulink, c'est l'extension graphique de Matlab qui permet de modéliser un système en schémas bloc et de mieux visualiser les résultats, et ce, en s'appuyant sur un ensemble de bibliothèques et de blocs applicatifs appelés « *blocksets* ». Pour nos modèles de simulation, nous avons surtout fait appel au « *Communication blockset* » qui permet de modéliser la couche physique d'un système et qui contient déjà un bloc générique pour simuler le codage CRC, en permettant de configurer les polynômes générateurs. Par contre, Simulink ne propose aucun bloc prédéfini pour simuler des codes Adler et Fletcher. Cependant, lors de travaux antérieurs [Naceur 2010], une première ébauche de tels blocs avait été initiée. Nous

avons donc nous-mêmes développé les blocs nécessaires, en utilisant les « *S-functions* ». La Figure III.3 donne une vue d'ensemble d'un modèle d'évaluation Matlab-Simulink.

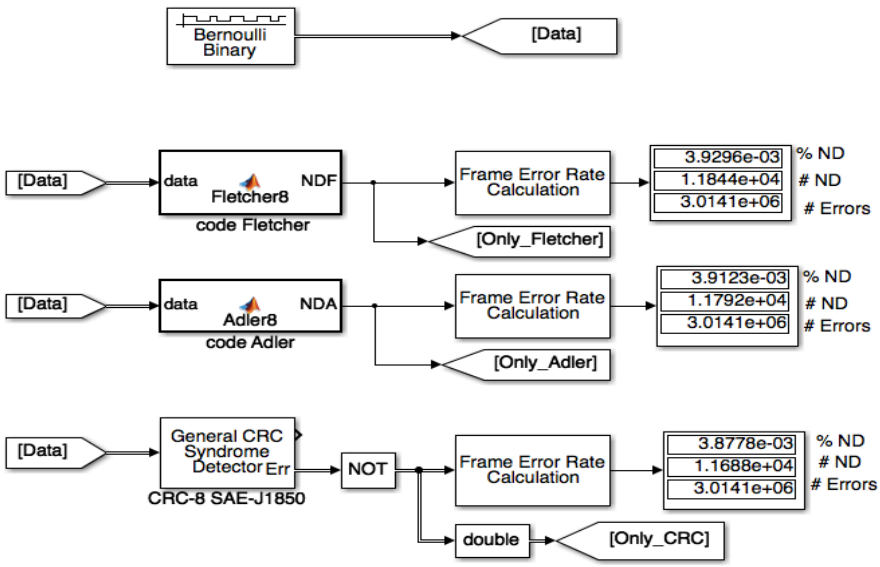


Figure III.3 : Exemple d'un modèle d'évaluation Matlab-Simulink

Grâce à Matlab-Simulink, nous avons eu des résultats préliminaires probants (décrits dans la section III.3.2.3 ci-après). Cependant, nous avons également été confrontés aux limitations de cet outil, notamment sur les temps de simulations beaucoup trop longs pour certains de nos scénarios, ce qui limite le périmètre de nos tests. Pour tenter d'accélérer les simulations, nous avons exploré deux pistes proposées dans cet environnement : le mode « *rapid accelerator* » de Simulink, et le mécanisme de « parallélisation » des traitements. Mais les résultats des essais du premier mode, et l'absence de retours de la part de MathWorks sur le second mécanisme, nous ont rapidement amené à conclure que ces pistes n'étaient pas pertinentes, et risquaient d'amener à des développements lourds et très coûteux en licences.

Par ailleurs, il faut également savoir que cet environnement Matlab-Simulink, présenté principalement ici, pour l'évaluation du pouvoir de détection dans le cadre de l'approche d'intégrité mono-code, a également été utilisé initialement pour ce même type d'évaluation dans l'approche multi-codes. Mais, dans l'approche multi-codes, c'est l'environnement développé en C qui a très rapidement été privilégié.

2) Environnement logiciel en langage C

Nous avons cherché à développer un tel environnement pour deux raisons :

- Pour s'abstraire des limitations précédentes et pouvoir lancer des simulations avec davantage de scénarios plus conséquents (par exemple, avec une taille de messages plus grande).
- Pour les évaluations sur les temps d'exécution, nous avions besoin d'un environnement capable de nous permettre de maîtriser, directement les implémentations des différents codes, puisque ces implémentations influent a priori directement sur les temps d'exécutions (or, Matlab ne permet pas cette maîtrise).

Nous avons donc décidé de développer un **environnement en langage C**, un langage utilisé communément pour le développement et l'évaluation des systèmes embarqués critiques. Pour cela, nous avons d'abord cherché s'il existait des outils proposant une mise en œuvre logicielle en langage C pour les différents codes retenus. Pour les codes CRCs, nous avons trouvé plusieurs outils et nous en avons retenus deux : *PyCRC*[1] [Pircher 2013] et *Universal CRC*[2] [McGougan 2011]. Par contre, pour Adler et Fletcher, nous n'avons pas trouvé d'outils, et nous avons alors nous-même codé notre outil de test. À partir de ces trois outils, nous avons développé notre propre environnement d'évaluation, qui a effectivement permis de répondre à notre besoin : disposer d'un environnement « rapide » pour des évaluations significatives du pouvoir de détection. Par contre, cet environnement n'a pas d'interface graphique.

Nous ne développons pas davantage ici cet environnement, parce que les détails précis seront exposés plus loin, dans la section sur l'évaluation du temps de calcul (voir section III.3.3.2). *PyCRC* et *Universal CRC* étaient les deux outils les plus aboutis, donc les plus exploitables, et que de plus, ils proposent plusieurs types implémentations dont certaines très optimisées.

III.3.2.2 Scénarios mis en place

Pour nos évaluations sur le pouvoir de détection, nous présentons les principales caractéristiques des scénarios conjointement dans une même sous-section (comme pour les environnements), dans le but d'avoir une première vision d'ensemble sur nos différents scénarios. Cette présentation conjointe permet également de mieux comprendre ce qui nous a conduits à ces scénarios.

1) Scénario n°1 : simulations Matlab avec génération exhaustive

Le premier scénario, mis en place pour l'environnement Matlab-Simulink, est fondé sur une stratégie exhaustive de génération des erreurs (avec toutes les erreurs possibles) sur une trame de 24 bits au total, composée de 16 bits de charge utile et de 8 bits de contrôle : le nombre de trames testées est donc égal à $2^{24} = 16777216$ trames. Le but de ce premier scénario est de comparer le nombre d'erreurs non détectées observé dans nos modèles Matlab-Simulink avec les valeurs théoriques prévues.

Cette expérimentation a duré presque 45 heures. De ce fait, pour la suite, il n'était bien sûr pas possible de garder ce principe de simulation exhaustive vu que le temps de simulation est proportionnel au nombre de cas testés : ajouter un simple octet à la trame conduit à multiplier par 256 le temps de simulation soit au final 480 jours de simulation !

2) Scénario n° 2 : simulations Matlab avec générations aléatoires

Nous nous sommes alors orientés vers une stratégie de génération aléatoire des erreurs, en gardant d'abord les mêmes conditions que pour le cas exhaustif (16 bits de charge utile et 8 bits de contrôle) afin de pouvoir comparer les résultats des deux simulations. Par la suite, nous avons progressivement incrémenté la taille de la charge utile, en rajoutant un octet à la charge utile (jusqu'à atteindre 112 bits), mais avec toujours seulement 8 bits de contrôle. Le but est d'étudier l'impact de la taille de la charge utile sur le pouvoir de détection, et également de comparer les pouvoirs de détection de trois types de codes. Mais ces simulations ne peuvent pas être poussées assez loin pour être suffisamment représentatives d'un système

[1] https://pycrc.org/

[2] http://www.mcgougan.se/universal_crc/

embarqué critique réel, toujours pour des problèmes de temps de simulations. Notamment, un contrôle sur 8 bits seulement est insuffisant.

3) Scénario N°3 : simulations C avec générations aléatoires

Grâce à notre développement de l'environnement C (initialement prévu pour l'évaluation des temps d'exécution), nous avons pu très rapidement utiliser cet environnement C pour l'évaluation du pouvoir de détection. Nous avons pu faire des simulations sur des messages **avec une taille de charge utile allant jusqu'à 32 octets (256 bits), et, désormais une taille de champ de contrôle (du code détecteur) allant jusqu'à 32 bits**, le tout avec des durées de simulation nettement inférieures. Globalement, les temps de simulation ont été divisés par 3600. Dit autrement, une simulation d'une heure sous C représenterait près de 150 jours (3600 h) de simulation sous Simulink !

Nous exposons maintenant les résultats préliminaires sur Matlab-Simulink, puis, dans un second temps, une sélection des résultats représentatifs obtenus dans l'environnement C.

III.3.2.3 Résultats préliminaires sur Matlab-Simulink

Dans le premier scénario, une génération exhaustive des erreurs sur une trame de 24 bits (dont 8 bits de contrôle) est utilisée, et donc 16777216 trames sont testées. Le Tableau III.1 présente la comparaison des résultats mesurés pour le taux de non détection (Pud) avec le maximum théorique du taux de non détection qui, pour une distribution uniforme des erreurs, vaut 2^{-n} pour un code détecteur de taille n [Paulitsch *et al.* 2005] ; soit $3{,}91*10^{-3}$ pour $n=8$.

L'analyse des résultats obtenus nous montre que les résultats pour cette simulation exhaustive sont conformes aux résultats théoriques attendus. Cela confirme deux choses :

- La validité de nos modèles par rapport l'hypothèse utilisée (Pud = 2^{-n}).
- Mais aussi que l'hypothèse théorique prise est à garder.

	Fletcher8	**Adler8**	**CRC8**
Taux de non détection : maximum théorique	$3{,}91*10^{-3}$	$3{,}91*10^{-3}$	$3{,}91*10^{-3}$
Taux de non détection mesuré	$3{,}91*10^{-3}$	$3{,}91*10^{-3}$	$3{,}91*10^{-3}$
Nombre d'erreurs non détectées	65536	65536	65536

Tableau III.1 : Pud des 3 codes, simulations exhaustives avec 8 bits de contrôle et 16 bits de charge utile

Dans le deuxième scénario, toujours avec 8 bits de contrôle (maximum théorique du Pud : $3{,}91*10^{-3}$), mais cette fois avec une génération aléatoire d'erreurs, nous avons testé 13 configurations de charges utiles, de 16 bits à 112 bits (total de 24 à 120 bits). Pour chaque cas, on a généré 16777216 trames erronées, pour être dans un cas comparable au précédent.

Le Tableau III.2 expose les résultats obtenus. Nous constatons que les écarts entre la valeur théorique et les valeurs mesurées n'excèdent jamais 1%. Mais, nous constatons aussi que les taux de non détection de ces trois codes fluctuent. Cependant, ces fluctuations ne sont pas proportionnelles à l'augmentation de la taille des données. En fait, il ne se dégage pas, à ce stade, d'explications sur ces fluctuations, hormis de soupçonner une origine dans le principe de génération aléatoire. Pour confirmer cette hypothèse, nous avons effectué des simulations dans lesquelles nous avons testé 8 graines différentes de la génération aléatoire des erreurs. Les résultats de ces simulations (donnés dans le Tableau III.3) montrent que le choix de la graine de la génération aléatoire a bien une influence sur la valeur du Pud. On peut déduire de

ces résultats que les fluctuations observées dans les simulations du Tableau III.2 ne sont pas à relier à la longueur du message mais bien à la génération aléatoire. De plus, ces fluctuations sont inférieures à 1%. Au final, nous considérons qu'elles ne sont pas significatives.

Taille du message	Fletcher8	Adler8	CRC8
24 bits	$3{,}91*10^{-3}$	$3{,}91*10^{-3}$	$3{,}91*10^{-3}$
32 bits	$3{,}90*10^{-3}$	$3{,}90*10^{-3}$	$3{,}91*10^{-3}$
40 bits	$3{,}89*10^{-3}$	$3{,}91*10^{-3}$	$3{,}88*10^{-3}$
48 bits	$3{,}88*10^{-3}$	$3{,}91*10^{-3}$	$3{,}91*10^{-3}$
56 bits	$3{,}90*10^{-3}$	$3{,}91*10^{-3}$	$3{,}91*10^{-3}$
64 bits	$3{,}88*10^{-3}$	$3{,}91*10^{-3}$	$3{,}91*10^{-3}$
72 bits	$3{,}89*10^{-3}$	$3{,}89*10^{-3}$	$3{,}93*10^{-3}$
80 bits	$3{,}92*10^{-3}$	$3{,}90*10^{-3}$	$3{,}91*10^{-3}$
88 bits	$3{,}92*10^{-3}$	$3{,}89*10^{-3}$	$3{,}89*10^{-3}$
96 bits	$3{,}90*10^{-3}$	$3{,}91*10^{-3}$	$3{,}89*10^{-3}$
104 bits	$3{,}90*10^{-3}$	$3{,}91*10^{-3}$	$3{,}92*10^{-3}$
112 bits	$3{,}89*10^{-3}$	$3{,}89*10^{-3}$	$3{,}89*10^{-3}$
120 bits	$3{,}88*10^{-3}$	$3{,}90*10^{-3}$	$3{,}87*10^{-3}$

Tableau III.2 : Pud des 3 codes, simulations aléatoires (8 bits de contrôle et de 1 à 112 bits de charge utile)

Taille du message	Fletcher8	Adler8	CRC8
Graine1	$3{,}91*10^{-3}$	$3{,}89*10^{-3}$	$3{,}90*10^{-3}$
Graine2	$3{,}91*10^{-3}$	$3{,}92*10^{-3}$	$3{,}93*10^{-3}$
Graine3	$3{,}91*10^{-3}$	$3{,}92*10^{-3}$	$3{,}93*10^{-3}$
Graine4	$3{,}92*10^{-3}$	$3{,}91*10^{-3}$	$3{,}90*10^{-3}$
Graine5	$3{,}87*10^{-3}$	$3{,}91*10^{-3}$	$3{,}90*10^{-3}$
Graine6	$3{,}89*10^{-3}$	$3{,}92*10^{-3}$	$3{,}90*10^{-3}$
Graine7	$3{,}90*10^{-3}$	$3{,}90*10^{-3}$	$3{,}91*10^{-3}$
Graine8	$3{,}89*10^{-3}$	$3{,}91*10^{-3}$	$3{,}89*10^{-3}$

Tableau III.3 : Impact de la graine de la génération aléatoire sur le Pud des 3 codes

Par conséquent, les résultats à retenir sont que **pour des erreurs multiples aléatoires** et pour un même nombre de bits de contrôle :

- Adler, Fletcher et CRC se valent en termes de pouvoir de détection,
- la longueur de la charge utile n'a pas d'influence sur le pouvoir de détection,
- les résultats observés concordent avec l'hypothèse théorique utilisée (Pud = 2^{-n}, n la taille du code), ce qui conforte l'utilisation de cette hypothèse et la validité de nos modèles.

En conclusion, ces simulations **Matlab-Simulink** nous ont donné une première idée sur les pouvoirs de détection des codes Adler, Fletcher et CRC. Néanmoins, les simulations menées sont très lentes, et nous limitent beaucoup trop en termes de scénarios testés et de paramètres

d'entrée à savoir la taille des données, puisque nous étions obligés de ne considérer que des données de petite taille (au maximum 112 bits de données et 8 bits de contrôle). Comme déjà dit, toutes ces limitations nous ont partiellement incités à développer un environnement d'évaluation de pouvoir de détection en C.

III.3.2.4 Résultats étendus dans l'environnement d'évaluation en C

Nous présentons maintenant une sélection des résultats étendus de nos expérimentations sur le pouvoir de détection des codes présélectionnées dans l'environnement d'évaluation en C. Il s'agit de résultats « étendus » car nous avons pu mener des expérimentations plus conséquentes (que dans le premier environnement), plus proches du cas des systèmes embarqués critiques ciblés. Nous ne présentons qu'une « sélection » de nos résultats, ceux qui sont en adéquation avec les caractéristiques des systèmes ciblés. Les scénarios que nous exposons ciblent toujours les erreurs multiples aléatoires. Ils se basent sur le cadre suivant :

- trois tailles des codes : 8, 16 et 32 bits de contrôle,
- chacune de ces tailles de code a été appliquée à plusieurs tailles de la charge utile : 8, 16 et 32 octets de données utiles,
- une génération aléatoire des erreurs sur les données en entrée, avec un nombre de cas de test pris égal à $\boldsymbol{2^n * 100}$ par scénario, avec *n* la taille des codes testés.

Ce dernier choix ($\boldsymbol{2^n*100}$ tests par scénario) découle d'une analyse a priori des intervalles de confiances. En effet, le passage à cet environnement C permet de gagner largement en temps de simulation, cependant ce gain n'est quand même pas suffisant pour envisager n'importe quelle longueur de simulation. D'où le choix d'une génération aléatoire des erreurs, qui permet, comme dans toute simulation de ce type (appelée aussi simulations de Monte Carlo), de ne pas avoir à explorer la totalité des cas de test possibles (parce que ce ne serait pas possible à cause de la lenteur des simulations ou du nombre de cas irréaliste à explorer). Vu l'espace à explorer dans notre cas, il faut se poser la question sur le nombre de tests à réaliser. Pour répondre à cette question, nous avons utilisé le principe des intervalles de confiance.

1) Intervalles de confiance et choix du nombre des cas de test

Les intervalles de confiance permettent d'estimer la crédibilité des résultats à obtenir ou obtenus parce que le calcul d'intervalles de confiance peut être utilisé à deux fins différentes :

- Il peut être fait « a priori » (avant le lancement de simulation) pour estimer, ou dimensionner, le nombre des cas de test qu'il faudra explorer pour estimer que les résultats observés seront suffisamment significatifs. Cela revient à chiffrer la condition d'arrêt des simulations (cette estimation se fera en fonction du nombre *n* de bits de contrôle pour notre cas).
- Il peut aussi être fait a posteriori (une fois les simulations faites) pour vérifier ou confirmer la pertinence et la représentativité des résultats observés.

Pour faire ces calculs, nous nous sommes appuyés sur la formule suivante, couramment utilisée, qui définit l'intervalle de confiance :

$$= \left[Pud - R * \sqrt{\frac{Pud*(1-Pud)}{CT}} \ , \ Pud + R * \sqrt{\frac{Pud*(1-Pud)}{CT}} \right]$$

Avec :

- *CT* : le nombre de Cas de Test,
- *Pud* : la proportion d'erreurs non détectées sur le nombre de cas de test qui correspond au taux de non détection,
- *R* : un coefficient qui traduit indirectement le risque d'erreur accepté (ou son complément, le niveau de confiance dans les résultats). Le Tableau III.4 donne les valeurs de R utiles à nos calculs

Seuil de l'intervalle de confiance	95 %	99 %
Niveau de risque correspondant	5 %	1 %
Valeur approchée du coefficient R	1,96	2,58

Tableau III.4 : Coefficients du niveau de confiance

Pour nos expérimentations sur le pouvoir de détection (évaluation du taux de non détection des différents codes), nous avons calculé les intervalles de confiance pour chaque scénario. Nous avons d'abord effectué des calculs a priori. Afin de ne pas surcharger la section, nous ne présentons (voir Tableau III.5) que deux exemples de calcul qui sont représentatifs :

- deux intervalles de confiance, à 95% et 99% de confiance,
- tous les deux pour le scénario avec des codes de taille 32 bits,
- à chaque fois, avec trois valeurs du nombre de cas de test (CT), notés « Court », « Long » et « Très long », correspondant respectivement à un nombre prévu d'erreurs non détectées (ND) de 1, 10 et 100.

Nous considérons que le nombre minimum prévu de cas de test est l'inverse du maximum théorique de non détection attendu (2^{-n} pour un code de taille n). En effet, c'est le seuil limite pour espérer pouvoir avoir au moins une erreur non détectée. Avec $n=32$, ce seuil est de 4294967296 tests. Dans la mesure où nous ne cherchons qu'un ordre de grandeur du nombre de tests, nous avons retenu les valeurs correspondant à 1, 10 et 100 de cas non détectés.

À partir de là, il est possible de calculer la limite supérieure (Sup) et la limite inférieure (Inf) de l'intervalle de confiance, et surtout le rapport entre ces deux bornes (rapport Sup/Inf). En effet, c'est ce dernier rapport qui va être déterminant pour choisir la configuration du nombre de cas de tests à réaliser. Dans le principe, on cherche un intervalle de confiance de taille réduite. Si l'intervalle est trop grand, on considère que les résultats ne seront pas assez significatifs. Dans la pratique, quantifier la limite à partir de laquelle l'intervalle sera trop grand, dépend du contexte et des objectifs qu'on vise. Dans notre cas, nous considérons qu'un rapport positif et inférieur à 2 est tout à fait satisfaisant pour nos objectifs, alors qu'un rapport supérieur ou égal à 10 n'est pas pertinent. Enfin, précisons que nous éliminons toutes les configurations avec un rapport est négatif.

Au final, vu les résultats donnés dans le Tableau III.5, nous retenons que pour avoir un intervalle de confiance à 95 % (ou 99%) de confiance, il faut effectuer un nombre de test de $\boldsymbol{2^n*100}$ *(*dans l'exemple $n = 32$*)*. Précisons que les calculs menés sur les scénarios avec $n=16$ et $n=8$ bits de contrôle ont conduit à la même conclusion : dans tous les cas, nous lancerons des simulations qualifiées (selon le Tableau III.5) de « très long » avec $\boldsymbol{2^n*100}$ *cas de test,* ***n*** *étant la taille du code étudié*.

	Court	**Long**	**Très long**
Nombre prévu de cas de test (CT)	4294967296	42949672960	429496729600
Nombre prévu d'erreurs non détectées (ND)	1	10	100
Nominal (proportion Pud = ND/CT)	$2{,}33*10^{-10}$	$2{,}33*10^{-10}$	$2{,}33*10^{-10}$
Intervalle de confiance à 95% de confiance			
Limite Supérieure (Sup)	$6{,}89*10^{-10}$	$3{,}77*10^{-10}$	$2{,}78*10^{-10}$
Limite Inférieure (Inf)	$-2{,}24*10^{-10}$	$8{,}85*10^{-11}$	$1{,}87*10^{-10}$
Écart en ± (%)	196%	62%	20%
Rapport Sup/Inf	**-3,1**	**4,3**	**1,5**
Intervalle de confiance à 99% de confiance			
Limite Supérieure (Sup)	$8{,}34*10^{-10}$	$4{,}23*10^{-10}$	$2{,}93*10^{-10}$
Limite Inférieure (Inf)	$-3{,}68*10^{-10}$	$4{,}29*10^{-11}$	$1{,}73*10^{-10}$
Écart en ± (%)	258%	82%	26%
Rapport Sup/Inf	**-2,3**	**9,9**	**1,7**

Tableau III.5 : Calculs a priori des intervalles de confiance à 95 % et 99 % de confiance pour les simulations sur des codes de 32 bits

2) Résultats des évaluations des codes de 8, 16 et 32 bits

Nous pouvons maintenant présenter les résultats de nos évaluations sur les codes de 8, 16 et 32 bits, pour des longueurs de charges utiles de 8, 16 et 32 octets.

Pour les codes de taille 8 bits, nous avons réalisé des simulations avec 2^8*100 *(25600)* trames de test. Globalement, les résultats (présentés dans la Figure III.4) montrent qu'on peut considérer que :

- Les pouvoirs de non détection convergent tous relativement vers le maximum théorique, égal à $3{,}9*10^{-3}$.
- Dans un contexte des erreurs multiples aléatoires, les pouvoirs de non détection des trois codes se valent.
- La longueur de la charge utile n'influe pas sur le pouvoir de détection, ce qui confirme le constat théorique qui dit que le pouvoir de détection d'un code dans le contexte des erreurs multiples dépend exclusivement de la taille du code (nombre de bits de contrôle). Ce résultat est différent de celui prouvé dans certains travaux qui considèrent une multiplicité faible des erreurs montrant que la distance Hamming (décrite dans le chapitre I, section I.4.1) est proportionnelle à la taille de la charge utile.

Dans le détail, on note quand même des écarts de valeurs, par rapport à la valeur théorique, et aussi entre les codes. On note également que l'ordre des performances (par rapport à la taille de la charge utile) est différent selon le type de code. Cependant, tous ces écarts sont négligés, car soit minimes, soit potentiellement imputables aux fluctuations liées à la génération aléatoire, comme déjà évoqué pour les résultats préliminaires sur Matlab-Simulink.

Globalement, les résultats obtenus sont cohérents avec les résultats préliminaires (pour la même taille des codes 8 bits).

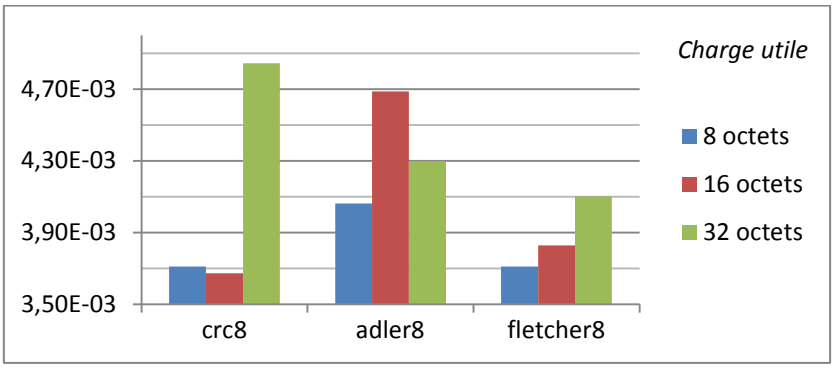


Figure III.4 : Pud des codes de taille 8 bits (charges utiles de 8, 16 et 32 octets)

Pour les codes de taille 16 bits, nous avons réalisé des simulations avec $2^{16}*100$ *(6553600)* trames de test. Globalement, les résultats (présentés dans la Figure III.5) montrent que là aussi, globalement, on peut considérer que : les résultats convergent tous vers le maximum théorique, égal à $1{,}52*10^{-5}$, les pouvoirs de détection des trois codes se valent, et ce, indépendamment de la taille de la donnée. Et là aussi, dans le détail, on note quand même des écarts et des fluctuations de même nature que ceux constatés pour les évaluations des codes 8 bits, et pour les mêmes raisons, ces écarts sont négligés.

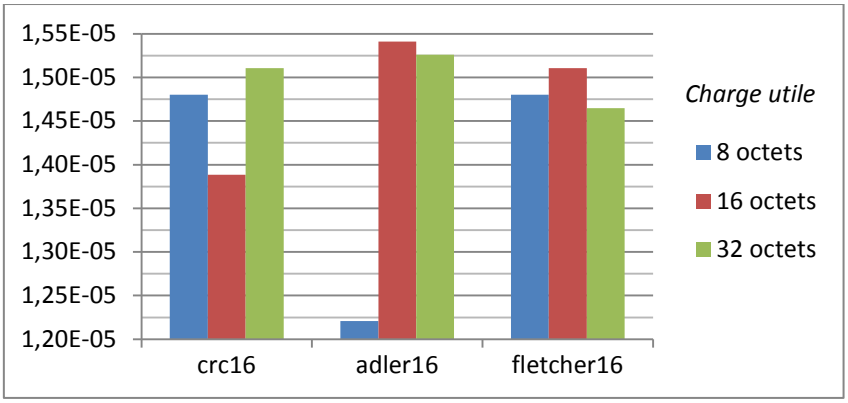


Figure III.5 : Pud des codes de taille 16 bits (charges utiles de 8, 16 et 32 octets)

Enfin, pour les codes de taille 32 bits, nous avons réalisé des simulations avec $2^{32}*100$ *(429496729600)* trames de test. Globalement, les résultats (présentés dans la Figure III.6) conduisent aux mêmes conclusions que précédemment avec notamment une convergence vers le maximum théorique égal à $2{,}32*10^{-10}$.

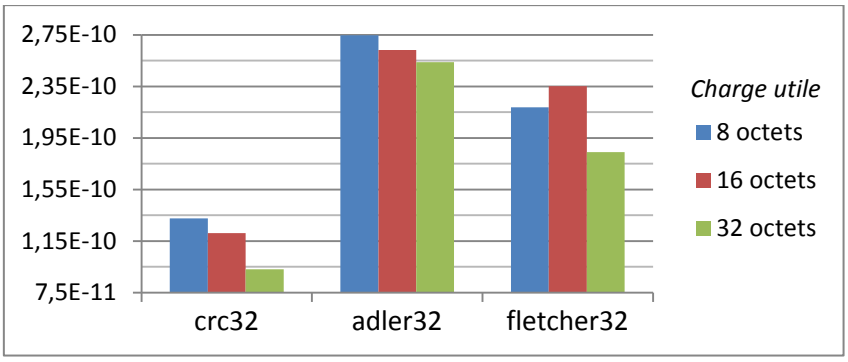


Figure III.6 : Pud des codes de taille 32 bits (charges utiles de 8, 16 et 32 octets)

Une conclusion générale à partir de toutes ces évaluations est que, dans un contexte d'erreurs multiples aléatoires, les pouvoirs de non détection des codes avec la même taille se valent et convergent assez bien vers les maximums théoriques.

3) Intervalles de confiance : calcul a posteriori

Suite à l'introduction des calculs d'intervalles de confiance pour dimensionner la taille de toutes nos expérimentations, nous avons utilisé cet outil pour faire des calculs a posteriori, pour confirmer la crédibilité des résultats obtenus. Nous ne donnons ici que le calcul pour un seul scénario. Il s'agit du scénario le plus lourd, avec 32 bits de contrôle et 32 octets de donnés. Nous avons fait les calculs pour les trois types de codes évalués (Adler, Fletcher et CRC) et pour un niveau de confiance à 99%. Les résultats sont donnés dans le Tableau III.6. Le constat principal est que pour les trois codes, le rapport Sup/Inf (qui nous sert d'indicateur de pertinence) est satisfaisant pour nos objectifs, même dans le cas du CRC32 (avec un écart légèrement supérieur à 2).

	adler32	**fletcher32**	**crc32**
Nombre de cas de test (CT)	429496729600	429496729600	429496729600
Nombre relevé d'erreurs non détectées (ND)	109	79	40
Nominal (proportion P= ND/CT)	$2{,}54*10^{-10}$	$1{,}84*10^{-10}$	$9{,}31*10^{-11}$
Limite Supérieure (Sup)	$3{,}17*10^{-10}$	$2{,}37*10^{-10}$	$1{,}31*10^{-10}$
Limite Inférieure (Inf)	$1{,}91*10^{-10}$	$1{,}31*10^{-10}$	$5{,}51*10^{-11}$
Écart en ± (%)	25%	29%	41%
Rapport Sup/Inf	1,7	1,8	2,4

Tableau III.6 : Calcul a posteriori des intervalles de confiance à 99,9% de confiance pour les simulations sur des codes de taille 32 bits et 32 octets de données

De toutes ces évaluations sur le pouvoir de détection des trois types de code que nous avions présélectionnés, la conclusion principale est que ces trois types, pour l'instant, sont toujours de « bons » candidats pour l'approche mono-code (le choix de la longueur du code adéquate dépend des objectifs d'intégrité de chaque système). Nous allons maintenant passer à la présentation de tout ce qui touche aux évaluations des temps de calculs pour ces trois codes.

III.3.3 Évaluations du coût de calcul des codes présélectionnés

Cette section est dédiée à nos évaluations des coûts, en termes de temps de calcul, des trois types de codes retenus (CRC, Adler et Fletcher). Rappelons que le but de ces évaluations est de déterminer si les temps d'exécution de ces types de codes sont compatibles avec les contraintes des systèmes embarqués critiques que nous ciblons.

III.3.3.1 Problématique de l'évaluation des temps de calcul

Le temps de calcul des codes dépend de nombreux facteurs. Nous pouvons principalement mentionner :

- Le code source obtenu après mise en œuvre d'un algorithme dans un langage de programmation donné : il y a différentes manières de programmer un même algorithme (calcul bit par bit ou octet par octet pour les CRC par exemple), et il peut même y avoir des règles de programmation imposées par le domaine des systèmes visés (règle sur la complexité, sur les branchements, sur les types d'instructions).
- Le compilateur qui génère le code exécutable : tous les compilateurs ne produisent pas nécessairement exactement le même code exécutable, et dans tous les cas, il y a également des options de compilation pour optimiser le code exécutable (par exemple, par rapport à l'occupation mémoire).

Nos évaluations n'ont pas pour objectif d'explorer en détail tous ces facteurs. Certains seront abordés, les autres pourraient être explorés en complément à ces travaux de thèse.

Plus précisément, toutes nos évaluations sont réalisées à l'aide de l'environnement que nous avons développé en langage C. Pour un même type de code, il faudrait pouvoir tester différents algorithmes puisque cela a un impact sur le temps de calcul. Nous commençons donc par une première section de présentation beaucoup plus poussée de notre environnement de simulation en C en décrivant les différentes implémentations que nous avons testées pour chaque type de code. Dans la deuxième section, nous décrivons notre démarche pour définir la fonction de mesure et la stratégie d'exploitation des résultats en nous focalisons sur celles que nous avons retenues. Enfin, puisque le temps d'exécution dépend également de l'environnement matériel (comme par exemple le processeur), nous exposons nos résultats sur deux environnements matériels différents : un environnement standard (un ordinateur de bureau) et un environnement embarqué (une carte d'évaluation pour un processeur STM32).

III.3.3.2 Environnement d'évaluation du temps de calcul

Pour les différentes raisons évoquées précédemment, nous avons décidé de développer un environnement d'évaluation en langage C. Nous avons d'abord cherché s'il existait des outils proposant des mises en œuvre logicielles pour les différents codes retenus. Pour les CRCs, nous avons effectivement trouvé un ensemble d'outils, mais tous ne sont pas assez aboutis ou matures pour être intéressants pour nos besoins (comme par exemple celui proposé par « BARR group » [Barr 2007]). Nous avons finalement retenu les deux outils *PyCRC*[3] et *Universal CRC*[4] . Ces deux outils proposent :

- des implémentations basiques basées sur un calcul bit par bit,
- des implémentations « optimisées » basées sur un calcul octet par octet utilisant des tables de correspondance.

[3] https://pycrc.org/

[4] http://www.mcgougan.se/universal_crc/

PyCRC propose en plus des implémentations « optimisées » basées un calcul octet par octet utilisant des expressions logiques.

L'utilisation des expressions logiques ou des tables de correspondance repose sur la linéarité des codes CRC (voir chapitre II, section II.4.1) pour accélérer le calcul du CRC. En effet, cela consiste à calculer des valeurs (obtenues par exemple par la multiplication de certaines valeurs par le polynôme générateur) et les placer dans une table de correspondance (un calcul hors ligne) et consulter simplement la table pour avoir ces valeurs « toutes prêtes » sans refaire les calculs par la fonction du calcul de CRC. Un exemple ainsi que quelques explications de l'utilisation des tables de correspondance sont décrits dans l'annexe 1.

Ces différentes implémentations nous permettront d'évaluer l'apport de ces optimisations et leur pertinence par rapport aux contraintes de nos systèmes ciblés.

Pour Adler et Fletcher, nous n'avons pas trouvé d'outils, nous avons donc codé nous-même notre outil de test.

C'est à partir de ces trois outils, que nous avons développé notre propre environnement d'évaluation, qui n'est cependant pas graphique (contrairement à l'environnement Matlab-Simulink). Nous décrivons maintenant en détail ces différents outils.

1) L'outil « PyCRC » pour les codes CRC

L'outil « **PyCRC**[5] » est développé en langage Python par Thomas Pircher. Il permet de générer les codes C pour une multitude d'implémentations CRC. C'est l'outil d'évaluation des CRCs le plus complet et le plus documenté que nous ayons trouvé. Il est largement utilisé par divers industriels, organisations et projets. Il est aussi cité dans certains travaux académiques. En ce qui concerne les différentes implémentations proposées, on peut en distinguer six :

- **« bbb »** : algorithme bit de bit de base. Il effectue une itération sur chaque bit d'un message « augmenté ».
- **« bbf »**: c'est une variante de l'algorithme de base bit par bit. Cet algorithme itère aussi sur chaque bit du message, mais augmenter la fin d'une suite de bits à zéro.
- **« tb4 » :** algorithme basé sur une table de correspondance à 16 entrées
- **« tbl »**: algorithme basé sur une table de correspondance à 256 entrées.
- **« bwe »**: algorithme utilisant des expressions binaires (tels que, ET, NON, XOR, décalages binaires, etc.) pour un calcul octet par octet.
- **« bw4 »** : algorithme inspiré de « bwe » avec une utilisation différente des expressions binaires.

2) L'outil « Universal CRC » pour les codes CRC

L'outil « **Universal CRC**[6] » est développé en C par Danjel McGougan (en s'inspirant de l'outil PyCRC) pour une grande variété d'implémentations CRC. UniversalCRC propose sept différentes implémentations :

- « **bit** » : algorithme bit par bit.
- « **tab16** » : algorithme basé sur une table de correspondance à 16 entrées.
- « **tab16i** » : algorithme basé sur deux tables de correspondance indépendantes à 32 entrées chacune.

---

[5] https://pycrc.org/

[6] http://www.mcgougan.se/universal_crc/

- « **tab** » : algorithme basé sur une table de correspondance à 256 entrées.
- « **tabw** » : algorithme basé sur une table de correspondance à 256 entrées comme l'algorithme « Tab », mais qui lit 4 octets à la fois de la mémoire.
- « **tabi** » : algorithme basé sur 4 tables de correspondance indépendantes (1024 entrées). Il est inspiré par l'algorithme CRC32 dans zlib [Deutsch & Gailly 1996].
- « **tabiw** » : algorithme basé sur 4 tables de correspondance indépendantes (1024 entrées) comme l'algorithme « Tabi » mais il lit 4 octets à la fois de la mémoire.

3) L'outil pour les codes Adler et Fletcher

Contrairement au cas des codes CRC, il n'existe presque aucun outil d'évaluation déjà implémenté pour **Adler et Fletcher**. Nous avons alors implémenté notre propre outil de test en C dédié à ces codes. Nous avons d'abord commencé par implémenter les deux algorithmes tels qu'ils sont décrits par leurs inventeurs [Deutsch & Gailly 1996] et [Fletcher 1982].

Dans l'objectif de réduire les temps de calcul, nous avons implémenté une deuxième version « optimisée » de ces codes en nous basant sur l'optimisation de Fletcher proposée par « John Kodis » [Kodis 1992]. Nous avons adopté la même optimisation pour Adler, puisque ce code n'est qu'une amélioration de Fletcher.

Notre outil d'évaluation d'Adler et Fletcher propose donc quatre implémentations :

- **« fletcherM »** : algorithme de base de Fletcher.
- **« fletcher »** : algorithme optimisé de Fletcher.
- **« adlerM »** : algorithme de base d'Adler.
- **« adler »** : algorithme optimisé d'Adler.

En conclusion, l'environnement d'évaluation que nous avons développé intègre ces trois outils. Mais cela ne suffit pas à le rendre totalement opérationnel. Il faut en particulier compléter les moyens pour mesurer le temps, ce qui est l'objet de la section suivante.

III.3.3.3 Fonction de mesure des temps d'exécution, métriques et scénarios ciblés

Évaluer des temps d'exécution est une tâche délicate. Cela dépend déjà des fonctions de mesures disponibles dans l'environnement de simulation et des plateformes sur lesquelles se déroulent les exécutions. Notamment, les fonctions de base ne permettent pas toujours d'avoir la précision souhaitée. De plus, il faut souvent instrumenter le programme de simulation, ce qui veut dire rajouter des instructions, qui viennent donc augmenter le temps d'exécution que l'on veut justement évaluer. Du fait de toutes ces difficultés, nous présentons dans le détail les différentes étapes qui ont conduit à la solution que nous avons utilisée pour mesurer le temps, en précisant que nous cherchons autant que possible à accéder à des mesures de temps réel (c'est-à-dire avec la plus petite granularité de temps possible).

