République Algérienne Démocratique et Populaire
Ministère de l'Enseignement Supérieur et de la Recherche Scientifique
Université BATNA 2

T H È S E

Pour obtenir le titre de

Docteur en Sciences

Spécialité : Informatique

Présentée par

Malika BACHIR

Ordonnancement Tolérant Aux Fautes Pour Les Systèmes Distribués Temps Réel Embarqués

Thèse dirigée par

Pr. Hamoudi KALLA

Soutenue le : 17/01/2019

Devant le jury composé de :

Directeur :	Pr. Hamoudi KALLA	Prof	Université de Batna 2
Président :	Dr. Saber BENHARZELLAH	MCA	Université de Batna 2
Examinateurs :	Dr. Abdelwaheb ALOUI	MCA	Université de Béjaïa
	Dr. Mourad AMAD	MCA	Université de Bouira
	Dr. Omar MOULOUD	MCA	Université de Béjaïa
	Dr. Moumen HAMOUMA	MCA	Université de Batna 2

A ma très chère petite RAZANE

Remerciement

الشكر الأول لخالقي وحافظي وولي كل نعمي: ربّ الكريم

Je tiens ensuite à remercier mon directeur de thèse Hammoudi Kalla, professeur à l'université de Batna 2, pour la confiance qu'il m'a accordée, l'encadrement qu'il m'a offert pour ce travail et tout le soutien qui m'a permis d'achever ce travail. Un vrai esprit scientifique, rigoureux, et très précis, … merci monsieur pour tout.

Je remercie aussi les membres de mon jury de thèse, qui ont tous pris de leur temps pour lire et juger mes travaux et en particulier monsieur Saber BENHARZELLAH, maitre de conférences "A" à l'université de Batna 2, de présider mon jury de thèse et l'attention qu'il à accorder à sa lecture. Je remercie également Monsieur Abdelwaheb ALOUI, maitre de conférences "A" à l'université de Béjaïa, Monsieur Mourad AMAD, maitre de conférences "A" à l'université de Bouira, Monsieur Omar MOULOUD, maitre de conférences "A" à l'université de Béjaïa et Monsieur Moumen HAMOUMA maitre de conférences "A" à l'université de Batna 2, d'avoir bien voulu accepter la charge d'examinateur.

Je remercie vivement mon mari pour son aide, sa patience, ses conseils, … pour tous.

Je remercie sincèrement mon amie et ma sœur Derdour Khadidja, pour tous le bien de la vie qu'elle m'a appris et de m'avoir accueillie durant six ans à M'Sila. Je n'oublie pas ma chère Belkhiri Louazna.

Je remercie encore toutes personnes de près ou de loin qui m'a encouragé, aidé, guidé, … durant toute ma vie, et plus particulièrement mon oncle Ziadi.

Enfin, le meilleur pour la fin, je remercie de tout cœur ceux que je ne pourrai jamais remercier : ma mère et mon père ; mon frère, mes sœurs et toute la famille grands et petits.

Table des matières

Remerciements iii
Table des matières iv
Table des figures vii
Liste des tableaux ix

Chapitre 1 Introduction Générale

1.1.Introduction ... 1

Chapitre 2 Introduction aux systèmes distribués temps réels embarqués

2.1. Intrduction ... 5
2.2. Systèmes distribués ... 6
2.2.1. Définition ... 6
2.2.2. Caractéristiques ... 6
2.3. Systèmes temps réel ... 7
2.3.1. Le concept de temps réel ... 7
2.3.2. Les spécificités des systèmes temps réel ... 7
2.3.2.1. Définitions ... 7
2.3.3. Les tâches temps réel ... 8
2.4. Systèmes embarqués ... 9
2.4.1. Définition ... 9
2.4.2. Caractéristiques principales d'un système embarqué ... 9
2.4.3. Classement des systèmes embarqués ... 10
2.5. Systèmes distribués temps réel embarqués ... 11
2.5.1. Architecture des Systèmes distribués temps réel embarqués ... 11
2.5.2. Contraintes des Systèmes distribués temps réel embarqués ... 11
2.6. Conclusion ... 13

Chapitre 3 Sûreté de fonctionnement et Tolérance aux fautes

3.1. Introduction ... 14
3.2. Sûreté de fonctionnement ... 14
3.2.1. Définition de la sûreté de fonctionnement ... 14
3.2.2. Coût de la sûreté de fonctionnement ... 15

3.2.3. Taxonomie ... 15
3.3. Tolérance aux fautes ... 17
3.3.1. Taxonomie des fautes, des erreurs et des défaillances ... 17
3.3.2. Etapes de la tolérance aux fautes ... 20
3.3.3. Techniques de tolérance aux fautes ... 20
3.3.4. Techniques de redondance ... 24
3.3.5. Tolérance aux fautes logicielles et matérielles ... 26
3.3.6. Redondances logicielles pour la tolérance aux fautes matérielles ... 26
3.4. Conclusion ... 27

Chapitre 4 Introduction à l'ordonnancement temps réel

4.1. Introduction ... 28
4.2. Typologie des algorithmes d'ordonnancement ... 29
4.2.1. Monoprocesseur ou multiprocesseur ... 29
4.2.2. Hors-ligne ou en-ligne ... 29
4.2.3. Préemptif ou non-préemptif ... 30
4.2.4. Oisif ou non oisif ... 30
4.2.5. Centralisé ou distribué ... 30
4.3. Propriétés des algorithmes d'ordonnancement ... 30
4.4. Complexité des algorithmes d'ordonnancement ... 31
4.5. Présentation de la méthodologie AAA développée à l'INRIA Rocquencourt ... 32
4.5.1. Modèle de l'algorithme ... 33
4.5.2. Modèle d'architecture ... 35
4.5.3. Modèle d'implantation ... 36
4.5.4. Optimisation de l'implantation ... 38
4.5.5. Formalisation de la méthode AAA ... 39
4.5.6. Présentation de l'algorithme de distribution et d'ordonnancement de AAA . 40
4.6. Conclusion ... 43

Chapitre 5 Etat de l'art sur l'ordonnancement temps réel tolérant aux fautes des processeurs

5.1. Introduction ... 44
5.2. Algorithmes tolérants aux fautes basés sur la redondance logicielle ... 45
5.2.1. Réplication active ... 45
5.2.2. Réplication passive ... 47
5.2.3. Réplication semi-active (hybride) ... 49
5.3. Conclusion ... 52

Chapitre 6 Présentation de la méthodologie AAA-FAULT^{DT} et la méthodologie FT-TDEP

6.1. Introduction 53

6.2. Présentation de l'algorithme de distribution/ordonnancement tolérant aux fautes des tâches dépendantes AAA-FAULT^{DT} 54

6.2.1. Modèle de fautes 54

6.2.2. Données du problème 55

6.2.3. Principe général de la méthodologie AAA-FAULT^{DT} 56

6.2.3.1. Tâches principales de l'heuristique d'adéquation 61

6.2.3.2. Principe de l'heuristique 62

6.2.3.3. Présentation de l'heuristique 62

6.3. Evaluation de la méthodologie AAA-FAULT^{DT} 64

6.3.1. Paramètres d'évaluation 64

6.3.2. Les résultats 65

6.4. Présentation de la méthodologie FT-TDEP 67

6.5. Evaluation de la méthodologie FT-TDEP 71

6.6. Prédiction du comportement temps réel 73

6.7. Conclusion 74

Chapitre 7 Présentation de l'algorithme de distribution/ordonnancement tolérant aux fautes des tâches indépendantes AAA-FAULT^{IDT}

7.1. Présentation de l'algorithme de distribution/ordonnancement tolérant aux fautes des tâches indépendantes AAA-FAULT^{IDT} 75

7.1.1. Modèle de fautes 75

7.1.2. Données du problème 75

7.1.3. Principes généraux de la méthodologie AAA-FAULT^{IDT} 76

7.2. Evaluation de la méthodologie AAA-FAULT^{IDT} 83

7.2.1. Les résultats 83

7.3. Conclusion 85

Chapitre 8 Conclusion Générale et Perspectives

8.1. Conclusion et perspectives 856

Bibliographie 89

Table des figures

3.1. Arbre de la sûreté de fonctionnement 15
3.2. La chaine fondamentale des entraves à la sureté de fonctionnement........ 16
3.3 Défaillances du service 19
3.4. Techniques de la tolérance aux fautes 21
3.5. Différents types de redondance dans les systèmes embarqués critiques........ 24
4.1. Méthodologie AAA 33
4.2. Exemple d'un modèle d'algorithme........ 34
4.3. Communication intra-processeurs 34
4.4. Communication inter-processeurs........ 34
4.5. Exemple de graphe matériel (architecture multiprocesseur)........ 36
4.6. L'heuristique de distribution/ordonnancement AAA........ 42
5.1. Exemple de la réplication active 46
5.2. Exemple de la réplication passive 48
6.1. Architecture liaison à bus........ 55
6.2. Exemple d'un graphe d'algorithme........ 55
6.3. Exemple de transformation d'un graphe d'algorithme 57
6.4. Méthodologie AAA-FAULT^{DT} 58
6.5. Schéma de transformation de ALG........ 59
6.6. Distribution/ordonnancement tolérante aux fautes avec absence de défaillance 60
6.7. Distribution/ordonnancement tolérante aux fautes avec présence de défaillance........ 60
6.8. Nouveau distribution/ordonnancement après défaillance 61
6.9. L'algorithme AAA-FAULT^{DT} 63
6.10. Etapes de génération aléatoire d'un graphe d'algorithme..... **Erreur ! Signet non défini.**5
6.11. Effet de N sur AAA-FAULT^{DT} pour p = 5 et CCR = 2 66
6.12. Effet du P sur AAA-FAULT^{DT} pour N = 40 et CCR = 1........ 66
6.13. Effet du CCR sur AAA-FAULT^{DT} pour P = 6 et N = 50........ 67
6.14. Example d'un graphe d'algorithme........ 69
6.15. Example d'une connexion point à point 69
6.16. Schéma de transformation du graphe d'algorithme........ 69
6.17. Réplication Active des tâches 70

6.18. Réplication passive des tâches 70
6.19. Distribution/ordonnancement après défaillance de P1 71
6.20. Effet de N sur AAA-FAULT^{DT} et FT-TDEP pour p=5 et CCR=2 en absence de fautes 72
6.21. Effet de N sur AAA-FAULT^{DT} et FT-TDEP pour p=5 et CCR=2 en présence de fautes 72
6.22. Effet de P sur AAA-FAULT^{DT} et FT-TDEP pour N=40 et CCR=1 en absence de fautes 72
6.23. Effet de P sur AAA-FAULT^{DT} et FT-TDEP pour N=40 et CCR=1 en présence de fautes 72
6.24. Effet de CCR sur AAA-FAULT^{DT} et FT-TDEP pour P = 6 et N =50 en absence de fautes 73
6.25. Effet de CCR sur AAA-FAULT^{DT} et FT-TDEP pour P = 6 et N =50 en présence de fautes 73
7.1. Architecture liaison à bus 76
7.2. Exemple d'un graphe d'algorithme avec tâches indépendantes 76
7.3. Méthodologie AAA-FAULT^{IDT} 78
7.4. Exemple de transformation d'un graphe d'algorithme 79
7.5 Architecture matérielle 81
7.6 Architecture logicielle 81
7.7 Distribution/ordonnancement sans défaillance 82
7.8. Distribution/ordonnancement avec défaillance de P1 82
7.9. L'algorithme AAA-FAULT^{IDT} 83
7.10. Effet de N sur AAA-FAULT* pour P=5 et CCR=2 84
7.11. Effet de P sur AAA-FAULT* pour N=40 et CCR=1 84

Liste des tableaux

3.1. Comparaison entre les trois approches de redondance 27
4.1. Durée d'exécution des tâches sur les processeurs 38
4.2. Durée d'exécution des dépendances de données 38
6.1. Coût d'exécution des tâches sur différents processeurs 69
6.2. Coût de transfert de données entre tâches 70

Chapitre 1

Introduction générale

1.1. Introduction

Les systèmes temps réel embarqués sont de plus en plus présents dans tous les domaines de la vie quotidienne, du domaine d'applications à grand public (électronique grand public, automobile, …) aux domaines d'applications critiques (espace, nucléaire, santé, …). Ces systèmes sont conçus pour exécuter des tâches complexes et critiques, et sont soumis à des contraintes de temps et de ressources très strictes. Dans les systèmes temps réels embarqués critiques, une défaillance du système peut avoir des conséquences catastrophiques (perte de temps, d'argent ou pire, perte de vies humaines), et vu que les fautes sont inévitables et pouvant apparaître à tout moment ; ces systèmes doivent donc être sûre de fonctionnement. La sûreté de fonctionnement d'un système informatique est la propriété qui permet de placer une confiance justifiée dans le service qu'il délivre [19] [20] [52], plusieurs méthodes sont proposées dans la littérature pour la garantir ; la tolérance aux fautes est la technique la plus adoptée et elle constitue l'objectif principal de notre travail. La tolérance aux fautes permet au système de continuer à fournir le service attendu même en présence de défaillance. Comme les défaillances peuvent être causées à la fois par le matériel et le logiciel, la tolérance aux fautes peut alors être matérielle ou logicielle. La solution matérielle consiste à répliquer des composants matériels dont les systèmes embarqués n'en y peuvent supporter à cause de leurs contraintes de : coût, taille, énergie, … ; c'est pourquoi on opte dans la plupart des cas pour des solutions logicielles.