1) Fonctions clock() et times()

Lorsque nous avons commencé nos expérimentations, nous ne dispositions pas encore d'une plateforme qui serait représentative de celle d'un système embarqué. Nous avons donc cherché à faire une première mesure des tests d'exécution des implémentations des codes CRC sur un poste de travail standard (un ordinateur de bureau). Une telle approche pose quelques problèmes pour obtenir des valeurs représentatives d'un cas réel, problèmes qui sont dus au contexte du système d'exploitation qui est un système multitâches à temps partagé. Les processeurs présents dans ce type de machine disposent généralement des mécanismes

d'accélération (caches, prédicteurs de branchement, etc.) qui ne sont pas forcément présents dans des processeurs embarqués et qui peuvent favoriser certains algorithmes par rapport à d'autres.

Dans un premier temps, pour essayer de disposer d'une grande portabilité, nous avons cherché à utiliser la fonction *clock()* qui fait partie de la bibliothèque standard « *time.h* » du C. Les temps étant fournis avec une granularité de 1 ms, il est nécessaire d'exécuter un grand nombre de fois le code pour arriver à avoir une valeur suffisamment précise. En effet, exécuter un grand nombre de fois un programme permet d'atténuer (de lisser) les fluctuations existantes entre deux exécutions sous un système d'exploitation standard. Mais en contrepartie, l'exécution répétée d'un même programme met en jeu des mécanismes d'accélération du processeur (dont notamment l'utilisation de cache), qui a pour effet de réduire les durées de certaines exécutions, et donc, au final de donner des temps d'exécution insuffisamment représentatifs d'un contexte embarqué, qui ne bénéficierait pas de ces mécanismes.
Conscients des problèmes évoqués, nous avons exploré d'autres pistes. Dans le programme de test de performance de l'outil PyCRC, l'auteur utilise la fonction *times()*. Mais le principe de cette fonction se heurte aux mêmes difficultés que la méthode précédente.

2) Fonction et stratégie de mesure

Faute d'avoir une fonction de mesure adéquate, nous avons cherché à comprendre le principe des mesures effectuées dans l'environnement « UniversalCRC ». Pour cela, nous avons pris directement contact avec son auteur qui nous a fourni son programme de mesure de temps. Le principe de mesure élémentaire repose sur l'instruction RDTSC des processeurs x86. Ce principe a le gros avantage de donner une précision égale au cycle horloge du processeur, donc une meilleure précision que les fonctions précédemment évoquées. Mais il a aussi l'inconvénient de ne pas être portable puisqu'il s'appuie sur une instruction spécifique d'une famille donnée de processeur, et donc, il faut adapter le code lorsqu'on change de processeur. Cependant, cet inconvénient est mineur au regard de la précision que l'on peut avoir avec cette solution, que nous avons donc décidé d'utiliser pour une solution basée sur l'utilisation de l'instruction RDTSC.

À titre d'exemple, la Figure III.7 donne le code des fonctions de l'auteur de « Universal CRC que nous avons retenu pour nos mesures actuelles. C'est le code élémentaire d'utilisation de l'instruction RDTSC. Il reste à voir plus précisément, comment l'utiliser pour fournir au final des temps d'exécution, c'est-à-dire définir la stratégie ou la politique pour exploiter les temps mesurés et en exploiter les plus significatifs.

Pour la stratégie de mesure qui consiste en l'exploitation des mesures obtenues (ou la stratégie de mesure), nous avons identifié deux stratégies différentes. La première stratégie est présentée dans un article d'Intel. Il s'agit d'un article [Gopal *et al*. 2012] qui s'intéresse plus précisément à la performance de l'instruction « CRC32 » présente dans les processeurs Core i7. La deuxième stratégie est celle adoptée par l'auteur de « Universal CRC ». À préciser que les deux stratégies sont basées sur l'instruction RDTSC des processeurs x86.

```
// fonctions mesures fournies par Danjel McGougan
//  * Copyright (C) 2011 Danjel McGougan
//  * Contact me at <danjel.mcgougan@gmail.com>

/*
 * Serialize the CPU (wait for all outstanding ops to retire)
 */
static inline void serialize()
{
  asm volatile ("xor %%eax, %%eax\n"
                "cpuid\n" : : :
                "%eax", "%ebx", "%ecx", "%edx");
}

/*
 * Read the time stamp counter of the CPU
 */
static inline unsigned __tsc()
{
  uint32_t tmp;
  asm volatile ("rdtsc\n" : "=a" (tmp) : : "%edx");
  return (unsigned)tmp;
}

/*
 * Read the time stamp counter with serialization before and after
 */
unsigned tsc()
{
  unsigned tmp;
  serialize();
  tmp = tsc();
  serialize();
  return tmp;
}
```

Figure III.7 : Fonction de mesure du temps de calcul

La différence entre les deux stratégies peut se résumer comme suit :

- La stratégie retenue par l'auteur de « Universal CRC » consiste à extraire un temps minimal d'exécution. Plus précisément, le code du CRC est exécuté 32 fois pour être sûr que tout le programme soit d'abord placé dans les caches (si l'environnement matériel dispose de cache), et on retient alors la valeur minimale du temps (parmi les 32 exécutions).
- La stratégie d'Intel est de fournir un temps moyen : le code CRC est exécuté 256 fois et la mesure de temps fournie est la moyenne des 128 exécutions donnant des temps intermédiaires (les 64 valeurs les plus faibles et les 64 valeurs les plus élevées sont éliminées).

Pour résumer, nos mesures de temps de calcul sont basées sur l'instruction RDTSC (pour l'avantage de la meilleure précision), et notre stratégie pour exploiter les mesures est inspirée de la stratégie d'Intel, qui, à notre sens, est la plus susceptible de fournir des valeurs plus réalistes. Toutefois, il reste alors le souci des effets des caches du processeur sur les temps mesurés. Pour limiter ces effets, nous avons préféré utilisé une boucle externe au programme de mesure (la boucle est déportée dans un shell Unix) pour ne pas favoriser la mise en cache des programmes et des données.

Plus précisément, en termes d'unité, les temps moyen d'exécution que nous mesurons sont exprimés en « ticks », ce qui correspond à un nombre de cycles d'horloge processeur

(typiquement 1 tick = 1 cycle d'horloge). Cette unité a l'avantage de faire abstraction de la fréquence du processeur.

Nous avons maintenant décrit pratiquement tous les éléments de notre environnement d'évaluation. Il manque encore des précisions plus spécifiques aux évaluations réellement faites, et qui sont introduites dans les sections III.3.3.4 et III.3.3.5. Une partie des codes des programmes et des shells développés est fournie dans l'annexe 2 avec des explications supplémentaires sur leur utilisation.

III.3.3.4 Évaluations sur un environnement matériel standard

Cette section a pour objectif d'exposer nos évaluations et résultats, mais avant cela, il reste encore à donner des précisions sur l'environnement matériel et sur le compilateur que nous avons utilisés, et les scénarios mis en place.

1) Environnement matériel et compilateur

L'environnement de test a été développé et les évaluations ont été menées sur un MacMini, avec un processeur Core i7 à 2,3 Ghz) fonctionnant sous MacOS X 10.9.4. Le compilateur utilisé est clang/LLVM[7], qui est une alternative à GCC.

2) Algorithmes implémentés et principes des scénarios évalués

Nous avons implémenté 17 types différents d'algorithmes sur la base des types d'algorithmes présentés dans la section III.3.3.2. Voici la liste complète de leur nom, tels qu'ils apparaissent sur les graphiques et dans le texte, à ceci près que la lettre X est ici un caractère générique qui correspond à la longueur du code testé :

- CRC bit par bit : *crcX_bbb, crcX_bbf, crcX_bit,*
- CRC par expressions binaires : *crcX_bwe, crcX_bw4,*
- CRC à base de table : *crcX_tbl, crcX_tb4, crcX_tab16, crcX_tab16i, crcX_tab, crcX_tabw, crcX_tabi, crcX_tabiw,*
- Adler et Fletcher : *adlerMX, adlerX, fletcherMX, fletcherX.*

Les scénarios ciblés sont, pour chaque type d'algorithme implémenté :

- un nombre de bits de contrôle égal à 8, 16 et 32 bits (c'est ce nombre qui instancie le caractère générique X ci-dessus),
- une taille de données égale à 8, 16, et 32 octets.

Ces tailles sont naturellement identiques à celles des évaluations sur le pouvoir de détection, afin de caractériser complètement (pourvoir de détection et temps de calcul) les codes que nous avons présélectionnés pour l'approche mono-code. Rappelons que ces tailles sont représentatives du fait que nous considérons que dans les systèmes embarqués critiques, on ne peut pas se permettre d'utiliser des codes détecteurs d'erreurs de trop grande taille, et que les messages échangés ont souvent une taille de quelques dizaines à quelques centaines d'octets.

De plus, après une première série de simulations, certains résultats nous ont incités à engager une deuxième série de simulation pour évaluer si le fait de favoriser la mise en cache des programmes et des données impacte les temps d'exécution. Nous présentons ces séries d'évaluation : par abus de langage, nous appelons la première série « sans favorisation du cache » et la deuxième « avec favorisation du cache », même si dans la réalité, les caches sont

[7] http://clang.llvm.org

toujours présents. La différence réside dans le fait que, dans la première série, nous favorisons la mise en cache des programmes et des données avant de commencer à prendre des mesures.

3) Scénarios « sans favorisation du cache »

Nous avons commencé par évaluer les temps de calcul des différentes implémentations des codes de taille 8 bits. L'intégralité des résultats est présentée sur la Figure III.8. Par rapport à nos objectifs de comparaison des performances respectives des différents codes et de leurs différentes implémentations, les principales conclusions sont les suivantes :

- Le temps de calcul est proportionnel à la taille des données de la charge utile, excepté pour 2 des 17 implémentations, qui sont *crc8_tabi* et *crc8_tabiw*, pour lesquelles le temps de calcul avec 16 octets de charge utile est inférieur à celui avec 8 octets.
- Les implémentations bit par bit des codes CRCs (*crc8_bbb*, *crc8_bbf*, *crc8_bit*) sont les plus coûteuses en temps de calcul et ce, quelle que soit la taille de la charge utile.
- Les implémentations CRC par expressions binaires (*crc8_bwe*, *crc8_bw4)* ont des performances meilleures que les précédentes, mais restent néanmoins moins performantes que les implémentations ci-après.
- Les implémentations CRC basées sur des tables de correspondance ont des performances très intéressantes (les performances sont généralement meilleures avec des tables plus grandes) et sont globalement parmi les plus performantes. Certaines de ces implémentations ont des performances comparables à celles d'Adler et Fletcher.
- Quelle que soit l'implémentation, Adler et Fletcher ont également des performances très intéressantes, du même ordre, voire même meilleures, que les implémentations des CRC basées sur des tables.
- Enfin, l'implémentation « optimisée » de Fletcher (*fletcher8*) ne réduit que très légèrement le temps de calcul par rapport son implémentation de base (*fletcherM8),* et pour Adler, son implémentation optimisée (*adler8) est même moins performante que son implémentation de base (adlerM8).*

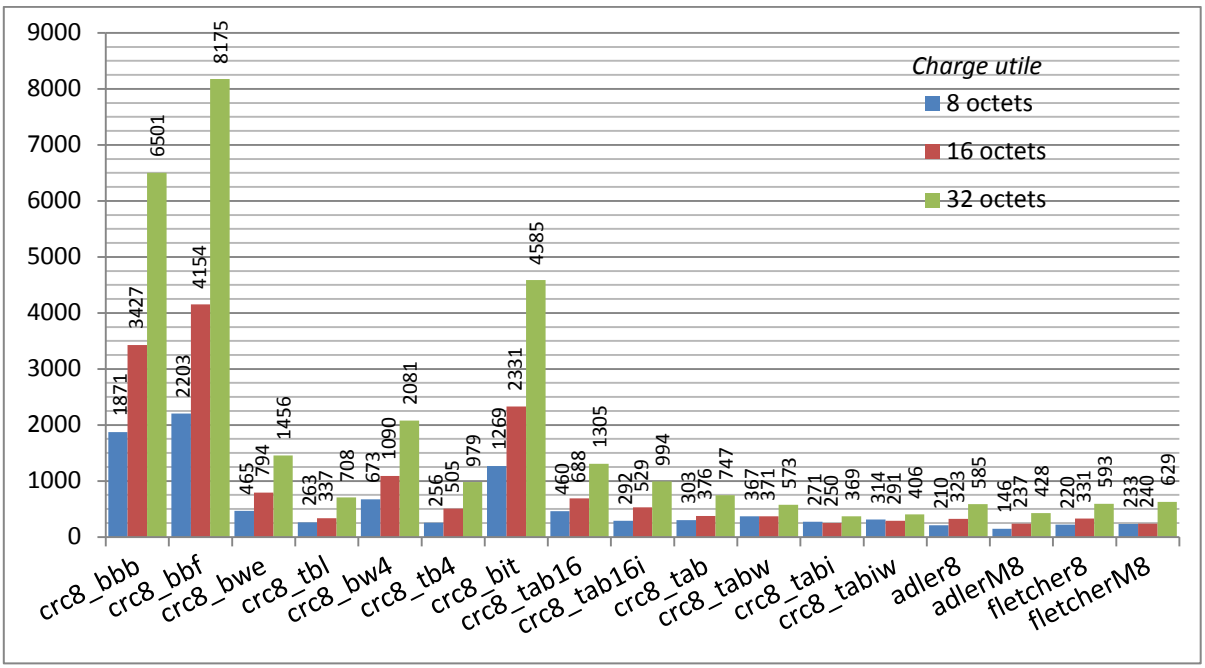


Figure III.8 : Temps de calcul (tick) « sans favorisation du cache » des codes 8 bits

Pour les scénarios avec des codes de taille 16 et 32 bits, l'intégralité des résultats est présentée sur la Figure III.9 et la Figure III.10. Les conclusions principales sont pratiquement du même ordre que précédemment, avec cependant, deux nouveaux constats particuliers que nous préciserons après les conclusions communes suivantes :

- Le temps de calcul est proportionnel à la taille des données de la charge utile, excepté pour toutes les implémentations de type *crcX_tabi* et *crcX_tabiw* (*X* correspond à 16 et 32 bits de contrôle), et, accessoirement, les implémentations spécifiques *adlerM16*, *fletcherM16* et *adler32*.
- Les implémentations bit par bit des codes CRCs (*crcX_bbb, crcX_bbf, crcX_bit*) sont toujours les plus coûteuses en temps de calcul.
- Les implémentations CRC utilisant des expressions binaires (*crcX_bwe, crcX_bw4)* ont toujours des performances entre les précédentes et les suivantes, a une exception près : *crc32_bw4*, qui est meilleure qu'une des implémentations par table.
- Les implémentations CRC basées sur des tables de correspondance ont toujours des performances très intéressantes, dont certaines qui sont parmi les plus performantes, et toujours assez comparables à celles d'Adler et Fletcher qui néanmoins sont globalement les plus performantes (ayant les plus faibles temps d'exécution).
- Pour Adler et Fletcher, sans détailler, leurs implémentations optimisées sont parfois plus performantes, parfois moins performantes que leur implémentations de base : cela fluctue selon à la fois la longueur du code et la longueur de la charge utile.

Le premier constat supplémentaire qui nous interpelle est que pour des codes de 32 bits, certaines implémentations ont de performances meilleures que pour des tailles de codes plus petites. C'est en particulier le cas des codes Adler et Fletcher. Ce constat est très encourageant (pour les systèmes que nous ciblons), mais semble aller à l'encontre de l'idée reçue que plus le code est long, plus son temps de calcul sera long. En réalité, ce n'est pas si surprenant que cela, si on envisage comme piste d'explication le fait qu'aujourd'hui, la plupart des processeurs sont des processeurs 32 bits, on peut imaginer que plus la taille du code est identique à la taille des registre des processeurs, plus le calcul sera performant car demandant moins d'opérations.

Le deuxième constat supplémentaire qui nous interpelle, est le comportement des implémentations de type *crcX_tabi* et *crcX_tabiw* qui ont systématiquement des performances indépendantes de la taille des données, et surtout, dans le cas de 32 bits, des performances comparables à celles des implémentations bit par bit, donc de mauvaises performances, alors qu'il s'agit d'implémentations à base de tables.

Si on analyse ce qui distingue ces implémentations des autres implémentations utilisant des tables, on trouve que ces implémentations sont celles qui utilisent les plus grandes tables de correspondance : 4 tables de correspondance indépendantes avec au total 1024 entrées.

Nous avons donc cherché à savoir si la taille des tables était à l'origine des écarts de comportements observés. En fait, on soupçonne comme origine le fait de favoriser la mise en cache des programmes et des données, phénomène qui avait déjà été soulevé lors de l'analyse des stratégies de mesure du temps par Intel et Universal CRC. Si on fait des mesures en faisant en sorte de ne pas favoriser la mise en cache des données et des programmes, les implémentations basées sur des tables de correspondance de grandes tailles seront pénalisées et auront les pires performances.

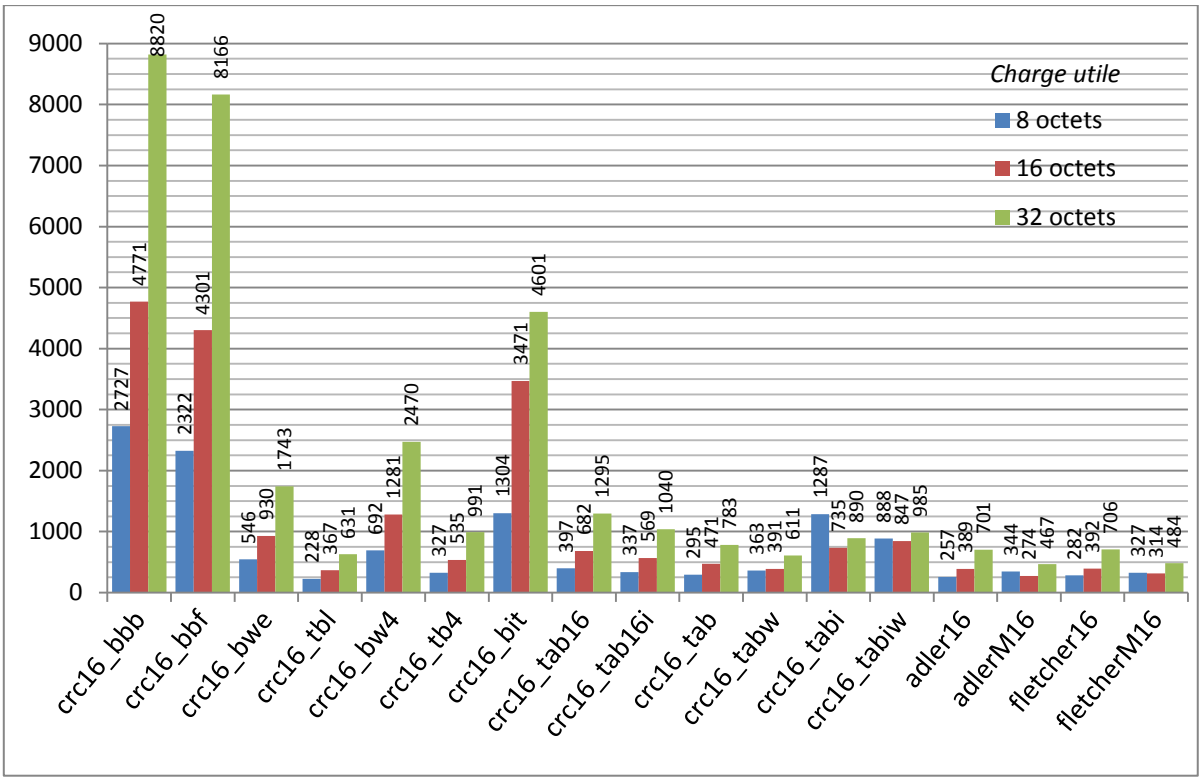


Figure III.9 : Temps de calcul (tick) « sans favorisation du cache » des codes 16 bits

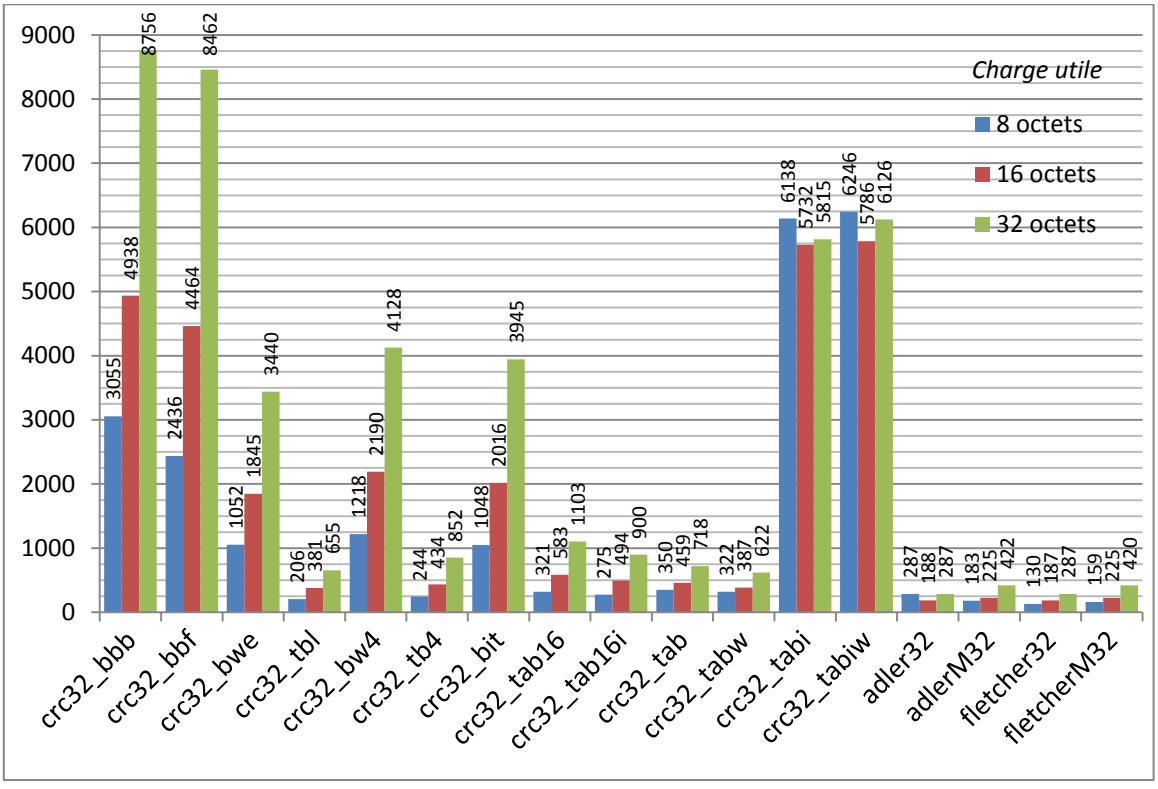


Figure III.10 : Temps de calcul (tick) « sans favorisation du cache » des codes 32 bits

4) Scénarios « avec favorisation du cache »

Pour vérifier notre hypothèse, nous avons alors relancé toutes les simulations précédentes, mais cette fois en favorisant la mise en cache des programmes et des données et ce, via le

lancement d'une boucle avant d'effectuer la mesure du temps afin d'avoir le temps nécessaire de la mise en cache. Les résultats sont présentés sur la Figure III.11, la Figure III.12 et la Figure III.13.

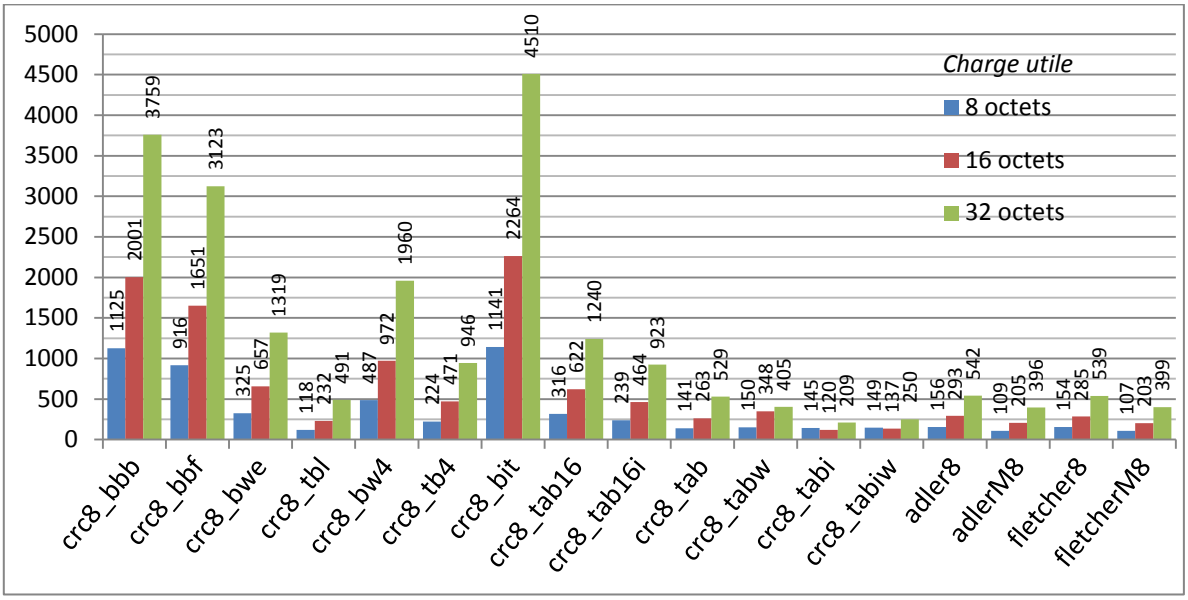


Figure III.11 : Temps de calcul (tick) « avec favorisation du cache » des codes 8 bits

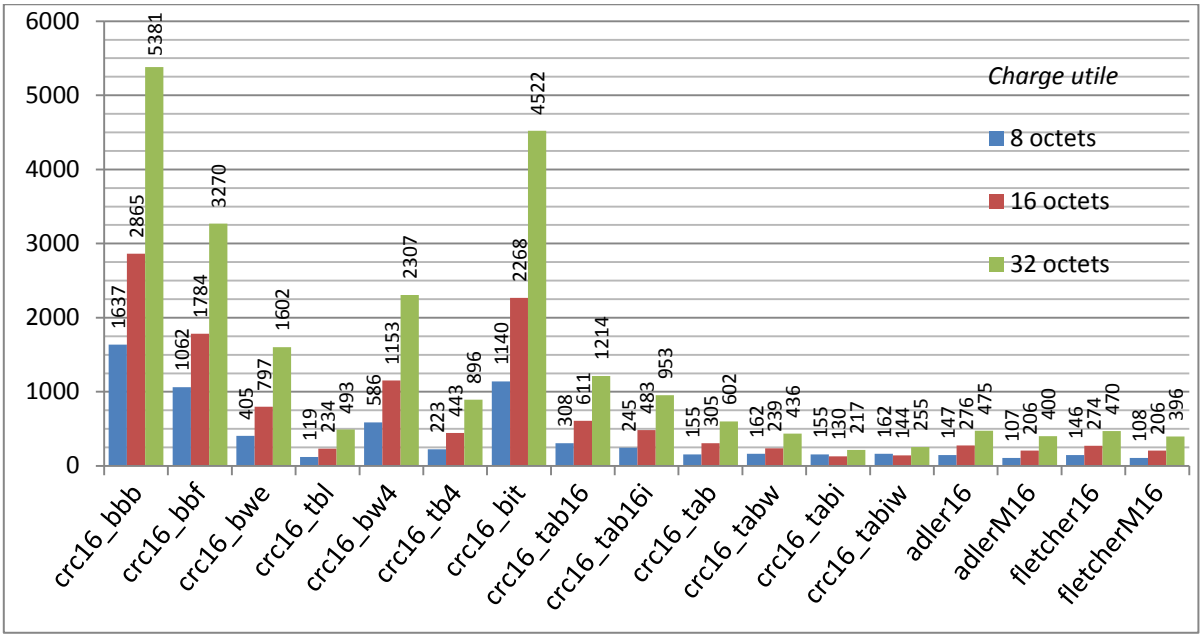


Figure III.12 : Temps de calcul (tick) « avec favorisation du cache » des codes 16 bits

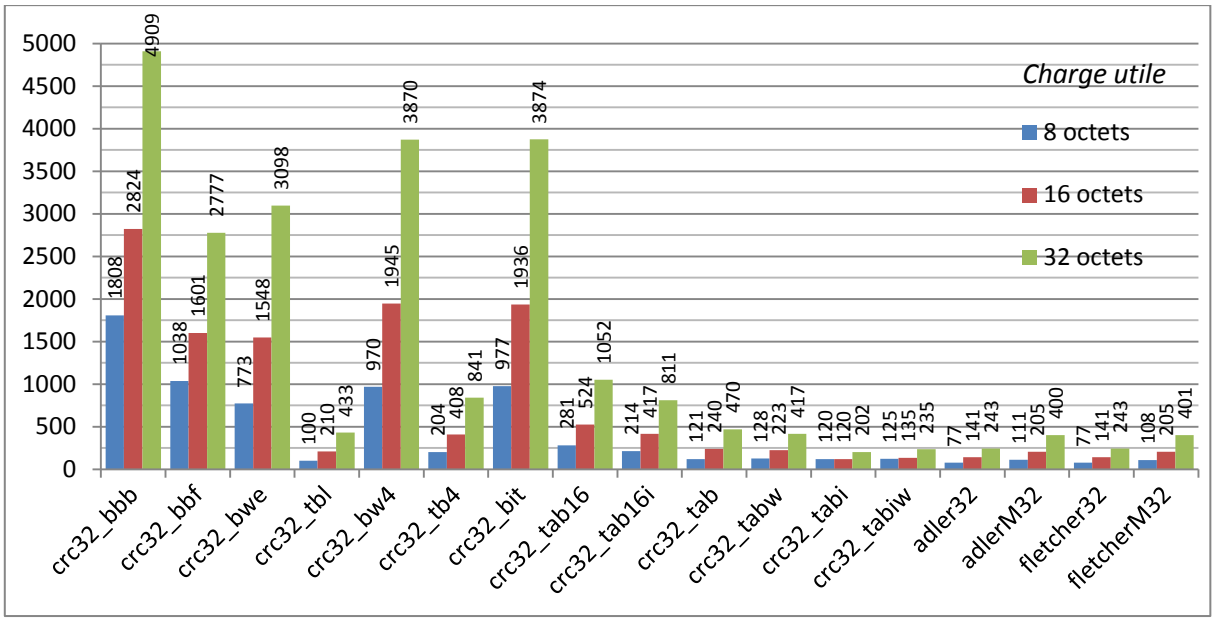


Figure III.13 : Temps de calcul (tick) « avec favorisation du cache » des codes 32 bits

On constate que les temps de calcul des implémentations *crcX_tabi* et *crcX_tabiw* dans les scénarios « avec favorisation du cache » sont réduits par rapport aux scénarios « sans favorisation du cache ». Cette réduction est telle que l'implémentation *crc32_tabi* a le temps de calcul le plus faible de toutes les implémentations. Ce qui confirme notre hypothèse que l'efficacité des implémentations à base de grandes tables de correspondance est favorisée par la mise en cache des tables de correspondance.

Pour voir plus clairement l'impact de la favorisation de la mise en cache des données et des programmes sur les temps de calcul des codes détecteurs d'erreurs, la Figure III.14 récapitule les résultats de la **comparaison de deux scénarios avec et sans favorisation du cache**, pour des codes de taille 32 bits et 8 octets de données de charge utile.
L'impact de l'utilisation du cache processeur apparaît clairement : les temps de calcul « avec favorisation du cache » sont systématiquement inférieurs aux temps de calcul « sans favorisation du cache », mais dans des proportions différentes. Si pour beaucoup d'implémentions le gain est mineur, il devient important pour quelques implémentations bit, et il est très important pour les implémentations *crc32_tabi* et *crc32_tabiw*, puisqu'il atteint un facteur de quelques dizaines. Nous posons alors la question de savoir si le cache ne permettra pas de reconsidérer de manière positive des implémentations et des codes qui, à ce jour, sont considérés comme lourds et donc non adéquats pour les systèmes embarqués critiques. Dans le même ordre d'idée, la piste de verrouillage des fonctions de calcul des codes dans les caches est également à explorer pour résoudre le problème du déterminisme qui pourrait être soulevé par l'utilisation des caches en rapport avec la caractérisation des pires temps d'exécution pour les systèmes embarqués critiques pour être certifiés.

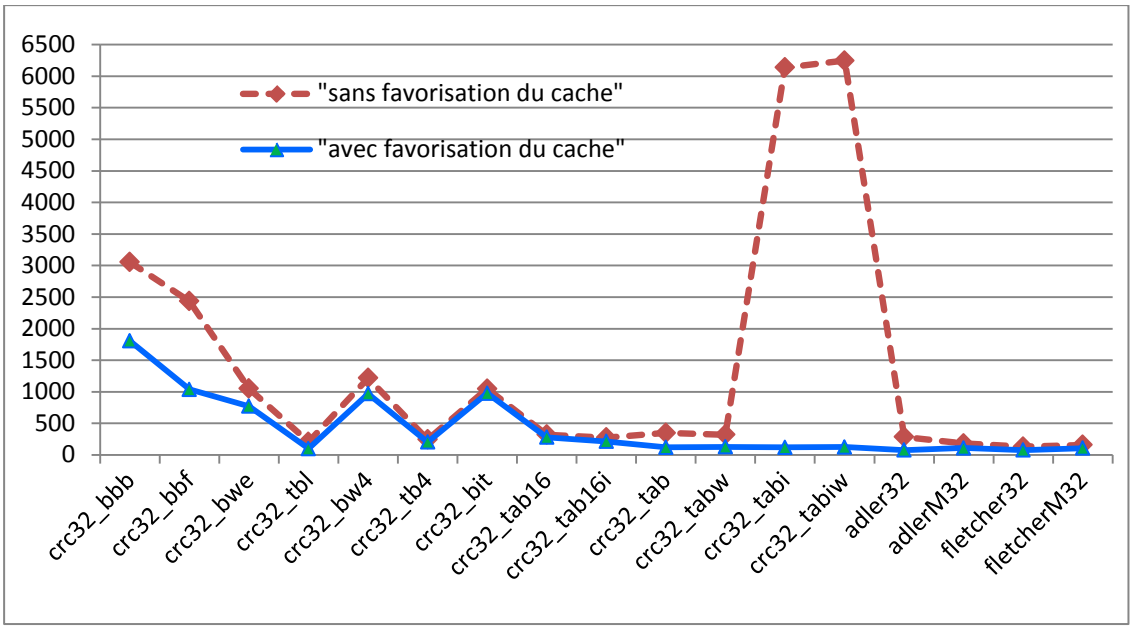


Figure III.14 : Temps de calcul « sans et avec favorisation du cache » et « avec favorisation du cache » pour les codes 32 bits avec 8 octets de charge utile

III.3.3.5 Évaluations sur un environnement matériel embarqué (carte STM32)

Après nos expérimentations sur un environnement matériel standard (ordinateur de bureau), nous avons effectué des expérimentations sur un processeur plus représentatif d'un environnement embarqué. Nous avons opté pour la famille des processeurs STM32, qui sont d'une utilisation très courante dans le milieu académique.

1) Description générale de la plateforme et de l'environnement

Les processeurs STM32 sont des processeurs 32 bits à architecture RISC fabriqués par la société STMicroelectronics à partir d'une IP (bloc propriété intellectuelle) développée par la société ARM. Le processeur que nous avons choisi pour nos expérimentations est un **processeur STM32F417**[8], qui fait partie de la série STM32 F4, premiers processeurs à utiliser un cœur ARM Cortex-M4F[9]. Ce choix a été motivé par le fait que ce processeur comporte des accélérateurs matériels pour le calcul d'un CRC32, ainsi que pour des fonctions de hachage ou cryptographiques. L'utilisation d'accélérateurs matériels pourrait être une piste à creuser pour obtenir des temps de calcul d'une fonction de contrôle compatibles avec un système temps-réel ayant des contraintes très sévères. Le processeur STM32F417 se caractérise par :

- une fréquence de 168 Mhz,
- 192 Ko de Ram et 1024 Ko de mémoire Flash
- pas de mémoire cache.

La carte d'évaluation utilisée pour exploiter ce processeur est une **carte STM3241G-EVAL**[10] (voir Figure III.15) qui dispose d'une sonde JTAG intégrée dans l'outil ST-LINK V2 présent

[8] http://www.st.com/web/en/catalog/mmc/FM141/SC1169/SS1577/LN11

[9] http://www.arm.com/products/processors/cortex-m/cortex-m4-processor.php

[10] http://www.st.com/web/en/catalog/tools/PF252217

sur la carte et accessible via le port USB de la carte. Les résultats des mesures sont récupérés via le port RSC232 présent sur la carte.
Enfin, côté logiciel, nous avons utilisé la solution ***µVision de Keil***[11] (en version 5.14) qui est un environnement de développement intégré pour processeurs ARM. Nous avons utilisé cet environnement en mode académique (version gratuite), ce qui impose des limitations, comme, notamment, de ne pas dépasser la taille de 32 Ko pour les exécutables générés.

2) Mesures du temps

Les mesures de temps sont exprimées en « tick », comme pour les évaluations sur la plateforme standard. Par contre, le processeur n'étant pas de la même famille que cette précédente plateforme, la fonction de base de mesure du temps que nous avons utilisé est différente. De même, la stratégie de calcul des valeurs à retenir est également différente, même si au final, cela revient au même. En effet, les durées d'exécution sont « déterministes » : l'exécution répétée d'un même code exécutable donne toujours le même temps. Et cela, parce les codes exécutables peuvent être chargés dans le processeur, de sorte à être exécuté sans passer par l'intermédiaire d'un système d'exploitation, ce qui élimine les fluctuations que ce dernier introduit (comme déjà évoqué).

3) Évaluations : scénarios et résultats présentés

Nous avons réalisé les mêmes séries d'évaluations que sur la plateforme standard, avec 8, 16 et 32 bits de contrôles, et 8, 16 et 32 octets de charge utile. Nous avons réalisé les simulations dans les configurations « sans favorisation du cache » puisque le processeur utilisé ne dispose pas de cache. Mais, il faut préciser que comme il dispose quand même de mémoire Flash, l'impact de mécanismes d'anticipation sur les temps de calcul n'est pas à exclure (ce n'est pas l'objet de nos simulations). Enfin, par manque de temps, nous n'avons pas étudié les apports éventuels des accélérateurs matériels qui avaient pourtant attiré initialement notre attention (cette piste reste donc à explorer).

Globalement, l'ensemble de ces simulations ont conduit à des résultats de même nature que ceux obtenus sur la plateforme standard. Les différences observées n'entraînent pas de nouvelles conclusions, hormis pour les temps réel de calcul, qui eux, seront discutés dans la section III.3.3.6 plus loin.
Par contre, nous avons pu évaluer un nouveau facteur d'influence sur les temps d'exécution : les options d'optimisation proposées par le compilateur. Nous avions déjà tenté des expérimentions sur la plateforme standard, mais sans succès, en raison d'incompatibilité de la mesure avec l'optimisation.
Pour ne pas surcharger le document, nous ne présentons dans cette section que des résultats sur un scénario avec 32 bits de code et 32 octets de charge utile, évalué avec deux options du compilateur :

- l'option o2, qui génère du code exécutable optimisé pour le temps d'exécution.
- l'option o0, qui n'entraîne aucune optimisation.

Les résultats sont d'abord présentés dans le Tableau III.7, puis dans la Figure III.16. On constate que l'option d'optimisation o2 du compilateur permet systématiquement de réduire les temps d'exécution même si, pour une bonne partie des implémentations, le gain est mineur. Mais il devient intéressant pour la majorité des implémentations qui ont les plus

[11] http://moodle.insa-toulouse.fr/pluginfile.php/34872/mod_resource/content/1/Prise_en_main_Keil.pdf

mauvais temps d'exécution. On voit également, même si les résultats sont présentés sous une forme différente que, l'ordre de grandeur des temps d'exécutions (en tick), et les classements en performance des d'implémentation suivent les même tendances que sur l'environnement matériel standard.