La redondance est l'une des méthodes utilisées pour atteindre la tolérance aux fautes. Elle peut être active, passive ou les deux. La réplication active consiste à exécuter la même tâche en parallèle sur plusieurs processeurs distincts. La réplication passive consiste à répliquer chaque tâche sur n répliques. Un seul des n répliques, appelé primaire, est exécuté. Les n-1 autres répliques sont en attente et ne s'exécutent que lorsque la tâche primaire échoue.

Les fautes peuvent être classées en fonction de leur durée en tant que : fautes permanentes, fautes intermittentes ou fautes transitoires. Les fautes permanentes sont persistantes, elles continuent d'exister jusqu'à ce que le composant défectueux soit réparé ou remplacé. Ces erreurs peuvent être causées par des pannes systèmes catastrophiques telles que des pannes de processeurs. Les fautes transitoires surviennent une fois puis disparaissent. Par exemple, un message réseau n'atteint pas sa destination. Les fautes intermittentes sont caractérisées par une faute survenue, puis disparaissent, puis se reproduisent, puis disparaissent à nouveau, ... Elles sont difficiles à deviner, mais leurs effets sont souvent corrélés. Une connexion lâche est un exemple de ce type de faute [49]. Nous nous concentrons sur les fautes permanentes d'un processeur unique.

Dans cette thèse, nous étudions l'intégration de la tolérance aux fautes dans les systèmes temps réel embarqués et distribués en partant d'un algorithme flot de données et d'une architecture distribuée hétérogène avec des liaisons multipoints ou point à point. Ce problème est NP difficile dont la solution optimale et exacte ne peut être trouvée que par des algorithmes exacts de complexité exponentielle, nous proposons des heuristiques qui approchent seulement cette solution tout en restant de complexité polynômiale.

Alors, notre objectif est de générer automatiquement une distribution/ordonnancement des tâches sur les processeurs qui est en plus tolérantes aux fautes. Les fautes considérées sont des fautes permanentes d'un seul processeur avec un comportement arrêt sur défaillance, c-à-d ou bien le système fonctionne, et donne un résultat correct, ou bien il est en panne, et ne fait rien. Pour ce faire, nous présentons trois nouvelles heuristiques basées sur la redondance logicielle qui génèrent un ordonnancement statique tolérant aux fautes. En prenant en compte la durée d'exécution de toutes les tâches sur tous les processeurs et les durées de communication de toutes les dépendances de données sur les liaisons de communication, nous sommes en mesure de calculer le temps total d'exécution de l'ordonnancement obtenue, en présence et en absence de défaillance. Les heuristiques proposées, qui sont une extension de la méthodologie AAA (Adequation Algorithm Architecture) de l'outil SynDEx [50] [51], tentent de trouver des solutions satisfaisantes les contraintes de temps réel, de distribution et de tolérance aux fautes.

Les trois méthodologies proposées sont :

La première méthodologie nommée AAA-FAULT^{DT} sert à optimiser d'une part la génération automatique de distribution/ordonnancement et d'autre part pour tolérer les fautes permanentes d'un seul processeur durant un cycle d'exécution de l'architecture

logicielle sur l'architecture matérielle. L'architecture logicielle est composée de plusieurs tâches dépendantes et l'architecture matérielle est constituée de plusieurs processeurs hétérogènes connectés par un bus de communication. Dans cette méthodologie, la tolérance aux pannes est assurée par le mécanisme de la redondance passive des composants logiciels en reposant sur un mécanisme de détection d'erreur.

La deuxième méthodologie notée FT-TDEP, est une optimisation de la première, elle se distingue par le média de communication qui est une liaison point à point et par le type de la redondance utilisé qui est la redondance hybride.

La troisième méthodologie appelée AAA-FAULT^{IDT} consiste à générer des distribution/ordonnancements tolérantes aux fautes, adaptées aux architectures matérielles distribuées et hétérogènes munies d'un réseau de communication bus. Elle permet de tolérer les fautes des processeurs en utilisant la redondance active dans une architecture logicielle avec tâches indépendantes.

La thèse est composée de deux parties :

La première partie comporte quatre chapitres décrivant les concepts de base des systèmes distribués, temps réels et embarqués, sur l'ordonnancement temps réel tolérant aux fautes et un état de l'art sur les techniques proposées dans la littérature pour assurer la tolérance aux fautes dans ces systèmes. La deuxième partie est constituée de deux chapitres consacrés à la présentation de nos propres solutions.

Le premier chapitre présente des concepts de bases et des terminologies liés aux systèmes temps réel, embarqués et distribués. Il définit leurs caractéristiques, leurs spécificités et leurs classifications. Il décrit ainsi leur architecture et leurs contraintes à savoir, les contraintes temporelles, matérielles et les contraintes de placement.

Le deuxième chapitre, présente des notions de base sur la sûreté de fonctionnement, comme : ses propriétés et ses moyens, il décrit spécialement les terminologies liées à la tolérance aux fautes et les méthodes utilisées pour l'assurer, en particulier il présente la redondance logicielle pour la tolérance aux fautes matérielles.

Le troisième chapitre est une introduction à l'ordonnancement temps réel d'où plusieurs définitions sont exprimées en mettre l'accent sur l'algorithme d'ordonnancement AAA de SynDEx.

Le quatrième chapitre expose un état de l'art sur les différents moyens et techniques pour assurer la tolérance aux fautes utilisées jusqu'à aujourd'hui.

Le cinquième et le sixième chapitres présente trois nouvelles heuristiques qui servent à optimiser d'une part la génération automatique de distribution/ordonnancement et d'autre part à tolérer les fautes permanentes d'un seul processeur durant un cycle d'exécution de l'architecture logicielle sur l'architecture matérielle. Les deux premières appelées AAA-FAULT^{DT} et FT-TDEP, essayent de

trouver une solution de tolérance aux fautes dans des architecture avec des tâches dépendantes en se basant sur la redondance passive et hybride. La seconde appelée AAA-FAULT^{IDT}, tolère les fautes permanentes d'un processeur unique dans une architecture avec des tâches indépendantes en utilisant la redondance active.

En conclusion, nous résumons nos contributions et présentons les perspectives de recherche future.

Chapitre 2

Introduction aux systèmes distribués temps réels embarqués

2.1. Introduction

Les systèmes informatiques qui interagissent avec l'environnement peuvent être classés en systèmes conversationnels et systèmes réactifs ou temps réel. Dans Les systèmes conversationnels l'ordinateur est maître de l'interaction et le client attend d'être servi comme les systèmes d'exploitation. Dans les systèmes réactifs c'est l'environnement qui est maître de l'interaction, c-à-d le rôle de l'ordinateur est de réagir de façon continue aux stimuli externes en produisant des réponses instantanées. On trouve ces systèmes dans les domaines où le client ne peut pas attendre à savoir les systèmes temps réel qui doivent fournir un service dans un contexte où le temps intervient, la plupart des systèmes temps réel sont embarqués dans un équipement spécialisé, leur but étant de contrôler l'équipement et/ou son environnement. L'évolution technologique et l'évolution des besoins en informatique (réseaux de télécommunications, performance des voies de télécommunication, communication, partage et accès universel à l'information, …) ont conduit à l'apparition des systèmes distribués dont lesquels les systèmes temps réel critiques s'appliquent mieux. Parmi les objectifs des systèmes distribués, la réplication de ressources non visible et donc la tolérance aux fautes qui permet à un utilisateur de ne pas s'interrompre à cause d'une panne d'une ressource.

Nous intéressons dans cette thèse aux systèmes distribués temps réel pour des applications embarquées, donc nous introduisons d'abord les caractéristiques de ces systèmes dans les différents travaux que nous menons.

2.2. Systèmes distribués

2.2.1. Définition

Plusieurs définitions sont données aux systèmes distribués : selon Tanenbaum [61], un système distribué est un ensemble d'ordinateurs indépendants qui apparaît à un utilisateur comme un système unique et cohérent ; selon Lamport [63], Un système réparti est un système qui vous empêche de travailler quand une machine dont vous n'avez jamais entendu parler tombe en panne. Nous donnons cette définition :

Définition 2.1 : un système distribué est défini comme étant un ensemble des ressources physiques et logiques géographiquement dispersées et reliées par un réseau de communication dans le but est de réaliser une tâche commune. Cet ensemble donne aux utilisateurs une vue unique des données du point de vue logique.

Les systèmes distribués ont plusieurs raisons de leur existence, à savoir le partage des ressources, l'accès distant, l'amélioration des performances, la confidentialité et la disponibilité des données en raison de l'existence de plusieurs copies, …etc.

2.2.2. Caractéristiques

La performance d'un système distribué se révèle dans ces caractéristiques. Nous citons ci-dessous les plus important dans notre travail.

2.2.2.1. L'interopérabilité

L'interopérabilité est une caractéristique importante qui désigne la capacité à rendre compatibles deux systèmes quelconques. En effet, l'interopérabilité vise à réduire le vrai problème de l'hétérogénéité en la masquant par l'utilisation d'un protocole unique de communication.

2.2.2.2. L'ouverture

Cette caractéristique fait mention de l'extensibilité dans la mesure où des composants peuvent être ajoutés, remplacés ou supprimés dans un système distribué sans en affecter les autres.

2.2.2.3. La tolérance aux fautes

Une faute peut être comprise comme une défaillance au sein du système pouvant conduire à des résultats erronés comme aussi engendrer l'arrêt de toute ou partie d'un système distribué. Les fautes peuvent résulter des différentes couches et se propager éventuellement aux autres. Elle peut être matérielle ou logicielle. Un système distribué doit être conçu pour masquer ce genre des fautes aux utilisateurs et ne doit pas perturber l'utilisation du système en termes de fonctionnalité.

2.3. Systèmes temps réel

2.3.1. Le concept de temps réel

Un système d'exploitation classique n'a pour seule contrainte de temps que celle d'un temps de réponse satisfaisant pour les utilisateurs avec lesquels il dialogue. En système temps réel, le concept de temps réel signifie l'aptitude d'un système d'exploitation de fournir le niveau de service requis au bout d'un temps de réponse borné, ainsi on ne doit pas confondre entre temps réel et rapidité. Un système temps réel n'est pas un système qui va vite mais un système capable de réagir à des stimuli externes dans des délais spécifiés (contraintes temporelles).