Figure III.15 : Carte d'évaluation STM3241G-EVAL du processeur STM32F417

Option de compilation / **Code 32 bits**	o2	o0
crc32_tabiw	356	544
adler32	366	434
crc32_tabw	384	468
crc32_tabi	422	656
fletcher32	434	462
crc32_tbl	486	620
crc32_tab16	496	826
crc32_tab	500	868
fletcherM32	542	604
adlerM32	574	640
crc32_tb4	578	720
crc32_tab16i	602	1236
crc32_bit	2024	4210
crc32_bbf	3238	4346
crc32_bwe	4398	6272
crc32_bbb	5202	6768
crc32_bw4	6616	8092

Tableau III.7 : Temps d'exécution sur STM32 avec options de compilation o0 et o2

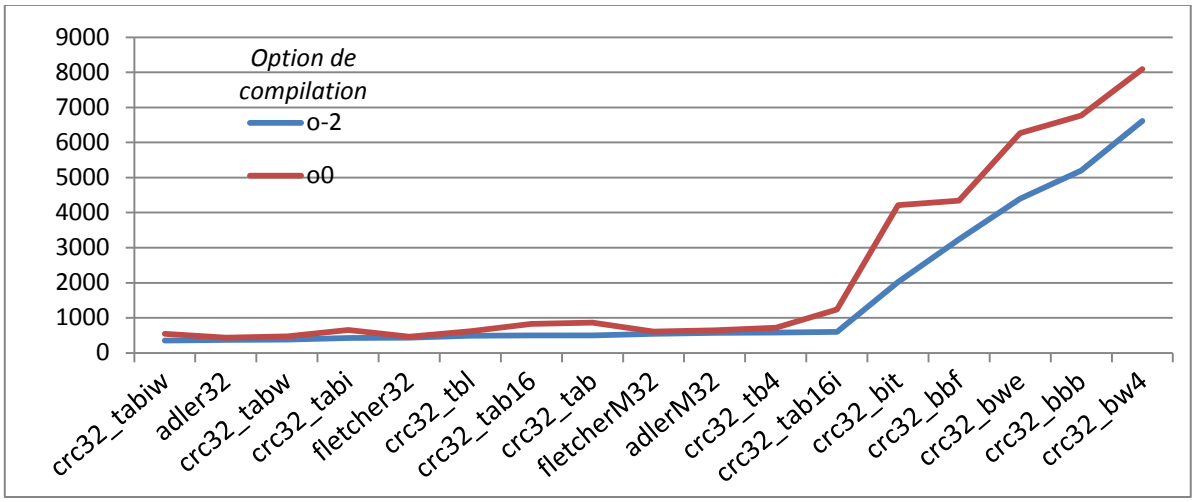


Figure III.16 : Temps d'exécution sur STM32 avec deux options de compilation

III.3.3.6 Comparaison des résultats sur environnements matériels standard et embarqué

Après avoir évalué les temps d'exécution sur deux environnements matériels différents, il est logique d'établir une comparaison entre les résultats obtenus dans ces deux cas, et également de rapprocher ces résultats de la réalité des systèmes que nous ciblons. Toutes les mesures des évaluations faites ont été exprimées en nombre de « tick », qui correspond, en première approche, à un « nombre » de cycles d'horloge du processeur (avec comme base : 1 tick = 1 cycle d'horloge). Cette manière de procéder permet de rendre les évaluations indépendantes de la valeur chiffrée d'un tel cycle, qui, par nature, est totalement dépendant d'un processeur donné, et des paramètres de son utilisation. Et donc, on peut ainsi plus facilement comparer des évaluations faites sur différentes plateformes. Il n'en reste pas moins, qu'in fine, en vue d'implémentation dans des systèmes opérationnels, il faut regarder ce que donne ce nombre de tick une fois traduit en temps réel, même si le résultat d'une telle traduction est à relativiser très fortement, tant il est dépendant de la configuration des processeurs. On ne peut donc donner que des ordres de grandeurs.

Sur l'environnement matériel standard (la plateforme i7), les temps de calcul des 17 différentes implémentations pour les codes 32 bits (dans les scénarios « sans favorisation du cache ») s'échelonnent entre 287 cycles (adler32) à 8756 cycles (crc32_bbb). Sachant que le cycle horloge du processeur utilisé est de 0,43 ns (fréquence de 2,3 Ghz), cela correspond à des valeurs allant de 0,12 µs à 3,77 µs. Mais, par exemple, avec exactement le même processeur et le même compilateur, il faudrait multiplier ces résultats par 2 si l'on avait une machine de bas de gamme avec un processeur à 1,15 Ghz. Ce simple exemple illustre pourquoi dans le milieu académique, on préfère s'abstraire de la fréquence d'horloge en donnant le temps de calcul en *tick*.

Pour ce qui est de la plateforme STM32, les valeurs vont de 434 cycles (adler32) à 8092 cycles (crc32_bw4), soit respectivement 2,58 µs à 48,15 µs, sachant que le cycle processeur est à 5,95 ns (fréquence de 168 Mhz).

On voit donc que les valeurs en µs sont donc de l'ordre de 14 fois plus importantes (c'est le rapport des horloges) sur la deuxième plateforme. Si l'on compare les architectures i7 et STM32 en cycles horloge, cela donne + 49% pour l'implémentation la plus rapide et -7,6 % pour l'implémentation la plus lente quand on passe de l'i7 au STM32.

Sachant que, d'une part, les jeux d'instructions des processeurs sont différents et que l'on passe d'un processeur 64 bits à 32 bits, et que, d'autre part, les compilateurs sont différents, on peut considérer finalement que les écarts entre les résultats ne sont pas si importants que ça.

Enfin, il est important de préciser que les valeurs exprimées en temps réel sont simplement « indicatives ». Il n'est pas possible d'en déduire des conclusions plus précises. Cela ne peut être fait qu'en mettant ces valeurs en regard avec les contraintes temps réel d'un système donné et de l'application nécessitant l'usage de fonction de contrôle : processeur, fréquence de rafraîchissement des opérations dédiés de l'application, niveau d'intégrité requis, nombre d'appels des fonctions de contrôle, etc. Le chapitre IV, dédié à l'application de notre approche sur un cas d'étude industriel, sera l'occasion d'illustrer tous ces points.

III.4 APPROCHE D'INTÉGRITÉ MULTI-CODES

Après avoir présenté et évalué l'approche d'intégrité mono-code, nous nous focalisons maintenant sur la deuxième approche proposée qui est l'approche d'intégrité multi-codes dont le principe général est décrit dans la section III.1.3. Nous commençons par décrire les caractéristiques des systèmes que nous ciblons pour appliquer cette approche, nous discutons par la suite le problème du mode commun de défaillance qui nous a conduits à proposer l'approche multi-codes. Nous discutons enfin le critère de complémentarité qui distingue l'approche multi-codes.

III.4.1 Caractéristiques des systèmes ciblés

Dans les systèmes embarqués critiques, atteindre le haut niveau d'intégrité requis se heurte souvent aux contraintes de ces systèmes qui sont réduits en volume et poids et munis des performances et ressources limitées. Ces systèmes, étant complexes, ne peuvent pas être conçus avec le concept « zéro fautes » assuré généralement par la prévention et l'élimination de toutes fautes (voir chapitre I, section I.2.3). Ces systèmes doivent être plutôt tolérants aux fautes en mettant en place un ensemble de stratégies de redondance permettant de masquer et recouvrir les fautes (ou les erreurs). Le niveau de redondance diffère d'un système à l'autre. Dans le premier volet de ce chapitre, nous avons ciblé les systèmes « très contraignants » où le niveau de redondance est réduit, cela correspond dans notre cas au fait qu'un seul exemplaire du même message est envoyé. Dans ce deuxième volet, nous ciblons plutôt des systèmes embarqués critiques avec un niveau de redondance plus important. Nous appelons ces systèmes « systèmes contraignants », cela se traduit dans notre cas par le fait qu'une information peut donner lieu à la transmission de plusieurs exemplaires du même message. Comme décrit en section III.1.3, l'intégrité des communications est considérée différemment dans les deux types de systèmes. Particulièrement, dans l'approche multi-codes, l'intégrité est quantifiée sur un lot d'exemplaires du même message d'où la possibilité de tolérer la non détection de certains exemplaires erronés. L'objectif dans l'approche multi-codes est de détecter au moins une fois l'erreur si on suppose que tous les exemplaires sont affectés.

Différents systèmes ont un niveau de redondance permettant de transmettre plusieurs exemplaires du même message, les caractéristiques de ces différents systèmes sont identiques à ceux appartenant à une de ces trois classes suivantes.

- La classe des systèmes mettant en place une **redondance spatiale** (définie dans chapitre I, section I.3.3.4) : l'exemple le plus commun de la transmission redondante via une redondance spatiale est le cas de duplication des canaux de communication permettant la transmission de plusieurs exemplaires du message, un exemplaire par canal de communication. La duplication des canaux de communication est généralement adoptée pour prévenir la perte des données et augmenter le niveau de sûreté des communications.
- La classe des systèmes mettant en place une **redondance temporelle** (définie dans chapitre I, section I.3.3.4) délibérée permettant la transmission de plusieurs exemplaires du même message dans des instants (des cycles) différents (successifs ou pas): nous considérons la redondance temporelle délibérée quand le système met en place une redondance temporelle volontaire dans l'objectif de réduire le risque de perte de messages, d'augmenter la fiabilité, de détecter les erreurs en utilisant un système de vote.

- La classe des systèmes mettant en place une **redondance temporelle** non délibérée : un exemple de redondance temporelle non délibérée est le cas des systèmes dits « à **dynamique lente** » [Youssef 2005] [Zammali *et al.* 2013]. En effet, nous classifions les systèmes en deux classes: 1) systèmes à dynamique rapide et 2) systèmes à dynamique lente. Nous définissons la propriété « dynamique lente » par le fait que la durée de l'intervalle entre deux changements significatifs (le changement significatif se traduit par la variation de la valeur de la donnée à transmettre) est beaucoup plus importante que le cycle de rafraichissement (la période d'envoi des messages). Par conséquent, la donnée à transmettre reste la même (en valeur de donnée utile) durant un ensemble c de cycles (c>=2) avant l'occurrence d'un changement significatif. Dans l'exemple de la Figure III.17, trois exemplaires du même message D1 sont envoyés. Par contre, dans les systèmes à dynamique rapide, la durée de cet intervalle est très proche de la durée du cycle de rafraichissement et donc, pour une valeur de donnée, le message n'est envoyé qu'une seule fois. Notre système de référence de la classe des systèmes à dynamique lente est le système de commande de vol avion (voir chapitre I, section I.1.2). Les surfaces de l'avion sont conçues pour se déplacer lentement et le résultat du calcul de la trajectoire restant le même pendant un ensemble de cycles de rafraichissement, plusieurs exemplaires du même message sont alors envoyés.

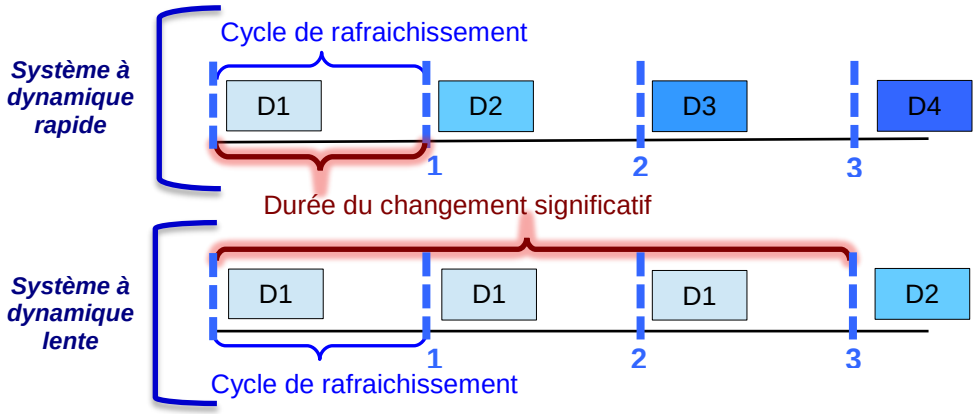


Figure III.17 : Système à dynamique rapide VS système à dynamique lente

Notre approche multi-codes s'applique à toutes les classes des systèmes décrits précédemment. La section suivante discute le problème du mode commun de défaillance qui nous a conduits à la proposition de l'approche multi-codes, ce problème met en évidence l'apport et la pertinence de l'approche multi-codes.

III.4.2 Problème du mode commun de défaillance

Dans le premier volet de ce chapitre, nous avons proposé une approche d'intégrité mono-code dont l'efficacité est probante pour les systèmes « très contraignants ». Pour les systèmes « contraignants » qui permettent une transmission de plusieurs exemplaires du même message, l'approche mono-code peut être améliorée. En effet, dans le cas où l'erreur est répétitive (affectant tous les exemplaires) (Figure III.18) et non détectée une fois par le code détecteur, elle ne sera jamais détectée puisque le même code est utilisé pour tous les exemplaires. Ce scénario est un cas du mode commun de défaillance (défini dans le chapitre I, section I.2.2.1) où la redondance symétrique qui consiste à utiliser le même code détecteur d'erreurs pour tous les messages est vulnérable aux erreurs répétitives.

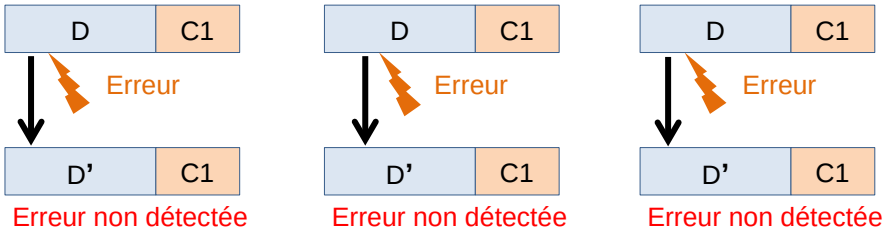


Figure III.18 : Problème du mode commun de défaillance dans l'approche mono-code

Un exemple pour illustrer ce problème du mode commun de défaillance dans le cas de l'utilisation d'un CRC est quand l'erreur est répétitive et multiple de son polynôme générateur. L'erreur ne sera jamais détectée (sur tous les exemplaires affectés par l'erreur). L'utilisation du même code détecteur dans le cas où on a plusieurs exemplaires du même message est une forme de redondance symétrique. La redondance symétrique est intrinsèquement vulnérable au problème du mode commun de défaillance. L'approche multi-codes que nous proposons s'inspire des techniques de « *design diversity* » recommandées pour résoudre ce problème.

III.4.3 Approche multi-codes et complémentarité des codes détecteurs d'erreurs

L'approche multi-codes se base sur le concept de la redondance asymétrique (voir chapitre I, section I.3.3.1) et est inspirée des travaux antérieurs de notre équipe de recherche [Youssef 2005] ciblant exclusivement les codes CRC. Les systèmes ciblés sont des systèmes avec un niveau de redondance permettant d'avoir plusieurs exemplaires du même message et donc de tolérer la non détection de certains exemplaires erronés. La Figure III.19 décrit le principe général de l'approche multi-codes

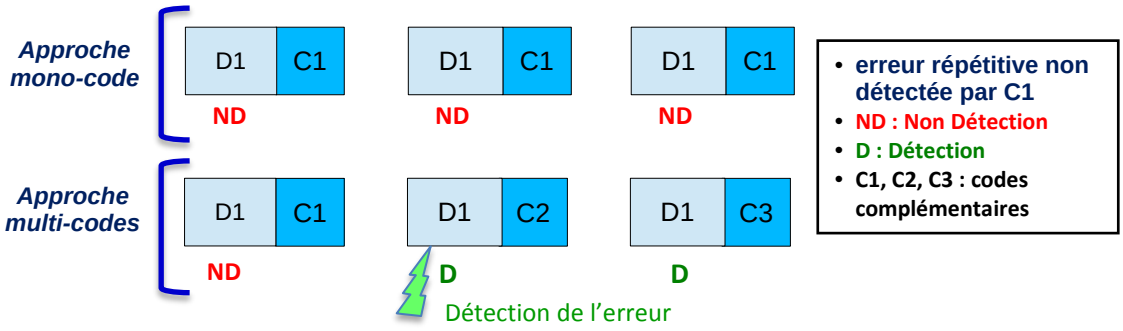


Figure III.19 : Principe général de l'approche multi-codes

Comme nous l'avons défini dans la section III.1.4, la stratégie de sélection des codes à adopter pour l'approche multi-codes consiste en un compromis entre trois critères : le pouvoir de détection intrinsèque de chaque code, son coût en termes de temps d'exécution et la complémentarité entre l'ensemble de codes considérés. Les deux premiers critères sont longuement détaillés dans le premier volet de ce chapitre sur l'approche mono-code. Nous nous concentrons ici sur le troisième critère qui distingue l'approche multi-codes et qui concerne la « complémentarité entre les codes ».

Nous définissons la **complémentarité** entre deux codes détecteurs d'erreurs par le fait qu'un code est capable de détecter des erreurs non détectées par l'autre code. Si on schématise cette définition avec la théorie des ensembles, deux codes sont dits complémentaires si leurs ensembles respectifs de couverture d'erreurs ne se recouvrent pas.

Cela peut être formalisé (**Figure III.20**) comme suit :

C1 et C2 sont complémentaires SSi E1 U E2 <> E1 et E1 U E2 <> E2

Avec C1 et C2 deux codes détecteurs d'erreurs, E1 et E2 sont l'ensemble des erreurs couvertes respectivement par C1 et C2

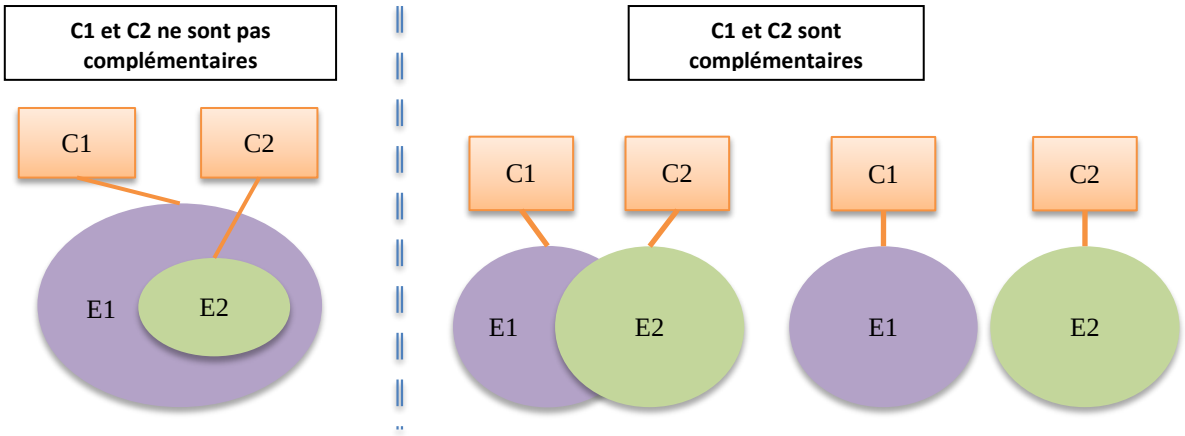


Figure III.20 : Principe de complémentarité entre deux codes détecteurs d'erreurs

Puisque nous avons présélectionné Adler, Fletcher et CRC pour nos deux approches d'intégrité, nous avons analysé les travaux existants sur ces codes pour chercher des éléments qui peuvent indiquer une possible complémentarité entre eux. Ce que nous avons retenu est le fait que Adler et Fletcher sont vulnérables aux erreurs en rafale qui inversent les bits de tout-zéro à tout-un [Nakassis 1988], Fletcher est vulnérable à certaines erreurs de taille 2 où les bits sujets de l'erreur sont initialement différents et non les deux de valeur 0 ou 1 et CRC ne détecte pas les erreurs multiples de son polynôme générateur. Le fait que ces codes n'ont pas les mêmes vulnérabilités pourrait supposer que ces codes peuvent être complémentaires, une hypothèse à prouver par les simulations.

III.5 ÉVALUATIONS POUR L'APPROCHE MULTI-CODES

Nous présentons ici les évaluations faites pour valider l'efficacité de l'approche multi-codes ; nous commençons par présenter le contexte général de ces évaluations, puis nous décrivons nos résultats prouvant la complémentarité des codes présélectionnés.

III.5.1 Contexte général

Cette section décrit en détail nos expérimentations dont l'objectif est d'évaluer la complémentarité (en mesurant le taux de non détection global) des codes présélectionnés en termes de pouvoir de détection. Les deux autres critères sur lesquels se base la stratégie de sélection de l'approche multi-codes à savoir le coût de calcul et le pouvoir intrinsèque de chaque code sont décrits dans le volet consacré à l'approche mono-code de ce chapitre. Il ne reste qu'à évaluer la complémentarité des codes pour estimer l'apport de l'approche multi-codes. Tout comme pour l'approche mono-code, pour l'évaluation de la complémentarité des codes en termes de pouvoir de détection, nous avons utilisé deux environnements logiciels d'évaluation : des évaluations préliminaires (dont les résultats sont décrits dans [Zammali *et al.* 2014] et [Zammali *et al.* 2015a]) sur Matlab-Simulink (décrit dans la section III.3.2.1) puis sur un environnement en C (le même que pour les évaluations pour l'approche mono-code en ajustant la fonction de mesure du taux de non détection). Pour les mêmes limitations soulignées dans les évaluations de l'approche mono-code, c'est l'environnement développé en C (voir section III.3.2.1) qui a très rapidement été privilégié.

Pour l'environnement matériel et puisqu'il n'a pas d'impact sur le pouvoir de détection, nous avons mené nos évaluations uniquement sur un environnement matériel standard (dont les caractéristiques sont décrites dans la section III.3.3.4).

Nous ne présentons que nos évaluations et résultats sur l'environnement en C puisque sur l'environnement Matlab-Simulink nous n'avons pu mener des évaluations que pour un nombre de cas de test très limité donc les résultats risquent de ne pas être suffisamment représentatifs, deux exemples de modèle Matlab-Simulink de l'approche multi-codes sont présentés dans l'annexe 3.

Puisque nous visons dans ces travaux de thèse les erreurs multiples et puisque les expérimentations exhaustives requièrent un temps irréaliste, la génération des erreurs sur laquelle se basent nos évaluations est aléatoire en se basant sur le principe de Monte Carlo. Le pouvoir de détection global de l'approche multi-codes est évalué en calculant le taux de non détection global (appelé **Pud**). Plus précisément, notre métrique est le taux de non détection global qui est le rapport du nombre de messages erronés non détectés par tous les codes de l'approche multi-codes et le nombre de messages testés. Dans nos simulations, tous les messages générés sont erronés et donc le nombre de messages erronés est égal au nombre total des messages testés. Un message erroné est considéré comme non détecté par l'approche multi-codes quand aucun des codes n'arrive à détecter qu'il est erroné (et considéré comme détecté quand il y'a au moins un code qui arrive à le détecter).

Comme pour l'approche multi-codes, les calculs des intervalles de confiance nous a conduits à fixer le nombre de cas de tests à $\boldsymbol{2^n*100}$ avec *n* le nombre global de bits de contrôle (la somme de toutes les longueurs des codes utilisés), soit la somme de tous les bits de contrôle de tous les codes utilisés dans l'approche multi-codes. Nous n'allons pas présenter les calculs des intervalles de confiance qui ont été longuement détaillés dans le premier volet du chapitre.

Dans nos évaluations, l'erreur est toujours répétitive (tous les exemplaires sont affectés par la même erreur).

Pour les codes de taille 32, nous n'avons pas pu avoir des résultats sur des approches multi-codes vu que le nombre de cas de test pour une fonction bi-codes avec une taille de 32 bits pour les deux codes, pour avoir un bon niveau de confiance, il faut 1.84e+21 cas de test ce qui donne un temps d'expérimentations irréaliste et donc non réalisable.

Nous présentons alors les résultats pour les trois scénarios suivants :

- Scénario n°1 : une approche bi-codes, la taille de chaque code est de 8 bits, charge utile de 8, 16 et 32 octets et $2^{16}*100 = 6553600$ cas de test.
- Scénario n°2 : une approche tri-codes, la taille de chaque code est de 8 bits, charge utile de 8, 16 et 32 octets et $2^{24}*100 = 1677721600$ cas de test.
- Scénario n°3 : une approche bi-codes, la taille de chaque code est de 16 bits, charge utile de 8, 16 et 32 octets et $2^{32}*100 = 429496729600$ cas de test.

III.5.2 Résultats des évaluations de la complémentarité des codes

III.5.2.1 Scénario n°1 : approche bi-codes combinant des codes 8 bits

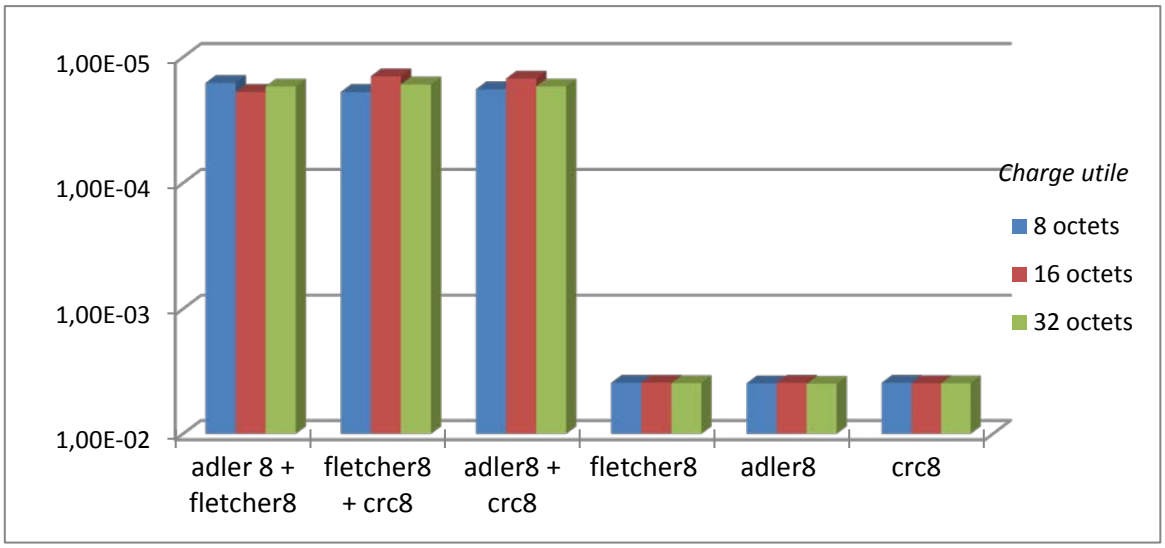


Figure III.21 : Taux de non détection de l'approche multi-codes, scénario de 2 codes de 8 bits

Des résultats décrits par la Figure III.21 comparant le taux de non détection de l'approche bi-codes (combinant adler8 et fletcher8, puis fletcher8 et crc8, et enfin adler8 et crc8) et de l'approche mono-code (fletcher8 ou adler8 ou crc8) utilisant des codes détecteurs 8 bits, nous constatons que :

- L'approche multi-codes a un pouvoir de détection nettement important (taux de non détection plus faible) que celui de l'approche mono-code pour le même nombre de bits de contrôle par message.
- Le taux de non détection global converge vers un maximum théorique 2^{-n} avec *n* la somme des bits de contrôle de l'ensemble de codes utilisés (16 dans ce cas, maximum théorique égal à $1{,}53*10^{-5}$).
- Comme constaté dans les évaluations précédentes, la taille de la charge utile n'as pas d'impact sur le pouvoir de détection des codes détecteurs d'erreurs (quelques fluctuations dues à la génération aléatoire).

III.5.2.2 Scénario n°2 : approche tri-codes combinant des codes 8 bits

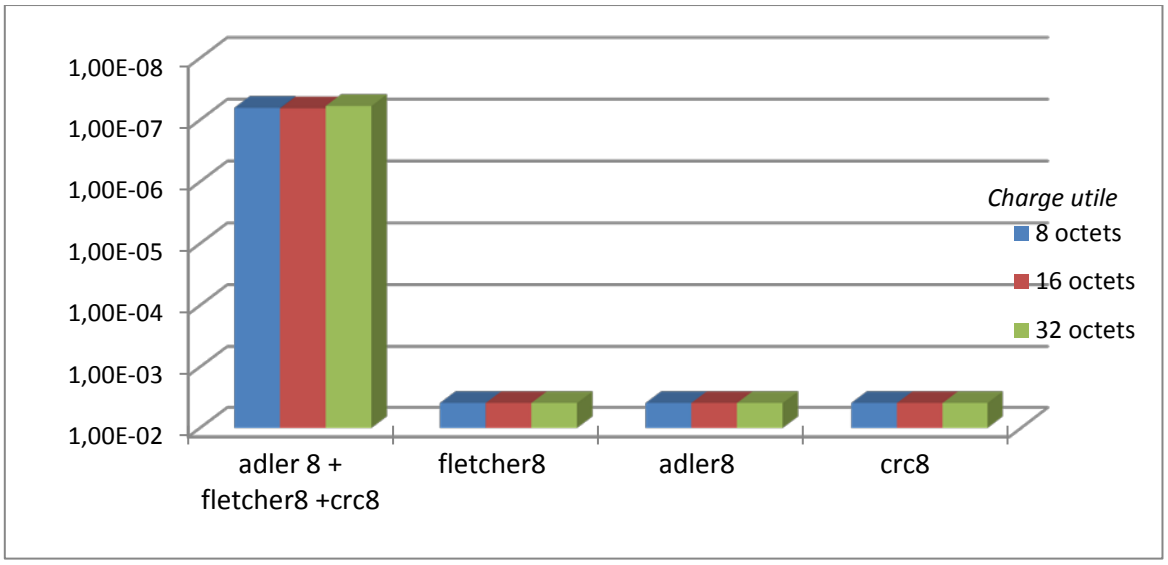


Figure III.22 : Taux de non détection de l'approche multi-codes, scénario de 3 codes de 8 bits

Pour l'approche tri-codes combinant adler8, fletcher8 et crc8, les résultats ont les mêmes tendances (Figure III.22) que pour l'approche bi-codes combinant des codes 8 bits. Le taux de non détection de l'approche tri-codes est beaucoup plus faible que les approches mono-code avec un seul code 8 bits (fletcher8, adler8 ou crc8), il converge vers $2^{-24} = 5{,}96*10^{-8}$ qui est le maximum théorique de l'approche mono-code avec un code 24 bits. Même constat sur l'absence de l'impact de la taille de la charge utile sur les pouvoirs de détection.

III.5.2.3 Scénario n°3 : approche bi-codes combinant des codes 16 bits

Enfin, pour l'approche bi-codes combinant des codes 16 bits, les résultats conduisent globalement aux mêmes conclusions (Figure III.23) que les deux précédents scénarios avec notamment une convergence vers le maximum théorique égal à $2^{-32} = 2{,}32831*10^{-10}$. Un résultat qui nous a interpellés est la **complémentarité parfaite** entre **Adler** et **Fletcher 16 bits dont le taux de non détection est égal à 0** (non représenté sur la Figure III.23). Cette complémentarité parfaite est sûrement due à la génération aléatoire dont l'aspect aléatoire se dégrade quand il s'agit des simulations très longues tel le cas de ce scénario donc nous ne considérons pas qu'Adler16 et Fletcher16 sont totalement complémentaires (ceci n'exclut pas la nécessité de faire une recherche mathématique sur les deux codes qui pourrait expliquer cette « complémentarité parfaite ». Nous avons tenté d'explorer cette piste, mais c'est un travail purement mathématique et donc hors périmètre de nos travaux.

Le résultat que nous retenons est que l'approche multi-codes a un meilleur pouvoir de détection comparée à une approche mono-code pour le même nombre de bits de contrôle par message (même longueur du code détecteur d'erreurs par message).

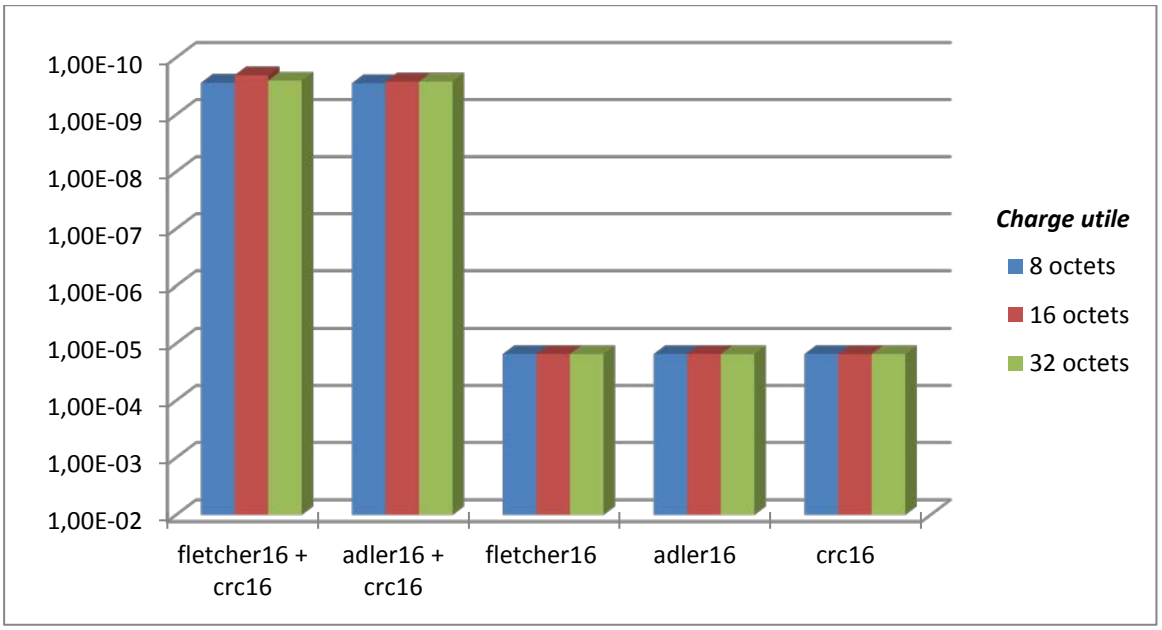


Figure III.23 : Taux de non détection de l'approche multi-codes, scénario de 2 codes de 16 bits

De tous ces résultats, on peut constater que la combinaison des codes Adler, Fletcher et CRC augmente nettement le pouvoir de détection global donc on peut confirmer qu'ils sont complémentaires.

III.5.3 Conclusion sur l'approche multi-codes

Une conclusion générale à partir de ces évaluations et résultats est que dans le contexte des systèmes embarqués critiques, et si les caractéristiques des systèmes en question le permettent (déployant certaines formes de redondance), nous pouvons atteindre un même niveau d'intégrité des communications utilisant moins de bits de contrôle et ce, en utilisant l'approche multi-codes. Par exemple, pour un système de commande de vol avion qui est un système à dynamique lente, on peut utiliser deux codes détecteurs 16 bits dans un contexte d'approche multi-codes au lieu d'utiliser une approche mono-code basée sur un code 32 bits. Cela nous permet d'avoir le même niveau d'intégrité (taux de non détection inférieur ou égal à $2^{-32} = 2{,}33*10^{-10}$) avec moins de bits de contrôle par message.

III.6 CONCLUSION

Dans ce chapitre, nous avons présenté la principale contribution de ces travaux de thèse qui consiste en une approche d'intégrité bout en bout basée sur le concept du « canal noir » et utilisant les codes détecteurs d'erreurs. Dépendamment du niveau de redondance dans le système ciblé, notre approche générique se décline en deux approches différentes : une approche mono-code pour les systèmes « très contraignants » et une approche multi-codes pour les systèmes « contraignants ». Nous avons commencé d'abord par présenter le principe général de l'approche d'intégrité bout en bout, puis nous nous sommes focalisés sur l'approche mono-code en décrivant d'abord sa philosophie puis les évaluations que nous avons faites pour évaluer le temps de calcul et le pouvoir de code d'un ensemble de codes présélectionnés. De même, pour l'approche mono-code, nous avons d'abord décrit son principe général puis les évaluations qui ont prouvé la complémentarité des codes présélectionnés et l'apport de l'utilisation de cette approche.

Des évaluations menées pour l'approche mono-code, nous avons prouvé que :

- les codes CRC peuvent bien être utilisés dans le contexte des systèmes embarqués critiques. En effet, les implémentations basées sur des tables de correspondance ont des performances en termes de temps de calcul très prometteuses et équivalentes (parfois meilleures) que celles des sommes arithmétiques comme Adler et Fletcher connus par leurs coûts de calcul théoriquement inférieurs à ceux du CRC. L'utilisation des tables de correspondance est généralement freinée par deux problèmes :
    - Le fait que certains systèmes embarqués critiques ne disposent pas d'assez de mémoire. Mais grâce aux progrès technologiques, l'utilisation des tables devient de plus en plus commune, les systèmes embarqués critiques disposent de plus en plus d'accélérateurs matériels, de mécanismes d'anticipation et de caches permettant de profiter de ces implémentations CRC optimisées.
    - Le problème de déterminisme auquel s'oppose l'utilisation des tableaux de correspondance. Pour certains systèmes critiques où la certification impose une caractérisation déterministe des pires temps d'exécution des différentes fonctions et applications, nous proposons de creuser la piste de verrouillage des données et programmes dans le cache pour traiter ce problème.

- Dans un contexte des erreurs multiples et uniformément distribuées, les pouvoirs de détection des codes détecteurs d'erreurs de la même longueur se valent et convergent vers un maximum théorique égal à 2^{-n} (n longueur du code) même si on a constaté dans nos expérimentations quelques fluctuations dues à l'aspect aléatoire de nos expérimentations.
- Adler et Fletcher sont un très bon compromis pour les systèmes embarqués critiques avec un bon pouvoir de détection et un temps de calcul faible, les implémentations optimisées de CRC le sont aussi.

De notre proposition de l'approche multi-codes et des évaluations menées, nous tirons ces constats :

- Certains systèmes embarqués critiques contraignants ont un niveau de redondance permettant de repenser l'intégrité des communications et raisonner sur un lot de messages et non pas sur un seul message.
- L'approche multi-codes n'engendre pas l'augmentation du nombre de bits (le même nombre de bits que l'approche mono-codes) mais a un apport très important en termes de pouvoir de détection. Les évaluations ont montré que le taux de non détection d'une approche multi-codes converge vers un maximum théorique 2^{-n} avec n la somme des longueurs des codes utilisés.
- Nous avons évalué l'approche multi-codes dans un contexte des erreurs répétitives (la même erreur affectant tous les exemplaires du message), c'est le contexte le plus favorable pour montrer la complémentarité entre les codes (puisque cela évalue le pouvoir de détection sur la même erreur à chaque fois). Mais il n'en reste pas moins vrai que l'approche multi-codes est efficace dans le cas où l'erreur n'est pas répétitive. Puisque le pouvoir de détection est cumulatif et sera au moins égal au maximum des pouvoirs de détection des différents codes utilisés par l'approche multi-codes.

Dans le chapitre suivant, nous allons présenter les résultats de l'application de l'approche d'intégrité mono-code dans un cas d'étude industriel qui est celui des futurs systèmes de commande de vol d'Airbus Helicopters.