2.3.2. Les spécificités des systèmes temps réel

En informatique, on parle d'un système temps réel lorsque ce système est capable de contrôler un procédé physique à une vitesse adaptée à l'évolution du procédé contrôlé. Les système informatiques temps réel se différencient des autres systèmes informatiques par la prise en compte de contraintes temporelles dont le respect est aussi important que l'exactitude du résultat délivré [1].

2.3.2.1. Définitions

Plusieurs définitions des systèmes temps réel sont données dans la littérature [2, 3, 9], celle largement adoptée est la suivante [4] :

Définition 2.2 : *Le comportement d'un système informatique est qualifié de « temps réel » lorsqu'il est assujetti à l'évolution d'un procédé qui lui est connecté et qu'il doit piloter ou suivre en réagissant à tous ses changements d'états.*

De plus, un système temps réel est un système dont la correction ne dépend pas seulement des valeurs des résultats produits mais également des délais dans lesquels les résultats sont produits.

2.3.2.2. Catégories des systèmes temps réel

Selon le niveau de criticité des contraintes temporelles, les systèmes temps réel sont classés en trois catégories : système temps réel dur, système temps réel souple et système temps réel ferme.

- **Système temps réel dur ou critique** (hard real time en anglais) : Dans ces systèmes la réponse dans les délais est vitale. Le dépassement ou l'absence de réponse est catastrophique en termes de temps, d'argent ou plus grave de vie humaine.
- **Système temps réel souple** (soft real time en anglais) : Dans ces systèmes la réponse du système après les délais réduit progressivement son intérêt. Les pénalités ne sont pas catastrophiques.
- **Système temps réel ferme** (firm real time en anglais) : Dans ces systèmes la réponse du système dans les délais est essentielle. Le résultat ne sert plus à rien une fois le deadline passe.

Dans ce travail, nous intéressons aux système temps réel critiques (à contrainte stricte).

2.3.3. Les tâches temps réel

Un programme temps réel est composé d'un ensemble d'entités appelées tâches temps réel (task en anglais). Chacune ayant un rôle qui lui est propre, comme par exemple : réaliser un calcul, être associé à une alarme, traiter des entrées / sorties, …etc.

2.3.3.1. Définitions

Définition 2.3 : Une tâche temps réel se définit comme une séquence d'instructions constituant l'unité de base d'un système temps réel et destinées à être exécutées en séquence sur un ou plusieurs processeurs en respectant des contraintes de temps.

Dans un environnement multitâche, à tout instant, chaque tâche peut être dans l'un des états suivants [1] :

- **Elue ou en cours (running)** : c'est le cas de la tâche en train de s'exécuter. Quand il n'y a qu'un processeur, une seule tâche est en cours d'exécution à un instant donné (celle-ci est choisie selon la politique d'ordonnancement considérée et le mode d'exécution : préemptif ou non),
- **Prête (ready)** : c'est le cas d'une tâche en attente de la disponibilité de la ressource de traitement,
- **Suspendue (suspended)** : c'est le cas des tâches qui sont en attente d'évènements qui provoqueront leur réveil.

2.3.3.2. Caractérisation d'une tâche temps réel

Une tâche temps réel requière des besoins en ressources logiques ou matérielles pour quelle s'exécute, elle peut être soumise à des contraintes (autres que les contraintes temporelles) liées aux spécifications du système. Nous pouvons citer :

- **Les contraintes de précédence** : des relations de précédence peuvent exister entre les tâches d'où un ordre d'exécution de ces tâches s'impose. Dans ce cas les tâches sont dites dépendantes, elles sont dites indépendantes si aucune relation de précédence existe [62].
- **Les contraintes d'exécution** : il existe deux modes d'exécution, le mode préemptif et le mode non-préemptif. Dans le mode préemptif, une tâche peut être interrompue à un instant donné et être reprise ultérieurement. Dans le mode non-préemptif, une tâche s'exécute complètement sans interruption jusqu'à sa terminaison.
- **Les contraintes de placement** : Les tâches doivent s'exécuter sur un processeur quelconque, ou peuvent s'exécuter sur des processeurs différents.

Ainsi, selon la loi d'arrivée d'une tâche qui définit sa nature et qui s'agit des contraintes temporelles qui définissent la répartition des dates d'activation des instances d'une tâche dans le temps, il est possible de classer les tâches en trois catégories [5] :

- **Une tâche périodique** est une tâche dont l'activation est régulière et le délai Pi (période) entre deux activations successives est constant. Une tâche Ti est caractérisée par une durée d'exécution Ci, une période d'activation Pi, une échéance Di et la date de la première activation Ri

- **Une tâche sporadique** est une tâche caractérisée par un délai minimum entre deux activations successives. Au contraire des tâches périodiques, les dates d'activation des différentes instances d'une tâche sporadique ne peuvent pas être déterminées a priori.
- **Une tâche apériodique** est une tâche dont on ne connaît aucune caractéristique, sauf son échéance. Elle est généralement activée par l'arrivée des évènements (message ou requête de l'opérateur) qui peuvent être produits à tout instant.

Dans notre travail, nous considérons les tâches périodiques dépendantes et indépendantes qui s'exécutent en mode non-préemptif sur des processeurs distincts.

2.4. Systèmes embarqués

2.4.1. Définition

Les systèmes embarqués sont de plus en plus utilisés dans notre quotidien. En plus des ordinateurs et microordinateurs qui sont les systèmes embarqués les plus connus, l'électronique embarquée est utilisée sur de nombreux objets comme les avions, les trains et les voitures. L'objectif de l'utilisation des systèmes embarqués est de donner la possibilité de réagir à l'environnement. Alors un système embarqué peut être définit comme :

Définition 2.4 : Le système embarquée est un système électronique et informatique autonome, intégré à un autre objet pour y réaliser des tâches précises, souvent en temps réel.

Définition 2.5 *Un système embarqué est un système électronique et informatique destiné à une tâche spécifique intimement liée au procédé dans lequel il est intégré. A ce titre il s'agit d'un système réactif. Le terme désigne aussi bien le matériel informatique que le logiciel utilisé. On parle également de système enfoui et en anglais embedded system* [6].

Les grands secteurs de l'embarqué concernent les domaines suivants :

- Jeux et calcul général : application similaire à une application de bureau mais empaquetée dans un système embarqué : jeux vidéo, set top box...
- Contrôle de systèmes : automobile, process chimique, process nucléaire, système de navigation...
- Traitement du signal : radar, sonar, compression vidéo...
- Communication et réseaux : transmission d'information et commutation, téléphonie, Internet...

2.4.2. Caractéristiques principales d'un système embarqué

La conception d'un système embarqué doit respecter un certain nombre de caractéristiques, nous citons ci-dessous les plus importants :

- **La fiabilité** : probabilité que le système fonctionne correctement à partir du moment où c'est le cas à l'instant t = 0 ;

- **La maintenabilité** : probabilité que le système fonctionne correctement sur un laps de temps d'après une défaillance ;
- **La disponibilité** : probabilité que le système fonctionne à un instant t ;
- **La sûreté** : le système ne cause aucun dommage ;
- **La sécurité** : confidentialité et l'authenticité des informations.
- Un système embarqué doit être efficace d'un point de vue :
  - Energétique ;
  - Logiciel ;
  - Intégration (encombrement) ;
  - Autonomie (exécution) ;
  - Coût.
- Un système embarqué est destiné à une tâche à prendre en compte dès la conception du système (matériel et logiciel)
- Un système embarqué doit avoir une interface utilisateur dédiée (pas de clavier, pas d'écran, etc.)
- Un système embarqué :
  - Respecte des contraintes temporelles ;
  - Restitue une réponse déterministe ;
  - Est connecté à un environnement réel par l'intermédiaire de capteurs et d'actionneurs ;
  - Est un système hybride (combinaison de signaux analogiques et de signaux numériques) ;
  - Est un système réactif.

2.4.3. Classement des systèmes embarqués

Un premier classement des architectures pour système embarqué dépend du nombre de processeurs [7] : architecture monoprocesseur ou multiprocesseur. Pour les architectures multiprocesseurs, on trouve différents classements à savoir :

- Homogène/hétérogène selon la nature des processeurs de l'architecture :
  - Homogène : dans ce cas les processeurs sont **identiques,** c-à-d ils sont interchangeables et ils ont la même capacité de calcul ;
  - Hétérogène : les processeurs sont soit **indépendants,** c-à-d les processeurs ne sont pas destinés à exécuter les mêmes tâches, ou **uniformes,** c-à-d les processeurs exécutent les mêmes tâches mais chaque processeur à sa propre capacité de calcul.
- Homogène/hétérogène selon la nature des communications entre processeurs :
  - Homogène : les coûts de communication entre chaque paire de processeurs de l'architecture sont toujours les mêmes ;
  - Hétérogène : les coûts de communication entre processeurs varient d'une paire de processeurs à une autre ;
- Parallèle/distribuée selon le type de mémoire de l'architecture :
  - Parallèle : les processeurs communiquent par mémoire partagée ;

- Distribuée : les processeurs ne partagent pas de mémoire et ont leur propre mémoire qui est ainsi dite distribuée. Ils communiquent par envoi/réception de messages.

2.5. Systèmes distribués temps réel embarqués

2.5.1. Architecture des Systèmes distribués temps réel embarqués

La conception des systèmes embarqués nécessite une approche qui intègre les méthodes de conception matérielle, les méthodes de conception logicielles et la théorie de contrôle d'une manière cohérente. En suivant quelques méthodologies de conception des systèmes embarqués [8], on les partage en deux classes différentes : l'ingénierie des systèmes et la conception basée modèle.

Les nouvelles approches sont des approches de conception basée modèle (Model-Based Design ou MBD), elles se basent sur la conception conjointe matérielle-logicielle (Hardware-Software codesign) et elles soulignent la séparation du niveau de la conception du niveau de l'implémentation.

Les méthodes d'Adéquation Algorithme Architecture ou AAA et les méthodes de transformation de modèles du MBD sont apparues dans les années 90. Depuis le début des années 2000, il y a également une tendance vers la conception à base de plateforme (Platform-Based Design ou PBD).

Dans nos travaux, nous focalisons sur les méthodes AAA [8]. La méthodologie AAA vise le prototypage rapide et l'implantation optimisée d'applications distribuées temps réel embarquées, elle est fondée sur des modèles de graphes, autant pour spécifier les Algorithmes applicatifs et les Architectures matérielles distribuée, que pour déduire les implantations possibles en termes de transformations de graphes. L'Adéquation revient à résoudre un problème d'optimisation consistant à choisir une implantation dont les performances, déduites des caractéristiques des composants matériels, respectent les contraintes temps réel et d'embarquabilité.

2.5.2. Contraintes des Systèmes distribués temps réels embarqués

On peut distinguer deux contraintes principales temporelles et matérielles ainsi plusieurs d'autres comme les contraintes de distribution ou de placement.

2.5.2.1. Contraintes temporelles

Les systèmes temps réel peuvent avoir deux types de contraintes temporelles [10] : contraintes strictes (dures) et contraintes souples. Un système temps réel est dit à contraintes strictes quand une faute temporelle (non-respect d'une échéance, arrivée d'un message après les délais, irrégularité d'une période d'échantillonnage, dispersion temporelle trop grande dans un lot de mesures simultanées) peut avoir des conséquences catastrophiques du point de vue humain, économique ou écologique. Un système temps réel est à contraintes souples lorsque le non-respect de contraintes temporelles est acceptable. On admet alors de ne pas respecter certain pourcentage d'échéances par exemple. Dans les applications de contrôle/commande temps réel critiques, les traitements de capteurs/actionneurs et les traitements de commande de

procédés ne doivent pas avoir de gigue sur les entrées issues des capteurs et sur les sorties fournies aux actionneurs. Dans un tel système, une faute temporelle peut avoir des conséquences catastrophiques autant au plus qu'une faute de calcul.