Chapitre IV

APPLICATION : FUTURS SYSTÈMES DE COMMANDE DE VOL D'AIRBUS HELICOPTERS

Dans ce chapitre, nous décrivons les résultats de l'application d'une partie des contributions de la thèse sur un cas d'étude. De par l'expérience déjà acquise dans notre équipe lors de travaux précédents, nous avions initialement envisagé comme cas d'étude les systèmes de Commandes De Vol (CDV) d'avions civils. Ce choix a changé, lorsqu'en début de la dernière année de ces travaux de thèse, une collaboration est née avec Airbus Helicopters autour de la problématique d'intégrité des communications dans les systèmes de commande de vol de leurs futurs hélicoptères. Cette problématique est cruciale parce que les CDV hélicoptères présentent des différences, par rapport aux CDV avions, plus importantes qu'il n'y paraît au premier abord. Cette collaboration s'est axée particulièrement sur l'utilisation des codes détecteurs d'erreurs pour l'intégrité des communications de bout en bout.

Nous commençons ce chapitre par une section d'introduction élémentaire sur les CDV, pour présenter de manière très synthétique le rôle d'un CDV et ses propriétés. La deuxième section a pour but d'amener progressivement à la description détaillée de l'architecture dans les CDV Électriques (CDVE) des hélicoptères actuels d'Airbus Helicopters, particulièrement celle du NH 90, tout en présentant ses limitations. Pour en arriver à cette description, cette section retrace d'abord brièvement l'historique des évolutions des commandes de vol dans les deux industries avions et hélicoptères. Ceci permet d'enchaîner, avec une étude de l'existant sur les architectures des CDV, toujours dans ces deux industries, suivie d'une description plus fine des principaux composants du CDVE hélicoptère actuel. Les limitations de cette architecture actuelle, mises en lumière dans cette section, les contraintes de certification ainsi que les nouveaux défis comme celui du passage au « tout numérique » ont conduit à la conception d'une nouvelle architecture que nous décrivons dans la troisième section en nous focalisant sur la politique d'intégrité des communications adoptée.

Le deuxième volet de ce chapitre s'intéresse aux expérimentations qui ont pour objectif l'évaluation d'un ensemble de codes détecteurs d'erreurs, les résultats ainsi que nos analyses aident Airbus Helicopters à choisir le code qui assurera l'intégrité des communications bout en bout dans les futurs CDVE.

IV.1 SYSTÈME DE COMMANDE DE VOL : DESCRIPTION GÉNÉRALE

Cette section est une introduction élémentaire sur les CDV dont l'objectif est de dresser le cadre de certaines bases communes aux aéronefs de type avion et hélicoptère. C'est un préalable nécessaire à la description plus détaillée, dans les deux sections suivantes, des architectures CDV actuelles et futures.

IV.1.1 Rôle du système de commande de vol

Le principal rôle du système de commande de vol est le contrôle de la trajectoire de l'aéronef dans les trois phases : décollage, vol et atterrissage. Notons que, par abus de langage, dans la suite, lorsqu'on utilisera les termes « vol » ou « voler », c'est pour couvrir ces trois phases. Pour parvenir à voler, quatre forces sont mises en jeu, définies ci-après, et illustrées sur le cas d'un hélicoptère sur la Figure IV.1 :

- La poussée : c'est la force qui permet à l'aéronef de décoller et d'avancer ;
- La trainée : c'est une force de ralentissement causée par la perturbation des flux d'air par les surfaces « portantes » (par exemple, les ailes ou le rotor), le fuselage et d'autres surfaces saillantes, force qui s'oppose à la poussée ;
- La portance : c'est la force produite par l'effet dynamique de l'air agissant sur la ou les surfaces portantes et agit perpendiculairement à la trajectoire du vol ;
- Le poids : c'est la force qui tire l'hélicoptère vers le bas par la force de la gravité, elle s'oppose à la portance.

Figure IV.1 : Les quatre forces appliquées à un hélicoptère

Pour contrôler la trajectoire d'un aéronef, le pilote doit pouvoir jouer sur la combinaison de ces quatre forces. Même si l'aéronef est motorisé (ce qui n'est pas toujours le cas, comme par exemple les planeurs), cela ne suffit pas. L'aéronef doit nécessairement être équipé de **« surfaces mobiles »**, c'est-à-dire de surfaces dont la position peut être modifiée au cours du vol. Ces surfaces sont donc équipées, a minima, d'actionneurs, mais également, depuis longtemps, de capteurs (de position par exemple).

Ainsi, entre le pilote et les surfaces mobiles, il doit y avoir un système qui permet au pilote de donner des ordres de contrôle de la trajectoire, et de « transformer » ces ordres en actions adéquates sur les surfaces mobiles. À titre d'exemple, la Figure IV.2 montre dans le cas d'un hélicoptère avec uniquement les trois organes de commande à partir desquels le pilote « contrôle » les surfaces mobiles : la commande du pas collectif (via un manche), la commande du pas cyclique (via un manche) et la commande anti-pédales (via des pédales ou palonniers) [FAA Helicopter].

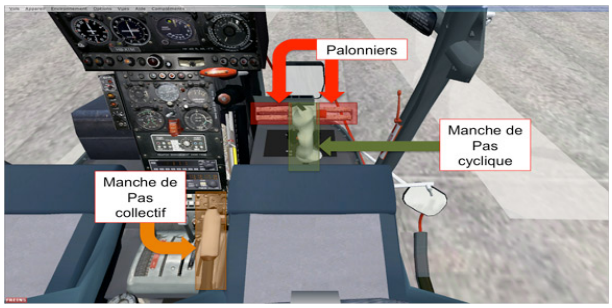


Figure IV.2 : Les trois organes de commandes d'un hélicoptère

On appelle **système de Commande de Vol** (CDV) le système complet de contrôle de trajectoire, complet au sens qu'il comprend tous les composants entre le pilote et les surfaces mobiles. Dans les architectures actuelles de ces systèmes, l'essentiel des composants sont « électriques » (par opposition à « mécanique ») : calculateurs, actionneurs, capteurs, et liaisons entre ces éléments. La composition plus détaillée et les évolutions de ce type de système font l'objet de la section IV.2. Avant cela, nous rappelons quelques propriétés d'un tel système.

IV.1.2 Propriétés d'un système de commande de vol Hélicoptère

Un système de commande de vol (qu'il soit pour un avion ou un hélicoptère) est un système critique, embarqué et temps réel. Rappelons ici brièvement ce qui a été défini dans le chapitre I, section I.1 :

1) Un système « critique » : cela veut dire qu'il est soumis à des exigences très sévères en sûreté de fonctionnement étant données les conséquences graves qui pourraient être causées par des éventuelles défaillances. Dans un tel système, le taux de défaillance doit classiquement être inférieur à 10^{-9}/h. Ces exigences en sûreté de fonctionnement sont imposées par les standards de certification comme l'ARP4754A [ARP4754A 2010] et l'ARP4761 [ARP4761 1996].
2) Un système « embarqué » : la propriété embarquée implique des ressources limitées en mémoires et performances des processeurs, poids et consommation d'énergie et cela implique aussi des communications basées sur des messages de taille limitée.
3) Un système « temps réel » : cela veut dire des contraintes temporelles strictes.

Cependant, pour le système de commande de vol hélicoptère, par rapport à celui d'un avion, les deux premières propriétés ont un impact accru, du fait, notamment, de la différence du principe même de vol, et également de la taille réduite d'un hélicoptère :

- Un hélicoptère ne peut pas planer, alors qu'un avion oui.
- Il n'y a pas (ou extrêmement peu) de redondance des surfaces mobiles, contrairement à un avion, où la perte de certaines surfaces mobiles peut être compensée, tout ou en partie, par l'utilisation des autres surfaces.

La taille réduite accroît les limitations sur les ressources, mais également la proximité des ressources avec les structures mécaniques de vol et la motorisation, avec comme conséquences des vibrations et des variations de températures accrues. En résumé, bien que les CDV présentent certains fondements communs à tous les aéronefs, ils présentent également, certaines différences, parfois très importantes, entre les différentes catégories d'aéronefs, et tout particulièrement entre les avions et les hélicoptères.

IV.2 CDV : ÉTUDE DE L'EXISTANT

Nous détaillons dans cette section une étude de l'existant sur les CDV, et nous commençons par décrire l'historique des architectures CDV allant de celles basées sur une commande mécanique à celles basées sur la technique « Fly By Wire » pour les CDV avion et hélicoptère. Puis, nous présentons les principales architectures de référence pour les systèmes de Commande de Vol Électriques (CDVE) avion et hélicoptère aussi. Ensuite, nous exposons l'architecture de base d'un CDVE hélicoptère. Nous clôturons cette section en décrivant l'architecture du NH90 et ses limitations qui ont incité à la conception d'une nouvelle architecture pour les futurs CDVE d'Airbus Helicopters.

IV.2.1 Historique des CDV : du mécanique au Fly By Wire (FBW)

IV.2.1.1 Le tout mécanique

Dans les débuts de l'aviation, puis durant des décennies, les systèmes de commande de vol étaient basés sur une architecture purement mécanique : les commandes données par le pilote étaient transmises aux « actionneurs » des surfaces mobiles via un ensemble de composants mécaniques (tringles, câbles, etc.) [Traverse *et al.* 2006]. Avec l'évolution de la taille des aéronefs, ces architectures mécaniques sont devenues de plus en plus complexes, lourdes et coûteuses à mettre en œuvre et à maintenir, et ont considérablement augmenté la charge de travail du pilote, à tel point qu'une autre solution était nécessaire, qui a donné naissance aux systèmes de Commande De Vol « Électriques » (CDVE) dans les années 1970.

IV.2.1.2 Le concept de Fly By Wire

Le concept qui consiste à remplacer l'architecture mécanique par une architecture fondée sur l'utilisation de composants électriques et électroniques est appelé **Fly By Wire (FBW)**. Basées sur des calculateurs, des capteurs, des actionneurs et des liens électriques, les architectures FBW offrent un ensemble d'avantages, dont les principaux sont les suivants. En effet, le FBW permet de réduire le volume et la masse du système de commande de vol, et donc faciliter aussi sa maintenance. Il permet également de réduire globalement la charge de travail du pilote, lui facilitant ainsi sa mission. Enfin, le FBW rend possible l'introduction de lois de commande plus complexes, ce qui augmente la manœuvrabilité de l'aéronef (surtout pour les hélicoptères) [Boczar & Hull 2004].

IV.2.1.3 Les évolutions du Fly By Wire dans le monde avion

Bien entendu, la transition vers des architectures FBW s'est faite, et continue à se faire, par étapes. D'une part, tous les éléments mécaniques n'ont pas été remplacés par des composants électriques et électroniques en une seule étape. D'autre part, ces composants, initialement à base de technologies analogiques, s'appuient de plus en plus sur des technologies numériques.

Ainsi, dans les années 1970, le FBW était une technologie analogique. Le premier aéronef commercial qui a mis en œuvre les commandes de vol électriques analogiques est le Concorde [Traverse *et al.* 2006]. Ensuite, le FBW a commencé à intégrer des technologies numériques. Cette évolution n'a d'abord concerné que le cœur des commandes des vols : les calculateurs. Le FBW numérique était initialement limité aux avions militaires. Puis la technologie a été introduite dans les avions commerciaux. L'A320 d'Airbus faisait partie de la première génération d'avions volant avec le FBW numérique. Il a été certifié et mis en service en 1988.

Depuis, d'autres composants ont évolué vers le numérique, dont notamment le remplacement de liens directs entre les calculateurs et les actionneurs, par des réseaux numériques de communication. Aujourd'hui encore, cette mutation n'est pas totalement achevée, et le passage au « tout numérique » est encore une évolution en cours.

IV.2.1.4 Les évolutions du Fly By Wire dans le monde hélicoptère

Avec un décalage de plusieurs années par rapport au monde avion, l'industrie des hélicoptères a également adopté le concept FBW dans sa version analogique. Cet important décalage dans le temps est dû au fait que les principes et la mécanique du vol d'un hélicoptère sont différents de ceux d'un avion.

Comme pour les avions, le FBW analogique a été d'abord introduit dans les hélicoptères militaires. **L'hélicoptère militaire NH90** [Vidal *et al.* 2000] d'Airbus Helicopters a fait son premier vol avec un FBW analogique en Mars 1997. Puis, le FBW a évolué vers une architecture « partiellement numérique ». Avec l'architecture FBW numérique, le pilote n'agit plus sur les actionneurs de rotor (notamment) en « déplaçant » des pédales et des manches mécaniques classiques (illustrés dans la Figure IV.2.). Au lieu de cela, il n'a plus qu'à déplacer des « bâtons électriques » secondaires qui génèrent des signaux électriques à transmettre par des fils aux calculateurs. Ceux derniers calculent et génèrent des signaux électriques de commande à transmettre par des fils aux actionneurs du rotor (notamment). La Figure IV.3 illustre le principe global des évolutions d'un CDV hélicoptère conventionnel vers un CDVE mixte (analogique-numérique du NH90), sans détailler pour l'instant l'architecture elle-même d'un CDV.

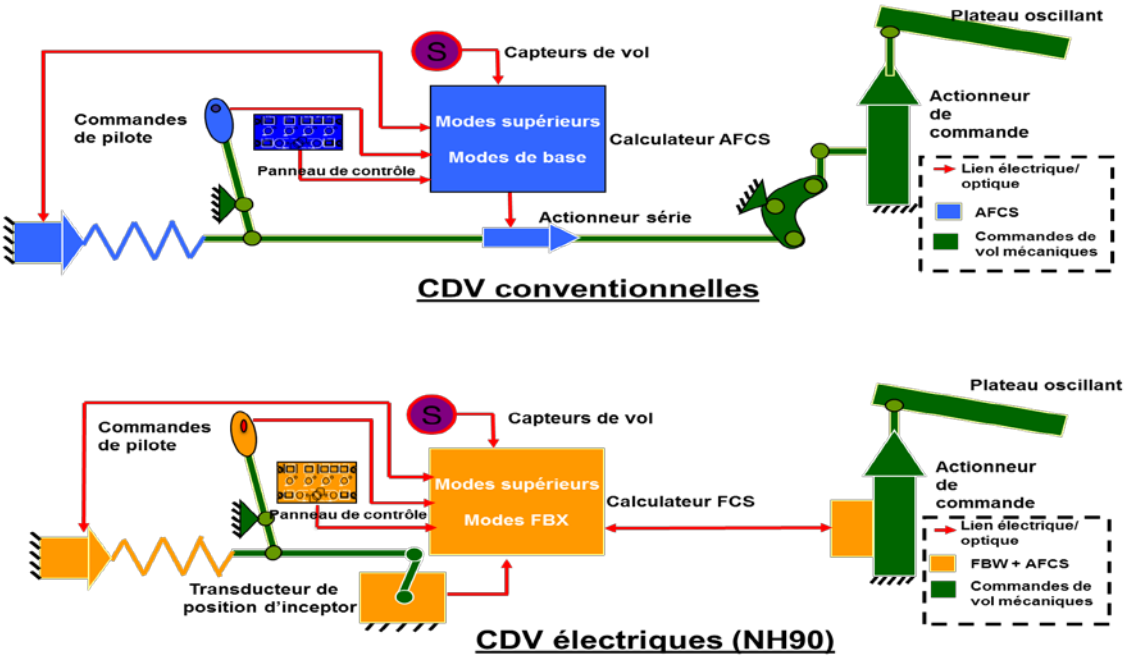


Figure IV.3 : CDV Hélicoptère conventionnelles Versus CDV Hélicoptère électriques (« Full FBW », NH90)

Comme le montre la figure IV.3, le schéma CDV conventionnel décrit bien ce que sont les CDV mécaniques où la commande pilote est liée mécaniquement à l'actionneur mécanique via un actionneur série (communément appelé vérin série). Le schéma intitulé CDV électrique représente l'architecture électrique mixte du NH90 qui a des liens conventionnels pour les axes de pilotage, et des liens électriques entre les commandes pilote et l'actionneur.

À l'heure actuelle, les systèmes de Commandes De Vol Électriques (CDVE) comme celui du NH90 sont basés sur une architecture mixte avec des calculateurs numériques et des calculateurs analogiques. Ces « architectures mixtes » présentent un ensemble de limitations (détaillées dans la section IV.2.4), ce qui a incité à faire évoluer l'architecture des commandes de vol électriques actuelles. Pour les futurs programmes d'Airbus Helicopters, à savoir le X6, sur lequel nous travaillons, l'architecture adoptée est une architecture FBW tout numérique.

IV.2.2 Architectures actuelles de référence des CDVE

Avant de présenter une description détaillée des architectures hélicoptères, il est nécessaire de retracer, dans les grandes lignes, les architectures de commande de vol électriques des avions commerciaux qui ont fortement inspirées celles des hélicoptères. Pour ce faire, nous considérons l'architecture d'Airbus et l'architecture de Boeing, sans décrire ici en détail les architectures des CDVE des différents programmes d'Airbus et Boeing. Nous nous concentrons uniquement sur les philosophies générales d'architecture que ces deux constructeurs ont adoptées.

L'architecture **Boeing** (avions et hélicoptères) est basée sur le concept « **Triplex** », appelé aussi « **Redondance Modulaire Triple** ». Elle comprend trois calculateurs numériques centraux, dont le rôle est le calcul des lois de commande, et quatre calculateurs analogiques pour l'asservissement des actionneurs et la réalisation des conversions numériques-analogiques. Les calculateurs de même type sont identiques du point de vue matériel. Une diversification logicielle est adoptée pour assurer la dissimilarité des calculateurs centraux. C'est une architecture « **Actif/Actif simplex surveillée** » [Yeh 1998] [Yeh 2001]. « Actif/Actif » signifie que la commande est calculée dans toutes les voies, « simplex » signifie un seul canal de calcul (par opposition à duplex) et « surveillé » veut dire que les calculs issus des voies simplex sont échangés et comparés. Les communications numériques sont basées sur l'ARINC 629. Les hélicoptères issus de chez Boeing et Sykorsky adoptent presque la même architecture des systèmes de commande de vol des avions Boeing.

Pour **Airbus**, les architectures adoptées sont plutôt de type « Quadruplex » et « Sextuplex » en mode Actif/Standby (avions civil, comme l'A380). « Standby » signifie que les redondances des calculateurs calculent la loi aussi mais leur calcul n'est pas pris en compte. Présentons la plus complexe : elle comprend six calculateurs numériques, trois de type primaire et trois de type secondaire. Une diversification matérielle entre les calculateurs de type différent est déployée pour assurer en partie la dissimilarité. Pour compléter cette dissimilarité, les calculateurs du même type ont des logiciels différents. Les communications sont basées sur l'ARINC 429. Les hélicoptères d'Airbus (le NH90 à l'heure actuelle) adoptent une architecture Quadruplex mixte (numérique-analogique) inspirée de la philosophie des avions d'Airbus.

Pour assurer le niveau requis en sûreté de fonctionnement, les deux architectures se basent sur la redondance asymétrique (voir chapitre I, section I.3.3.1) et la diversification que ce soit au niveau matériel ou logiciel. Cela permet de réduire le risque du mode commun de défaillance [Traverse *et al.* 2004]. Les principales caractéristiques des deux CDVE avion sont décrites dans le Tableau IV.1.

Boeing	Airbus
Architecture Triplex	Architecture Quadruplex ou Sextuplex
Calculateurs numériques et analogiques	Calculateurs numériques
Communications basées sur l'ARINC 629	Communications basées sur l'ARINC 429

Tableau IV.1 : Principales caractéristiques des CDVE avion Airbus et Boeing

IV.2.3 Architecture de base d'un CDVE hélicoptère

Les CDVE hélicoptère peuvent être décomposés en trois lots fonctionnels principaux :

- **L'interface du pilote** : elle comprend principalement les écrans, le manche cyclique, le manche collectif, le palonnier, des dispositifs de sélection du mode de vol, le dispositif de réglage de la donnée de référence du mode, etc.
- **Le traitement des commandes de vol**, aussi appelé « **Flight Control Processing** » (**FCP**) : il élabore des requêtes de positionnement d'actionneurs, reflétant le contrôle de la trajectoire désirée par le pilote. Le FCP met en œuvre une partie du mode de pilotage manuel (PFCS) et une partie du mode automatique (AFCS) appelé aussi mode supérieur. Dans la pratique, ces deux « modes » ne sont pas distincts, mais sont plutôt distribués et/ou entremêlés. Le traitement des paramètres de vol est également fait par le FCP.
- **L'actionnement d'asservissements**, aussi appelé « **servo actuation** » : il assure que les requêtes du FCP de positionnement des actionneurs sont bien effectuées par les actionneurs. Cet actionnement d'asservissement s'appuie sur :
    - les actionneurs eux-mêmes,
    - le traitement de commande d'actionneur, aussi appelé « Actuators Control Processing » (ACP) lui-même composé de trois parties : la sélection, l'asservissement et la surveillance. Une particularité de l'ACP est qu'il gère deux types d'actionneurs : les actionneurs des surfaces primaires et les actionneurs des surfaces secondaires. Selon les surfaces d'actionnement, le découpage des sous-fonctions de l'ACP est différent.

IV.2.4 Architecture du CDVE du NH90 : description et limites

Comme dit à la section IV.2.2, les architectures des systèmes de commande de vol des hélicoptères d'Airbus suivent la même philosophie adoptée pour les systèmes de commande de vol des avions d'Airbus. Nous décrivons dans cette section l'architecture mise en place dans l'hélicoptère NH90 [Vidal et al. 2000] tout en décrivant les limitations de cette architecture qui ont incité à la conception d'une nouvelle architecture pour les futurs systèmes de commande de vol d'Airbus Helicopters

D'un point de vue matériel, le CDVE du NH 90 est basé sur une architecture Quadruplex et est composé de (Figure IV.4) :

- Deux boîtes de calculateurs de vol (FCC1 et FCC2) dont chacune contient deux calculateurs :
    - un calculateur de vol numérique « Flight Control Digital Computer (FCDC) », comprenant lui-même deux canaux (FCDC1 et FCDC2),
    - et un calculateur de vol analogique (« Primary Flight Control Analog Computer (PFCAC) »), comprenant lui aussi deux canaux (PFCAC1 et PFCAC2).
- Deux boîtes de calculateurs d'actionneurs « Actuators Control Computer (ACC) » dont chacune contient deux calculateurs d'actionneurs analogiques comprenant eux-mêmes, chacun, deux canaux.
- Des contrôles conventionnels (à savoir un manche cyclique central).

Pour ce qui est des communications, elles sont basées :

- sur le bus ARINC 429 (qui assure une transmission duplex) pour les communications inter-calculateurs de vol (FCC), ainsi que pour celles entre les calculateurs de vol (FCC) et les calculateurs des actionneurs (ACC),
- sur des liens analogiques pour les communications entre les ACC.

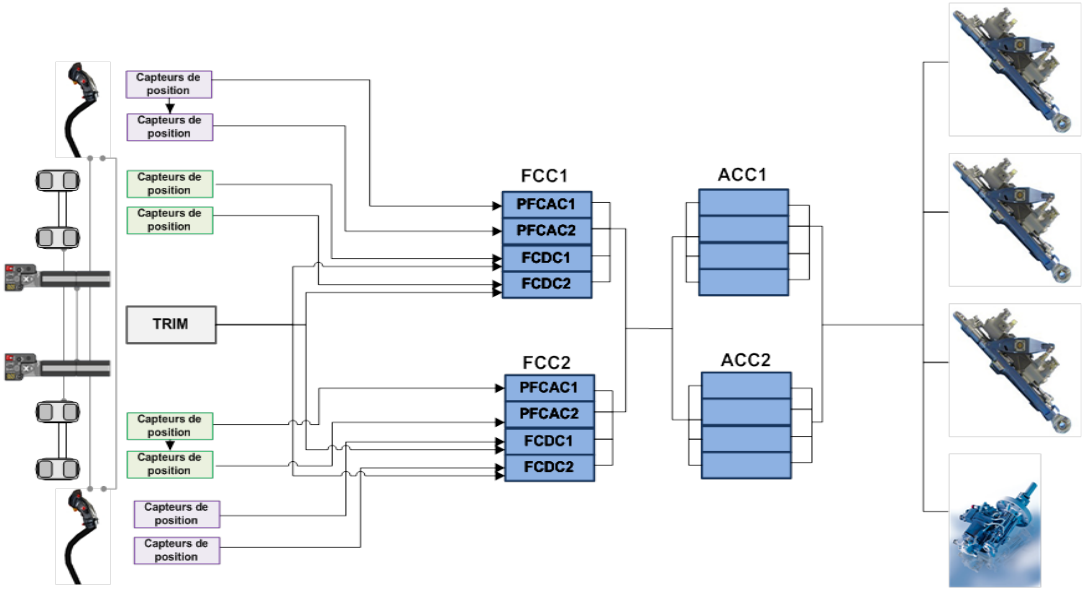


Figure IV.4 : L'architecture du système de commande de vol du NH 90

Les limites de cette architecture actuelle viennent essentiellement de l'hybridation numérique-analogique, non seulement au niveau des traitements, mais aussi des communications.

Ainsi, une première limite, intrinsèque à cette hybridation, est la nécessité de gérer deux technologies différentes. Une deuxième limite est le poids du câblage résultant de cette hybridation ; un objectif des futures architectures est donc d'alléger le poids de câblage.

Cependant, la limite principale qui a incité à l'évolution vers les futures architectures est l'effet falaise « cliff effect » qui résulte de la différence de la qualité de manœuvrabilité, notée HQ pour « Handling Quality » (la qualité de vol) entre les calculateurs numériques « Flight Control Digital Computer » (FCDC) et les calculateurs analogiques « Flight Control Analog Computer » (FCAC). Le facteur « HQ » mesure la facilité de pilotage. Les calculateurs numériques ont HQ = 1 qui signifie un niveau très élevé de finesse de manœuvrabilité (cela a permis au NH90 d'être certifié selon l'ADS33C [Holh *et al.* 1989]), alors que les calculateurs analogiques ont un HQ égal à 2 ou 3 selon les manœuvres, ce qui signifie un niveau plus faible de finesse de manœuvrabilité. La différence entre ces deux niveaux engendre l'effet de falaise.

En définitive, pour résoudre le problème de l'effet falaise et pour alléger le poids de câblage, Airbus Helicopters a décidé de passer à une architecture « tout numérique » pour ses futurs programmes. Nous décrivons dans ce qui suit l'architecture des futurs systèmes de commande de vol d'Airbus Helicopters.

IV.3 ARCHITECTURE DES FUTURS CDVE D'AIRBUS HELICOPTERS

Dans cette section, nous présentons l'architecture des futurs CDVE d'Airbus Helicopters. Cette description adopte le point de vue communications. En effet, l'objectif de cette section n'est pas de présenter en détail l'intégralité de la nouvelle architecture, mais uniquement les aspects en rapport avec les communications, puisque notre contribution s'inscrit dans ce cadre. Nous commençons d'abord par une description générale en nous focalisons sur les communications, puis nous détaillons la politique adoptée par Airbus Helicopters pour assurer l'intégrité des communications intra et inter-calculateurs.

IV.3.1 Description générale

L'architecture de référence du système de commande de vol considérée à la fin de l'étude de faisabilité du projet de nouvelle architecture (c'est la phase qui consiste à prouver que le projet est techniquement faisable et économiquement viable). C'est une architecture qui s'inspire beaucoup de l'architecture du système de commande de vol du NH90 détaillée dans la section précédente. Elle est toujours fondée sur le principe du quadruplex avec comme grande nouveauté le fait de se baser entièrement sur la technologie FBW numérique : cela veut dire une architecture « tout numérique », au sens que **tous les calculateurs et toutes les communications intra et inter-calculateurs sont numériques**.

Les briques de base de cette architecture sont deux types de calculateurs numériques :

- Les calculateurs primaires également appelés calculateurs **PRIM**. Le rôle des PRIM est de calculer les lois évoluées : ce sont les lois de pilotage qui sont valables pour tout le domaine de vol.
- Les calculateurs secondaires également appelés calculateurs **SEC**. Le rôle des SEC est de calculer les lois normales issues du FCDC : ce sont les lois de pilotage qui sont valables pour tout le domaine de vol (C'est grâce à l'utilisation de nouvelles lois et des lois issues du FCDC que tout le domaine de vol est couvert sans effet de falaise).

Qu'il soit primaire ou secondaire, un calculateur comprend toujours deux voies :

- la première voie (notée #a) est communément appelée COM, pour son rôle principal de commande,
- la deuxième voie (notée #b) est communément appelée MON, pour son rôle principal de moniteur (ou surveillance) de la voie COM.

En effet, le principe de base du fonctionnement COM/MON est que les 2 voies effectuent les mêmes traitements dont les résultats sont comparés entre eux pour une validation avant communication avec les autres éléments de l'architecture. Cependant, pour des raisons d'optimisation (par exemple, afin d'équilibrer la charge entre les deux voies en termes de nombre d'entrées et de sorties à gérer par le calculateur), une voie pourrait prendre la main sur une partie des fonctionnalités de l'autre voie (par exemple, la voie MON pourrait assurer des fonctionnalités de commande).

Ces briques de base sont « assemblées » pour assurer les traitements de types FCP et ACP définis dans la section IV.2.3. Un lot de 4 calculateurs, composé de 2 PRIM et de 2 SEC, va assurer les traitements FCP, et les traitements ACP. Par abus de langage, on donnera aux lots de calculateurs le nom des traitements qu'ils réalisent. De ce fait, « FCP », respectivement « ACP », sera utilisé pour désigner indifféremment soit les calculateurs, soit les traitements/fonctions réalisés. De plus, quel que soit le rôle joué par une de ces fonctions, ils contribuent tous aux commandes de vol, et seront donc tous, désormais, considérés comme étant, de manière très générique, des FCC (Flight Control Computer). On parlera donc de **« calculateurs » FCP et ACP**, qui tous les deux peuvent être appelés des **FCC**.

Maintenant, concernant le système de communication FBW des futurs CDVE, ses fondements sont les suivants :

- Une topologie de type maillage total entre tous les éléments communicants avec **des liens numériques de communication simplex rebouclés**, et chaque lien assure une transmission sûre et sécurisée unidirectionnelle.
- Les **interfaces matérielles** sont basées sur la norme **EIA-485** qui a été choisie pour plusieurs raisons, à savoir : la simplicité de l'implémentation, le haut débit, la faible latence et la facilité de certification.
- Le **protocole** utilisé est issu de la norme **HDLC**.
- Pour toute communication, le débit des liens est fixé à une valeur unique comprise entre 15 et 20 Mbps.

Pour des raisons de sûreté de fonctionnement, chaque FCP ou ACP doit être en mesure de transmettre des données à tous les autres FCP / ACP. C'est pour cette raison que l'architecture de communication globale est basée sur une topologie de type maillage total. De plus, ce choix de la topologie est renforcé par un besoin d'augmentation de la disponibilité du système. Ainsi, lorsqu'un FCP ou un ACP émetteur ou récepteur défaille, il n'a pas d'impact sur la communication entre tous les autres FCP et ACP valides.

Enfin, en rappelant que toutes les liaisons sont unidirectionnelles, l'architecture complète d'interconnexion est représentée sur la Figure IV.5 et sur la Figure IV.6, chacune représentant une partie de cette architecture (la représentation sur une seule figure n'aurait pas été lisible).

La Figure IV.5 illustre tous les liens de communication sortant de tous les calculateurs FCP, d'une part, vers les autres calculateurs FCP, et d'autre part, vers tous les calculateurs ACP. Les liens dessinés en pointillés illustrent le sous-ensemble de liens sortant d'un seul calculateur (le FCP PRIM 1). La Figure IV.6 illustre tous les liens de communication sortant de tous les calculateurs ACP, d'une part, vers les autres calculateurs ACP, et d'autre part, vers tous les calculateurs FCP. Les liens dessinés en pointillés illustrent le sous-ensemble de liens sortant d'un seul calculateur (l'ACP PRIM 1).

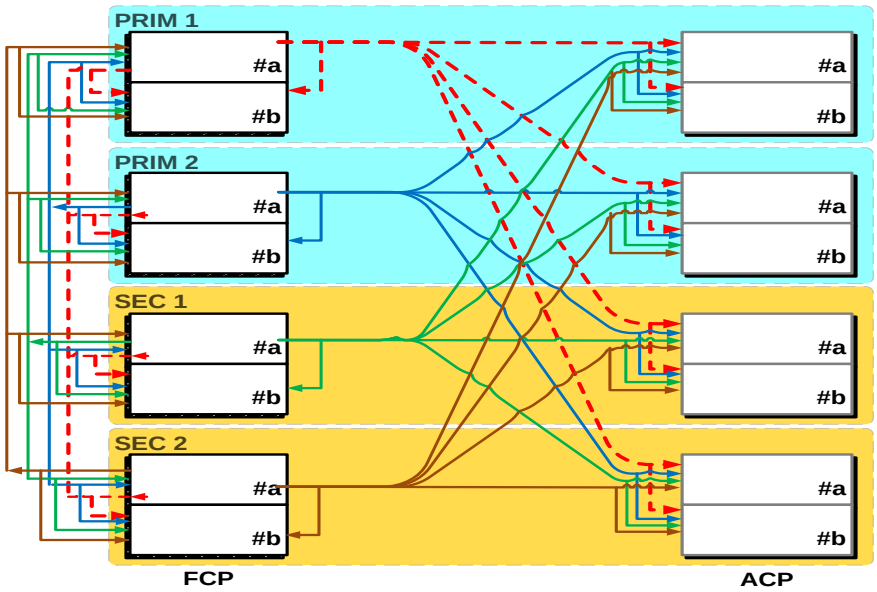


Figure IV.5 : Les liens de communication FCP vers FCP et FCP vers ACP

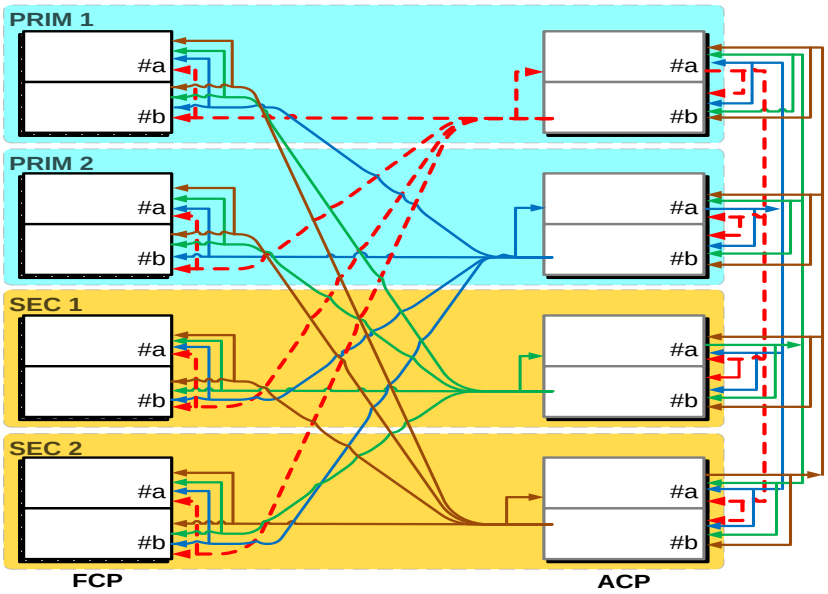


Figure IV.6 : Les liens de communication ACP vers ACP et ACP vers FCP

Donc si on fait un calcul sur l'ensemble du FCS, nous avons au total :

- Côté FCP :
  - 24 liens sortants
  - 24 liens entrants
  - 1 lien interne entrant dans chaque FCC
  - 1 lien interne sortant dans chaque FCC
- Côté ACP :
  - 12 liens sortants
  - 12 liens entrants

Donc, en liens physiques, le FCS comprend 48 liens côté FCP et 24 liens coté ACP.

Les éléments de communications de la nouvelle architecture ayant été maintenant présentés, nous pouvons décrire maintenant la politique d'intégrité qui a été prévue.

IV.3.2 Politique d'intégrité des communications

IV.3.2.1 Cadre général

La figure IV.7 donne le format des trames qui seront utilisées. Pour assurer l'intégrité des données, les communications des futurs CDVE d'Airbus Helicopters seront protégées par deux fonctions de contrôle, qui seront des codes de détection d'erreurs. Ce sont le « FCS » et le « CRC » dans la Figure IV.7.

Délimiteur	Adresse	Contrôle	Charge utile + CRC	FCS	Délimiteur
1 octet	*2 octets*	*1 octet*	*36 octets (32 de charge utile et 4 de CRC)*	*4 octets*	*1 octet*

Figure IV.7 : Format de la trame EIA-485

La première fonction de contrôle sert à protéger toute la trame (elle couvre donc tous les champs de la trame) et c'est un code CRC à **implémenter dans la couche liaison de données**. Ce code, noté FCS est un CRC d'une taille de 4 octets, qui est celui du protocole HDLC, ayant pour polynôme générateur :

$$G(x) = x^0 + x^1 + x^2 + x^4 + x^5 + x^7 + x^8 + x^{10} + x^{11} + x^{12} + x^{16} + x^{22} + x^{23} + x^{26} + x^{32}$$

La deuxième fonction de contrôle sert à protéger exclusivement la charge utile (données utiles). Elle est à **implémenter** dans la **couche applicative** pour assurer une intégrité de bout en bout.

C'est sur ce deuxième code détecteur (applicatif), et plus spécifiquement sur la problématique de sa sélection, que porte l'objectif de notre collaboration avec Airbus Helicopters.

Nous prenons, à titre d'exemple, un code CRC, également de taille 4 octets. Ce n'est qu'un exemple, le choix définitif pour ce code détecteur ne sera pas nécessairement un CRC, mais pourrait aussi être un code Fletcher ou de Adler, ou autre.

Nous décrivons dans les deux sous-sections suivantes le processus de détection d'erreurs avec les deux codes détecteurs d'erreurs CRC et FCS dans les communications :

- Internes aux calculateurs FCP (intra-calculateurs FCP ou inter-voies),
- Externes aux calculateurs FCP et ACP (inter-calculateurs).

Nous précisons que nous nous intéressons uniquement aux processus de détection d'erreurs. On ne s'intéresse pas aux mécanismes de « recouvrement » d'erreur en cas de détection d'erreurs. Nous ne nous intéressons pas non plus à l'utilisation faite ultérieurement par un FCP ou un ACP de la charge utile (qui sera noté Pi).

IV.3.2.2 Processus de détection d'erreurs dans les communications internes

Les communications internes correspondent aux communications entre les deux voies #a et #b d'un même calculateur.
Les notations et conventions que nous prenons à partir d'ici sont les suivantes :

- les boîtes P1 bleu et P1 rouge représentent les charges utiles, respectivement de la voie #a et de la voie #b,
- les boîtes E1 bleu et E1 rouge représentent le CRC des charges utiles, respectivement de la voie #a et de la voie #b,
- les boîtes C1 bleu et C1 rouge représentent le champ de contrôle de toute la trame (c'est le FCS défini plus haut), respectivement de la voie #a et de la voie #b,
- le gestionnaire des Entrées/Sorties est noté « gestionnaire d'E/S ».

A l'étape 1 (Figure IV.8), le cœur de traitement de chaque voie *i* (c'est-à-dire #a et #b) élabore la charge utile (Pi), calcule le CRC (Ei) correspondant, et envoie le tout (Pi et Ei) à son gestionnaire d'E/S. A l'étape 2 (Figure IV.9), chaque gestionnaire d'E/S élabore la trame complète à émettre, en y incluant le champ de contrôle (Ci) de toute la trame. Puis, la voie #a envoie sa trame ainsi construite sur le lien inter-voies (intra-calculateurs) dédié.