Les contraintes temps-réel peuvent être de différents types. Dans un système temps réel critique, la réaction à chaque événement doit être bornée, c'est-à-dire que le temps de réponse à cet évènement ne doit jamais dépasser une certaine valeur critique appelée date d'échéance (deadline) ou contrainte temps réel (c_{tr}).

2.5.2.2. Contraintes matérielles

Un système embarqué doit satisfaire des contraintes matérielles comme la gestion de la mémoire où l'espace mémoire est généralement limité et la consommation énergétique qu'elle doit être la plus faible possible. Ces contraintes sont engendrées à cause des ressources limitées (espace mémoire, puissance, processeur, poids, taille, coût, énergie, …) qu'un système embarqué doit respecter.

Par exemple, par rapport à la vitesse des processeurs qui ne cesse de croitre, l'accès aux mémoires reste toujours plus lent ; malgré que la capacité mémoire dans les systèmes embarqués augmente, elle reste toujours insuffisante pour des applications qui deviennent de plus en plus complexes. La complexité des systèmes embarqués ne cesse d'augmenter, ce qui engendre des consommations d'énergie de plus en plus importantes. La consommation d'énergie est l'un des paramètres les plus importants de la conception et le développement des applications embarquées. L'utilisation de processeurs puissants qui tournent à des fréquences élevées génère des consommations d'énergie élevées [11].

2.5.2.3. Contraintes de placement

Les contraintes de placements (ou de distribution) définissent la possibilité d'affecter ou de placer une tâche ou une dépendance de données de l'algorithme sur un composant de l'architecture (processeurs ou média de communication). Ces contraintes de placement sont généralement liées aux capteurs et aux actionneurs, mais peuvent aussi dépendre de la présence ou non de co-processeurs spécialisés [11].

2.6. Conclusion

Ce chapitre présente des concepts de bases des systèmes temps réel, embarqués et distribués. Nous avons considéré dans ce travail les systèmes temps réel critiques, càd ceux qui doivent satisfaire les contraintes temporelles pour prévenir le système des différentes catastrophes possibles. Comme la caractéristique principale de ces systèmes est d’être sûr de fonctionnement, nous allons présenter dans le chapitre suivant des généralités sur la sureté de fonctionnement des systèmes temps réel embarqués et bien sûr des notions de bases sur le moyen de l’assurer entrepris dans notre thèse à savoir la tolérance aux fautes.

Chapitre 3

Sûreté de fonctionnement et Tolérance aux fautes

3.1. Introduction

La sûreté de fonctionnement (Dependability en anglais) est apparue comme une nécessité au cours du XXème, notamment avec la révolution industrielle. Le terme dependability est apparu dans une publicité sur des moteurs Dodge Brothers dans les années 1930. L'objectif de la sûreté de fonctionnement est d'atteindre l'idéal de la conception de système : zéro accident, zéro arrêt, zéro défaut et même zéro maintenance. Pour pouvoir y arriver, il faudrait tester toutes les utilisations possibles d'un produit pendant une grande période ce qui est impensable dans le contexte industriel voire même impossible à réaliser tout court. La sûreté de fonctionnement est un domaine d'activité qui propose des moyens pour augmenter la fiabilité et la sûreté des systèmes dans des délais et avec des coûts raisonnables [12]. L'un des moyens les plus utilisés dans la littérature pour assurer la sûreté de fonctionnement est la tolérance aux fautes.

Tous au long de ce chapitre, nous allons présenter en détaille les concepts fondamentaux caractérisant les deux fameux domaines en se concentrant sur ce qui nous concerne dans notre travail.

3.2. Sûreté de fonctionnement

3.2.1. Définition de la sûreté de fonctionnement

La sûreté de fonctionnement (SdF) est souvent appelée la science des défaillances; elle inclut leur connaissance, leur évaluation, leur prévision, leur mesure et leur maîtrise. Il existe de nombreuses définitions, de standards (qui peuvent varier selon les domaines d'application - nucléaire, spatial, avionique, automobile, rail . . .), nous prenons celle de J.C. Lapris [16].

Définition 3.1 : La sûreté de fonctionnement d'un système informatique est la propriété qui permet de placer une confiance justifiée dans le service qu'il délivre.

3.2.2. Coût de la sûreté de fonctionnement

Le coût d'un haut niveau de sûreté de fonctionnement est très onéreux. Le concepteur doit faire des compromis entre les mécanismes de sûreté de fonctionnement nécessaires et les coûts économiques. Le coût d'une défaillance peut être extrêmement élevé. Comme nous visons des systèmes embarqués, nous ne tenons compte qu'aux solutions logicielles.

3.2.3. Taxonomie

La sûreté de fonctionnement manipule un certain nombre de concepts que nous précisons dans cette partie en donnant des définitions précises. La sûreté de fonctionnement peut être vue comme étant composée des trois éléments suivants :

- **Attributs** : points de vue pour évaluer la sûreté de fonctionnement ;
- **Entraves** : évènements qui peuvent affecter la sûreté de fonctionnement du système;
- **Moyens** : moyens pour améliorer la sûreté de fonctionnement.

Ces notions sont résumées dans la figure 3.1

Sûreté de fonctionnement	**Attributs**	**Disponibilité**
		Fiabilité
		Sécurité-innocuité
		Confidentialité
		Intégrité
		Maintenabilité
	Entraves	**Fautes**
		Erreurs
		Défaillances
	Moyens	**Prévention des fautes**
		Tolérance aux fautes
		Elimination des fautes
		Prévision des fautes

Figure 3.1. Arbre de la sûreté de fonctionnement [15]

3.2.3.1. Attributs

Les attributs de la Sdf se définissent selon les applications auxquelles le système est destiné, la sûreté de fonctionnement peut être vue selon des attributs différents, mais complémentaires, qui permettent de la caractériser. Ce sont des aptitudes (ex. : fiabilité, disponibilité) que le système doit présenter ou vérifier avec un certain niveau [13].

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

D'autres attributs de sûreté de fonctionnement ont été identifiés comme par exemple la testabilité (le degré d'un composant ou d'un système à fournir des informations sur son état et ses performances), ou la diagnosticabilité (capacité d'un système à exhiber des symptômes pour des situations d'erreur) survivabilité (capacité d'un système à continuer sa mission après perturbation humaine ou environnementale) et ainsi de suite [12].

3.2.3.2. Entraves

Les entraves de la sureté de fonctionnement peuvent affecter le système et dégrader la sureté de fonctionnement. Elles consistent en la faute, l'erreur et la défaillance qui s'enchainent comme illustré dans la figure 3.2. Les définitions sont récursives car la défaillance d'un composant est une faute pour le système qui le contient.

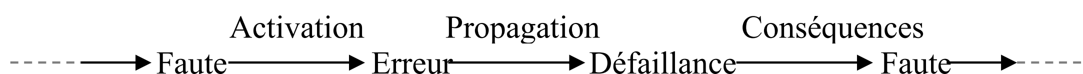


Figure 3.2. La chaine fondamentale des entraves à la sureté de fonctionnement [15]

Une défaillance (ou panne) est l'évènement qui survient lorsque le comportement du système dévie de sa fonction. L'erreur est la partie de l'état du système qui est susceptible d'entrainer une défaillance. La défaillance survient lorsque l'erreur affecte le service délivré à l'utilisateur. La faute est définie comme la cause adjugée ou supposée de l'erreur [14].

3.2.3.3. Moyens

Les moyens d'assurer la sureté de fonctionnement sont définis comme les méthodes utilisées pour assurer cette propriété. On distingue quatre méthodes principales [14].

- **La prévention des fautes** vise à empêcher l'apparition ou l'introduction des fautes dans le système. Elle repose sur des règles de développement (modularisation, utilisation de langage fortement typé, preuve formelle, etc.).

- **L'élimination des fautes** s'attache à réduire la présence (nombre, sévérité) des fautes. Cette méthode opère à la fois lors du développement (vérification des conditions, test de régressions, injection de fautes, etc.) ou lors de l'utilisation (maintenance).
- **La prévision des fautes** cherche à estimer (qualitativement et quantitativement) l'occurrence et les conséquences des fautes. Elle est réalisée par la modélisation et l'évaluation de systèmes.

Ces trois méthodes visent à empêcher l'occurrence de fautes, elles peuvent être vues comme constituant de l'évitement des fautes, càd comment tendre vers un système exempt de fautes.

Par opposition, la dernière méthode, tolérance aux fautes, vise à préserver le service malgré l'occurrence de fautes

- **La tolérance aux fautes** consiste à mettre en place des mécanismes qui maintiennent le service fourni par le système, même en présence de fautes. On accepte dans ce cas un fonctionnement dégradé.

L'objectif initial de ce travail est la tolérance aux fautes dans les systèmes temps réels embarqués donc, dans la suite de ce chapitre, nous allons présenter les différentes approches pour réaliser la tolérance aux fautes.

3.3. Tolérance aux fautes

Quelles que soient les précautions prises, l'occurrence de fautes est inévitable (erreur humaine, malveillance, vieillissement du matériel, catastrophe naturelle, etc.). Cela ne veut pas dire qu'il ne faut pas essayer de prévenir ou d'éliminer les fautes, mais les mesures prises peuvent seulement réduire la probabilité de leur occurrence [22]. Il faut donc concevoir les systèmes de manière à ce qu'ils continuent à rendre le service attendu (éventuellement un service dégradé) même en présence de fautes par la destruction de la chaine décrite à la figure 3.2, qui conduit de la faute à la défaillance. Avant de décrire les techniques de la tolérance aux fautes, nous représentons d'abord les différentes classes de ces trois menaces.

3.3.1. Taxonomie des fautes, des erreurs et des défaillances

3.3.1.1. Fautes

Les sources de faute, qui est la cause de l'apparition de l'erreur, sont extrêmement diverses. Elles sont classées selon huit critères : phase de création ou d'occurrence, frontières du système, cause phénoménologique, dimension, intention, capacité et persistance. La combinaison pertinente de ces critères permet de donner un classement exhaustif de tous les types de fautes. Pour simplifier, on peut les regrouper en trois grandes classes non exclusives [20] :

- **Les fautes de développement** : sont celles qui peuvent survenir durant le développement ;
- **Les fautes physiques** : sont celles qui affectent le matériel ;

- **Les fautes d'interactions** : elles rassemblent les fautes externes, c'est-à-dire celles qui sont localisées à l'extérieur des frontières du système et qui propagent des erreurs à l'intérieur du système par interaction ou interférence.

3.3.1.2. Erreurs

Une erreur a été définie comme étant susceptible de provoquer une défaillance. Qu'une erreur conduise ou non à défaillance dépend de trois facteurs principaux [16]:

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

3.3.1.3. Défaillances

Un système ne défaille généralement pas toujours de la même façon, ce qui conduit à caractériser les modes de défaillances selon trois points de vue : leurs domaines, leur perception par les utilisateurs, et leurs conséquences sur l'environnement. La figure 3.3 résume les modes de défaillance selon J.C. Laprie [21].

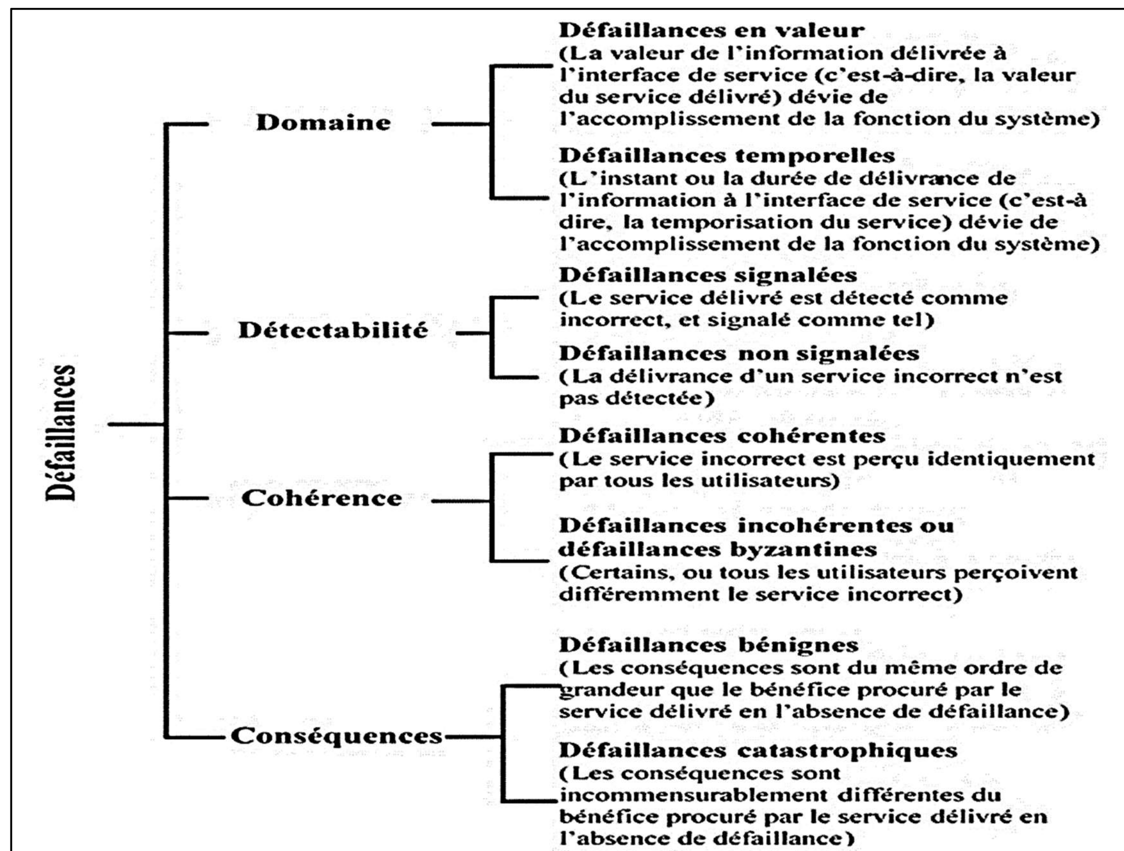


Figure 3.3 Défaillances du service

Le domaine de défaillance conduit à distinguer [16] :

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

Un cas particulier est celui de défaillance franche, dit aussi arrêt sur défaillance (fail stop) : ou bien le système fonctionne, et donne un résultat correct, ou bien il est en panne (défaillant), et ne fait rien [17]. Fail stop est le modèle de faute étudié dans ce travail.

Quand un système a plusieurs utilisateurs, il faut distinguer, selon leur perception des défaillances :

- **Les défaillances cohérentes :** tous les utilisateurs ont la même perception des défaillances;
- **Les défaillances incohérentes :** les différents utilisateurs peuvent avoir différentes perceptions d'une défaillance donnée ; une défaillance incohérente est généralement dénommée défaillance byzantine [18].

Ainsi les défaillances n'ont pas un comportement uniforme dans le temps [22]

- **Défaillance transitoire** : se produit de manière isolée

- **Défaillance intermittente** : se reproduit sporadiquement
- **Défaillance permanente** : persiste indéfiniment (jusqu'à réparation) après son occurrence

Les défaillances non permanentes sont difficiles à modéliser et à traiter, dans ce travail nous considérons les défaillances permanentes.

3.3.2. Etapes de la tolérance aux fautes

On peut distinguer plusieurs phases successives, non obligatoirement toutes présentes :

- La détection a pour rôle de découvrir l'existence d'une erreur (état incorrect) ou d'une défaillance (comportement incorrect)
- La localisation qui identifier le point précis (dans l'espace et le temps) où l'erreur (ou la défaillance) est apparue
- L'isolation pour confiner l'erreur pour éviter sa propagation à d'autres parties du système
- La réparation qui permet de remettre le système en un état de fournir un service correct

3.3.3. Techniques de tolérance aux fautes

La tolérance aux fautes est mise en œuvre par la détection des erreurs et le rétablissement du système [15]. La figure 3.4 liste les techniques de tolérance aux fautes.

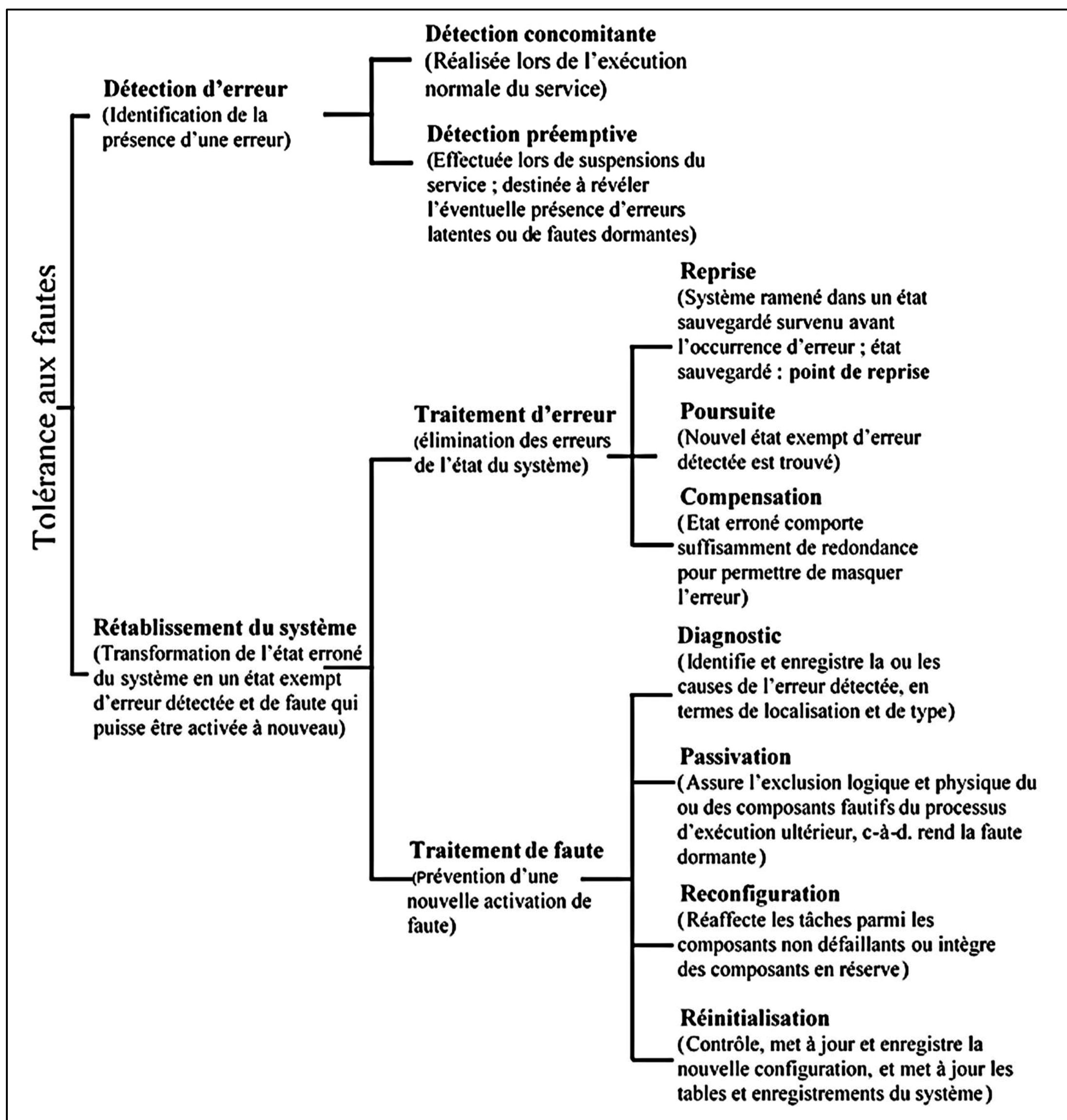


Figure 3.4. Techniques de la tolérance aux fautes [15]

Le traitement d'erreur est, à la demande, suivi du traitement de faute constituent le rétablissement du système, d'où le qualificatif de la stratégie de tolérance aux fautes correspondante : détection et rétablissement. Habituellement le traitement de faute est complété par des opérations de maintenance corrective, afin d'éliminer les composants passivés. Reprise et poursuite sont invoquées à la demande, après qu'une ou plusieurs erreurs aient été détectées, alors que la compensation peut être appliquée à la demande ou systématiquement, indépendamment de la présence ou de l'absence d'erreur. Détection et traitement d'erreur préemptifs sont des pratiques courantes lors de l'initialisation, d'un système informatique.

3.3.3.1. Détection d’erreur

La détection d’erreur est basée sur une redondance qui peut prendre plusieurs formes: redondance au niveau information ou composant, redondance temporelle ou algorithmique. Dans les techniques de détection concomitante, la forme la plus sophistiquée de détection d’erreur consiste à construire des composants autotestables en adjoignant au composant purement fonctionnel des éléments de contrôle permettant de vérifier que certaines propriétés entre les entrées et les sorties du composant sont satisfaites [15]. Les formes de détection d’erreur les plus couramment utilisées sont les suivantes :

- Codes détecteurs d’erreur ;
- Doublement et comparaison ;
- Contrôles temporels et d’exécution ;
- Contrôles de vraisemblance ;
- Contrôles de données structurées.

3.3.3.2. Rétablissement du système

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

a) Reprise

La reprise représente la technique la plus fréquemment utilisée pour assurer le rétablissement du système. Elle consiste en la sauvegarde périodique de l’état du système de façon à pouvoir, après avoir détecté une erreur, ramener le système dans un état antérieur, appelé « point de reprise ». La sauvegarde périodique de l’état du système doit s’effectuer au moyen d’un mécanisme de mémorisation, ou « support stable » qui protège les données contre les effets des fautes.

b) Poursuite

Le rétablissement par poursuite constitue une approche alternative ou complémentaire à la reprise, après avoir détecté une erreur, et après avoir éventuellement tenté une reprise, la poursuite consiste en la recherche d’un nouvel état acceptable pour le système à partir duquel celui-ci pourra fonctionner (éventuellement en mode dégradé).

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

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

c) Compensation

La compensation d'erreur nécessite que l'état du système comporte suffisamment de redondance pour permettre, en dépit des erreurs qui pourraient l'affecter, sa transformation en un état exempt d'erreur. Avec la compensation, il n'est pas nécessaire de réexécuter une partie de l'application (cas de la reprise) ou d'exécuter une procédure dédiée (cas de la poursuite) pour permettre de continuer le traitement fonctionnel. La compensation d'erreur peut être initialisée par une détection d'erreur (détection et compensation), être systématique (masquage) ou une dernière possibilité de compensation est fournie par les codes correcteurs d'erreur, notamment pour la transmission et ou le stockage de l'information.

- **Détection d'erreur et compensation**

Un exemple typique de la détection et compensation d'erreur est l'utilisation des composants autotestables exécutant en redondance active le même traitement ; en cas de défaillance de l'un d'entre eux, il est déconnecté et le traitement se poursuit sans interruption sur les autres. La compensation, dans ce cas, se limite à une commutation éventuelle de composants [15].

- **Masquage de faute**

Le masquage de faute est une méthode de compensation d'erreur où la compensation est effectuée de manière systématique, sans détection préalable d'erreur. Un exemple typique est celui du vote majoritaire : les traitements sont exécutés par au moins trois composants identiques dont les sorties sont votées ; les résultats majoritaires sont transmis, les résultats minoritaires (supposés erronés) sont éliminés. Comme le vote est appliqué systématiquement, le traitement, et par conséquent le temps d'exécution, sont identiques qu'il y ait ou non erreur. C'est ce qui différencie le masquage de la technique de détection et compensation [15].