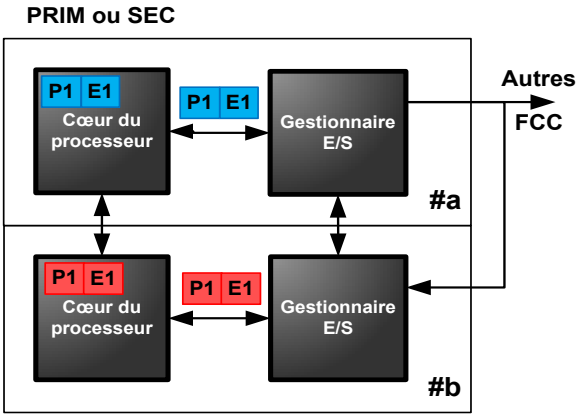


Figure IV.8 : Étape 1 du processus interne de détection d'erreurs

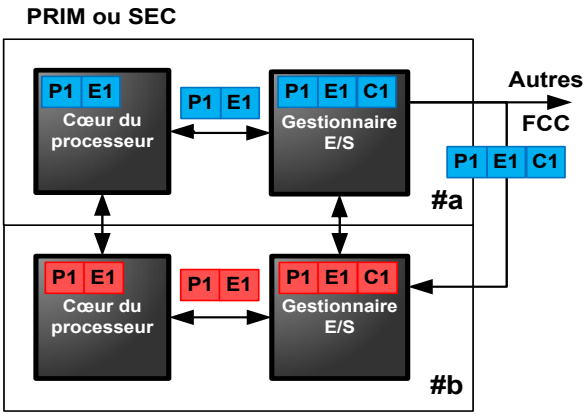


Figure IV.9 : Étape 2 du processus interne de détection d'erreurs

Dans l'étape 3 (Figure IV.10), la trame émise par la voie #a est reçue sur la voie #b, dont le gestionnaire d'E/S réalise une première vérification de l'intégrité de la trame reçue en vérifiant le Ci reçu (C1 bleu). Puis, dans l'étape 4 (Figure IV.11), ce même gestionnaire E/S de la voie #b fait une seconde vérification d'intégrité, cette fois en comparant le Ci qu'il avait lui-même établi et le Ci reçu de la voie #a (comparaison des C1 rouge et bleu). Puis, il retire ce Ci reçu et transmet vers son cœur de traitement la charge utile et le CRC associé qu'il a reçu de la voie #a (envoi de P1 et E1 bleu).

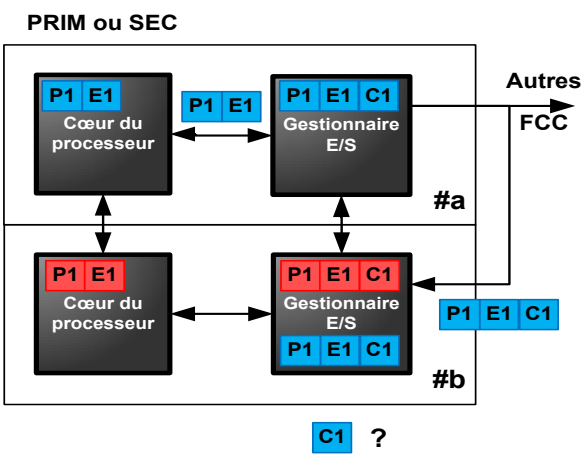


Figure IV.10 : Étape 3 du processus interne de détection d'erreurs

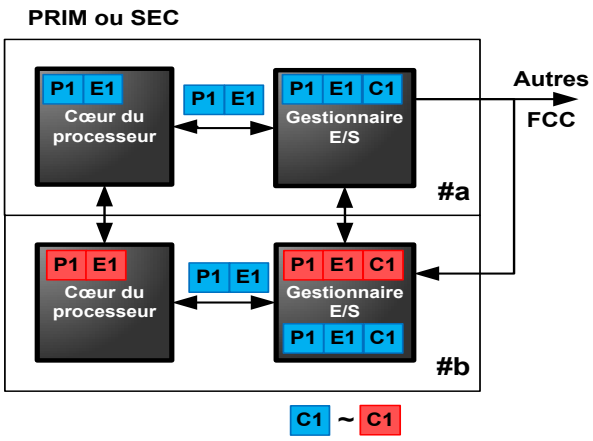


Figure IV.11 : Étape 4 du processus interne de détection d'erreurs

A l'étape 5 (Figure IV.12), quand le cœur de traitement de la voie #b reçoit la charge utile et le CRC de la voie #a (P1 et E1 bleu), il vérifie l'intégrité de la charge utile en comparant les deux Ei : celui qui a été émis par la voie #a (E1 bleu) et celui que lui-même avait calculé (E1 rouge). Et enfin, il élimine ce Ei pour ne garder que la charge utile Pi.

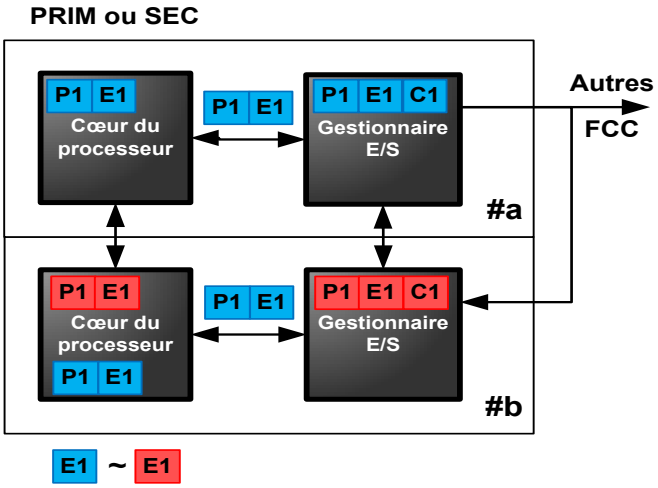


Figure IV.12 : Étape 5 du processus interne de détection d'erreurs

Ce processus de détection d'erreurs assure, en fait, à la fois l'intégrité des données transmises d'une voie à l'autre au sein d'un même calculateur et l'intégrité de la fonction de calcul grâce à la comparaison entre les CRCs (Ei) calculés sur les deux voies.

IV.3.2.3 Processus de détection d'erreurs dans les communications externes

Les communications externes correspondent aux communications entre calculateurs, quel que soit le type de ces calculateurs. Comme pour les communications internes, deux niveaux d'intégrité (Ci et Ei) sont considérés. Rappelons qu'on a 4 types possibles de communications, puisque tous les FCC (FCP et ACP) communiquent entre eux.

On peut regrouper ces échanges selon deux points de vue, qui seront illustrés par la Figure IV.13 et la Figure IV.14 :

1) des FCP vers les autres FCP et vers tous les ACP,
2) des ACP vers les autres ACP et vers tous les FCP.

Donc, du point de vue des communications sortantes des FCP, le processus est résumé sur la Figure IV.13. Un FCP envoie vers les 2 voies #a et #b de tous les autres FCC une trame contenant Pi, Ei et Ci (boites bleus sur la Figure IV.13). Dans chaque FCC qui reçoit cette trame, le gestionnaire d'E/S de chacune de ses 2 voies effectue une vérification sur le Ci reçu : il compare le Ci reçu et le Ci qu'il recalcule à partir de la trame reçue. Puis, il retire ce Ci, et transmet à son cœur de traitement, qui applique le même principe de vérification, mais cette fois sur Ei.

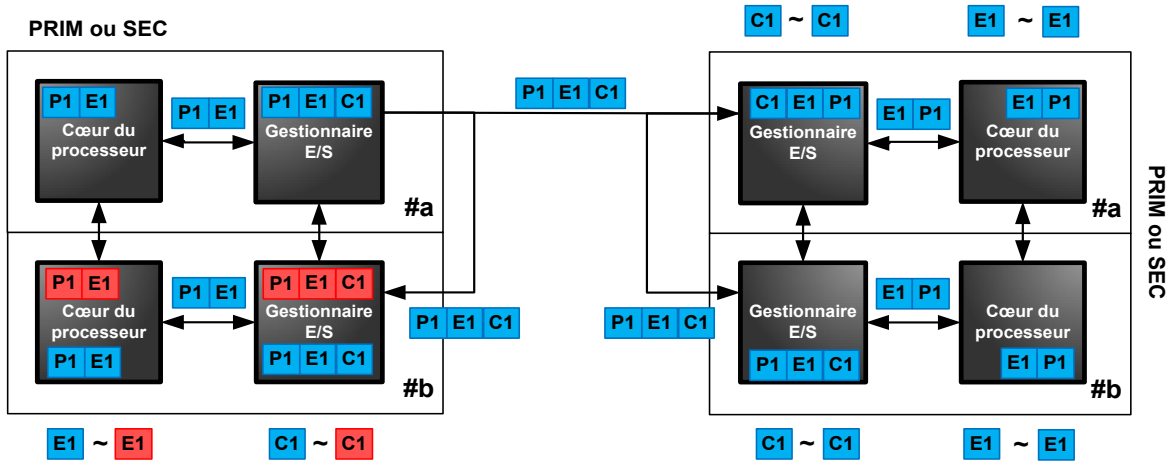


Figure IV.13 : Détection d'erreurs dans les communications inter-calculateurs

Du point de vue des communications sortantes des ACP, le processus est résumé sur la Figure IV.14. Le principe de la chaîne de transmission reste le même que précédemment. Étant donné que les FCP et les ACP sont différents, les FCP sont en Actif/Standby et les ACP sont en Actif/Actif, une information supplémentaire de validité (appelée Val) est ajoutée pour renforcer la vérification d'intégrité entre les deux ensembles (FCP et ACP). Cette information supplémentaire permet aussi de confirmer l'intégrité de l'information par la voie opposée (et aussi pour connaitre le niveau de la validité de l'ACP pour l'ensemble des FCP).

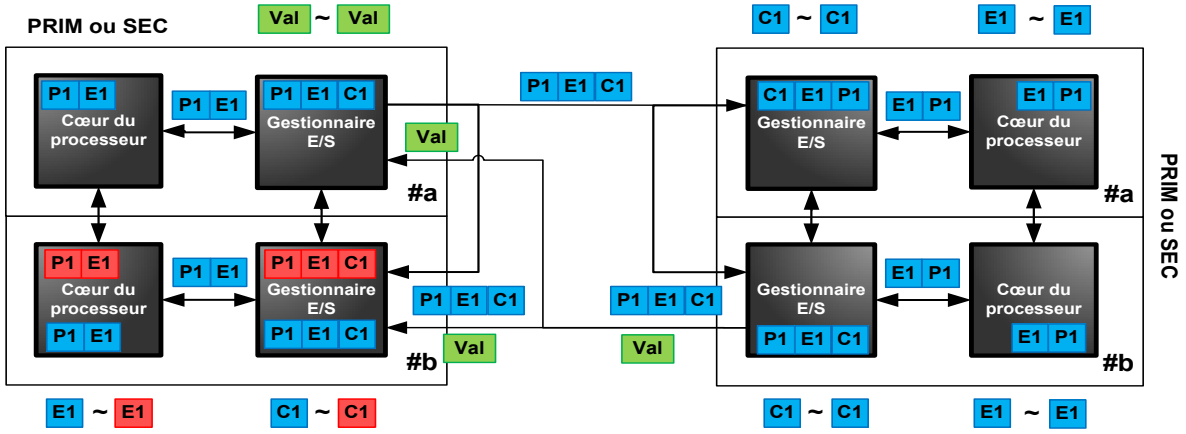


Figure IV.14 : Détection d'erreurs dans les communications externes

IV.4 ENVIRONNEMENT D'EXPÉRIMENTATION

Dans cette section, nous présentons tous les éléments de l'environnement que nous avons utilisé pour nos expérimentations. Il s'agit d'une partie de l'environnement mis en place par Airbus Helicopters pour ses premières études de mise en œuvre de l'architecture présentée à la section précédente. Précisons que les éléments retenus à ce stade ne seront pas forcément ceux qui seront utilisés lors de mise en production des futurs hélicoptères (pour des raisons d'échelle de temps, et parce que tous les choix, dépendant de la vitesse d'évolution des cœurs processeurs, ne sont pas encore définitifs). Ainsi, dans cette section, nous abordons successivement le choix des processeurs, la présentation des deux cartes d'évaluation et le type de plateforme d'expérimentation utilisée.

IV.4.1 Choix des processeurs

Les processeurs sont les composants clés d'une architecture. Rappelons que le chapitre I a montré que, dans les systèmes critiques, il est primordial de réduire le risque du mode commun de défaillance, et que pour ce faire, on diversifie au maximum les composants tout en prenant en compte l'impact de cette diversification sur le coût et donc les contraintes économiques. C'est pourquoi il est nécessaire de choisir deux processeurs présentant le plus possible de dissimilarité. De plus, ces deux processeurs doivent répondre à un certain nombre de critères : fonctionnels, et non fonctionnels (liés à la sûreté de fonctionnement). Il faut également tenir compte de la gestion de l'obsolescence des processeurs. Il faut ensuite identifier des processeurs répondant à ces critères, puis en sélectionner deux. Enfin, il faut étudier les cartes qui permettent de les évaluer.

IV.4.1.1 Critères fonctionnels

Pour satisfaire les aspects **fonctionnels**, le processeur doit disposer de ressources suffisamment importantes. En premier lieu, pour être capable d'exécuter l'application de commande de vol en un temps inférieur au temps de cycle de rafraîchissement des lois de commande (10 ms). En fait, à cette contrainte naturelle de temps réel, s'ajoute une autre contrainte : l'obligation de garder une marge de temps suffisamment grande pour intégrer des évolutions futures. Donc, le temps d'exécution doit être largement inférieur à un temps de cycle. Cela concerne les capacités de traitement et la fréquence de fonctionnement du processeur, le dimensionnement des mémoires internes (caches, RAM et mémoires de masse Flash). En second lieu, le processeur doit aussi disposer de ressources de communications numériques, et supportant un débit compatible avec la taille, le type et la fréquence de données à échanger.

Précisons ici qu'il faut aussi être très vigilant sur la consommation électrique, du fait des limitations des sources d'énergie, mais également du fait des conséquences directes sur la dissipation thermique. Et ce, d'autant plus que, d'un autre côté, le processeur doit avoir la capacité de fonctionner à des températures extrêmes.

IV.4.1.2 Critères non-fonctionnels

Sur le plan **non-**fonctionnel (sûreté de fonctionnement), le processeur doit avoir des capacités à traiter (par détection/correction) les altérations des mémoires **RAM internes** induites par les rayonnements cosmiques. Il doit aussi disposer d'une unité de gestion mémoire capable, d'une part, d'effectuer le même type de traitements sur des mémoires **RAM externes** et d'autre part, d'offrir une solution de partitionnement spatial.

Enfin, il serait intéressant de pouvoir bénéficier d'un « **crédit de certification** » afin de réduire la quantité de justifications à apporter pour la certification des calculateurs. Cela peut être satisfait assez facilement, puisque aujourd'hui la plupart des fabricants de processeurs s'engage à produire des composants compatibles avec des développements de systèmes critiques devant répondre aux normes IEC 61508 et ISO 26262. A titre d'illustration, nous pouvons citer le programme *SafeAssure*[1] de Freescale dans le cadre duquel Freescale propose des solutions permettant d'alléger la phase de certification.

IV.4.1.3 Obsolescence des composants

Le traitement de l'**obsolescence** des processeurs est très important dans le cas des systèmes dont la durée de vie est de plusieurs dizaines d'années, comme c'est le cas pour de aéronefs, tant au niveau de leur production que sur leur vie opérationnelle. Cela passe par un engagement du fabricant à fournir le même processeur sur une longue période et une gestion des stocks appropriée chez l'avionneur. Tout comme sur le plan du crédit de certification, il y a de plus en plus de fabricants qui s'engagent sur une longue durée de vie de certains processeurs. Ainsi, Freescale propose deux programmes de longévité de produits (« *Product Longevity Program*[2]») : un sur 10 ans et un sur 15 ans. Le programme sur 15 ans est plus particulièrement dédié aux domaines de l'automobile, du médical et des télécommunications.

IV.4.1.4 Les processeurs candidats et retenus

Durant la phase de sélection des processeurs, quatre processeurs ont été analysés par Airbus Helicopters : trois de chez Freescale (**MPC8548**[3]**, P2020**[4] **et P5020**[5]) et un de chez Texas Instrument (**TMS320C6657**[6]). Globalement, les quatre processeurs remplissaient les critères de sélection, mais un compromis entre ces critères a été recherché sur les trois processeurs Freescale. Le processeur **MPC8548** a été rapidement écarté car il ne fait plus partie du programme de longévité de Freescale (mentionné plus haut). Il restait donc à finaliser le choix entre le processeur P2020 (processeur 32 bits fonctionnant à 1 Ghz) et le processeur P5020 (processeur 64 bits fonctionnant à 2 Ghz). Le processeur **P5020** est globalement 4 fois plus puissant que le processeur P2020, mais sa consommation électrique (et donc sa dissipation thermique) est aussi 4 fois plus importante (30 W au lieu de 8 W pour le P2020), et il a été finalement écarté vu que les capacités du P2020 devraient suffire. Les processeurs retenus sont donc les suivants, et seront détaillés dans les deux prochaines sous-section :

1) le TMS320C6657 de Texas Instrument,
2) le P2020 de Freescale.

---

[1] http://www.freescale.com/SafeAssure

[2] http://www.freescale.com/pages/product-longevity-archived:LONGEVITY-ARCHIVED

[3] http://www.freescale.com/products/power-architecture-processors/powerquicc-processors/powerquicc-iii-85xx/powerquicc-iii-processor-with-ddr2-pci-pci-express-serial-rapidio-serdes-1-gb-ethernet-security:MPC8548E

[4] http://www.freescale.com/webapp/sps/site/prod_summary.jsp?code=P2020

[5] http://www.freescale.com/products/power-architecture-processors/qoriq-power-architecture-processors/qoriq-p5020-10-64-bit-single-and-dual-core-communications-processors:P5020

[6] http://www.ti.com/product/TMS320C6657/description

IV.4.1.5 Caractéristiques détaillées des TMS320C6657 et P2020

Le processeur P2020[7] de Freescale est un processeur 32 bits de la série P2 de la famille QorIQ à architecture POWER. C'est un processeur bi-cœur basé sur le noyau « core E500v2 » cadencé à 1Ghz. Il est muni, pour chaque cœur, d'un cache de niveau 1 (L1) de 32 Ko pour les données et 32 Ko pour les instructions), ainsi que d'un cache, partagé entre les deux cœurs, de niveau 2 (L2) de 512 Ko et configurable en SRAM. Par contre, il ne dispose pas de mémoire de masse (Flash). Le cache L1 est protégé par un code à parité tandis que le cache L2 est protégé par un mécanisme de correction d'erreur (ECC). Le processeur dispose d'une gestionnaire de mémoire permettant, d'une part, d'assurer un partionnement spatial entre zones mémoires, et d'autre part, de connecter des mémoires RAM externes dotés du mécanisme ECC (Error Correcting Codes).

Le processeur TMS320C6657[8] de chez Texas Instruments est un processeur de la série C6000 au sein de la famille TMS320 à architecture RISC. Il a globalement des caractéristiques similaires à celles du P2020. Il est basé sur un noyau C6600 et, par rapport au P2020, il possède des caches L2 de taille plus importante (1024 Ko propre à chaque cœur). Toutefois, une différence notable, par rapport au P2020, concerne la présence d'une mémoire partagée interne de 1024 Ko, configurable en cache de niveau 3 (L3).

Le Tableau IV.2. résume les principales caractéristiques des deux processeurs retenus.

	P2020	**TMS320C6657**
Consommation/Dissipation thermique	Basse	Basse
Plage de fonctionnement étendue	Oui	Oui
Bus E/S	Oui	Oui
Cœur	E500v2	C6600
Nombre de cœurs	2	2
Fréquence cœur	1000/1200 Mhz	1000 Mhz
Cache L1	32 Ko instructions 32Ko CB data	32 Ko instructions 32 Ko CB data
Cache L2/SRAM	512 Ko unifié WT	1024 Ko unifié CB
Cache L3/SRAM	N/A	1024 Ko unifié CB /oui
Taille cache L2	512 Ko	2*1024 Ko
Taille cache L3	NA	1024 Ko
Détection/correction d'erreur	- Parité sur cache L1 - ECC sur cache L2	- Parité sur cache L1 - ECC sur caches L2 et L3

Tableau IV.2 : Récapitulatif des propriétés des deux processeurs P2020 et TMS320C6657 selon les critères de sélection

[7] http://www.freescale.com/products/power-architecture-processors/qoriq-power-architecture-processors/qoriq-p2020-10-single-and-dual-core-communications-processors:P2020

[8] http://www.ti.com/product/tms320c6657

IV.4.2 Cartes d'évaluation

Pour pouvoir évaluer les performances des processeurs P2020 et TMS320C6657 et vérifier leur adéquation aux besoins de l'architecture CDVE, Airbus Helicopters a fait l'acquisition de cartes d'évaluation pour ces deux processeurs. Les cartes d'évaluation entourent les processeurs de divers circuits périphériques permettant de tester les processeurs dans différents contextes.

Pour le processeur P2020, Freescale propose la carte P2020 DS (P2020 Development System)[9]. Pour le processeur TMS320C6657, Texas Instruments propose la carte TMDSEVM6657 (TMS320C6657 Lite Evaluation Modules)[10]. Pour ce qui nous concerne, nous noterons que les deux cartes permettent de disposer d'une mémoire RAM supplémentaire de 2Go dotée du mécanisme ECC.

Les deux cartes d'évaluation sont présentées sur la Figure IV.15

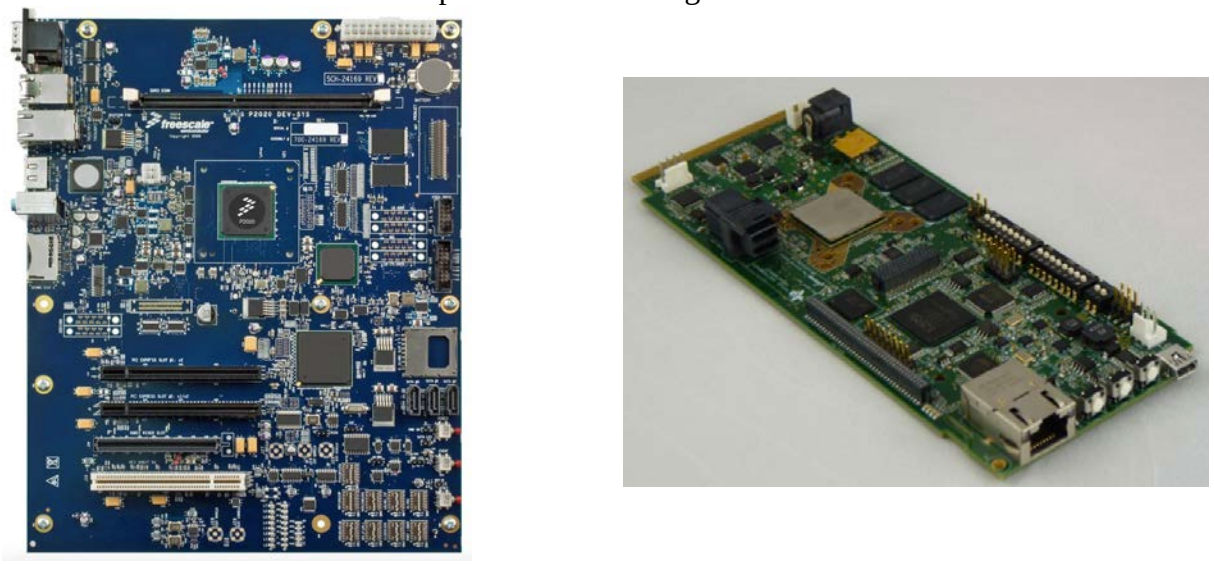


Figure IV.15 : Photos des deux cartes d'évaluation

Les caractéristiques principales des cartes d'évaluation sont résumées dans le Tableau IV.3

Caractéristiques	Carte pour P2020	Carte pour TMS320C6657
RAM	DDR SDRAM 2 Go / ECC	DDR SDRAM 2 Go / ECC
Fréquence carte	500/600 MHz	600 Mhz
Fréquence DDR	400/667 Mhz	667 MHz

Tableau IV.3 : Récapitulatif des caractéristiques des cartes d'évaluation pour les processeurs P2020 et TMS320C6657

[9] http://www.freescale.com/products/power-architecture-processors/qoriq-power-architecture-processors/p2020-development-system:P2020DS

[10] http://www.ti.com/tool/TMDSEVM6657

IV.4.3 Description de la plateforme d'expérimentation

La Figure IV.16 décrit la composition type de la plateforme d'expérimentation sur laquelle nous avons fait nos expérimentations.
Une plateforme d'expérimentation comprend deux composantes (Figure IV.16) : une composante consistant en un ordinateur de développement et une composante consistant en une des cartes d'évaluation présentées à la section précédentes. Les deux composantes sont reliées par une sonde JTAG USB qui sert d'interface entre la carte d'évaluation et l'ordinateur.

Figure IV.16 : Plateforme d'expérimentations (cas du P2020)

L'ordinateur comporte des environnements de développement croisé permettant de produire un code binaire pour la cible choisie à partir de programmes écrits en langage C. Le code binaire peut être ensuite téléchargé sur la cible via la sonde JTAG. L'ordinateur est également muni d'un logiciel servant à piloter les expérimentations et collecter les résultats.

En effet, pour exploiter une carte d'évaluation, il faut nécessairement un environnement de développement associé. Freescale propose deux solutions : une solution libre basée sur Linux (compilateur GCC[11]) et une solution commerciale basée sur CodeWarrior Development Studio[12] .
Texas Instruments propose aussi une solution libre (SYS/BIOS and Linux Multicore Software Development Kits - MCSDK[13]) et une solution commerciale basée sur Code Composer Studio (Code Composer Studio[14] - CCS Integrated Development Environment (IDE)).

D'un point de vue général, on pourrait penser que les solutions libres sont bien adaptées pour des phases de pré-développement, mais que pour des phases de développement définitives, les solutions commerciales sont à privilégier (surtout que les solutions libres ne disposent pas d'un support aussi pointu que celui des solutions commerciales), d'une part pour disposer d'une diversification au niveau des compilateurs (les deux solutions libre sont basées linux et donc sur un compilateur gcc) et d'autre part, pour pourvoir plus facilement bénéficier de crédits de certification.

Dans le cadre de nos expérimentations, les deux cartes d'évaluation sont configurées dans un mode dans lequel un seul cœur est utilisé (la configuration adoptée par Airbus Helicopters en exploitation normale) et le cache L2 est configuré en mode SRAM permettant d'obtenir les meilleures performances temporelles possibles.

---

[11] http://www.freescale.com/tools/embedded-software-and-tools/run-time-software/linux-sdk/linux-sdk-for-qoriq-processors-v1.8:SDKLINUX

[12] http://www.freescale.com/tools/embedded-software-and-tools/software-development-tools/codewarrior-development-tools/suite-for-networked-applications/codewarrior-development-studio-for-power-architecture-technology-eclipse-v10.5:CW-PA

[13] http://www.ti.com/tool/bioslinuxmcsdk

[14] http://www.ti.com/tool/ccstudio

IV.5 CODE DÉTECTEUR BOUT EN BOUT POUR LES FUTURS CDV D'AIRBUS HELICOPTERS

Cette section détaille les séries d'expérimentations réalisées et les résultats obtenus sur les deux cartes d'évaluation d'Airbus Helicopters présentées dans la section précédente. Le but de ces expérimentations est d'évaluer un ensemble de codes détecteurs (les codes déjà présentés dans le chapitre précédent) afin d'en choisir un à utiliser pour assurer l'intégrité des communications de bout en bout (voir la section IV.3.2). L'évaluation de ces codes sert à voir si ces derniers répondent aux exigences des futurs CDVE d'Airbus Helicopters. **Les expérimentations menées portent sur les coûts de calcul (temps d'exécution) des codes** puisque **les pouvoirs de détection ont été évalués dans le chapitre précédent et ce critère ne dépend pas de l'environnement matériel.**

Nous décrivons d'abord les hypothèses générales et la méthodologie retenues, puis les évaluations faites pour le choix de la longueur (en nombre de bits) des codes détecteurs. Puis, nous présentons les différentes étapes de toutes les autres évaluations : des évaluations « unitaires » qui ciblent les codes seuls jusqu'aux évaluations des codes intégrés dans l'application finale ciblée.

IV.5.1 Hypothèses et méthodologie des expérimentations

IV.5.1.1 Codes détecteurs évalués

Rappelons qu'un CDVE hélicoptère est à la fois un système critique, donc soumis à des exigences extrêmes de sûreté de fonctionnement, et un système embarqué et temps réel, donc disposant de ressources de calcul limitées et soumis à des contraintes temporelles strictes. De ce fait, le choix d'un code détecteur est nécessairement un compromis entre :

- son pouvoir de détection, qui doit être à la hauteur du niveau d'intégrité requis,
- et son temps d'exécution, qui doit être acceptable.

La problématique de sélection de code et l'analyse que nous avons exposées dans le chapitre III, nous ont conduits à limiter notre cadre de sélection aux trois types de codes : **CRC**, **Adler** et **Fletcher**. Pour chacun de ces 3 types, il existe différentes longueurs, exprimées en nombre de bits. Comme pour le chapitre précédent, nous avons retenu les longueurs les plus usuelles : 8, 16 et 32 bits.

IV.5.1.2 Métrique des expérimentations : le temps d'exécution

Pour guider le choix des codes, la principale **métrique utilisée au niveau des expérimentations** réalisées s'appuie sur des mesures de temps d'exécution de chacune des implémentations. Comme nos expérimentations ne peuvent pas être totalement exhaustives, nous ne pouvons pas mesurer le pire temps d'exécution, noté WCET (Worst Case Execution Time), qui est une métrique couramment utilisée et calculée usuellement selon des méthodes statiques ce qui est exigé par les standards de certification. Cependant, le nombre de nos cas de test par expérimentation est suffisamment grand pour donner une bonne approche d'un pire temps d'exécution. Concrètement, notre métrique est « le **pire temps d'exécution mesuré »**, que nous notons désormais **WCMT** (Worst Case Measured Time), et qui peut être vu comme un **WCET relatif**. Par ailleurs, pour chaque implémentation et pour chaque expérimentation, le nombre de cas de test a été de quelques centaines à quelques milliers.

De plus, rappelons que le temps d'un cycle de rafraichissement des commandes dans les CDVE hélicoptère est égal à 10 ms. Donc, pour être directement comparable à cette contrainte temps réel du système visé, l'unité de temps sera exprimée, selon le cas, en milliseconde (ms) ou microseconde (µs), alors que, dans le chapitre III, l'unité de temps était le nombre de « cycles d'horloge ».

IV.5.1.3 Algorithmes et langage d'implémentation

Généralement, pour un type de code détecteur, il faut prendre en compte l'existence :

- de plusieurs algorithmes pour l'implémenter,
- de plusieurs formes de programmation d'un même algorithme (dans un même langage) : les règles de programmation varient selon les domaines, par exemple dans l'avionique, certaines formes sont interdites ou imposées, le code doit être linéaire sans branches prédictives (sans boucles de type « Si Alors Sinon ») afin d'éviter de rentrer dans des boucles ou dans des branches non voulues. Et ce, pour répondre à des contraintes spécifiques du domaine.

Pour nos expérimentations :

- les algorithmes utilisés ont déjà été présentés dans le chapitre III, section III.3.2.1
- toutes les implémentations sont réalisées en langage C,
- toutes les implémentations sont issues des outils présentés dans le chapitre III, section III.3.2.1 et sont utilisées :
    - soit directement, c'est-à-dire réutilisées telles que nous les avons développées,
    - soit indirectement, c'est-à-dire qu'elles ont été « recodées» par Airbus Helicopters, pour être conformes à leurs règles de programmation.

IV.5.1.4 Démarche, Méthodologie pour les expérimentations

Les expérimentations sont faites à l'identique sur le P2020 et le TMS320C6657, et nous avons procédé par étapes progressives. Les premières séries d'expérimentations ont pour but d'évaluer :

- l'impact de la longueur d'un code sur le temps d'exécution,
- les performances temporelles (temps d'exécution), cette fois sur les deux processeurs P2020 et TMS320C6657 (via les deux cartes d'évaluation présentées précédemment), des mêmes programmes précédemment évalués sur les plateformes du LAAS-CNRS.

Dans les deux cas, **on évalue les codes détecteurs seuls** : les programmes ne comportent que ces implémentations des codes détecteurs, sans les intégrer dans aucune fonction de l'application cible.

Les séries suivantes d'expérimentations ont pour but **d'évaluer les codes détecteurs en les intégrant dans l'application de CDVE**. L'intégration est faite de manière progressive : d'abord dans 4 fonctions de base de l'application de CDVE (nommés « benchmarks unitaires »), puis dans une fonction plus complexe (nommée « benchmark intermédiaire »), puis dans une application proche de l'application complète de CDVE (nommée « benchmark complet »). Dans ces expérimentations, les programmes exécutés sont codés conformément aux règles de programmation d'Airbus.

Concernant les entrées de ces programmes, pour les premières séries d'expérimentations, nous utilisons une stratégie de génération aléatoire de données : le code détecteur d'erreurs est appliqué à différentes séquences de données générées d'une façon aléatoire. Cette stratégie est la plus réaliste vu que dans de telles communications, les données ne suivent pas un « pattern » particulier. Alors que pour les benchmarks, les données d'entrées sont générées selon une stratégie propre à Airbus Helicopters.

Enfin, précisons que au cours d'une exécution d'un programme, chaque fonction du calcul du code détecteur n'est appelée qu'une seule fois pour les premières séries d'expérimentations y comprises les expérimentations sur les benchmarks unitaires, deux fois pour le benchmark intermédiaire et 104 ou 46 fois (des choix faits par Airbus Helicopters) pour le benchmark complet.

Le Tableau IV.4 synthétise les étapes des expérimentations et leurs caractéristiques.

Étape	Intégration code détecteur / fonction CDVE	Type de programmation	Entrées des programmes
1) Impact de la longueur du code	Aucune intégration	LAAS (chapitre III)	Aléatoire
2) Temps d'exécution unitaire	Aucune intégration	LAAS (chapitre III)	Aléatoire
3) Benchmarks unitaires	Intégration dans 4 fonctions CDVE de base	Airbus	Airbus
4) Benchmark intermédiaire	Intégration dans une fonction CDVE complexe	Airbus	Airbus
5) Benchmark complet	Intégration dans toutes les fonctions CDVE de l'application quasi complète	Airbus	Airbus

Tableau IV.4 : Principales étapes des expérimentations

IV.5.2 Évaluations sans benchmark pour le choix de la longueur des codes considérés

Airbus Helicopters a réservé 32 bits pour le code détecteur de bout en bout, mais n'a pas encore fait de choix définitif entre un code de 8, 16 ou 32 bits. Rappelons que la longueur d'un code détecteur impacte à la fois son pouvoir de détection (le pouvoir de détection augmente avec la longueur du code) et son temps de calcul (logiquement, le temps de calcul augmente aussi avec la longueur du code, mais pourrait dépendre aussi des caractéristiques du processeur). Le choix final est toujours un compromis entre ces deux critères. Précisons également que les résultats, du chapitre III, sur le pouvoir de détection des codes évalués ici, ne permettaient pas encore de conclure sur le choix d'un de ces codes. Donc, pour choisir la longueur de code entre 8, 16 ou 32 bits, nous allons examiner les impacts de la longueur du code sur ces deux critères, en commençant par l'évaluation du temps de calcul.

IV.5.2.1 Hypothèses

Nous avons lancé des séries identiques de tests, sur les deux cartes d'évaluation, pour évaluer le temps d'exécution de chacun des trois codes (CRC, Adler et Fletcher), et ce, pour les trois longueurs 8, 16 et 32 bits. Les programmes de tests implémentent uniquement les codes détecteurs. La génération des données, sur lesquelles on calcule les bits de contrôle, est aléatoire. Enfin, pour les algorithmes d'implémentation de ces codes, nous avons seulement considéré une implémentation basique (voir chapitre III, section III.3.2.1) pour chaque type de code :

- une implémentation bit par bit basique pour le CRC,
- et une implémentation basique non optimisée pour Adler et Fletcher.

IV.5.2.2 Constats

Au total, nous effectuons nos évaluations sur les trois codes Adler, Fletcher et CRC. Pour chaque code, nous ciblons trois longueurs différentes 8, 16 et 32 bits. Dans toutes nos expérimentations, la taille de la charge utile est égale à 32 octets. Toutes ces expérimentations sont faites sur deux cartes d'évaluation avec deux processeurs différents. Tous les résultats (en µs) sont représentés dans le Tableau IV.5, et sous une autre forme dans la Figure IV.17 et la Figure IV.18, pour mieux mettre en lumière les similitudes de résultats entre le P2020 et le TMS320C6657.

L'analyse des résultats montre que, dans les grandes lignes, on obtient les mêmes types de constats sur le P2020 et le TMS320C6657. Plus en détails, nous constatons que :

1) Les différences de temps d'exécution entre les trois longueurs, pour un type donné de code, ne dépassent pas :
   - **10** µs sur le P2020 (cas de CRC).
   - **12** µs sur le TMS320C6657 (cas de CRC).
2) Accessoirement, on peut aussi remarquer que :
   - Les différences de temps d'exécution entre les types de codes sont assez similaires entre Adler et Fletcher, par rapport aux CRC où ils sont plus grands.
   - La longueur (8, 16 ou 32) qui a le meilleur temps de calcul est différente selon le type de code (CRC, Adler et Fletcher), mais pour un type donné de code la meilleure longueur est la même sur le P2020 et le TMS320C6657 : le meilleur temps de calcul est obtenu pour une longueur de 8 bits pour Fletcher et CRC et de 16 bits pour les Adler.

Code évalué	WCMT P2020 (µs)	WCMT TMS (µs)
Adler8	3,28	3,80
Adler16	2,29	2,20
Adler32	3,28	3,93
Fletcher8	2,96	3,80
Fletcher16	2,99	3,87
Fletcher32	2,99	3,95
CRC8	13,32	13,61
CRC16	13,38	13,63
CRC32	23,2	25,76

Tableau IV.5 : Temps de calcul sur P2020 et TMS selon la longueur du code

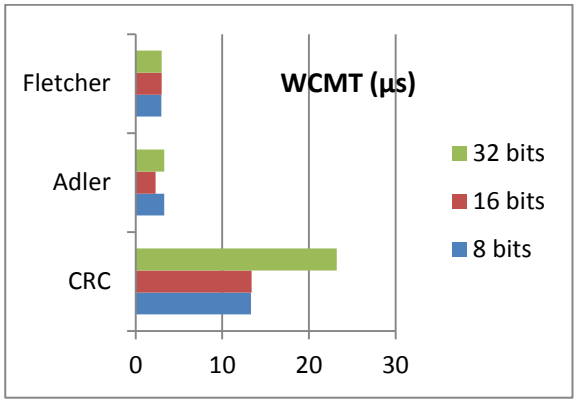


Figure IV.17 : WCMT sur P2020 selon la longueur du code

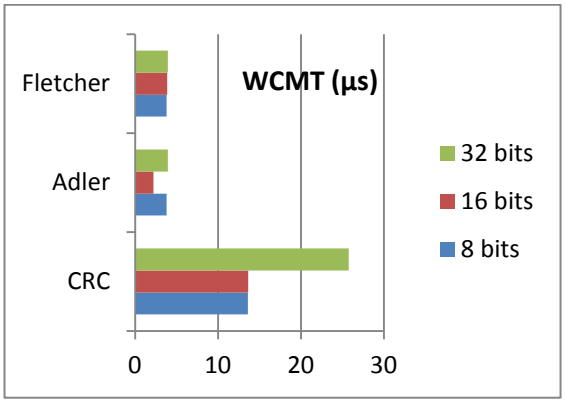


Figure IV.18 : WCMT sur TMS selon la longueur du code

IV.5.2.3 Interprétations et conclusions

L'interprétation de ces résultats est à faire en rappelant, d'une part, que les temps d'exécution sont à comparer au temps de cycle de rafraîchissement (10 ms) de l'application complète visée, et d'autre part, qu'on cherche seulement une première évaluation de l'ordre de grandeur de l'impact de la longueur du code sur le temps d'exécution. Dans ce contexte, nous pouvons tirer plusieurs conclusions :

1) Les temps d'exécution de tous les codes évalués (quelle que soit sa longueur) est largement compatible avec les contraintes temps réel de l'application : quelques µs à comparer avec quelques ms, soit un rapport de 1 pour 1000.
   - Le respect de ce critère de sélection sera bien entendu à revérifier dans les tests avec l'application finale. Mais, vu le rapport de 1 pour 1000 (qui est largement plus grand que le nombre d'appels du code détecteur du benchmark complet estimé à 104 au maximum), et vu qu'on n'a fait que des évaluations sur des algorithmes basiques non optimisées, alors, à ce stade, aucun des 3 types de codes ne peut être éliminé.
   - Et vu ce rapport 1 pour 1000, on peut considérer que les différents écarts (absolu par rapport aux 10 ms et relatif entre les codes) relevés dans les constats sont suffisamment significatifs pour en tirer des conclusions tranchées. Ce qui conduit aux conclusions 2) et 3)
2) Pour un type donné de code et pour une longueur donnée, les temps de calcul sont similaires sur les deux plateformes.
3) Pour un type donné de code, la longueur du code n'a pas toujours un grand impact sur le temps de calcul.