- **Codes correcteurs d'erreurs**

Le principe de codage de l'information, introduite à la section sur les codes détecteurs d'erreur afin de détection d'erreur, peut également être utilisé pour construire des codes correcteurs d'erreur (Wakerly 1978). La correction d'erreur nécessite cependant une distance de Hamming plus grande entre les symboles (mots) du code. Pour corriger une erreur simple, il faut que la distance de Hamming soit supérieure ou égale à 3 (au lieu de 2 pour la détection) [15].

Il n'existe pas de méthodes de tolérance aux fautes valables dans l'absolu, seulement des méthodes adaptées à des hypothèses particulières d'occurrence de fautes. Ces hypothèses doivent donc être explicitement formulées. Dans tous les cas, le principe de la redondance est utilisé.

3.3.4. Techniques de redondance

Les techniques de redondance mises au point pour protéger les systèmes contre les défaillances peuvent revêtir deux familles de redondance : 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 3.5 résume les différents types de redondance distingués [24].

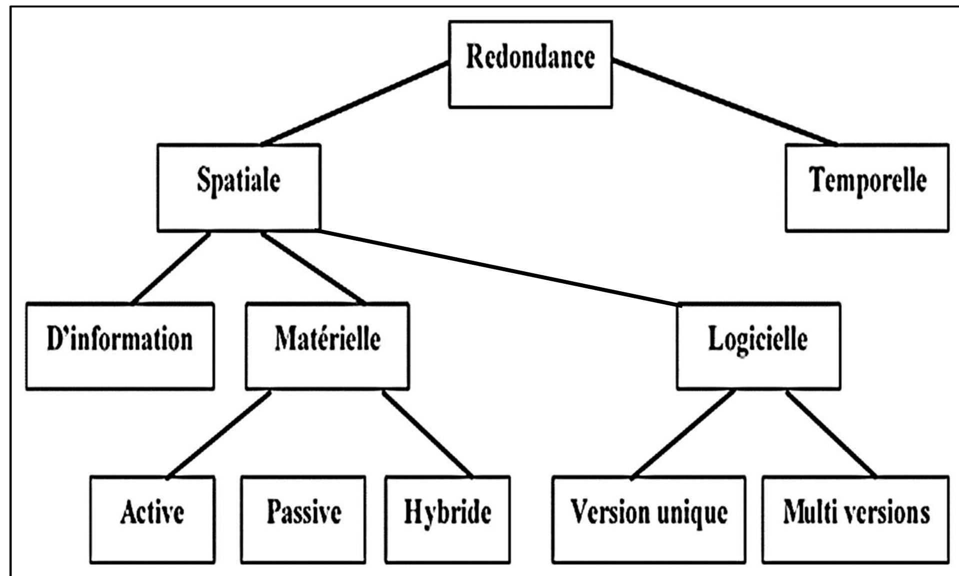


Figure 3.5. Différents types de redondance dans les systèmes embarqués critiques

3.3.4.1. Redondance spatiale

La redondance spatiale consiste à déployer en local ou d'une façon distribuée plusieurs copies d'un même composant [24]. 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 données). 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.

- **Redondance matérielle**

La redondance matérielle consiste à déployer un nombre N (N>=2) de copies physiques d'un composant matériel (processeur, 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. Les techniques de la redondance matérielle ont été classées en deux catégories :

redondance statique et redondance dynamique [23], redondance hybride est considérée comme une troisième catégorie.

- **Redondance logicielle**

La redondance logicielle 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 [24].

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 d'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 [24].

- **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 »). Ces extras données peuvent être regroupées dans un bloc à part des données, ou insérées/entrelacées dans les données.

3.3.4.2. Redondance temporelle (d'exécution)

La redondance temporelle est définie par le fait qu'un même composant exécute la même tâche plusieurs fois dans le temps [24]. Dans les systèmes critiques, la redondance temporelle est couramment déployée dans l'objectif d'assurer l'intégrité des communications. Elle consiste à envoyer à des instants distincts plusieurs copies du même message. 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. Il est généralement utilisé avec des techniques dynamiques de redondance matérielle et logicielle [23]. Les deux objectifs distincts de la redondance de temps sont la détection de fautes au moyen d'exécutions répétés ; et la récupération par le redémarrage du programme ou les tentatives d'opération après la détection ou la reconfiguration d'erreur.

3.3.5. Tolérance aux fautes logicielles et matérielles

La défaillance d'un système peut être à cause d'une faute matérielle ou d'une faute logicielle. Dans chaque type, des techniques différentes de tolérance aux fautes peuvent être utilisées.

- Pour la tolérance aux fautes logicielles, il existe deux techniques de base : uni-version et multi-versions du logiciel qui sont expliquées précédemment.
- Pour la tolérance aux fautes matérielles qui est le domaine le plus développé, beaucoup de techniques ont été développées et utilisées. Ces techniques sont basées sur des solutions matérielles ou logicielles ; sachant que la redondance est la méthode utilisée dans tous les cas, les solutions matérielles ont beaucoup d'inconvénients en termes de coût, d'énergie, de taille, …etc. Alors on opte généralement pour les solutions logicielles.

Dans ce travail nous considérons que les solutions logicielles pour tolérer les fautes matérielles.

3.3.6. Redondances logicielles pour la tolérance aux fautes matérielles

Les méthodes de tolérance aux fautes matérielles diffèrent selon l'origine des fautes matérielles (processeur, média de communication, capteurs et actionneurs, etc.), et le type de fautes pris en compte (fautes transitoires, fautes permanentes, etc.). La redondance logicielle nécessite une architecture distribuée afin de pouvoir allouer les répliques sur des processeurs différents. C'est la technique de tolérance aux fautes matérielles la plus adaptée aux systèmes embarqués car dans ces systèmes le nombre de composants matériels ne peut pas être facilement augmenté [10].

On distingue dans la littérature trois formes de redondance logicielle : redondance active, passive et hybride (mélange des deux première).

- **Redondance active**

La redondance active est basée sur la réplication des composants logiciels de l'algorithme. Toutes les répliques sont exécutées de sorte à ce qu'en présence de fautes il y'en ait toujours au moins une qui soit exécutée [10]. Le nombre n de répliques de chaque composant logiciel dépend directement du nombre de fautes k et aussi du type de ces fautes. Par exemple, pour tolérer au plus k fautes permanentes de processeurs, chaque composant logiciel est répliqué en une réplique primaire et en k répliques de sauvegardes, d'où $n = k + 1$.

- **Redondance passive**

La redondance passive est basée aussi sur la réplication des composants logiciels de l'algorithme. A la différence de la redondance active une seule des ré- pliques de chaque composant logiciel, appelée réplique primaire, est exécutée. Alors que les autres répliques, appelées répliques de sauvegardes, ne seront exécutées que si une faute provoque une erreur puis une défaillance du composant matériel implantant la réplique primaire. Le nombre n de répliques de chaque composant logiciel dépend directement du nombre de fautes k et aussi du type de ces fautes. Par exemple, pour

tolérer au plus k fautes permanentes de processeurs, chaque composant logiciel est répliqué en une réplique primaire et en k répliques de sauvegardes, d'où n = k + 1 [25].

- **Redondance hybride**

La redondance hybride est une combinaison de la redondance active et passive des composants logiciels. Par exemple, pour tolérer une faute permanente d'un processeur ou d'un médium de communication, on utilise la redondance active pour les composants logiciels de l'algorithme et la redondance passive pour les communications [25].

Comparaison entre les trois types de redondance

Le tableau 3.1 montre une comparaison entre les différentes approches de tolérance aux fautes basées sur la redondance active, passive ou hybride des composants logiciels d'un algorithme [10].

Critère de comparaison	Redondance active	Redondance passive	Redondance hybride
Surcoût	Un surcoût élevé	Un surcoût moins élevé	Le surcoût dépend du niveau de la réplication active par rapport à la réplication passive
Traitement de défaillances (Temps de réponse)	Un temps de réponse prévisible, et généralement rapide dans des architectures offrant un taux élevé de parallélisme	Meilleur temps de réponse en absence de défaillances. La défaillance de la réplique primaire peut de manière significative augmenter le temps de réponse	Le temps de réponse dépend du niveau de la réplication active par rapport à la réplication passive
Reprise après défaillance	Immédiate	Non immédiate	Non immédiate

Table 3.1. Comparaison entre les trois approches de redondance

3.4. Conclusion

Le but de ce chapitre a été d'introduire les concepts de sûreté de fonctionnement dans les systèmes temps réels embarqués critiques. Dans l'accomplissement de cet objectif, nous avons introduit les principales notions sur la tolérance aux fautes dans ces systèmes. Nous avons présenté différentes classes de fautes, d'erreurs et de défaillances. Nous avons introduit aussi les techniques de tolérance aux fautes et plus spécialement la redondance logicielle des fautes matérielles.

Chapitre 4

Introduction à l'ordonnancement temps réel

4.1. Introduction

Dans un système distribué temps réel multitâche, les tâches ont un accès concurrent aux processeurs. De ce fait, le rôle essentiel du système est d'organiser l'exécution des tâches dans le temps et de gérer leur concurrence au sein de chaque processeur. Ce processus est appelé ordonnancement temps réel des tâches. Cet ordonnancement doit garantir la satisfaction des contraintes de temps imposées à l'exécution de l'application. Le système est dit ordonnançable lorsque toutes les contraintes de temps sont respectées. Nous définissons un ordonnanceur comme étant un composant (un programme) permettant de choisir l'ordre d'exécution des tâches sur un processeur selon une stratégie appelée politique d'ordonnancement [26]. Il existe deux grandes classes d'ordonnanceurs :

- **Les ordonnanceurs en temps partagé** : Ils permettent d'attribuer les ressources de la manière la plus équitable possible entre les différents traitements dans le système.
- **Les ordonnanceurs temps réel** : Ils permettent de garantir le respect des échéances pour chaque traitement.

Dans le cadre de la thèse, nous nous intéressons aux ordonnanceurs temps réel. Nous présentons d'abord quelques terminologies pour bien assimiler les définitions décrites.

- **Ordonnancement temps réel** : Le problème d'ordonnancement met en relation des tâches à exécuter, des machines pour les exécuter et le temps [9]. Il détermine l'ordre d'exécution des tâches selon les critères (contraintes) spécifiés (échéance, temps de réponse, durée d'exécution, contraintes d'enchainement, …). L'ordonnancement est dit faisable s'il respecte toutes les contraintes temporelles.

- **Ordonnancement statique et dynamique** : dans l'ordonnancement statique, la séquence d'exécution des tâches est déterminée avant le début d'exécution de l'algorithme et ne change pas durant cette exécution, tandis que dans l'ordonnancement dynamique la séquence déterminée au préalable est mise à jour et réordonnancée en fonction des nouvelles tâches créées.
- **Allocation statique et dynamique** : l'allocation statique consiste à placer un ensemble de tâches sur un réseau de processeurs avant leur exécution, en respectant et optimisant certains critères. Par contre l'allocation dynamique désigne le placement des tâches créées durant l'exécution.
- **Fonction de coût** : Le rôle d'une fonction de coût est d'associer un poids à chaque composant logiciel, et ceci afin de calculer un ordre d'exécution entre ces composants.
- **Algorithme de distribution/ordonnancement temps réel** : c'est l'algorithme qui permet d'ordonnancer et de placer les tâches sur les processeurs en utilisant la fonction de coût dans un environnement distribué pour optimiser les performances du système temps réel.

Pour des raisons de lisibilité, parfois nous utilisons dans la suite de cette thèse le terme de « distribution/ordonnancement » au lieu de « distribution et ordonnancement temps réel »

4.2. Typologie des algorithmes d'ordonnancement

En général, les ordonnanceurs sont classifiés selon des caractéristiques du système sur lequel ils sont implantés. On aura donc différentes typologies parmi lesquelles :

4.2.1. Monoprocesseur ou multiprocesseur

L'ordonnancement est de type monoprocesseur si toutes les tâches ne peuvent s'exécuter que sur un seul et même processeur. Si plusieurs processeurs sont disponibles dans le système, l'ordonnancement est de type multiprocesseur.

4.2.2. Hors-ligne ou en-ligne

Les algorithmes d'ordonnancement peuvent être classés en deux catégories, à savoir les algorithmes d'ordonnancement hors-ligne et en-ligne :

- **Dans l'ordonnancement hors-ligne** une séquence fixe d'exécution des tâches est établi avant le lancement de l'application à partir de toutes les différentes caractéristiques et contraintes de celles-ci. Ensuite cette séquence est rangée dans une table et exécutée en ligne par le processeur.
- **Un ordonnanceur en-ligne** consiste à construire une séquence d'exécution des tâches dynamiquement en fonction des événements qui surviennent. Cependant il s'appuie sur des données collectées suite à une analyse préalable du système effectuée hors-ligne afin d'assurer le respect des contraintes temporelles des tâches.

4.2.3. Préemptif ou non-préemptif

Un ordonnanceur est préemptif si l'exécution de toute tâche peut être interrompue pour réquisitionner le processeur au profit d'une autre tâche jugée plus urgente ou plus prioritaire. L'ordonnancement est dit non préemptif si une fois lancée la tâche en cours d'exécution ne peut pas être interrompue avant la fin de son exécution en dépit du réveil d'une tâche plus prioritaire. Dans le cas d'ordonnanceurs monoprocesseur non-préemptifs, en l'absence de préemption, il ne peut y avoir d'accès concurrent aux ressources c'est-`a-dire qu'à partir du moment où une tâche est élue, celle-ci continue son exécution sans pouvoir être interrompue.

4.2.4. Oisif ou non oisif

Un ordonnanceur est dit non oisif lorsqu'il possède la propriété suivante : à partir du moment où au moins une tâche est prête et que la ressource processeur est libre, alors l'ordonnanceur élit forcément une tâche et cette dernière commence son exécution sans attendre. L'ordonnanceur fonctionne alors sans insertion de temps creux. Dans le cas contraire, si l'ordonnanceur est oisif, lorsqu'une tâche est prête, elle peut attendre un certain temps avant d'être élue même si la ressource processeur est libre. On dit que l'ordonnanceur fonctionne par insertion de temps creux.

4.2.5. Centralisé ou distribué

Un ordonnancement est distribué si les décisions d'ordonnancement sont prises par un algorithme localement placé en chaque nœud. Il est centralisé lorsque l'algorithme d'ordonnancement pour tout le système, distribué ou non, est déroulé sur un nœud privilégié.

4.3. Propriétés des algorithmes d'ordonnancement

Le choix d'un algorithme d'ordonnancement dépend essentiellement du contexte défini par les différentes caractéristiques du système temps réel sur lequel il sera mis en œuvre [27]. Nous citons dans ce qui suit, quelques propriétés et définitions utilisées dans l'analyse d'ordonnançabilité et de faisabilité d'une application temps réel.

- **Validité :** Une séquence d'ordonnancement d'une configuration de tâches est valide si et seulement si toutes les échéances des tâches sont respectées.
- **Faisabilité :** Une configuration de tâches est dite faisable s'il existe au moins une séquence d'ordonnancement dans laquelle toutes les tâches respectent bien leur échéance.
- **Ordonnançabilité :** Une configuration de tâches est dite ordonnançable si et seulement si, il existe une séquence d'ordonnancement valide de longueur infinie.
- **Optimalité :** L'algorithme d'ordonnancement A est dit optimal s'il est capable d'ordonnancer toute configuration de tâches faisable, càd si un ordonnanceur optimal ne parvient pas à construire une séquence valide pour une configuration de tâches donnée, alors aucun autre ordonnanceur ne pourra trouver une séquence valide pour cette même configuration.
- **Test d'ordonnançabilité :** Un test d'ordonnançabilité permet de déterminer, avant exécution, si un ordonnanceur donné pourra fournir une séquence

d'ordonnancement valide pour une configuration de tâches donnée. Alors, Concevoir une application de type temps réel dur impose donc avant son démarrage de vérifier que l'application est réalisable.

- **Un algorithme est déterministe** s'il ne fait intervenir aucune composante aléatoire dans la prise de décisions d'ordonnancement. Par conséquent, une configuration de tâches périodiques ordonnancée plusieurs fois par un même algorithme déterministe présentera la même planification de tâches.

4.4. Complexité des algorithmes d'ordonnancement

L'efficacité d'un algorithme d'ordonnancement n'est pas uniquement évaluée en termes de métriques de performance de celui-ci mais aussi en fonction de sa simplicité de mise en œuvre et/ou de calcul, on parle donc de complexité [1]. Celle-ci est calculée en évaluant la quantité de ressources en temps (nombre d'instructions) et en espace (mémoire) nécessaire pour la résolution de problèmes au moyen de l'exécution d'un algorithme.

Dans l'étude de la complexité, il faut distinguer deux catégories de problèmes : problèmes de décision et problèmes d'optimisation.

Un problème de décision est un problème dont la réponse est soit oui soit non. Les travaux théoriques dans ce domaine ont permis d'identifier différentes classes de problèmes de décision en fonction de la complexité de leur résolution [1] :

- Un problème de décision appartient à la classe P, s'il peut être résolu par un algorithme polynomial en n.
- Un problème de décision appartient à la classe NP (Non-déterministe Polynomial) s'il peut être résolu par un algorithme non déterministe polynomial capable de vérifier la validité d'une solution donnée.
- Les problèmes les plus difficiles de classe NP appartiennent à la classe des problèmes NP-complets dans le sens où l'on ne trouve pas d'algorithme polynomial pour les résoudre avec une machine déterministe.

Un problème d'optimisation est un problème pour lequel on doit chercher à déterminer une solution en vue d'optimiser un critère. A chaque problème d'optimisation, un problème de décision peut être associé. Plus précisément, lorsque le problème de décision est NP-complet, le problème d'optimisation est dit NP-difficile.

Lorsqu'on aborde la résolution d'un problème d'ordonnancement, on peut choisir entre deux grands types de stratégies, visant respectivement à l'optimalité des solutions par rapport à un ou plusieurs critères, ou plus simplement à leur admissibilité vis-à-vis des contraintes [28]. Une méthode utilisant un critère d'optimisation est exacte si elle garantit l'optimalité des solutions trouvées ; sinon elle est dite approchée, ou heuristique, lorsqu'on observe empiriquement qu'elle fournit de « bonnes » solutions.

Les algorithmes hors-ligne et en-ligne qui trouvent une solution optimale, si celle-ci existe, appartiennent à la classe des algorithmes exacts puisqu'ils renvoient toujours une solution optimale. Cependant, dans le cas général, ce problème est NP-difficile et

la complexité est exponentielle. Trouver une solution optimale pour le problème de distribution/ordonnancement n'est pas usuellement une contrainte d'un système temps réel embarqué à contraintes strictes. C'est pourquoi, pour résoudre ce problème dans un temps polynomial, plusieurs heuristiques ont été développées dans la littérature pour chercher une solution valide et si possible proche de la solution optimale. Ces algorithmes de distribution et d'ordonnancement appartiennent à la classe des algorithmes approchés.

Parmi tous les algorithmes proposés dans la littérature, nous intéressons dans ce travail aux algorithmes hors/ligne approchés basés sur l'algorithme AAA[1] implanté dans l'outil SynDEx[2]. SynDEx est un environnement graphique interactif de développement pour applications temps réel.

4.5. Présentation de la méthodologie AAA développée à l'INRIA Rocquencourt

La méthodologie "Adéquation Algorithme-Architecture" (AAA) cherche à produire de façon automatique des exécutifs optimisés distribués temps réel pour les composants processeurs des architectures mono ou mufti-processeurs. Cette méthodologie, développée à l'INRIA-Rocquencourt (Projet SOSSO), part d'une spécification algorithmique de l'application, fournie par l'utilisateur, sous la forme d'un graphe flot de données. Pour générer l'exécutif optimisé, la méthodologie prend en compte un modèle d'architecture composée d'un ou plusieurs processeurs, sur lequel on doit effectuer l'implantation matérielle de la spécification algorithmique.

Cette méthodologie AAA de prototypage rapide optimisé est fondée sur une approche globale, formalisant l'algorithme, l'architecture et l'implantation, à l'aide d'un modèle unifié de graphes factorisés (voir figure 4.1).

L'implantation d'un algorithme sur une architecture est un problème d'allocation de ressources. L'allocation statique est la plus appropriée pour l'implantation d'algorithmes réactif temps réel sur des systèmes embarqués à contrainte stricte. Dans ce cas, l'algorithme et son environnement sont bien connus, les ressources matérielles sont normalement limitées et les contraintes temporelles sont très sévères. En tenant compte de ces caractéristiques, AAA utilise l'ordonnancement statique (même les communications sont routées et ordonnancées statiquement).

L'adéquation cherche à mettre en correspondance l'algorithme et l'architecture, en réduisant le parallélisme potentiel de l'algorithme au parallélisme disponible de l'architecture et en transformant le graphe logiciel correspondant à l'algorithme en un graphe matériel correspondant à l'architecture qui implante l'algorithme, comme le montre la figure suivante.

[1] Adéquation Algorithme Architecture pour l'implantation optimisée d'application distribuées temps réel embarquées

[2] SynDEx est un environnement graphique interactif de développement pour applications temps réel. http ://www.syndex.org

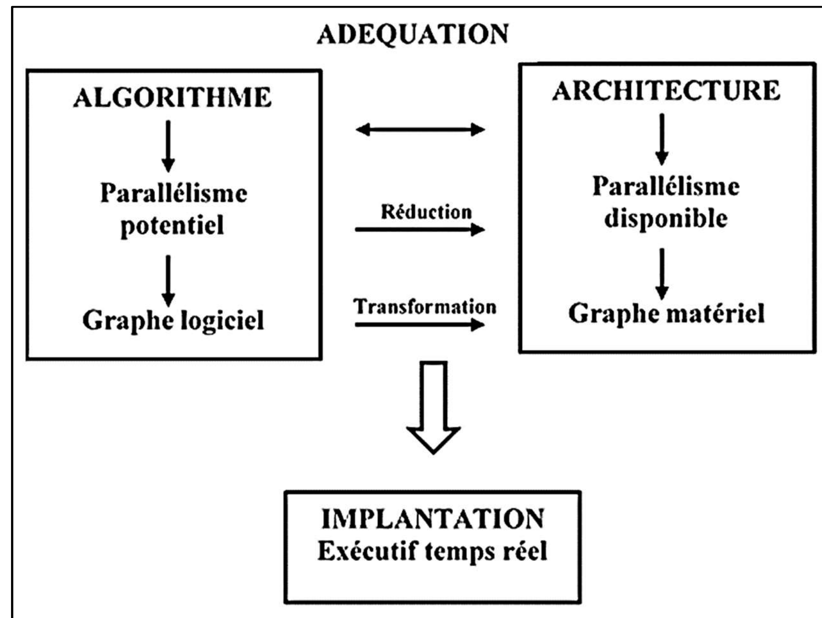


Figure 4.1. Méthodologie AAA [29]

4.5.1. Modèle de l'algorithme

Un algorithme est une séquence finie d'opérations directement exécutable par une machine à états finie. Cette définition doit être étendue afin de permettre d'une part la prise en compte du parallélisme disponible dans les architectures distribuées, composées de plusieurs machines à états finies interconnectées, et d'autre part la prise en compte de l'interaction infiniment répétitive de l'application avec son environnement [29].