Ainsi, ni l'impact de la longueur des codes sur les temps d'exécution ni la différence des résultats des trois longueurs pour chaque code (pour CRC, les longueurs 8 et 16 bits ont nettement des meilleures performances, pour Adler et Fletcher, ces différences sont mineures) ne permettent pas de trancher sur le choix de la longueur à retenir.

Il faut maintenant étudier le critère du pouvoir de détection, sachant que **le modèle d'erreurs ciblé par Airbus Helicopters est les erreurs multiples avec l'hypothèse que les erreurs sont uniformément distribuées**. Comme présenté dans le chapitre III, on s'appuie sur la propriété mathématique qui dit que, pour un code détecteur d'erreurs avec une longueur de *n* bits de contrôle, la limite supérieure du taux de non-détection des erreurs multiples et uniformément distribuées est égale à 2^{-n} [Paulitsch *et al.* 2005]. Pour les codes de longueur 32 bits, cette limite est égale à $2{,}32*10^{-10}$ ce qui est une performance largement suffisante pour répondre à l'exigence du taux de défaillances qui doit être inférieure à 10^{-9} dans les systèmes critiques.

IV.5.2.4 Conclusion globale

En se basant sur ces résultats, nous avons décidé de retenir, pour assurer l'intégrité de bout en bout, la longueur 32 bits puisqu'elle garantit de meilleures performances en pouvoir de détection par rapport aux longueurs 8 et 16 bits, tout en ayant un temps d'exécution quasiment similaire (pour le cas du CRC, ces performances est nettement améliorables si on utilise des implémentations optimisées) à celui de ces deux longueurs. De ce fait, dans ce qui suit, nous ne présenterons que les résultats pour des codes de longueur 32 bits, même si les expérimentations ont également été menées sur les codes 8 et 16 bits.

Dans les prochaines étapes, nous nous focalisons désormais sur une analyse beaucoup plus fine des temps d'exécution des différentes implémentations d'Adler, Fletcher et CRC, avec une longueur de 32 bits, dans le but de sélectionner les implémentations ayant les performances les plus intéressantes.

IV.5.3 Évaluations des temps d'exécution des codes 32 bits sans benchmarks

Cette série d'expérimentations a pour but d'évaluer le temps d'exécution « unitaire » des codes détecteurs seuls (sans être intégrés dans une fonction de l'application cible), pour une longueur de 32 bits.

La Figure IV.19 décrit les performances des différentes implémentations sur le processeur P2020. Nous notons d'abord que l'on retrouve les mêmes grandes tendances que dans les résultats obtenus sur les plateformes du LAAS-CNRS. En effet, nous notons une grande différence des résultats entre les implémentations, avec un rapport de 1 à 10 entre les valeurs extrêmes, et un seuil très net au-delà duquel le rapport est de 1 à 4. On peut donc classer les implémentations en deux lots :

1) le lot des implémentations ayant les meilleures performances avec dans l'ordre décroissant : Fletcher32, Adler32, crc32_tbl, crc32_tabw, crc32_tabiw, crc32_tabi, Fletcher32M, crc32_tab, crc32_tb4, Adler32M, crc32_tab16i, crc32_tab16
2) le lot des implémentations qui ont les plus mauvaises performances avec dans l'ordre décroissant : crc32_bbb, crc32_bbf, crc32_bw4, crc32_bwe, crc32_bit

Les implémentations qui ont les meilleures performances sont celles de Fletcher et d'Adler optimisés (Fletcher32 et Adler32), puis celles de CRC se basant sur les tables de correspondance (crc32_tbl, crc32_tabw, etc.). Nous notons que les implémentations optimisées d'Adler et Fletcher ont des performances légèrement meilleures que celles des implémentations basiques de ces deux codes. Par contre, les implémentations bit par bit de CRC (par exemple, CRC32_bbf et CRC32_bbb) ont les pires performances.

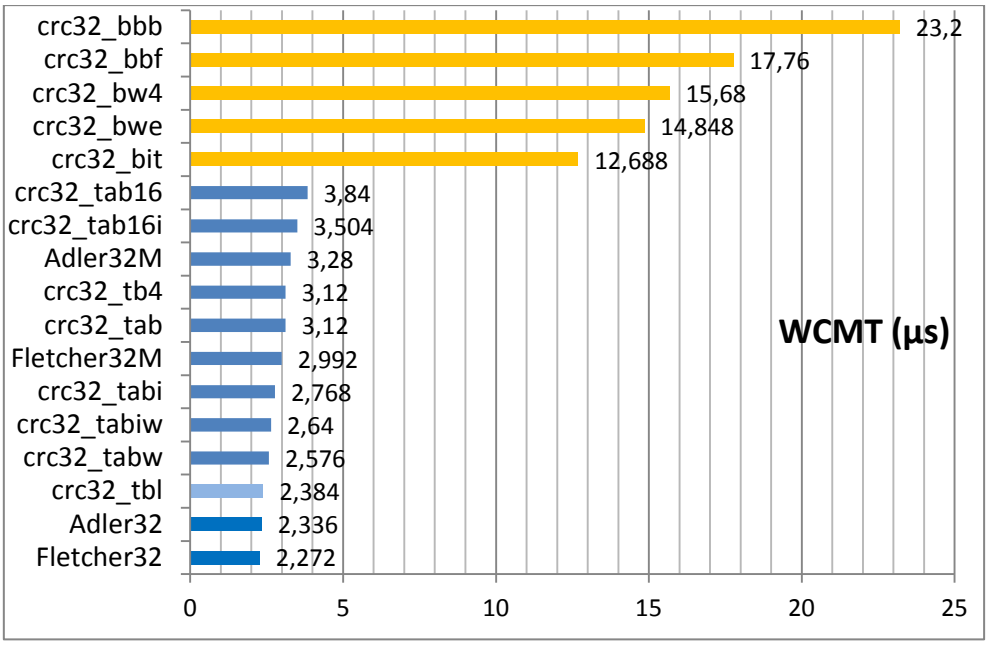


Figure IV.19 : Temps de calcul sur P2020 des codes seuls (sans benchmark)

La Figure IV.20 décrit les performances des différentes implémentations sur le processeur TMS320C6657 et qui conduit aux conclusions suivantes :

- Les constats majeurs par rapport au P2020 sont qu'on retrouve globalement les mêmes ordres de grandeur et tendances, pour les deux mêmes lots d'implémentation,
- dans le détail, il y a quelques différences :
  - la principale est que certaines implémentations CRC basées sur les tables de correspondance (crc32_tabi, crc32_tabiw, etc.) ont des meilleures performances que celles d'Adler et Fletcher,
  - plus généralement, à l'intérieur de chaque lot, on ne trouve pas exactement le même ordre des implémentations que celui du lot analogue de la P2020
  - accessoirement, on note que le seuil entre les 2 lots est moins marqué

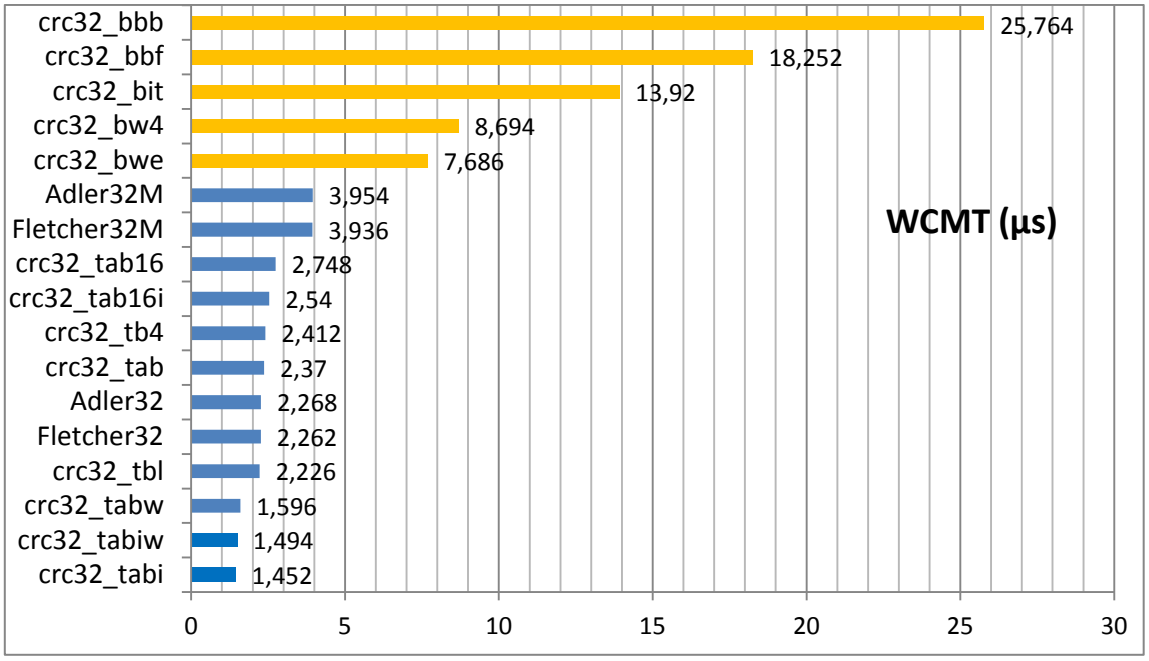


Figure IV.20 : Temps de calcul sur TMS320C6657 des codes seuls (sans benchmark)

En croisant les résultats sur les deux cartes d'évaluation (voir Figure IV.21), nous pouvons tirer ces conclusions :

- les performances des différentes implémentations testées sont quasi similaires sur les deux cartes d'évaluation,
- Pour les CRCs, l'utilisation des tables de correspondance réduit le temps de calcul et permet, dans le cas du processeur TMS320C665 d'avoir des performances meilleures que celles d'Adler et Fletcher,
- les implémentations bit par bit du CRC sont lourdes en temps de calcul. Donc, ils ne sont pas à retenir.

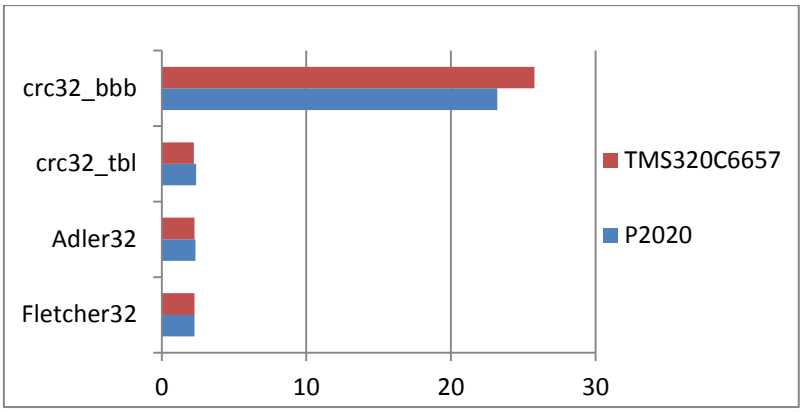


Figure IV.21 : Comparatif des temps d'exécution des codes sur P2020 et TMS320C6657

Les points à retenir pour les expérimentations suivantes sont :

- Nous gardons une implémentation de la classe des implémentations CRC bit à bit qui est le CRC32_bit qui servira simplement de point de référence pour montrer les apports des implémentations retenues,
- Les implémentations Adler32, Adler32M et Fletcher32M ont les mêmes performances, donc nous gardons juste un représentant qui est Fletcher32,
- Nous garderons toutes les implémentations basées sur des tables de correspondances de CRC qui sont : crc32_tbl, crc32_tabw, crc32_tabiw, crc32_tabi, crc32_tab, crc32_tb4, crc32_tab16i et crc32_tab16.

IV.5.4 Expérimentations avec benchmarks

Après avoir évalué les coûts de calcul des différentes implémentations seules sans être intégrées dans les fonctions CDVE, nous évaluons maintenant leurs coûts de calcul intégrées dans les benchmarks des applications constituant le système de commande de vol hélicoptère et exécutés par les différents calculateurs. Trois types de benchmarks sont évalués pour se rapprocher progressivement de l'application finale visée. C'est Airbus Helicopters qui a défini et codé ces benchmarks, a choisi la nature de la génération des données en entrées et les règles de codage des programmes (codage en C).

Les benchmarks réalisés dans une première étape sont des benchmarks unitaires sur des fonctions de base à différents niveaux de complexité : ACP_IO_S1_V41, ACP_IO_S2_V41, IO_FP_V41 et APSW_V41. Les trois premières applications gèrent principalement des entrées/sorties, alors que la quatrième est une application plus complexe, comportant elle-même plusieurs fonctions unitaires.

Nous ciblons dans une deuxième étape un benchmark intermédiaire. Ce benchmark, appelé **INER_FCP_V41**, concerne l'applicatif de haut niveau FCP supporté dans les calculateurs des commande de vol, et qui comprend notamment une nouvelle fonction permettant de faire des calculs d'inertie.

Finalement, nous ciblons un benchmark qui est une extension d'**INER_FCP_V41**. Ce benchmark n'est pas la version définitive puisque certaines hypothèses et paramètres sont encore en train d'être affinés au moment de l'écriture de ce manuscrit. Cependant, nous considérons que ce benchmark est suffisamment représentatif de la version définitive du benchmark complet.

IV.5.4.1 Benchmarks unitaires

Pour évaluer les coûts de calcul des différents codes et implémentations, nous avons retenu Fletcher32, toutes les implémentations basées sur des tables de correspondance de CRC et le CRC32_bit qui servira d'élément de comparaison. Nous considérons également « l'ancien CRC » (celui initialement adopté par Airbus Helicopters avant notre collaboration avec eux) qui est également une implémentation CRC bit par bit (en effet, ces deux dernières implémentations font partie des implémentations très coûteuses en temps de calcul). Nous avons ciblé les 4 benchmarks suivants :

- Les deux benchmarks **ACP_IO_S1_V41** et **ACP_IO_S2_V41** qui sont les benchmarks des entrées/sorties de la fonction ACP.
- Le benchmark **IO_FP_V41** des entrées sorties de l'application.
- Le benchmark **APSW_V41** qui contient l'IO_FP, le FCP et les modes supérieurs et l'hybridation.

Le Tableau IV.6 décrit les résultats obtenus sur le P2020 (exprimés en µs). Il s'agit des durées d'exécution de tout le programme, c'est-à-dire les applications CDVE intégrant les codes détecteurs. Ce tableau exprime également ces temps en termes de leurs surcoûts (en pourcentage) calculés par rapport au cas dans lequel aucun code détecteur n'est intégré dans les benchmarks (appelé SansCRC dans le Tableau IV.6). Ces surcoûts sont également représentés de manière graphique dans la Figure IV.22. La présentation des surcoûts a pour but de donner une meilleure visibilité sur les résultats. Nous relevons plusieurs résultats intéressants :

1) Tous les nouveaux codes et implémentations améliorent nettement les performances en coût de calcul par rapport aux implémentations bit par bit, dont l'implémentation initiale d'Airbus Helicopters.
2) Certaines implémentations CRC tabulées ont des performances équivalentes voire légèrement meilleures (cas des crc32_tabi, crc32_tabiw et crc32_tabw) que celles de Fletcher32. On constate cela pour chacun des 4 benchmarks.
3) Pour le benchmark le plus complexe (APSW_V41), le temps d'exécution reste inférieur à **2 ms**, ce qui est largement compatible avec les contraintes temps-réel de l'application cible finale.

Benchmark / Implémentation	ACP_IO_S1_V41		ACP_IO_S2_V41		IO_FP_V41		APSW_V41	
	Temps (µs)	Surcoût %	Temps (µs)	Surcoût %	Temps (µs)	Surcoût %	Temps (µs)	Surcoût %
SansCRC	21	0%	39	0%	195	0%	1629	0%
AncienCRC	76	265%	141	262%	815	318%	2249	38%
crc32_bit	45	117%	81	107%	445	128%	1913	17%
crc32_tab	27	28%	50	27%	254	30%	1715	5%
crc32_tab16	26	27%	49	25%	256	31%	1713	5%
crc32_tab16i	26	25%	49	25%	257	32%	1719	6%
crc32_tabi	24	16%	44	13%	230	13%	1689	4%
crc32_tabiw	24	13%	43	11%	227	16%	1685	3%
crc32_tabw	24	17%	44	14%	230	18%	1688	4%
crc32_tb4	26	23%	47	21%	250	28%	1712	5%
crc32_tbl	26	25%	47	20%	241	24%	1702	5%
fletcher32	25	18%	47	19%	243	25%	1701	4%

Tableau IV.6 : Résultats d'expérimentation sur les 4 benchmarks unitaires

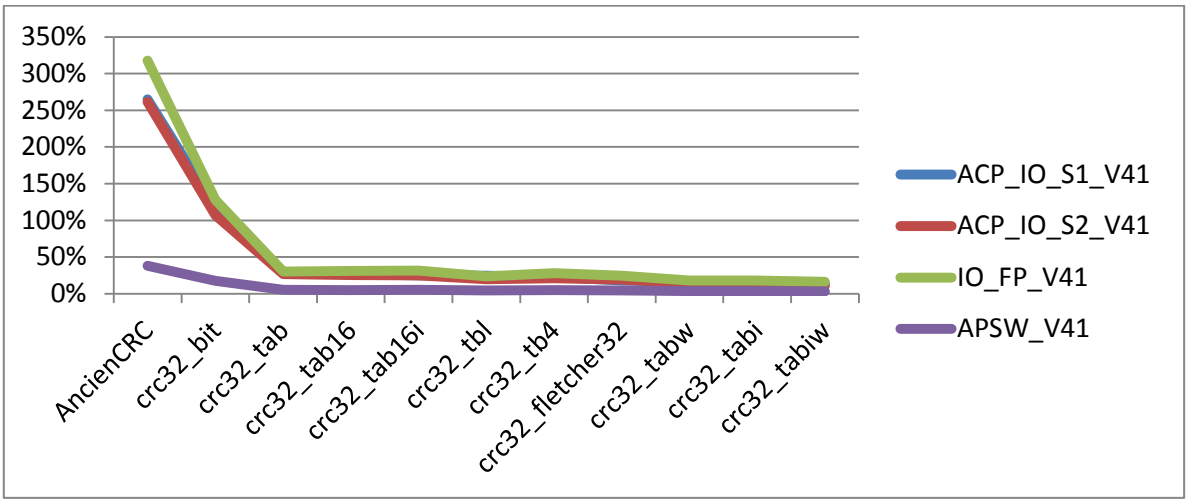


Figure IV.22 : Surcoûts sur P2020 des codes intégrés dans les 4 benchmarks unitaires

Les mêmes expérimentations menées sur le TMS320C6657 ont conduit à des résultats présentant des différences tellement mineures (avec ceux obtenus sur le P2020) que nous considérons que nous avons les mêmes résultats et constats principaux. De ce fait, nous n'allons pas présenter les résultats sur le TMS320C6657. De même, pour les prochaines expérimentations, nous nous limitons à présenter les résultats pour le benchmark intermédiaire et le benchmark complet sur le processeur P2020.

IV.5.4.2 Benchmark intermédiaire

Ce benchmark intermédiaire constitue une deuxième étape, plus poussée, dans l'évaluation des coûts de calcul des différents codes et implémentations. En effet, comme dit en introduction de cette section IV.5.4, ce benchmark, appelé **INER_FCP_V41** s'applique à un applicatif beaucoup plus complexe et de plus haut niveau que dans les benchmarks unitaires. Il s'agit d'un applicatif FCP supporté dans les calculateurs des commandes de vol. Cet applicatif s'approche déjà beaucoup de l'applicatif complet.

Nous avons sélectionné les codes et implémentations suivants : **Fletcher32** pour représenter le lot des codes basés sur des sommes arithmétiques ayant les mêmes caractéristiques et que les expérimentations précédentes ont prouvé qu'ils ont les mêmes performances et **toutes les implémentations basées sur des tables de correspondance** puisqu'elles ont des performances prometteuses mais des caractéristiques différentes (particulièrement en tailles de tables de correspondance). Pour avoir une référence pour apprécier l'apport en coût de calcul de ces implémentations, nous avons gardé le CRC32_bit et nous avons également pris en compte dans les résultats, le coût de calcul du benchmark sans CRC. Par contre, nous n'avons pas gardé l'ancien CRC adopté par **Airbus Helicopters**, puisque les résultats des benchmarks unitaires ont montré que cette implémentation a la pire performance comparée à toutes nos implémentations et donc elle n'était pas à retenir.

Comme l'applicatif qui fait l'objet du benchmark s'approche de l'applicatif complet, ce que nous présentons comme résultats cette fois, ce sont :

- Les temps d'exécution.
- Ce qui est nouveau est la « charge » qui représente le temps d'exécution par rapport au temps de cycle de rafraîchissement des commandes dans le système de CDVE, qui est égal à 10 ms, ce temps représente le temps d'exécution du benchmark ciblé intégrant à chaque fois une implémentation d'un code détecteur avec **2 appels à la fonction de calcul du code détecteur.**

Le Tableau IV.77 décrit les temps d'exécution et les charges, sur le P2020, du benchmark intermédiaire selon les différents scénarios considérés.

Implémentation	WCMT (ms)	Charge (en %)
SansCRC	4,087	**40,87%**
crc32_bit	4,165	41,65%
crc32_tab	4,153	41,53%
crc32_tab16	4,157	41,57%
crc32_tab16i	4,162	41,62%
crc32_tabi	4,135	41,35%
crc32_tabiw	4,135	41,35%
crc32_tabw	4,152	41,52%
crc32_tb4	4,162	41,62%
crc32_tbl	4,153	41,53%
Fletcher32	4,154	41,54%

Tableau IV.7 : Temps de calcul sur P2020 des codes dans le benchmark intermédiaire

La Figure IV.23 décrit les temps d'exécution des différentes implémentations considérées, à rappeler que nous avons deux appels de la fonction de calcul du code détecteur. Pour obtenir ces valeurs, on a retiré aux résultats précédents le coût de calcul du scénario SansCRC et les temps obtenus sont multipliés par 1000 (les nouveaux résultats sont alors en µs) pour être davantage lisibles.

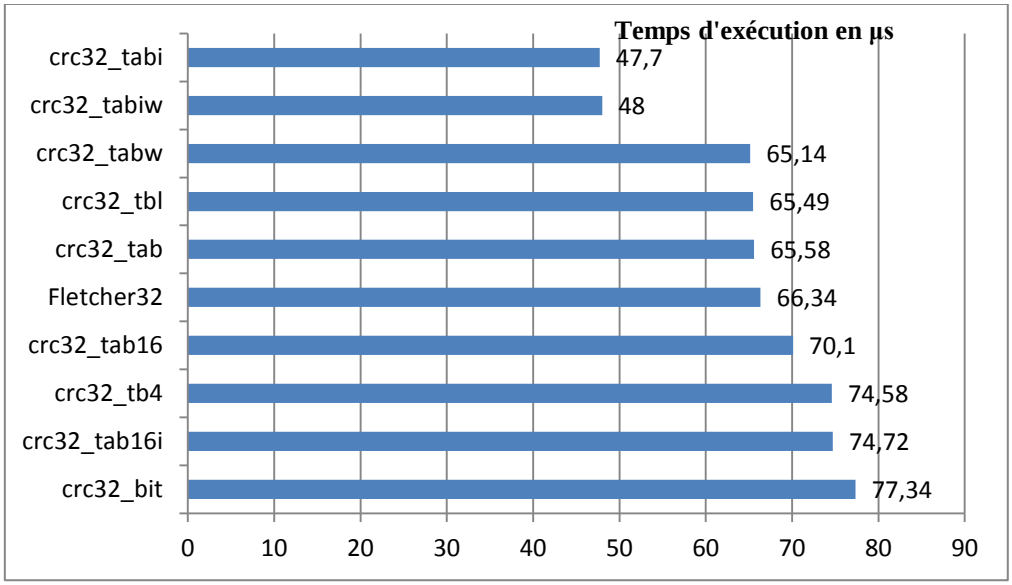


Figure IV.23 : Temps de calcul sur P2020 des codes dans le benchmark intermédiaire

Les résultats obtenus prouvent qu'avec l'intégration des différentes implémentations retenues pour ces expérimentations, nous respectons toujours les contraintes temporelles (les charges par rapport au temps de cycle de tous les scénarios sont inférieures à 50%, la charge maximale est égale à 41,65% pour CRC32_bit). Cependant, dans les expérimentations suivantes où on cible le benchmark complet, nous n'allons pas retenir toutes les implémentations évaluées lors de ces expérimentations, plus de détails sur le choix des implémentations à tester sur le benchmark complet seront dans la section suivante.

IV.5.4.3 Benchmark complet

Par rapport au benchmark intermédiaire, ce benchmark plus complet est encore plus proche de la version définitive (même s'il reste encore des hypothèses et des paramètres à affiner étant donné la phase d'avancement du projet des futurs CDVE d'Airbus Helicopters), en particulier, parce qu'il contient davantage d'accès au bus d'entrées/sorties, donc un nombre d'appels à la fonction de calcul du code détecteur très réaliste. Or, rappelons, que ce nombre d'appels impacte bien évidement le temps global d'exécution. Les écarts de temps d'exécution entre les différentes implémentations des codes, que l'on avait constaté jusqu'à maintenant, s'ils paraissent « mineurs » dans les appels unitaires, deviennent maintenant plus importants, quand on fait des dizaines d'appels à la fonction de calcul du code détecteur. Nous ciblons deux versions de ce benchmark, en précisant que « Crosstalk » signifie les échanges entre les deux voies d'un même FCC :

1) une version avec « full crosstalk » (les implémentations suivies par C dans le tableau) où il y a 104 appels de la fonction de contrôle d'intégrité (calcul du code détecteur)
2) et une deuxième version avec « light crosstalk » avec 46 appels de la fonction de contrôle d'intégrité.

Pour le benchmark complet, nous avons sélectionné les implémentations suivantes :

- **Fletcher32** : c'est la version optimisée de Fletcher, jusque-là toutes les expérimentations prouvent que Fletcher32 a de très bonnes performances en temps de calcul. Du fait qu'il est une somme arithmétique et qu'il n'est pas basé sur des tableaux de correspondance, il est un bon compromis avec un temps de calcul, une taille de code et une utilisation de mémoire intéressantes. Sa performance en pouvoir de détection est aussi prouvée (voir section IV.5.2). Pour nos expérimentations, il représente le lot des implémentations des sommes arithmétiques, selon nos analyses et résultats du chapitre III et des expérimentations précédentes de ce chapitre, Fletcher32 et Adler32 se valent selon les deux critères de sélection. Certains travaux [Maxino & Koopman 2009] confirment que Fletcher peut avoir des meilleures performances qu'Adler dans certains contextes d'où notre choix sur Fletcher32 pour ces expérimentations sur le benchmark complet.
- **CRC32_tb4 et CR32_tab16** : pour les implémentations CRC, nous avons déjà écarté les implémentations bit par bit. Notre choix consiste à un compromis entre le temps de calcul, la taille du code et de son utilisation mémoire. Nous avons gardé alors les implémentations avec le plus petit nombre d'entrées des tableaux de correspondance qui sont CRC32_tb4 et CRC_tab16 avec 16 entrées. Ce choix des implémentations avec des tables de correspondance avec un petit nombre d'entrées permet aussi de faciliter la caractérisation (selon des méthodes statiques) du WCET et facilite donc la certification.

Nous avons considéré le scénario SansCRC pour avoir une référence pour apprécier les coûts des implémentations retenues. Les résultats que nous présentons sont :

- les temps d'exécution,
- la « charge » que représente le temps d'exécution par rapport au temps de cycle de rafraîchissement des commandes dans le système de CDVE, qui est égal à 10 ms
- Les surcoûts en µs et en pourcentage des différentes implémentations après avoir soustrait le temps d'exécution du scénario SansCRC (démarche décrite précédemment).

Le Tableau IV.8 décrit les temps d'exécution de notre sélection d'implémentations ainsi que le scénario sans CRC. La Figure IV.24 décrit les surcoûts en µs des différentes implémentations en soustrayant le cas SansCRC.

Implémentation	WCMT (ms)	Surcoût (µs)	Surcoût (%)	CHARGE (%)
SansCRC	4,717	0	0,00%	47,17
CRC32_tb4_C	5,329	611,488	12,96%	53,29
CRC32_tab16_C	5,081	363,92	7,71%	50,81
Fletcher32_C	5,067	349,664	7,41%	50,67
CRC32_tb4	4,994	276,448	5,86%	49,94
CRC32_tab16	4,882	165,104	3,50%	48,82
Fletcher32	4,879	161,696	3,43%	48,79

Tableau IV.8 : Temps de calcul sur P2020 des différents codes intégrés dans le benchmark complet

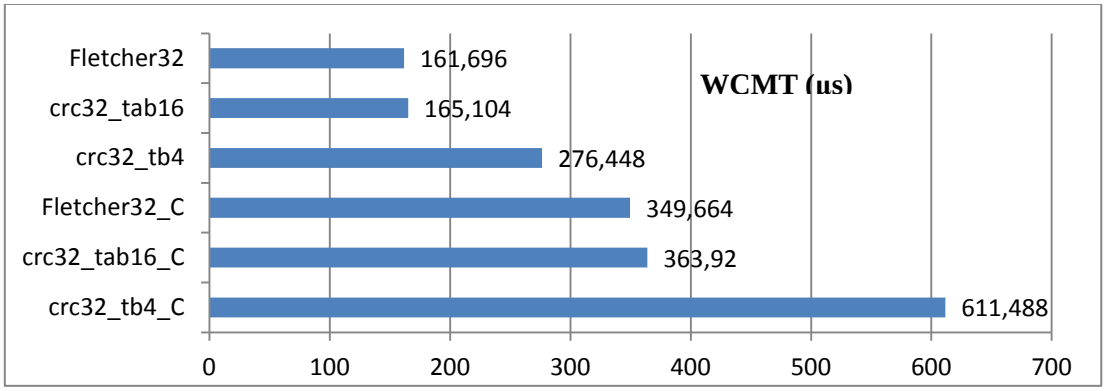


Figure IV.24 : Surcoûts sur P2020 des codes dans le benchmark complet (SansCRC 0%)

Nos expérimentations sur le benchmark complet ont prouvé, même dans le cas de la version avec crosstalk où on a 104 appels de la fonction de calcul du code détecteur d'erreur, que Fletcher32, CRC32_tab16 ou CRC32_tb4 intégrés dans le benchmark complet (application CDVE) fournissent des performances en temps d'exécution qui respectent les contraintes temporelles (charge maximale de 53,29 du temps de cycle pour CRC32_tb4_C), ce qui valide notre choix.

Suite à ces séries d'expérimentation et nos analyses, nous proposons ici nos conclusions tirées de notre expérience de l'utilisation des codes détecteurs dans les CDVE :

- Les algorithmes CRC bit par bit ne sont pas pertinents pour les systèmes critiques et temps réel, car ils sont coûteux en termes de temps de calcul.
- L'utilisation des tableaux de correspondance réduit le coût de calcul des CRC et les rend pertinents aux CDVE.
- Si le système est très contraignant et il est impossible d'utiliser les tables de correspondance, utiliser Adler ou Fletcher est un bon compromis, sinon on peut utiliser des CRC avec des tables de correspondance de petite taille (par exemple 16 entrées).

Si on compare maintenant nos résultats sur des codes détecteurs de taille 32 bits sur un CDVE hélicoptère décrits dans ce chapitre à ceux obtenus par [Youssef 2005] sur un CDVE avion où un CRC de taille 16 bits est utilisé à chaque cycle de rafraichissement, le CRC-16 avait une charge égale à 1% du cycle de rafraichissement, si on fait une centaine d'appels à la fonction du calcul de CRC-16 (comme on l'a fait quand on a intégré les codes détecteurs dans l'application complète du CDVE hélicoptère), la charge du calcul CRC seul sera égale à 100% du cycle de rafraichissement (plus de place pour les autres fonctionnalités) et donc un tel choix ne répond pas aux contraintes temporelles su système. Dans notre cas, le maximum de charge atteint par une centaine d'appels à la fonction de calcul des codes retenus est égale à 13% ce qui répond aux contraintes temporelles des systèmes. Ceci est dû d'un coté à la façon d'implémenter le code détecteur et aussi à la technologie utilisée, nos expérimentations étaient basées sur un processeur 32 bits fonctionnant à 1Ghz et avec des caches et les expérimentations menées dans [Youssef 2005] étaient basées sur un processeur 8 bits fonctionnant à 20 Mhz.

Dans ce qui suit, nous allons discuter de la complémentarité entre le code applicatif retenu et le CRC HDLC utilisé par Airbus Helicopters pour assurer l'intégrité des communications dans la couche liaison de données.

IV.5.5 Complémentarité entre le code détecteur applicatif retenu et le CRC HDLC

Après avoir retenu CRC et Fletcher (ou Adler puisqu'ils ont les mêmes performances) avec l'implémentation optimisée pour Fletcher ou Adler et l'implémentation basée sur une table de correspondance à 16 entrées pour le CRC, nous discutons ici de la complémentarité (voir chapitre III, section III.5.3) des codes retenus avec le CRC HDLC qui est un CRC implémentée dans la couche liaison de données (voir section IV.3.2). Nous rappelons que la politique d'intégrité d'Airbus Helicopters repose sur deux codes détecteurs d'erreurs, le CRC HDLC utilisé par le protocole HDLC et un code applicatif dont la sélection était l'objectif des expérimentations décrites précédemment. Comme dans nos travaux de thèse, nous proposons une approche multicodes [Zammali *et al.* 2015a] (voir chapitre III, section III.5) basée sur des codes détecteurs complémentaires et que nous nous intéressons à l'étude de la complémentarité entre les codes, l'objectif de cette section est de discuter la complémentarité du code retenu et du CRC HDLC. Même si l'approche multicodes que nous proposons considère plutôt la complémentarité d'un ensemble de codes applicatifs, la complémentarité inter-couches (couche applicative et couche liaison de données dans ce cas) a un sens aussi puisque le CRC applicatif peut couvrir les erreurs résiduelles non détectées par le CRC HDLC. À noter que la complémentarité ne couvre pas toutefois toutes les erreurs qui interviendraient entre la génération du code applicatif et celle de la génération du CRC HDLC sauf si on suppose que les liens dont directs et il n'y a pas de nœuds intermédiaires ce qui est le cas des futurs CDVE d'Airbus Helicopters.

Nos expérimentations ont conduit à deux pistes de choix du code applicatif, analysons la complémentarité pour les deux cas :

- ***Si CRC est retenu***

Dans le cas de l'utilisation d'un CRC, il faut étudier la complémentarité entre le polynôme générateur du CRC applicatif et le polynôme générateur du CRC HDLC. Le polynôme de CRC HDLC est :

$$G(x) = x^{32} + x^{26} + x^{23} + x^{22} + x^{16} + x^{12} + x^{11} + x^{10} + x^8 + x^7 + x^5 + x^4 + x^2 + x^1 + x^0$$

Pour la complémentarité, nous disposons de la base mathématique qui dit que deux CRC sont complémentaires si leurs polynômes générateurs ont un minimum de facteur en commun ([Youssef 2005], voir chapitre III). Le polynôme générateur du CRC HDLC est irréductible donc cette propriété mathématique est vérifiée peu importe le polynôme générateur du CRC applicatif choisi. Il faut donc choisir un « bon » polynôme générateur en termes de pouvoir de détection, même si dans notre contexte (les erreurs multiples sont ciblées) les polynômes générateurs se valent quand ils ont le même nombre de bits de contrôle (32 bits dans ce cas) mais pour renforcer la performance du CRC dans des contextes particuliers comme pour des erreurs de taille particulière, un bon polynôme est caractérisé dans [Koopman & Chakravarty 2004] par sa distance de Hamming (décrite dans le chapitre I) qui doit être grande.

Un polynôme générateur qui répond à ces conditions est le CRC-32 proposée par Koopman [Koopman 2002] :

$$G(x) = x^{32} + x^{30} + x^{29} + x^{28} + x^{26} + x^{20} + x^{19} + x^{17} + x^{16} + x^{15} + x^{11} + x^{10} + x^7 + x^6 + x^4 + x^2 + x + 1 = (x+1)(x^3 + x^2 + 1)(x^{28} + x^{22} + x^{20} + x^{19} + x^{16} + x^{14} + x^{12} + x^9 + x^8 + x^6 + 1)$$

Ce CRC peut détecter toutes les erreurs de taille 5 (HD=5) pour les messages avec une taille inférieure ou égale à 16360 bits qui est le cas des futurs CDVE d'Airbus Helicopters.

Donc, nous recommandons l'adoption de ce polynôme dans le cas où CRC est retenu pour être le code détecteur applicatif d'autant plus qu'il est recommandé dans le rapport du FAA [Koopman *et al.* 2015] qui est contient des recommandations pour la conception des systèmes avioniques.

- ***Si Fletcher ou Adler est retenu***

Contrairement au scénario où un CRC est retenu, il n'existe pas une base mathématique pour prouver la complémentarité entre le CRC HDLC et Fletcher (ou Adler). Néanmoins, Nous pouvons nous baser sur nos expérimentations décrites dans le chapitre III où il a été prouvé que Fletcher (ou Adler) et CRC sont complémentaires et que utilisés ensemble, Fletcher (ou Adler) et CRC fournissent de très bonnes performances en pouvoir de détection. La complémentarité analysée est aussi bonne que la complémentarité entre deux CRC.

IV.6 CONCLUSION

Ce chapitre était consacré à la présentation des résultats de l'application d'une partie de nos contributions sur un cas d'étude qui concerne les futurs systèmes de commande de vol d'Airbus Helicopters. Nous avons décrit d'abord le système étudié et ses propriétés. Nous avons fait par la suite une étude de l'existant tout en décrivant l'évolution des CDV des architectures mécaniques aux CDVE. Nous avons décrit l'architecture des CDVE actuels d'Airbus Helicopters particulièrement celle du NH9O tout en présentant ses limitations qui ont incité à la conception d'une nouvelle architecture pour les futurs CDVE. Nous avons décrit cette architecture en nous focalisant sur la politique d'intégrité des communications sur laquelle s'axe notre collaboration. Nous avons exposé dans le reste du chapitre nos expérimentations, résultats et analyses pour la sélection du code détecteur d'erreurs applicatif. Un premier volet était dédié à la présentation de l'environnement d'expérimentation et un deuxième volet est dédié aux expérimentations et résultats. La méthodologie d'expérimentation mise en place, ainsi que les résultats obtenus aideront Airbus Helicopters à effectuer un choix argumenté du code détecteur le plus favorable. Cependant, même si les expérimentations présentées ont été réalisées dans un cadre et pour des objectif très précis, toute une partie des hypothèses, des considérations et de la méthodologie peuvent s'appliquer bien au-delà de ce cadre, parce qu'elles ne sont pas spécifiques aux CDVE hélicoptère.

CONCLUSIONS ET PERSPECTIVES

L'objectif des travaux de thèse présentés dans ce mémoire était de définir une approche d'intégrité pour les communications dans les systèmes embarqués critiques.

Rappelons que les systèmes embarqués critiques sont des systèmes à hautes exigences en sûreté de fonctionnement, et ce, pour plusieurs raisons. La première est le fait que l'occurrence de certaines défaillances dans un tel contexte pourrait engendrer un évènement catastrophique, voire la perte de vies humaines. Du fait même de ce risque, les exigences de sûreté à respecter par ces systèmes sont souvent recommandées ou même dictées par des normes et des standards de certification qui sont, soit spécifiques à un domaine comme, par exemple, l'ISO2626 pour l'automobile, soit génériques comme l'IEC61508 qui définit les règles de certification pour tous les systèmes électroniques. À cela s'ajoute encore le fait que ces normes de certification ont même, dans certains domaines, une autorité légale (autorité de certification) qui autorise ou non, au final, la mise en service du système en évaluant son niveau de sûreté, comme par exemple l'AESA (Agence Européenne de la Sécurité Aérienne) pour l'aviation civile au niveau européen. Ceci fait de la sûreté de fonctionnement un enjeu primordial pour les concepteurs des systèmes embarqués critiques. Cependant, assurer la sûreté de fonctionnement rime souvent avec une augmentation des « coûts » (à prendre dans un sens très large), et en particulier avec l'augmentation du niveau et des formes de redondances déployées dans le système. Or, une telle augmentation se heurte aux contraintes des systèmes embarqués critiques, puisque ces systèmes sont généralement dotés de ressources restreintes, et dont le poids, le volume et la consommation énergétique sont à optimiser. À tout ceci s'ajoutent encore les contraintes temps réel strictes auxquelles sont soumis ces systèmes. Ainsi, atteindre le niveau requis en sûreté de fonctionnement est toujours un vrai défi.

Par ailleurs, depuis quelques années, les systèmes embarqués critiques sont de plus en plus nombreux à être davantage basés sur la philosophie de distribution de l'intelligence du système qui était jusque-là centralisée dans une ou plusieurs « unités de calcul », d'où le besoin du passage aux architectures distribuées. C'est le cas des systèmes de contrôle commande. Prenons l'exemple des systèmes de Commande De Vol Électriques (CDVE) des avions civils, pour lesquels les progrès technologiques et l'apparition des capteurs et actionneurs intelligents (c'est-à-dire dotés des capacités de mémorisation et de traitement de données) ont conduit ces systèmes à adopter des architectures de plus en plus distribuées où la prise de décision n'est plus seulement dévolue aux calculateurs, mais implique aussi les actionneurs et les capteurs intelligents via des systèmes de vote massif (par exemple) entre les décisions prises par les différents composants. Ces évolutions nécessitent naturellement l'adoption d'architectures distribuées fondées sur des réseaux de communication numériques, ce qui soulève la problématique de l'intégrité des communications (un attribut particulier de la sûreté de fonctionnement) et qui a conduit à l'objectif de ces travaux de thèse : définir une approche d'intégrité pour les communications dans les systèmes embarqués critiques.

Pour atteindre cet objectif, les travaux se sont articulés autour de quatre types de contributions.

1) Une analyse de l'existant sur plusieurs aspects, qui a permis de contextualiser nos travaux sur l'intégrité des communications, notamment par rapport à la problématique de certification, et d'analyser les solutions existantes, tant académiques que commerciales. Ceci nous a permis d'orienter nos travaux en nous inspirant de certaines approches et en nous appuyant sur leurs limitations pour proposer des améliorations.
2) La proposition d'une approche d'intégrité des communications bout en bout basée sur le concept « canal noir » présent dans la norme IEC61508. Notre approche utilise les codes détecteurs d'erreurs et se décline en deux approches différentes, selon le niveau de redondance déployée dans le système embarqué critique ciblé : l'approche « mono-code » et l'approche « multi-codes ».
3) Une étude fine (jusqu'à 17 algorithmes implémentés) des performances des codes détecteurs d'erreurs (CRC, Adler et Fletcher) en termes de pouvoir de détection et de coût de calcul, étude menée sur un environnement standard (« ordinateur de bureau ») et un environnement embarqué. Et cette étude a demandé le développement de deux environnements d'évaluation (sous Matlab-Simulink et en langage C). Il faut préciser que l'analyse fine de différentes implémentations possibles des codes détecteurs d'erreurs et de leurs impacts sur le coût de calcul est une des nouveautés par rapport aux travaux existants évaluant les performances des codes détecteurs d'erreurs.
4) L'application de l'approche d'intégrité bout en bout de type mono-code sur un cas d'étude industriel : les futurs systèmes de commande de vol d'Airbus Helicopters.

Plus précisément maintenant, l'analyse de l'existant a consisté dans un premier temps en un tour d'horizon pour étudier les concepts fondamentaux relatifs à l'intégrité des communications sur les systèmes embarqués critiques. Dans un deuxième temps, les principaux codes détecteurs ont été identifiés, puis les solutions existantes d'intégrité ont été étudiées, que ce soient celles déployées dans les couches basses des réseaux embarqués (ex. : ARINC 429, MIL-STD-1553, AFDX, etc.), ou celles déployées en tant que surcouches d'intégrité (telles que PROFIsafe, FSoE et openSAFETY) dans l'objectif d'assurer l'intégrité bout en bout. Enfin, les différentes approches dans des travaux académiques (proches de nos préoccupations) ont été également étudiées : ces approches s'appuient sur trois mécanismes différents à savoir les codes détecteurs d'erreurs, la redondance matérielle ou la diversification des données.

Cette analyse nous a conduits à identifier les limitations de certaines approches existantes. Nous en avons tiré six constats :

1) L'insuffisance des mécanismes d'intégrité implémentés dans les couches basses et le besoin d'une surcouche d'intégrité applicative pour couvrir les erreurs de toutes les couches sous-jacentes.
2) Le surdimensionnement dans les choix : pour augmenter le pouvoir de détection de l'approche d'intégrité, les solutions existantes ont souvent tendance à utiliser des codes détecteurs surdimensionnés en augmentant la taille des codes adoptés. Or, cette solution est très couteuse, ce qui est peu pertinent pour des systèmes embarqués critiques.
3) Le besoin de reconsidérer les codes dans de nouvelles hypothèses d'erreur : les travaux existants visent majoritairement des erreurs avec une faible multiplicité, alors qu'on ne peut pas exclure une multiplicité importante, pouvant atteindre la taille des messages.
4) Le besoin d'une caractérisation des temps de calcul des codes : pratiquement pas de travaux caractérisent le temps de calcul de manière précise.

5) Le besoin d'une meilleure sélection des codes détecteurs : un grand nombre de codes utilisés dans les approches existantes sont sous-optimaux en termes de pouvoir de détection. Le fait qu'un code soit déjà utilisé par des protocoles standards ne prouve pas son efficacité, et ne signifie pas non plus que ce code a les mêmes performances quelles que soient les caractéristiques du système. En réalité, cela dépend de plusieurs facteurs, à savoir la taille des messages échangés et la probabilité d'occurrence d'erreurs.
6) Les CRC ont un pouvoir de détection probant, mais leur coût de calcul est aujourd'hui encore considéré comme un frein à leur utilisation dans les systèmes embarqués critiques. Or il existe des pistes d'optimisation de leur coût de calcul qui méritent d'être explorées.

Finalement, de l'étude de l'existant, nous avons retenu le concept d'intégrité bout en bout qui permet de couvrir toutes les erreurs issues des différentes couches (par référence au modèle OSI). Nous nous sommes basés particulièrement sur le concept de « canal noir » introduit par l'IEC61508, qui fait abstraction des couches sous-jacentes, ce qui rend possible d'adapter notre approche quels que soient les protocoles et technologies sous-jacents. Un autre avantage du concept « canal noir » est le fait qu'il nécessite moins de justifications à apporter pour la certification, puisqu'il ne concerne que la couche applicative. Mais en contrepartie, il faut prouver que l'approche d'intégrité proposée suffit à elle-même pour atteindre le niveau d'intégrité requis sans s'appuyer sur les couches sous-jacentes. Nous avons retenu les codes détecteurs d'erreurs pour le compromis « pouvoir de détection - coût de calcul », un compromis qu'on ne pourrait pas satisfaire si on avait adopté des mécanismes plus complexes comme les fonctions de hachage, la cryptographie, qui ont des coûts de calcul importants. De plus, nous avons constaté une certaine maturité dans l'utilisation des codes détecteurs dans les systèmes embarqués critiques ce qui pourrait faciliter la certification de notre approche.

Nous avons donc présenté dans ces travaux une approche d'intégrité bout en bout basée sur les codes détecteurs d'erreurs, dont le but est de proposer des améliorations aux limitations citées plus haut. Il faut préciser qu'une hypothèse fondamentale pour notre travail est le fait que nous considérons que la multiplicité de l'erreur n'est pas bornée : c'est-à-dire que tous les bits de la trame peuvent être erronés. Cette hypothèse est différente de celle des travaux existants qui visent généralement une multiplicité faible (erreurs simples, doubles, etc.).

Une autre spécificité de nos travaux est que notre analyse des caractéristiques des systèmes embarqués critiques nous a conduits à distinguer deux classes de systèmes nommés « **très contraignants** » et « **contraignants** ». Cette notion de « contraignant » est relative aux mécanismes d'intégrité à mettre en place pour les communications : en effet, plus il y a certaines formes de redondance dans le système, plus cela offre d'opportunités de gérer l'intégrité. Les systèmes que nous considérons « **très contraignants** » sont ceux où une information à transmettre ne donne lieu qu'à un seul et unique message : un seul exemplaire de l'information est transmis. Dans ce cas, l'évènement redouté est la non détection d'erreur sur cet unique exemplaire. Dit autrement, si une erreur affecte cet unique exemplaire, et que cette erreur n'est pas détectée, alors, il n'y aucune autre opportunité de la détecter. Par conséquent, l'objectif d'intégrité doit nécessairement porter sur chaque message, pris individuellement. Les systèmes que nous considérons « **contraignants** » sont ceux possédant des formes de redondance (par exemple, une redondance spatiale ou temporelle) qui permettent qu'une information à transmettre donne lieu à plusieurs messages : plusieurs exemplaires de l'information sont transmis. Et dans ce cas, l'évènement redouté est la non détection d'une erreur sur tous les exemplaires (si on suppose qu'ils sont tous affectés par l'erreur). Par conséquent, l'objectif d'intégrité peut porter sur un lot de messages, typiquement les différents exemplaires d'une même information.

Cette classification est une généralisation d'une précédente classification, qui définissait des systèmes à « dynamique lente » ou dynamique « rapide », mais qui s'appuyait sur un cas de figure particulier pour notre deuxième classe de systèmes (dite « contraignante »), dont un exemple typique est les systèmes de commande de vol avion. En effet, ces systèmes sont considérés comme étant à « dynamique lente » parce que l'évolution physique des gouvernes est lente (de l'ordre de 50 degrés par seconde) relativement au cycle de rafraîchissement des calculs des commandes à appliquer aux gouvernes, et qui est égal à 10 ms. Ce temps de cycle est nettement inférieur à la durée nécessaire pour avoir une évolution significative au niveau des gouvernes (et donc au niveau de l'information à transmettre sur la position des gouvernes), ce qui se traduit par l'envoi de plusieurs messages successifs portant la même information avant l'occurrence d'une évolution significative. Ainsi, on dispose d'un ensemble d'exemplaires du même message, et l'intégrité peut porter donc sur ce lot d'exemplaires et non pas sur chaque message individuellement.

De cette classification, notre approche d'intégrité des communications générique se décline en deux approches différentes :

1) Pour les systèmes **« très contraignants »**, nous proposons une **approche d'intégrité « mono-code »**, c'est-à-dire utilisant un seul et même code détecteur pour protéger chacun des messages échangés.
2) Pour les systèmes **« contraignants »** qui rendent possible de tolérer la non détection d'erreur sur plusieurs messages, nous proposons une approche « **multi-codes** », c'est-à-dire que pour plusieurs messages portant la même information, on utilise un ensemble de codes détecteurs différents. L'objectif est qu'au moins un de ces codes soit apte à détecter l'erreur. Les différents codes doivent être « complémentaires » en termes de pouvoir de détection. Deux codes sont dits complémentaires, quand un code arrive à détecter des erreurs (toutes, idéalement) que l'autre code n'arrive pas à détecter.

Pour atteindre un niveau d'intégrité requis, il faut choisir le ou les « bons » codes, ce qui nécessite de mettre en place une stratégie de sélection des codes. Pour l'approche mono-code, les critères de sélection sont : le pouvoir de détection intrinsèque d'un code donné, et le coût (le temps) du calcul de ce même code. Pour chaque code, ces deux critères doivent répondre aux objectifs fixés vis-à-vis des contraintes et exigences du système. Pour l'approche multi-codes, aux deux critères précédents s'ajoute un troisième critère qui est la « complémentarité » entre les codes candidats.

Après avoir analysé les différents codes détecteurs d'erreurs et après avoir éliminé ceux qui ne répondent pas aux critères de sélection, nous avons présélectionné les codes CRC, Adler et Fletcher pour nos deux approches d'intégrité, que nous avons ensuite validées en réalisant des simulations évaluant les pouvoirs de détection et les coûts de calcul des codes présélectionnés et leur complémentarité pour l'approche multi-codes. Nous nous sommes appuyés pour nos expérimentations sur deux environnements logiciels différents : un environnement Matlab-Simulink et un environnement C et sur deux environnements matériels : un environnement standard (un « ordinateur de bureau ») et un environnement embarqué (STM32).

De ces expérimentations, nous avons tiré les conclusions majeures suivantes.

- La façon d'implémenter le code détecteur d'erreur impacte les performances en coût de calcul.
- Les implémentations bit par bit des CRC sont très lourdes, ce qui n'est pas adéquat pour les systèmes embarqués critiques.
- Les implémentations basées sur des tables de correspondance pour les CRC ont des performances prometteuses en coût de calcul, mais elles nécessitent une utilisation mémoire plus importante par rapport aux implémentations bit par bit.
- Pour une grande multiplicité de l'erreur, le pouvoir de détection d'un code dépend exclusivement de sa longueur (c'est-à-dire du nombre de bits de contrôle).
- L'utilisation combinée des codes CRC, Adler et Fletcher dans le contexte d'une approche multi-codes améliore le pouvoir de détection global si on compare avec une approche mono-code, ce qui prouve expérimentalement la complémentarité de ces codes.

Enfin, lors de la troisième année de thèse, une collaboration avec Airbus Helicopters est née autour de la problématique de l'intégrité de communication pour les futurs systèmes de commande de vol hélicoptère. En effet, Airbus Helicopters envisage de passer aux architectures totalement numériques pour les futurs programmes, à savoir, le X6 sur lequel nous avons travaillé. En effet, les architectures actuelles (par exemple, celle du NH9O) sont des architectures hybrides (numérique-analogique), ce qui présente un ensemble de limitations d'où l'objectif de l'évolution vers des architectures totalement numériques. Mais les futures architectures étant totalement numériques et basées sur une topologie totalement maillée, elles soulèvent une nouvelle problématique de l'intégrité des communications et nécessite la mise en place d'une nouvelle politique d'intégrité. L'objectif de cette collaboration était alors la sélection d'un code détecteur d'erreurs pour assurer l'intégrité bout en bout.

Nous avons alors mené des expérimentations sur deux cartes d'évaluation différentes (pour les processeurs P2020 et TMS320C6657) utilisées par Airbus Helicopters, dans le but d'évaluer les différentes implémentations des codes CRC, Adler et Fletcher sur ces cartes embarquées industrielles. Nous avons fait une première série d'expérimentations dans lesquelles les codes détecteurs ont été évalués seuls sans être intégrés dans l'application du système de CDVE. Cette série d'expérimentations nous a permis d'analyser l'impact de la longueur d'un code sur le temps d'exécution et les performances temporelles de codes sur les deux cartes. La deuxième série d'expérimentations a consisté à évaluer les codes détecteurs (retenus suite à la première série d'expérimentations) en les intégrant dans une application représentative de l'application du futur système de CDVE. L'intégration a été faite d'une façon progressive en partant des fonctions basiques jusqu'à l'application complète. Ces expérimentations nous ont permis de retenir un ensemble d'implémentations répondant aux contraintes temporelles et exigences en sûreté de fonctionnement d'Airbus Helicopters. Ces résultats aideront Airbus Helicopters pour son choix final du code détecteur applicatif à mettre en place dans les futurs programmes.

Les contributions de la thèse ont fait l'objet de quatre publications scientifiques dans quatre conférences internationales [Zammali *et al*. 2013], [Zammali *et al*. 2014], [Zammali *et al*. 2015a] et [Zammali *et al*. 2015b].

****

Pour les **perspectives** à ces travaux, nous pouvons déjà envisager des perspectives à **court terme** qui ciblent les résultats de nos travaux et leurs exploitations :

- L'application de l'approche multi-codes peut être étendue à d'autres scénarios d'utilisation (plusieurs codes complémentaires dans le même message). Une perspective pertinente pour l'approche multi-codes est de tester et valider cette approche sur un cas d'étude réel en menant une réflexion sur l'ordonnancement ou la distribution des codes complémentaires utilisés et en étudiant son coût d'implémentation et d'intégration dans un système réel. Un des défis de cette approche est d'éviter les fautes « byzantines » où les nœuds communicants ne voient pas le même état du système (il faut assurer leur synchronisation puisque le code détecteur utilisé change d'un exemplaire à l'autre).
- Lors de nos expérimentations sur les coûts de calcul des codes détecteurs d'erreurs, nous avons constaté que l'utilisation des caches accélère les temps de calcul de ces codes (constat validé par nos expérimentations qui simulaient l'utilisation des caches). Donc, l'utilisation des caches est une piste intéressante, mais pourrait ne pas être compatible avec les contraintes de déterminisme dans les systèmes embarqués critiques. Pour lever cet obstacle, la piste du « verrouillage des données et des programme dans le cache » pourrait être creusée en vue de respecter les contraintes de déterminisme.
- Les systèmes embarqués critiques, bien que dotés de ressources restreintes, ne cessent d'évoluer vers l'utilisation de composants de plus en plus complexes, implémentant des accélérations matérielles pour l'utilisation des codes détecteurs (ou correcteurs) ou même les fonctions de hachage ou la cryptographie. Par exemple, Freescale intègre dans certains composants un « *CRC engine* » pour la vérification de l'intégrité de la mémoire Flash et des communications. Ces mécanismes d'accélération matérielle peuvent inciter à repenser le choix de l'implémentation logicielle qui était jusque-là le choix le plus pertinent en termes de coût de calcul. Ces accélérations rendent aussi possible l'utilisation des mécanismes d'intégrité qui jusqu'à aujourd'hui sont considérés comme très lourd et non adéquats pour les systèmes embarqués critiques.

En se plaçant maintenant dans un point de vue plus large, comme perspectives à **long terme**, nous pouvons envisager deux axes :

- Une extension intéressante et importante est l'analyse quantitative des risques associés aux composants et leurs probabilités et modes de défaillance dans l'objectif de chiffrer le niveau d'intégrité requis et de le comparer au pouvoir de détection de l'approche d'intégrité. Cette analyse peut aider à éviter le surdimensionnement (concernant les codes détecteurs utilisés), parce que dans ces travaux de thèse, à défaut d'avoir pu disposer de tels chiffres pour faire cette analyse, notre approche reste pessimiste et donc surdimensionnée.
- Une extension possible est d'approfondir l'étude de la diversification de données et de son adoption dans l'approche d'intégrité en substitution ou en complément à l'utilisation des codes détecteurs d'erreurs, tout en définissant des fonctions de réexpression adéquates avec des expérimentations pour quantifier l'efficacité de l'approche par rapport au niveau d'intégrité visé et par rapport aux contraintes du système en coût de calcul.

RÉFÉRENCES BIBLIOGRAPHIQUES

[Akerberg *et al.* 2010] J. Akerberg, F. Reichenbach et M. Bjorkman, “Enabling Safety-Critical Wireless Communication using WirelessHART and PROFIsafe”, Proceedings 2010 IEEE Conference on Emerging Technologies and Factory Automation (ETFA), Bilbao, Espagne, 13-16 septembre 2010, pp. 1-8.

[Akerberg & M. Bjorkman 2009] J. Akerberg et M. Bjorkman, “Exploring Network Security in PROFIsafe”, Proceedings 28th International Conference on Computer Safety, Reliability and Security (SAFECOMP 2009), Hamburg, Allemagne, 15-18 septembre 2009, pp. 67-80.

[Ammann & Knight 1988] P. E. Ammann et J. C. Knight, “Data Diversity: An Approach to Software Fault Tolerance”, IEEE Transactions on Computers, vol. 31, no. 4, avril 1988, pp. 418-425.

[Arlat *et al.* 1990] J. Arlat, M. Aguera, L. Amat, Y. Crouzet, J.-C. Fabre, E. Martins, J.-C. Laprie et D. Powell, “Fault Injection for Dependability Validation — A Methodology and Some Applications”, IEEE Transactions on Software Engineering, Special Issue on Experimental Computer Science, vol.16, no. 2, février 1990, pp. 166-182.

[Arlat *et al*. 2003] J. Arlat, Y. Crouzet, J. Karlsson, P. Folkesson, E. Fuchs et G.-H. Leber, “Comparison of Physical and Software-Implemented Fault Injection Techniques”, IEEE Transactions on Computers, vol. 52, no. 9, septembre 2003, pp. 1115-1133.

[Armoush 2010] A. Armoush, “Design Patterns for Safety-Critical Embedded Systems”, Doctorat de l’Université Technique de Rhénanie-Westphalie, Aix-la-Chapelle, Allemagne, 15 juin 2010, 180 p. Disponible sur http://sunsite.informatik.rwth-aachen.de/Publications/AIB/2010/2010-13.pdf

[ARP4754A 2010] SAE International, “Lignes directrices pour le développement d’aéronefs et de systèmes civils” / “Guidelines for Development of Civil Aircraft and Systems”, révision A, 21 décembre 2010, 115 p.

[ARP4761 1996] ARP4761, “Guidelines and Methods for Conducting the Safety Assessment Process on Civil Airborne Systems and Equipment”, “Aerospace Recommended Practice” publié par SAE International, 1 décembre 1996, 331 p.

[Barr 2007] M. Barr, “CRC Implementation Code in C/C++”, CRC Series, Part 3, 2 décembre 2007. Disponible sur : http://www.barrgroup.com/Embedded-Systems/How-To/CRC-Calculation-C-Code

[Baufreton *et al.* 2010] P. Baufreton, J.P. Blanquart, J.L. Boulanger, H. Delseny, J.C. Derrien, J. Gassino, G. Ladier, E. Ledinot, M. Leeman, P. Quéré et B. Ricque, “Multi-Domain Comparison of Safety Standards”, Congrès ERTS2 des logiciels et systèmes embarqués temps réel, Toulouse, 19-21 mai 2010, 10 p.

[Baufreton *et al*. 2011] P. Baufreton, J.P. Blanquart, J.L. Boulanger, H. Delseny, J.C. Derrien, J. Gassino, G. Ladier, E. Ledinot, M. Leeman, P. Quéré et B. Ricque, “Comparaison de normes de sécurité-innocuité de plusieurs domaines industriels”, Revue de l'Electricité et de l'Electronique, no. 2, 2011, pp. 13-25.

[Boczar & Hull 2004] B. Boczar et B. J. Hull, “S-92 Fly-by-Wire Advanced Flight Control System”, 60th American Helicopter Society Annual Forum, Baltimore, Maryland, USA, 7-10 juin 2004, pp. 404-424.

[Chakravarty 2001] T. Chakravarty, “Performance of Cyclic Redundancy Codes for Embedded Networks”, rapport de master, Université de Carnegie Mellon, Pittsburgh, Pennsylvania, USA, décembre 2001, 26 p. Disponible sur http://users.ece.cmu.edu/~koopman/thesis/chakravarty.pdf

[Chun & Wolf 1994] D. Chun et J.K. Wolf, “Special Hardware for Computing the Probability of Undetected Error for Certain Binary CRC Codes and Test Results”, IEEE Transactions on Communications, vol.42, no. 10, octobre 1994, pp. 2769-2772.

[Cousin 1998] B. Cousin, “Protection contre les erreurs”, chapitre 3 du support de cours sur les réseaux et les protocoles informatiques, janvier 1998, disponible sur http://www.irisa.fr/armor/lesmembres/cousin/Enseignement/Reseaux-generalites/Cours/3.pdf

[Deutsch & Gailly 1996] P. Deutsch et J.-L. Gailly, “ZLIB Compressed Data Format Specification version 3.3.”, Network Working Group Request for Comments (RFC 1950), mai 1996. Disponible sur https://www.ietf.org/rfc/rfc1950.txt

[Dorow 2003] K. Dorow, “Flexible Fault Tolerance in Configurable Middleware for Embedded Systems”, Proceedings of International Conference on Computer Software and Applications (COMPSAC 2003), Dallas, Texas, USA, 3-6 novembre 2003, pp. 563-569.

[Dubrova 2013] E. Dubrova, “Fault-Tolerant Design”, Springer, no ISBN 978-1-4614-2113-9, 15 mars 2013,185 p.

[EN50128 2011] CENELEC, “Applications ferroviaires - Systèmes de signalisation, de télécommunication et de traitement - Logiciels pour systèmes de commande et de protection ferroviaire”, Norme Européenne, NF EN 50128, version française, octobre 2011, 150 p.

[Essamé 1998] D. Essamé, “Tolérance aux fautes dans les systèmes critiques – Application au pilotage de lignes de métro informatisées”, Doctorat de l’INP de Toulouse, 15 septembre 1998, 187 p.

[FAA Helicopter] “Federal Aviation Administration”, “Helicopter Flying Handbook” – chapitre 3 : “Helicopter Flight Controls”, 10 pages. Disponible sur: https://www.faa.gov/regulations_policies/handbooks_manuals/aviation/helicopter_flying_handbook/media/hfh_ch03.pdf.

[FAA] Federal Aviation Administration, System Safety Handbook – chapter 3: Principles of System Safety, 19 pages, Dec. 2000. Available at: http://www.faa.gov/regulations_policies/handbooks_manuals/aviation/risk_management/ss_handbook/media/Chap3_1200.pdf

[Feldmeier 1995] D. C. Feldmeier, “Fast Software Implementation of Error Detection Codes”, IEEE/ACM Transactions on Networking, vol. 3, no. 6, décembre 1995, pp. 640–651.

[Fletcher 1982] G. Fletcher, “An Arithmetic Checksum for Serial Transmissions”, IEEE Transactions on Communications, vol. 30, no. 1, janvier 1982, p. 247–252.

[Fuchs 1996] E. Fuchs, “An Evaluation of the Error Detection Mechanisms in MARS using Software-Implemented Fault Injection”, Proceedings of Second European Dependable Computing Conference (EDCC-2), 2-4 octobre 1996, Taormina, Italie, publié dans Springer-Verlag, Lecture Notes in Computer Science, vol. 1150, pp. 73-90.

[Fuchs 2012] C. M. Fuchs, “The Evolution of Avionics Networks from ARINC 429 to AFDX”, séminaire FI / IITM / AN SS2012, Network Architectures and Services, août 2012, 12 p. Disponible sur http://www.net.in.tum.de/fileadmin/TUM/NET/NET-2012-08-1/NET-2012-08-1_09.pdf

[Gailly & Adler 2013] J.L Gailly et M. Adler, “Zlib General Purpose Compression Library”, version 1.2.8, 28 avril 2013. Disponible sur : http://www.zlib.net/manual.html

[Gopal *et al*. 2012] V. Gopal, E. Ozturk, J. Guilford et W. Feghali, “Choosing a CRC Polynomial and Associated Method for Fast CRC Computation on Intel® Processors”, White Paper Intel Corporation, août 2012, 14 p. Disponible sur http://www.intel.com/content/dam/www/public/us/en/documents/white-papers/fast-crc-computation-paper.pdf

[Hiller 1998] M. Hiller, “Software Fault-Tolerance Techniques from a Real-Time Systems Point of View - An Overview », École polytechnique Chalmers, Göteborg, Suède. Rapport technique no 98-16, version1.0, novembre 1998, 45 p. Disponible sur http://webpages.iust.ac.ir/msharifi/SoftwareFaultToleranceWebSite/Files/Related%20Topics/Fault%20Tolerance%20in%20Real-Time%20Systems/hiller_tr98-16.pdf

[Hiller 1998] M. Hiller, “Software Fault-Tolerance Techniques from a Real-Time Systems Point of View - An Overview », École polytechnique Chalmers, Göteborg, Suède. Rapport technique no 98-16, version1.0, novembre 1998, 45 p. Disponible sur http://webpages.iust.ac.ir/msharifi/SoftwareFaultToleranceWebSite/Files/Related%20Topics/Fault%20Tolerance%20in%20Real-Time%20Systems/hiller_tr98-16.pdf

[IEC61508 2010] IEC, “Sécurité fonctionnelle des systèmes électriques/électroniques /électroniques programmables relatifs à la sécurité – Partie 2: Exigences pour les systèmes électriques/électroniques/électroniques programmables relatifs à la sécurité”, édition 2, International Electrotechnical Commission, avril 2010, 192 p.

[ISO26262 2011] ISO, “Véhicules routiers - Sécurité fonctionnelle”, norme de l’organisation internationale de normalisation, comité technique TC22 Road vehicles, sous-comité SC 3, novembre 2011, 10 parties.

[Jaber 2009] H. Jaber, “Conception architecturale haut débit et sûre de fonctionnement pour les codes correcteurs d’erreurs », Doctorat de l’université Paul Verlaine, Metz, France, 9 décembre 2009, 172p. Disponible sur http://docnum.univ-lorraine.fr/public/UPV-M/Theses/2009/Jaber.Houssein.SMZ0942.pdf

[Knight 2012] J. Knight, “Fundamentals of Dependable Computing for Software Engineers”, CRC Press, no. ISBN 9781439862551, 12 janvier 2012, 415 p.

[Kodis 1992] J. Kodis, “Fletcher’s Checksum”, Dr. Dobb’s Journal, vol. 17, no. 5, 1 mai 1992, pp 32-38. Disponible sur http://www.drdobbs.com/database/fletchers-checksum/184408761.

[Koopman & Chakravarty 2004] P. Koopman et T. Chakravarty, “Cyclic Redundancy Code (CRC) Polynomial Selection For Embedded Networks”, Proceedings 2004 International Conference on Dependable Systems and Networks (DSN 2004), Florence, Italie, 28 Juin-1 Juillet 2004, pp. 145-154.

[Koopman 2002] P. Koopman, “32-Bit Cyclic Redundancy Codes for Internet Applications”, Proceedings 2002 International Conference on Dependable Systems and Networks (DSN 2002), Bethesda, Maryland, USA, 23-26 juin 2002, pp. 459-468.

[Koopman *et al.* 2015] P. Koopman, K. Driscoll et B. Hall, “Cyclic Redundancy Code and Checksum Algorithms to Ensure Critical Data Integrity”, rapport FAA no. DOT/FAA/TC-14/49, mars 2015, 111 p. Disponible sur : http://www.tc.faa.gov/its/worldpac/techrpt/tc14-49.pdf

[Laprie *et al*. 1996] J.-C. Laprie, J. Arlat, J.-P. Blanquart, A. Costes, Y. Crouzet, Y. Deswarte, J.-C. Fabre, H. Guillermain, M. Kaaniche, K. Kanoun, C. Mazet, D. Powell, C. Rabejac et P. Thévenod-Fosse, “Guide de la sûreté de fonctionnement”, 2ème édition, Cépadues Éditions, no. ISBN 2854283821, 1996, 370 p.

[Liu & Song 2012] Y. Liu et Y. Song, “EtherCAT-based Functional Safety-Integrated Communication”, Proceedings International Conference on Automatic Control and Artificial Intelligence (ACAI 2012), Xiamen, Chine, 3-5 mars 2012, pp. 1005-1008.

[Marques *et al*. 2012] L. Marques, V. Vasconcelos, P. Pedreiras et L. Almeida, “Tolerating Transient Communication Faults with Online Traffic Scheduling”, Proceedings of IEEE International Conference on Industrial Technology (ICIT 2012), Athènes, Grèce, 19-21 mars 2012, p. 396-402.

[Maxino & Koopman 2009] T.C. Maxino et P. Koopman, “The Effectiveness of Checksums for Embedded Control Networks”, IEEE Transactions on Dependable and Secure Computing, vol. 6, no. 1, janvier-mars 2009, pp. 59-72..

[Maxino 2006a] T.C. Maxino, “The Effectiveness of Checksums for Embedded Networks”, Doctorat de l’université de Carnegie Mellon, Pittsburgh, Pennsylvania, USA, mai 2006, 32 p. Disponible sur http://users.ece.cmu.edu/~koopman/thesis/maxino_ms.pdf

[Maxino 2006b] T.C. Maxino, “Revisiting Fletcher and Adler Checksums”, Proceedings 2006 International Conference on Dependable Systems and Networks (DSN 2006), Student Forum, Philadelphia, Pennsylvania, USA, 25-28 juin. 2006, 3 p.

[McAuley 1994] A.J. McAuley, “Weighted Sum Codes for Error Detection and Their Comparison with Existing Codes”, IEEE/ACM Transactions on Networking, vol. 2, no. 1, février 1994, pp. 16-22.

[McGougan 2011] D. McGougan, “Universal CRC Environment”, version 1.3a., 2011, disponible sous licence GPL sur http://www.mcgougan.se/universal_crc/

[Naceur 2010] F. Naceur, “Etude et amélioration d'une fonction évolutive de controle d'erreur”, stage Master 2 Informatique et Télécommunications/Parcours Réseaux & Télécoms, Rapport LAAS no. 10416, 28 juin 2010, 51p.

[Nakassis 1988] A. Nakassis, “Fletcher’s Error Detection Algorithm: How to Implement It Efficiently and How to Avoid the Most Common Pitfalls”, ACM Computer Communication Review., vol. 18, no. 5, octobre 1988, pp. 63-88.

[Nguyen 2009] G. D. Nguyen, “Fast CRCs”, IEEE Transactions on Computers, vol. 58, no. 10, octobre 2009, pp. 1321-1331.

[Nguyen-Tuong *et al.* 2008] A. Nguyen-Tuong, D. Evans, J. C. Knight, B. Cox et J. W. Davidson, “Security through Redundant Data Diversity”, Proceedings of IEEE International Conference on Dependable Systems and Networks (DSN 2008), Anchorage, Alaska, USA, 24-27 juin 2008, pp. 187-196.

[Nguyen-Tuong *et al.* 2008] A. Nguyen-Tuong, D. Evans, J. C. Knight, B. Cox et J. W. Davidson, “Security through Redundant Data Diversity”, Proceedings of 2008 IEEE International Conference on Dependable Systems and Networks (DSN 2008), Anchorage, Alaska, USA, 24-27 juin 2008, pp. 187-196.

[Paret 2007] D. Paret, “Multiplexed Networks for Embedded Systems CAN, LIN, FlexRay, Safe-by-Wire…”, SAE International, no. ISBN 978-0768019384, 15 juin 2007, 434 p.

[Partridge *et al*. 1995] C. Partridge, J. Hughes et J. Stone, “Performance of Checksums and CRCs over Real Data”, Proceedings of Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication (SIGCOMM’95), vol. 25, no. 4, Cambridge, MA, USA, octobre 1995, pp. 68–76.

[Paulitsch & Hall 2007] M. Paulitsch et B. Hall, “Insights into the Sensitivity of the BRAIN (Braided Ring Integrity Network) - On Platform Robustness in Extended Operation”, Proceedings IEEE 2007 International Conference on Dependable Systems and Networks (DSN 2007), Edinburgh, UK, 25-28 juin 2007 pp. 154-163.

[Paulitsch & Hall 2008] M. Paulitsch et B. Hall, “FlexRay in Aerospace and Safety-Sensitive Systems”, IEEE Aerospace and Electronic Systems Magazine, vol. 23, no. 9, 30 septembre 2008, pp. 4-13.

[Paulitsch *et al.* 2005] M. Paulitsch, J. Morris, B. Hall, K. Driscoll, E. Latronico et P. Koopman, “Coverage and the use of Cyclic Redundancy Codes in Ultra-Dependable Systems”, Proceedings of the International Conference on Dependable Systems and Networks (DSN 2005), Yokohama, Japan, 28 juin-1 juillet 2005, pp. 346-355.

[Pircher 2013] T. Pircher, 2014, “pycrc Environment”, version 0.8.1, 17 mai 2013, disponible sous licence MIT sur https://pycrc.org/news.html

[Plummer 1978] W. W. Plummer, “TCP Checksum Function Design”, Internet Experiment Note 45 (IEN 45), 5 juin 1978, 12 p., disponible sur https://www.rfc-editor.org/ien/ien45.txt

[Powell 2001] D. Powell (Ed.), “A Generic Fault-Tolerant Architecture for Real-Time Dependable Systems”, Kluwer Academic Publishers Norwell, MA, USA, no. ISBN 978-1-4757-3353-2, 2001.

[Press *et al.* 2002] W.H. Press, S.A. Teukolsky, W.T. Vetterling, et B.P. Flannery, “Numerical Recipes in C: The Art of Scientific Computing”, 2ème édition, Cambridge University Press, juin 2002, 949 p.

[Rahmani *et al*. 2007] M. Rahmani, B. Muller-Rathgeber, E. Steinbach, “Error Detection Capabilities of Automotive Network Technologies and Ethernet – A Comparative Study”, Proceedings of IEEE Intelligent Vehicles Symposium (IV’07), Istanbul, Turkey, 13-15 juin 2007, pp. 674-679.

[Saiz-Adalid *et al.* 2013] L.J. Saiz-Adalid, P.J. Gil-Vicente, J.C. Ruiz-Garcia, D. Gil-Tomas, J.C. Baraza et J. Gracia-Moran, “Flexible Unequal Error Control Codes with Selectable Error Detection and Correction Levels”, Proceedings International Conference on Computer Safety, Reliability and Security (SAFECOMP 2013), Toulouse, 24-27 septembre 2013, pp. 178-189.

[Sghairi 2010] M. Sghairi, “Architectures innovantes de systèmes de commandes de vol”, Doctorat de l’Institut National Polytechnique de Toulouse, 27 mai 2010, 146 p. Disponible sur : http://tel.archives-ouvertes.fr/tel-00509156/fr/

[Sklower 1989] K. Sklower, “Improving the Efficiency of the OSI Checksum Calculation”, ACM Computer Communication Review, vol. 19, no. 5, octobre 1989, pp. 32–43.

[Soury *et al.* 2015] A. Soury, M. Charfi, D. Genon-Catalot et J.-M. Thiriet, “Performance Analysis of Ethernet Powerlink Protocol: Application to a New Lift System Generation”,

Proceedings IEEE 20th Conference on Emerging Technologies & Factory Automation (ETFA 2015), Luxembourg, Luxembourg, 8-11 septembre 2015, pp. 1-6.

[Stone & Partridge 2000] J. Stone et C. Partridge, "When the CRC and TCP Checksum Disagree", Proceedings of ACM Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication (SIGCOMM 2000), Stockholm, Suède, 28 août - 1 septembre 2000, pp. 309–319.

[Stone *et al*. 1998] J. Stone, M. Greenwald, C. Partridge et J. Hughes, "Performance of Checksums and CRC's over Real Data", IEEE/ACM Transactions on Networking, vol. 6, no. 5, octobre 1998. pp. 529–543.

[Storey 1996] N. Storey, "Safety-Critical Computer Cystems", Addison-Wesley, no. ISBN 0201427877, juillet 1996, 472 p.

[Traverse *et al.* 2004] P. Traverse, I. Lacaze et J. Souyris, "Airbus Fly-By-Wire: A Total Approach to Dependability", Proceedings of the 18th IFIP World Computer Congress (WCC 2004), Building the Information Society, *Topical Day 3: Fault Tolerance for Trustworthy and Dependable Information Infrastructures*, Toulouse, France, 23-24 août 2004, pp. 191-212.

[Traverse *et al.* 2006] P. Traverse, I. Lacaze et J. Souyris, "Airbus Fly-By-Wire: A Process Toward Total Dependability", Proceedings of the 25th International Congress of the Aeronautical Sciences (ICAS 2006), Hamburg, Allemagne, 3-8 septembre 2006, pp. 4293-4302.

[Verhulst *et al*. 2013], E. Verhulst, J. de la Vara, B.H.C. Sputh et V. de Florio "ARRL: A Criterion for Composable Safety and Systems Engineering", Workshop SASSUR (Next Generation of System Assurance Approaches for Safety-Critical Systems), International Conference on Computer Safety, Reliability and Security (SAFECOMP 2013), Toulouse, 24-27 septembre 2013. Disponible sur https://hal.archives-ouvertes.fr/hal-00848521.

[Vidal 1997] P.A. Vidal, "NH90 Helicopter Fly-By-Wire Flight Control System", Proceedings of the 53th American Helicopter Society Annual Forum,Virginia Beach, Virgiana, USA, 29 avril-1 mai 1997, pp. 915–923.

[Vidal *et al.* 2000] P.A. Vidal, E.G.J. Woirin, J-M. Massimi et P.L. Ressen, "Flight Control Device for an Aircraft, in particular a Helicopter", Brevet américain US6059225, mai 2000.

[Wagner 1986] M. Wagner, "On the Error Detecting Capability of CRC Polynomials", Informationstechnik it,28. Jahrgang, Heft 4/1986, p. 236-241.

[Wolf & Blakeney 1988] J.K. Wolf et R.D. Blakeney, "An Exact Evaluation of the Probability of Undetected Error for Certain Shortened Binary CRC Codes", Proceedings of 1988 IEEE Military Communications Conference (MILCOM 88), San Diego, Californie, USA, 23-26 octobre 1988, pp. 287-292.

[Yeh 1998] Y.C. Yeh, "Design Considerations in Boeing 777 Fly-By-Wire Computers", Proceedings of the 3rd IEEE International High-Assurance Systems Engineering Symposium (HASE'98), Washington, D.C, Etats-Unis, 13-14 novembre 1998, pp. 64-72.

[Yeh 2001] Y.C. Yeh, "Safety Critical Avionics for the 777 Primary Flight Control System", Proceedings of the 20th IEEE Conference on Digital Avionics Systems (DASC 2001), Daytona Beach, Florida, USA, 14-18 octobre 2001, pp. 1.C.2.1-1.C.2.11.

[Youssef 2005] A. Youssef, "Réseau de communication à haut niveau d'intégrité pour des systèmes de commande-contrôle critiques intégrant des nappes de microsystèmes",

Doctorat de l'Institut National Polytechnique de Toulouse, 22 novembre 2005, 164 p. Disponible sur : http://tel.archives-ouvertes.fr/tel-00111808/fr

[Youssef *et al.* 2006] A. Youssef, Y. Crouzet, A. de Bonneval, J. Arlat, J.J. Aubert et P. Brot, "Communication Integrity in Networks for Critical Control Systems", Proceedings of the European Dependable Computing Conference (EDCC-6), Coimbra, Portugal, 18-20 octobre 2006, pp. 23-34.

[Zammali *et al.* 2013] A. Zammali, A. de Bonneval et Y. Crouzet, "Communication Integrity for Slow-Dynamic Critical Embedded Systems", International Conference on Computer Safety, Reliability and Security (SAFECOMP 2013), Fast Abstracts, 24-27 septembre 2013, Toulouse, 2 p.

[Zammali *et al.* 2014] A. Zammali, A. de Bonneval et Y. Crouzet, "A Multi-Function Error Detection Policy to Enhance Communication Integrity in Critical Embedded Systems", International Conference on Software Security and Reliability - Companion (SERE-C 2014), 30 juin-02 juillet 2014, San Francisco (USA), pp.19-24.

[Zammali *et al.* 2015a] A. Zammali, A. de Bonneval et Y. Crouzet, "A Diversity-Based Approach for Communication Integrity in Critical Embedded Systems", Proceedings of the 16th IEEE International High-Assurance Systems Engineering Symposium (HASE 2015), Daytona Beach, Florida, USA, 8-10 janvier 2015, pp. 215-222.

[Zammali *et al.* 2015b] A. Zammali, A. de Bonneval, Y. Crouzet, P. Izzo et J. M. Massimi, "Communication Integrity for Future Helicopters Flight Control Systems", Proceedings of the 34th Digital Avionics Systems Conference (DASC 2015), Prague, République Tchèque, 13-17 septembre 2015, pp. 6D2-1-6D2-14.

Liens Internet du chapitre 2

Couche sûreté **PROFIsafe** : http://www.profibus.fr/technologies/profisafe

Couche sûreté **FSoE** :
site : http://www.ethercat.org/en/safety.html
description :
https://www.ethercat.org/download/documents/pcc0107_safety_over_ethercat_e.pdf
présentation :
http://www.ethercat.org/pdf/english/Safety_over_EtherCAT_Overview.pdf

Couche sûreté **openSAFETY**:
site : http://www.open-safety.org/
structure de la trame : http://www.open-safety.org/index.php?id=23&L=olrvrriytb

Liens Internet du chapitre 3

Outil pycrc
page principale : https://pycrc.org/
les différentes implémentations : https://pycrc.org/pycrc.html
tutoriel : https://pycrc.org/tutorial.html
modèles existants : https://pycrc.org/models.html

Outil Universal CRC : http://www.mcgougan.se/universal_crc/

Plateforme STM32

Famille ARM Cortex :
http://www.arm.com/products/processors/cortex-m/cortex-m4-processor.php
Processeur STM32F17 :
http://www.st.com/web/en/catalog/mmc/FM141/SC1169/SS1577/LN11
Carte d'évaluation :
http://www.st.com/web/en/catalog/tools/PF252217
Environnement de développement Keil : http://moodle.insa-toulouse.fr/pluginfile.php/34872/mod_resource/content/1/Prise_en_main_Keil.pdf

Liens Internet du chapitre 4

Programme SafeAssure Freescale : http://www.freescale.com/SafeAssure
Programme Longevity Freescale : http://www.freescale.com/pages/product-longevity-archived:LONGEVITY-ARCHIVED

Produits Freescale

Processeur MPC8548E :
http://www.freescale.com/products/power-architecture-processors/powerquicc-processors/powerquicc-iii-85xx/powerquicc-iii-processor-with-ddr2-pci-pci-express-serial-rapidio-serdes-1-gb-ethernet-security:MPC8548E
Processeur P2020 :
http://www.freescale.com/products/power-architecture-processors/qoriq-power-architecture-processors/qoriq-p2020-10-single-and-dual-core-communications-processors:P2020
Processeur P5020 :
http://www.freescale.com/products/power-architecture-processors/qoriq-power-architecture-processors/qoriq-p5020-10-64-bit-single-and-dual-core-communications-processors:P5020
Carte évaluation P2020 :
http://www.freescale.com/products/power-architecture-processors/qoriq-power-architecture-processors/p2020-development-system:P2020DS
Environnement de développement libre :
http://www.freescale.com/tools/embedded-software-and-tools/run-time-software/linux-sdk/linux-sdk-for-qoriq-processors-v1.8:SDKLINUX
Environnement de développement commercial :
http://www.freescale.com/tools/embedded-software-and-tools/software-development-tools/codewarrior-development-tools/suite-for-networked-applications/codewarrior-development-studio-for-power-architecture-technology-eclipse-v10.5:CW-PA

Produits Texas Instruments

Processeur TMS320C6657 : http://www.ti.com/product/tms320c6657
Carte évaluation pour TMS320C6657 : http://www.ti.com/tool/TMDSEVM6657
Environnement de développement libre : http://www.ti.com/tool/bioslinuxmcsdk
Environnement de développement commercial : http://www.ti.com/tool/ccstudio

ANNEXE 1 : UTILISATION DE TABLES DE CORRESPONDANCE

Algorithm:

1. Initialize the CRC register
2. XOR the CRC most significant byte with the incoming message byte
3. Use this byte to index into the 256 entry table
4. Shift the CRC register to the left by one byte
5. XOR the CRC register with the value indexed into the table
6. Continue with step 2 until no more message bytes are left
7. XOR the CRC register with the final XOR value

```
/* Code generated by universal_crc by Danjel McGougan
*
* CRC parameters used:
*   bits:       32
*   poly:       0x04c11db7
*   init:       0xffffffff
*   xor:        0xffffffff
*   reverse:    true
*   non-direct: false */

uint32_t crc_calc(const uint8_t *data, size_t len);

const uint32_t crc_table[256];

uint32_t crc_init(void){ return 0xffffffff;}

uint32_t crc_next(uint32_t crc, uint8_t data)
  { return (crc >> 8) ^ crc_table[(crc & 0xff) ^ data]; }

uint32_t crc_final(uint32_t crc){ return ~crc; }

const uint32_t crc_table[256] = {
     0x00000000, 0x77073096, 0xee0e612c, 0x990951ba,
     0x076dc419, 0x706af48f, 0xe963a535, 0x9e6495a3,
     0x0edb8832, 0x79dcb8a4, 0xe0d5e91e, 0x97d2d988,
     0x09b64c2b, 0x7eb17cbd, 0xe7b82d07, 0x90bf1d91,
     0x1db71064, 0x6ab020f2, 0xf3b97148, 0x84be41de,
     0x1adad47d, 0x6ddde4eb, 0xf4d4b551, 0x83d385c7,
     0x136c9856, 0x646ba8c0, 0xfd62f97a, 0x8a65c9ec,
     0x14015c4f, 0x63066cd9, 0xfa0f3d63, 0x8d080df5,
     0x3b6e20c8, 0x4c69105e, 0xd56041e4, 0xa2677172,
     0x3c03e4d1, 0x4b04d447, 0xd20d85fd, 0xa50ab56b,
     0x35b5a8fa, 0x42b2986c, 0xdbbbc9d6, 0xacbcf940,
     0x32d86ce3, 0x45df5c75, 0xdcd60dcf, 0xabd13d59,
     0x26d930ac, 0x51de003a, 0xc8d75180, 0xbfd06116,
     0x21b4f4b5, 0x56b3c423, 0xcfba9599, 0xb8bda50f,
     0x2802b89e, 0x5f058808, 0xc60cd9b2, 0xb10be924,
     0x2f6f7c87, 0x58684c11, 0xc1611dab, 0xb6662d3d,
     0x76dc4190, 0x01db7106, 0x98d220bc, 0xefd5102a,
     0x71b18589, 0x06b6b51f, 0x9fbfe4a5, 0xe8b8d433,
```

```
	0x7807c9a2, 0x0f00f934, 0x9609a88e, 0xe10e9818,
	0x7f6a0dbb, 0x086d3d2d, 0x91646c97, 0xe6635c01,
	0x6b6b51f4, 0x1c6c6162, 0x856530d8, 0xf262004e,
	0x6c0695ed, 0x1b01a57b, 0x8208f4c1, 0xf50fc457,
	0x65b0d9c6, 0x12b7e950, 0x8bbeb8ea, 0xfcb9887c,
	0x62dd1ddf, 0x15da2d49, 0x8cd37cf3, 0xfbd44c65,
	0x4db26158, 0x3ab551ce, 0xa3bc0074, 0xd4bb30e2,
	0x4adfa541, 0x3dd895d7, 0xa4d1c46d, 0xd3d6f4fb,
	0x4369e96a, 0x346ed9fc, 0xad678846, 0xda60b8d0,
	0x44042d73, 0x33031de5, 0xaa0a4c5f, 0xdd0d7cc9,
	0x5005713c, 0x270241aa, 0xbe0b1010, 0xc90c2086,
	0x5768b525, 0x206f85b3, 0xb966d409, 0xce61e49f,
	0x5edef90e, 0x29d9c998, 0xb0d09822, 0xc7d7a8b4,
	0x59b33d17, 0x2eb40d81, 0xb7bd5c3b, 0xc0ba6cad,
	0xedb88320, 0x9abfb3b6, 0x03b6e20c, 0x74b1d29a,
	0xead54739, 0x9dd277af, 0x04db2615, 0x73dc1683,
	0xe3630b12, 0x94643b84, 0x0d6d6a3e, 0x7a6a5aa8,
	0xe40ecf0b, 0x9309ff9d, 0x0a00ae27, 0x7d079eb1,
	0xf00f9344, 0x8708a3d2, 0x1e01f268, 0x6906c2fe,
	0xf762575d, 0x806567cb, 0x196c3671, 0x6e6b06e7,
	0xfed41b76, 0x89d32be0, 0x10da7a5a, 0x67dd4acc,
	0xf9b9df6f, 0x8ebeeff9, 0x17b7be43, 0x60b08ed5,
	0xd6d6a3e8, 0xa1d1937e, 0x38d8c2c4, 0x4fdff252,
	0xd1bb67f1, 0xa6bc5767, 0x3fb506dd, 0x48b2364b,
	0xd80d2bda, 0xaf0a1b4c, 0x36034af6, 0x41047a60,
	0xdf60efc3, 0xa867df55, 0x316e8eef, 0x4669be79,
	0xcb61b38c, 0xbc66831a, 0x256fd2a0, 0x5268e236,
	0xcc0c7795, 0xbb0b4703, 0x220216b9, 0x5505262f,
	0xc5ba3bbe, 0xb2bd0b28, 0x2bb45a92, 0x5cb36a04,
	0xc2d7ffa7, 0xb5d0cf31, 0x2cd99e8b, 0x5bdeae1d,
	0x9b64c2b0, 0xec63f226, 0x756aa39c, 0x026d930a,
	0x9c0906a9, 0xeb0e363f, 0x72076785, 0x05005713,
	0x95bf4a82, 0xe2b87a14, 0x7bb12bae, 0x0cb61b38,
	0x92d28e9b, 0xe5d5be0d, 0x7cdcefb7, 0x0bdbdf21,
	0x86d3d2d4, 0xf1d4e242, 0x68ddb3f8, 0x1fda836e,
	0x81be16cd, 0xf6b9265b, 0x6fb077e1, 0x18b74777,
	0x88085ae6, 0xff0f6a70, 0x66063bca, 0x11010b5c,
	0x8f659eff, 0xf862ae69, 0x616bffd3, 0x166ccf45,
	0xa00ae278, 0xd70dd2ee, 0x4e048354, 0x3903b3c2,
	0xa7672661, 0xd06016f7, 0x4969474d, 0x3e6e77db,
	0xaed16a4a, 0xd9d65adc, 0x40df0b66, 0x37d83bf0,
	0xa9bcae53, 0xdebb9ec5, 0x47b2cf7f, 0x30b5ffe9,
	0xbdbdf21c, 0xcabac28a, 0x53b39330, 0x24b4a3a6,
	0xbad03605, 0xcdd70693, 0x54de5729, 0x23d967bf,
	0xb3667a2e, 0xc4614ab8, 0x5d681b02, 0x2a6f2b94,
	0xb40bbe37, 0xc30c8ea1, 0x5a05df1b, 0x2d02ef8d
};

uint32_t crc_calc(const uint8_t *data, size_t len)
{
	uint32_t crc = crc_init();
	if (len) do { crc = crc_next(crc, *data++); }
		while (--len);
	return crc_final(crc);
}
```

ANNEXE 2 : UTILISATION DE PYCRC ET UNIVERSAL CRC POUR LA MESURE DU TEMPS DE CALCUL DES CODES CRC

Cette annexe présente comment nous avons exploité les outils pycrc et Universal CRC pour nos expérimentations concernant l'évaluation des temps de calcul des différentes implémentations des codes CRC. Nous présentons d'abord les commandes utilisées et nous décrivons ensuite comme ces commandes sont exploitées dans un shell Unix pour pouvoir de manière automatique : générer les différents sources, les associer à un programme main.c que nous avons développé suivant les besoins, compiler et lancer l'exécution des programmes afin de disposer des différents temps d'exécution dans des fichiers Excel.

II.1 Commandes utilisées au niveau de pycrc et Universal CRC

II.1.1 Cas de l'outil « PYCRC »

Pour l'utilisation de cet outil, on peut tout d'abord se référer à son tutoriel : https://pycrc.org/tutorial.html. L'utilisation de l'outil est très aisée si l'on utilise des modèles déjà présents dans le catalogue fourni par l'auteur : http://pycrc.org/crc-models.html. Dans ce cas, au niveau de la commande utilisée, il suffit de faire référence au nom du modèle. Pour notre étude, nous avons retenu les modèles crc-8, crc-16 et crc-32. Ainsi, dans un shell Unix, si la variable ${bits} contient le nombre de bits du CRC et si la variable ${algo} contient le type d'algorithme, il est possible de générer les fichiers .c et .h grâce aux deux commandes suivantes :

```
python ./outil/pycrc.py --model crc-${bits} --algorithm=${algo}
            --generate h -o crc${bits}_${algo}.h
python ./outil/pycrc.py --model crc-${bits} --algorithm=${algo}
            --generate c -o crc${bits}_${algo}.c
```

Pour disposer d'un fichier « main », il faut relancer une autre commande en précisant « c-main » dans l'option « --generate ». Il convient de noter que, dans le cas de l'outil « pycrc », il est possible de choisir le dialecte C utilisé grâce à l'option « --std » ; les choix possibles sont : C89, ANSI, C99.

II.1.2 Cas de l'outil « Universal CRC »

Pour l'utilisation de cet outil, on ne dispose que de sa page man que l'on peut retrouver sur le site : http://www.mcgougan.se/universal_crc/. L'utilisation de cet outil pourrait se révéler complexe s'il n'était pas un dérivé de « pycrc ». En effet, contrairement à « pycrc », il n'existe de modèles et il faut fournir au programme l'ensemble des paramètres qui caractérisent un code CRC. Dans le cas de notre étude, nous avons pu nous référer au catalogue de pycrc auquel il a été fait référence à la section précédente. Lors d'une commande de génération, même s'il existe un certain nombre de paramètres par défaut, il faut au moins fournir le nombre de bits du CRC et la valeur du polynôme générateur comme dans la commande suivante pour un CRC 8 bits :

```
./universal_crc -b 8 -p 0x07 -a ${algo} > crc8_${algo}.c
```

Pour disposer d'un fichier « main », il suffit d'indiquer l'option « --test ».

II.2 Shells UNIX pour la génération automatique des expériences

En se basant sur les commandes décrites précédemment, nous avons développé des shells Unix permettant d'enchaîner de manière automatique un ensemble d'opérations.
Un shell Unix est composé d'au moins 4 parties :

1) définition des paramètres des expériences avec notamment les différents algorithmes et tailles retenus,
2) génération des sources C pour tous les algorithmes et toutes les tailles retenus en associant le code C généré par les outils avec un programme d'exécution (main.c) développé suivant les besoins des expérimentations (incluant, par exemple, une génération aléatoire des entrées soumises aux fonctions CRC,
3) compilation de tous les sources C,
4) exécution de tous les programmes pour les différentes longueurs souhaitées au niveau de la charge utile avec accumulation des résultats de chaque exécution dans un fichier Excel,
5) lancement d'un programme de traitement des résultats bruts permettant d'obtenir des valeurs statistiques (valeurs min, max, moyenne).

II.2.1 Cas de l'outil « PYCRC »

```
#!/bin/sh

if [ $# -ne 2 ]
  then
    echo Usage: $0 préciser l\'option de compilation -Ox et le nombre de
runs
    exit 1
fi

CC_OPTS=$1
NB_RUNS=$2
NB_RUNS_1=`expr $2 - 1`
DATE=`date "+%d-%m-%Y/%H:%M"`
ALGOS="bbb bbf bwe tbl bw4 tb4"
TAILLES="8 16 32"
NB_CONFIGS=18
NB_PLAYLOADS=4

#==== génération des sources pour différentes tailles du CRC ===

for bits in $TAILLES
do
  echo "Génération des sources pour un CRC de $bits bits"
  for algo in $ALGOS
  do
    echo "    Génération des sources pour l'algorithme $algo"
    case $algo in
      bbb | bbf | bwe | tbl)
        python ./outil/pycrc.py --model crc-${bits} --algorithm=${algo}
           --generate h -o crc${bits}_${algo}.h
        python ./outil/pycrc.py --model crc-${bits} --algorithm=${algo}
           --generate c -o crc${bits}_${algo}.c
      ;;
      bw4)
        python ./outil/pycrc.py --model crc-${bits} --algorithm=bwe
           --generate h -o crc${bits}_${algo}.h --table-idx-width 4
        python ./outil/pycrc.py --model crc-${bits} --algorithm=bwe
```

```
            --generate c -o crc${bits}_${algo}.c --table-idx-width 4
      ;;
      tb4)
        python ./outil/pycrc.py --model crc-${bits} --algorithm=tbl
           --generate h -o crc${bits}_${algo}.h --table-idx-width 4
        python ./outil/pycrc.py --model crc-${bits} --algorithm=tbl
           --generate c -o crc${bits}_${algo}.c --table-idx-width 4
      ;;
    esac
    mv crc${bits}_${algo}.c temp
    cat temp main_LAAS.c > crc${bits}_${algo}.c
  done
done
rm temp

#==== compilation pour différentes tailles du CRC et algorithmes ===

echo
echo "Compilation des sources"
echo

for bits in $TAILLES
do
  for algo in $ALGOS
  do
    cc crc${bits}_${algo}.c -o crc${bits}_${algo} $CC_OPTS
  done
done

#=== création entête fichier résultats avec mémorisation des paramètres ===

echo  "Payload\t8  bytes\t16  bytes\t24  bytes\t32  bytes\t(CC_OPTS=$CC_OPTS)
\t$NB_RUNS\t$NB_CONFIGS\t$NB_PLAYLOADS
\t(DATE=$DATE)" > $FICHIER_RESULTS

#==== exécution pour différentes tailles du CRC et algorithmes ===

echo "Exécution avec l'option $CC_OPTS"
for test in `seq 0 $NB_RUNS_1`
do

  echo "      Test numéro $test"
  echo "Test numéro $test" >> $FICHIER_RESULTS

  for bits in $TAILLES
  do
    for algo in $ALGOS
    do
      ./crc${bits}_${algo}
    done
  done
done

#==== traitement du fichier résultats (extraction moyenne, min, max) ===

echo
echo "Début du traitement du fichier"
cc traitement_fichier.c -o traitement_fichier
./traitement_fichier $FICHIER_RESULTS $FICHIER_STATS
echo "Fin du traitement du fichier"
```

II.2.2 Cas de l'outil « Universal CRC »

```
#!/bin/sh

if [ $# -ne 2 ]
  then
    echo Usage: $0 préciser l\'option de compilation -Ox et le nombre de
runs
    exit 1
fi

CC_OPTS=$1
NB_RUNS=$2
NB_RUNS_1=`expr $2 - 1`
DATE=`date "+%d-%m-%Y/%H:%M"`
ALGOS="bit tab16 tab16i tab tabw tabi tabiw"
TAILLES="8 16 32"
NB_CONFIGS=21
NB_PLAYLOADS=4

#==== compilation outil ========

cc universal_crc.c -o universal_crc

#==== quelques tests ========

for algo in $ALGOS
do
  echo "Test algorithme type $algo : "
  ./universal_crc -b 8 -p 0x07 -a ${algo} --test > crc8_${algo}_test.c
  cc crc8_${algo}_test.c -o crc8_${algo}_test
  ./crc8_${algo}_test
done

#==== génération des sources pour différentes tailles du CRC ===

echo "Génération des sources pour un CRC de $bits bits"
for algo in $ALGOS
do
  echo "    Génération des sources pour l'algorithme $algo"
  ./universal_crc -b 8 -p 0x07 -a ${algo} | sed '/crc.h/d' > temp
  cat temp main_LAAS.c > crc8_${algo}.c
  ./universal_crc -b 16 -p 0x8005 -r -a ${algo} | sed '/crc.h/d' > temp
  cat temp main_LAAS.c > crc16_${algo}.c
  ./universal_crc -b 32 -p 0x04c11db7 -i 0xffffffff -x 0xffffffff -r
    -a ${algo} | sed '/crc.h/d' > temp
  cat temp main_LAAS.c > crc32_${algo}.c
done
rm temp

#==== compilation pour différentes tailles du CRC et algorithmes ===

echo
echo "Compilation des sources"
echo

for bits in $TAILLES
do
  for algo in $ALGOS
  do
    cc crc${bits}_${algo}.c -o crc${bits}_${algo} $CC_OPTS
  done
done
```

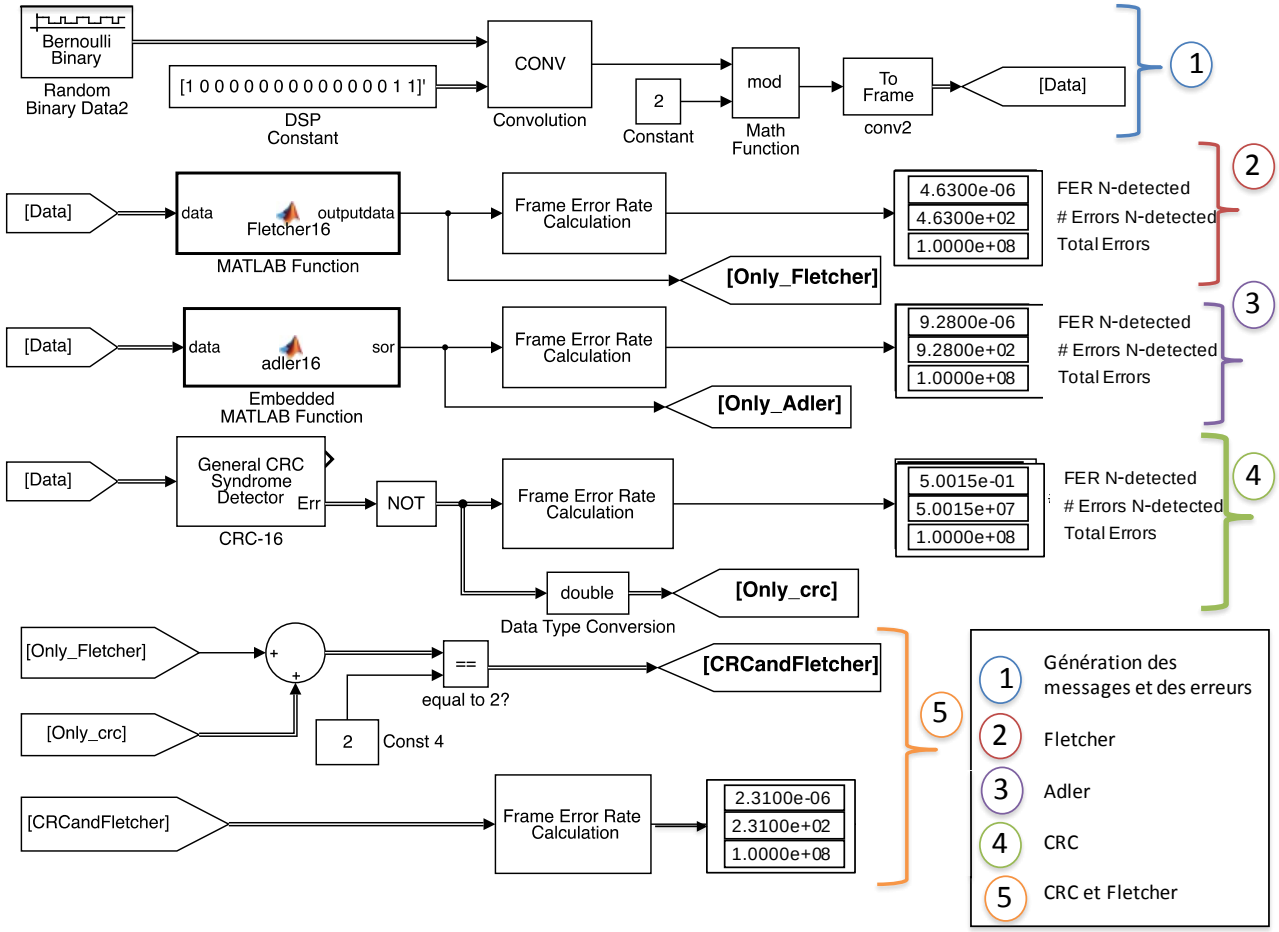
```
#=== création entête fichier résultats avec mémorisation des paramètres ===

echo "Payload\t8 bytes\t16 bytes\t24 bytes\t32 bytes\t(CC_OPTS=$CC_OPTS)
\t$NB_RUNS\t$NB_CONFIGS
\t$NB_PLAYLOADS\t(DATE=$DATE)" > $FICHIER_RESULTS

#==== exécution pour différentes tailles du CRC et algorithmes ===

echo "Exécution avec l'option $CC_OPTS"
for test in `seq 0 $NB_RUNS_1`
do

  echo "      Test numéro $test"
  echo "Test numero $test" >> $FICHIER_RESULTS

  for bits in $TAILLES
  do
    for algo in $ALGOS
    do
      ./crc${bits}_${algo}
    done
  done
done

#==== traitement du fichier résultats (extraction moyenne, min, max) ===

echo
echo "Début du traitement du fichier"
cc traitement_fichier.c -o traitement_fichier
./traitement_fichier $FICHIER_RESULTS $FICHIER_STATS
echo "Fin du traitement du fichier"
```

II.3 Programme de traitement des fichiers bruts

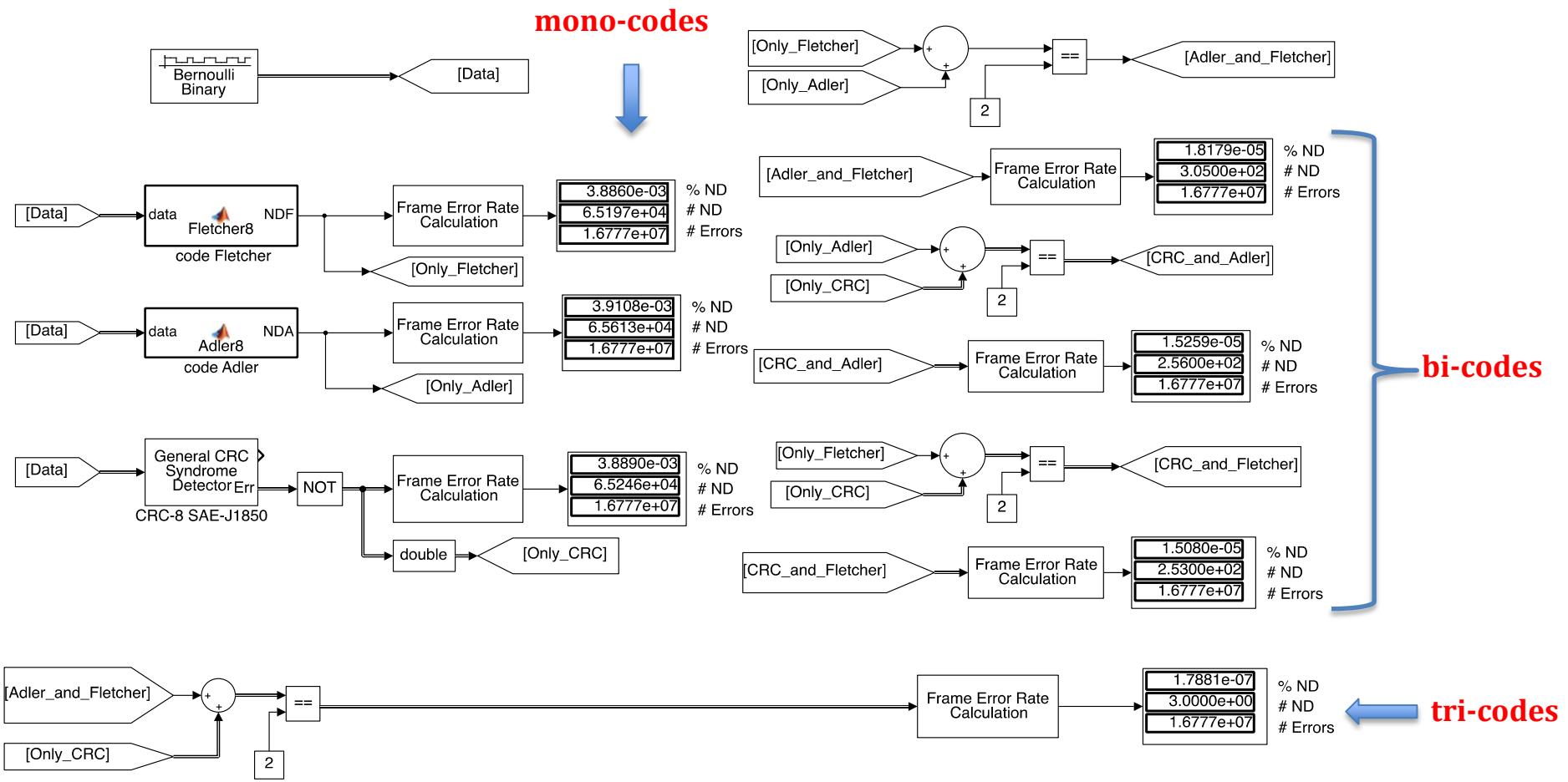
```
#include <stdio.h>
#include <stdlib.h>

#define max_runs 256
#define max_configs 40
#define max_payloads 10

int compare (const void * a, const void * b)
{
  return ( *(int*)a - *(int*)b );
}

int main(int argc, char *argv[])

{

char *file_inputs, *file_outputs ;

FILE * f;
char chaine1[32], chaine2[32], unite[32], options[32], date[32] ;
int p1, p2, p3, p4 ;
int nb_runs, nb_configs, nb_payloads ;
char nom_config [max_configs][32] ;
int num_run, num_config, num_payload ;
int texec[max_runs][max_configs][max_payloads] ;
int tab_tmp [max_runs] ;
int intervalle ;
int somme ;
double mean ;
```

```
  file_inputs = argv[1];
  file_outputs = argv[2];

// lecture du fichier brut
  f = fopen(file_inputs, "r");

// lecture de l'entête
  fscanf(f, "%s\t%d%s\t%d%s\t%d%s\t%d%s\t%s\t%d\t%d\t%d\t%s\n",
     chaine1, &p1, unite, &p2, unite, &p3, unite, &p4, unite,
     options, &nb_runs, &nb_configs, &nb_payloads, date);

// lecture de tous les runs
  for (num_run = 0 ; num_run < nb_runs ; num_run++)
  {
   // lecture du numéro du test
    fscanf(f,"%s\t%s\t%d\n", chaine1, chaine2, &num_run) ;

   // lecture des différentes configurations
    for (num_config = 0 ; num_config < nb_configs ; num_config++)
    {
      fscanf(f,"%s", nom_config[num_config]) ;
      for (num_payload = 0 ; num_payload < nb_payloads ; num_payload++)
        { fscanf(f, "\t%d", &(texec[num_run][num_config][num_payload])) ; }
      fscanf(f, "\n") ;

     } /* fin bcle sur les configs */
  } /* fin bcle sur les runs */
  fclose(f) ;

// création du fichier de stats
  f = fopen(file_outputs, “w");
// enregistrement entête
  fprintf(f, "Conditions\t%s\t%d runs\t%s\n", options, nb_runs, date) ;
  fprintf(f,   "Payloads\t8   Bytes\t\t\t16   Bytes\t\t\t24   Bytes\t\t\t32
Bytes\n") ;
  fprintf(f,
"\tMin\tMax\tMean\tMin\tMax\tMean\tMin\tMax\tMean\tMin\tMax\tMean\n") ;
  for (num_config = 0 ; num_config < nb_configs ; num_config++)
  {
    fprintf(f, "%s\t", nom_config[num_config]) ;
    for (num_payload = 0 ; num_payload < nb_payloads ; num_payload++)
    {
      for (num_run = 0 ; num_run < nb_runs ; num_run++)
      // Duplication des valeurs dans un tableau temporaire
      { tab_tmp[num_run] = texec[num_run][num_config][num_payload] ; }
      // Tri des valeurs et calcul de la moyenne
     qsort (tab_tmp, nb_runs, sizeof(int), compare);
     intervalle = (nb_runs+1) / 4 ;
     somme = 0 ;
      for (num_run = intervalle ; num_run < 3*intervalle ; num_run++)
        { somme = somme + tab_tmp[num_run] ; }
      mean = (double)somme / (2*intervalle) ;
      fprintf (f, "%d\t", tab_tmp[0]) ;
      fprintf (f, "%d\t", tab_tmp[nb_runs-1] ) ;
      fprintf (f, "%.2f\t", mean) ;
    } /* fin bcle sur les payloads */
    fprintf(f, "\n") ;
  } /* fin bcle sur les configs */
fclose(f) ;
return 0;
}
```

ANNEXE 3 : MODÈLES SIMULINK DE L'APPROCHE MULTI-CODES



Premier modèle Simulink illustrant l'approche « Multi-codes »
(cas d'un bi-code CRC16+Fletcher16)



Deuxième modèle Simulink illustrant l'approche « Multi-codes »
(cas de 3 bi-codes et d'un tri-codes Adler8+Fletcher8+CRC8)

RÉSUMÉ

Dans les systèmes embarqués critiques, assurer la sûreté de fonctionnement est primordial du fait, à la fois, des exigences en sûreté dictées par les autorités de certification et des contraintes en sûreté de ces systèmes où des défaillances pourraient conduire à des évènements catastrophiques, voire la perte de vies humaines. Les architectures de ces systèmes sont aujourd'hui de plus en plus distribuées, s'appuyant sur des réseaux numériques complexes, ce qui pose la problématique de l'intégrité des communications. Dans ce contexte, nous proposons une approche bout en bout pour l'intégrité des communications, basée sur le concept du « canal noir » introduit par l'IEC 61508. Elle utilise les codes détecteurs d'erreurs CRC, Adler et Fletcher. Selon le niveau de redondance des systèmes, nous proposons une approche multi-codes (intégrité jugée sur un lot de messages) pour les systèmes dotés d'un niveau de redondance important et une approche mono-code (intégrité jugée sur chaque message) pour les autres cas. Nous avons validé ces propositions par des expérimentations évaluant le pouvoir de détection intrinsèque de chaque code détecteur et la complémentarité entre ces code en termes de pouvoir de détection, ainsi que leurs coûts de calcul avec une analyse de l'impact du type de leur implémentation et de l'environnement matériel (standard et embarqué : processeurs i7, STM32, TMS320C6657 et P2020). L'approche mono-code a été appliquée à un cas d'étude industriel : les futurs systèmes de commande de vol d'Airbus Helicopters.

Mots clés : systèmes embarqués critiques, réseaux de communication numériques, sûreté de fonctionnement, intégrité des communications, intégrité bout en bout, codes détecteurs d'erreurs, systèmes de commande de vol.

ABSTRACT

In critical embedded systems, ensuring dependability is crucial given both dependability requirements imposed by certification authorities and dependability constraints of these systems where failures could lead to catastrophic events even loss of human lives. The architectures of these systems are increasingly distributed deploying complex digital networks which raise the issue of communication integrity. In this context, we propose an end to end approach for communication integrity. This approach is based on the “black channel” concept introduced by IEC 61508. It uses error detection codes particularly CRC, Adler and Fletcher. Depending on the redundancy level of targeted systems, we propose a multi-codes approach (integrity of a set of messages) for systems with an important redundancy level and a single-code approach (integrity per message) for the other cases. We validated our proposals through experiments in order to evaluate intrinsic error detection capability of each error detection code, their complementarity in terms of error detection and their computational costs by analyzing the impact of the type of implementation and the hardware environment (standard or embedded: i7, STM32, TMS320C6657 and P2020 processors). The single-code approach was applied to an industrial case study: future flight control systems of Airbus Helicopters.

Keywords: critical embedded systems, dependability, communication integrity, digital networks, end to end integrity, error detection codes, flight control systems.