Le modèle d'algorithme utilisé est un graphe de dépendances factorisé conditionné: c'est un hypergraphe orienté acyclique, dont les sommets sont des tâches partiellement ordonnées (exprimant le parallélisme potentiel de l'algorithme) par leurs dépendances de données. Ce graphe de dépendance, appelé graphe acyclique orienté, exprime le parallélisme potentiel de l'algorithme : deux opérations qui ne sont pas en relation de dépendance de données peuvent être exécutées dans n'importe quel ordre par le même opérateur ou simultanément par des opérateurs différents.

Nous nous intéressons à des systèmes réactifs qui interagissent constamment avec l'environnement qu'ils contrôlent. Ainsi, les tâches nécessaires au calcul des événements de sortie pour les actionneurs, à partir des événements d'entrée acquis par les capteurs, sont exécutées à chaque interaction avec l'environnement.

L'algorithme est donc modélisé par un graphe de dépendances, infiniment large mais périodique, réduit par factorisation à son motif répétitif, appelé graphe flot de données (graphe algorithmique) [29]. La factorisation de ce motif permet de transformer la répétition en itération, outre la réduction du volume de la spécification. Dans le cas d'une implantation multiprocesseurs, la distribution des motifs est faite pour que chaque processeur exécute une partie de l'itération.

Une tâche peut être, soit un calcul, soit une entrée-sortie. Les sommets qui ne possèdent pas de prédécesseur sont les interfaces d'entrée (capteurs recevant des

stimuli de l'environnement) et ceux qui ne possèdent pas de successeur représentent les interfaces de sortie (actionneurs produisant les réactions vers l'environnement). Dans le cas d'une tâche de calcul, la consommation des entrées précède la production des sorties. Dans le cas d'un conditionnement, la sortie n'est produite que si l'entrée booléenne est vraie.

Le graphe présenté ci-dessous montre un exemple de la modélisation de l’algorithme où les nœuds nommés Ti sont les tâches accomplies par le système, et les arcs entre ces nœuds sont les dépendances de données qui expriment un ordre partiel d’exécution sur les tâches.

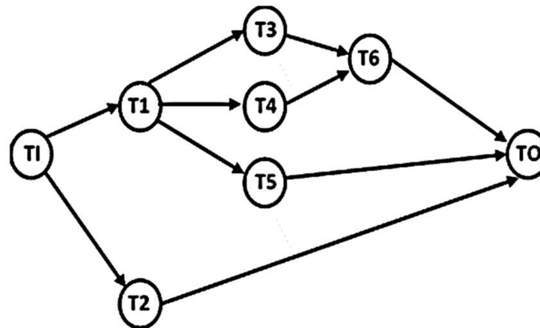
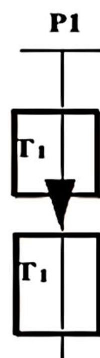


Figure 4.2. Exemple d’un modèle d’algorithme

Dans le graphe d’algorithme, une tâche T_i est définie par ces caractéristiques temporelles (temps d’exécution sur chaque processeur). Les arcs sont valués par la quantité de données échangées. Il y a deux types de communication entre tâches :

- **Communication locale sans délai** appelé, communication intra-processeur (voir figure 4.3) : dans ce cas, les tâches sont sur un même processeur, et utilisent sa mémoire pour communiquer.
- **Communication distante avec délai** appelé, communication inter-processeurs (voir figure 4.4) : dans cette communication, les tâches sont placées sur des processeurs distincts, et utilisent le réseau pour communiquer.



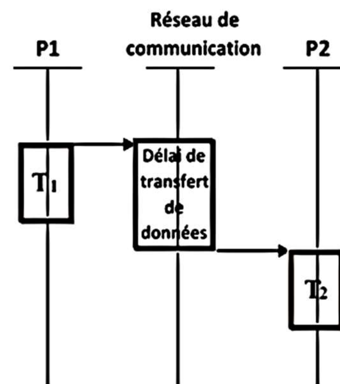


Figure 4.3. Communication intra-processeurs Figure 4.4. Communication inter-processeurs

4.5.2. Modèle d'architecture

Le PRAM (Parllel Random Accès Machine) et le DRAM (Distributed Random Accès Machine) sont les modèles les plus souvent utilisés pour la spécification d'architectures distribuées ou parallèles. Le PRAM correspond à un ensemble de processeurs communiquant à travers une mémoire partagée. Le DRAM correspond à un ensemble de processeurs communiquant par passage de message à travers une mémoire distribuée. Ces modèles suffisent pour décrire la distribution et l'ordonnancement des calculs sur des machines homogènes, mais ils ne sont pas appropriés pour la description de machines hétérogènes.

Le modèle d'architecture multicomposant hétérogène utilisé est un hypergraphe non-orienté (graphe matériel), dont les sommets sont des machines à états finies (séquenceurs d'opérations de calcul ou séquenceurs d'opérations de communication), qu'on appelle opérateurs, et dont les hyperarcs sont des ressources partagées entre séquenceurs (conducteurs, mémoires RAM ou FIFO-First In First Oui et arbitres), que nous appelons médias. Chaque média de communication exécute des opérations de communication de façon séquentielle. Un média n'inclut pas seulement les conducteurs électriques nécessaires pour transporter les données entre les opérateurs, mais aussi les unités de transformation (DMA-Direct Memory Access ou UART- Universal Asynchronous Receiver/Transmitter) qui séquencent les accès mémoires de chaque côté des conducteurs [29].

Dans le cas d'une architecture multi-processeurs, chaque sommet du graphe matériel représente un processeur et chaque arc représente une liaison physique bidirectionnelle de communication, permettant de transférer des données entre les mémoires des processeurs.

Un processeur comprend une unité de calcul (CAL), une unité d'entrée/sortie [E/S) pour chaque interface avec l'environnement, une unité de communication (COM) pour chaque arc adjacent et une unité mémoire partagée (MEM) avec les autres unités. Chaque liaison de communication comprend un médium physique de communication et les unités de communication connectées au médium.

La figure 4.5 montre un exemple de graphe matériel : une architecture multiprocesseur (P1, P2, P3), où chaque processeur est connecté à un bus commun et les processeurs P1 et P2 peuvent s'échanger des données à travers une connexion directe entre eux sans utiliser le bus.

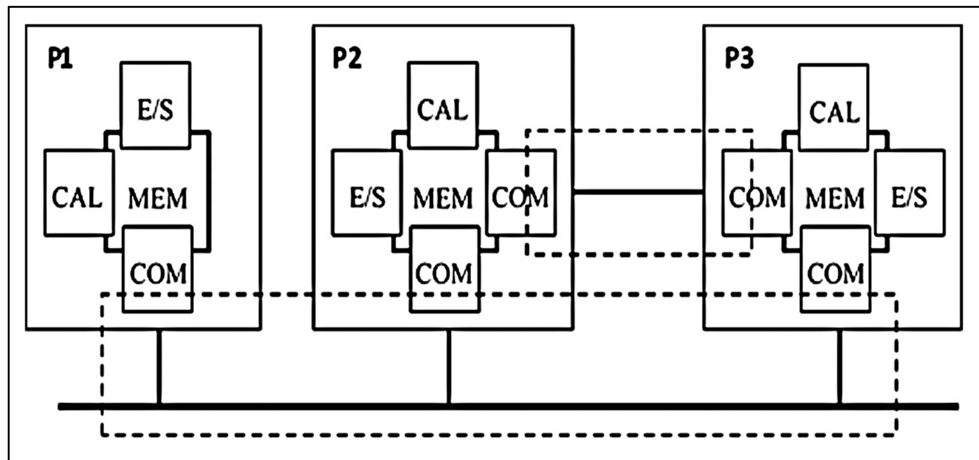


Figure 4.5. Exemple de graphe matériel (architecture multiprocesseur)

4.5.3. Modèle d'implantation

Le principe du modèle d'implantation définit que chaque tâche du graphe algorithmique n'exige qu'un seul processeur pour son exécution, et qu'il y a, au minimum, un processeur du graphe matériel capable d'exécuter une tâche du graphe algorithmique. Un de ces processeurs doit être choisi pour l'exécuter. Quand deux tâches en relation de dépendance de données sont exécutées, la tâche qui produit les données (appelée prédécesseur) doit être exécutée avant l'opération qui consomme les données (appelé successeur). Dans les architectures multiprocesseurs, l'implantation d'un algorithme sur une architecture est une distribution et un ordonnancement des tâches de l'algorithme sur les processeurs de l'architecture, il va de même pour les données de communication qui découlent de la distribution sur les médias de communication. Nous pouvons dire que l'implantation d'un algorithme sur une architecture distribuée est un ordonnancement non seulement des tâches de l'algorithme sur les processeurs, mais aussi des données de communication, la manière d'exécuter cette implantation est définie par plusieurs modes [60], on distingue dans la littérature deux grandes classes, l'exécution périodique et l'exécution cyclique.

- L'exécution périodique consiste à activer les tâches automatiquement sur des intervalles de temps réguliers à l'aide d'une horloge périodique externe au système, elle utilise l'ordre de priorité quand plusieurs tâches sont activées au même temps. Ce mode d'exécution est réalisé par l'ordonnanceur dynamique préemptif (une tâche est interrompue par une autre plus prioritaire), il a la forme suivante :

A chaque top faire

- Lire les entrées
- Calculer
- Écrire les sorties

Fin faire

- L'exécution cyclique permet l'exécution répétitive des tâches de l'algorithme au sein du système lui-même, une exécution d'une tâche définit une répétition de l'exécution de l'algorithme sur l'architecture (parfois dans certaines applications une tâche peut avoir plusieurs exécutions dans un seul cycle d'exécution). Suivant le type d'application temps réel, les dates de début d'exécution des tâches peuvent être régulières ou non, mais elles sont toutes bornées entre la borne minimale et la borne maximale de la taille d'un cycle. Ce mode d'exécution est réalisé par l'ordonnanceur statique non préemptif. L'exécution cyclique a donc la forme suivante (en prenant le graphe de la figure 4.2 un exemple) :

While système ok do

Exécuter TI
Transmettre le résultat de TI à t_1 et t_2
Exécuter t_1 et t_2
Transmettre le résultat de t_1 à t_3, $t_4$et à t_5
Exécuter t_3, $t_4$et t_5
Transmettre le résultat de t_2 à TO
Transmettre le résultat de t_3, t_4 à t_6
Transmettre le résultat de t_4 à t_6
Exécuter t_6
Transmettre le résultat de t_6 à TO
Exécuter TO

End while

Dans ce mémoire, nous nous intéressons à l'exécution cyclique de l'algorithme sur l'architecture, et nous supposons que chaque tâche est exécutée une seul fois durant chaque cycle d'exécution.

De plus, durant l'exécution de l'architecture logicielle sur l'architecture matérielle deux contraintes doivent être prise en compte, les contraintes de distribution et les contraintes d'embarquabilité (C_{mt}). Elles sont liées à plusieurs critères souvent d'optimisation, par exemple certaines tâches nécessitent dans leur exécution certaines ressources particulières, logicielles ou matérielles, qui ne sont disponibles que sur tel ou tel processeur (ex. processeurs dédiés aux opérations de traitement du signal). Ainsi, pour satisfaire les contraintes d'embarquabilité, seulement certains processeurs sont physiquement placés devant les capteurs et les actionneurs, alors les tâches d'entrée sortie doivent être placées sur ces processeurs.

Les contraintes de distribution : consistent à assigner à chaque paire (processeur, tâche) une valeur du temps d'exécution de cette tâche sur ce processeur. De même, elles assignent une durée de communication à chaque paire (dépendance de données, lien de communication).

Le tableau ci-dessous représente les durées d'exécution de chaque tâche du graphe d'algorithme de la figure 4.2 sur les opérateurs de calcul de chaque processeur du graphe d'architecture de la figure 4.5. La valeur ∞ signifie que cette tâche ne peut pas être exécutée sur cet opérateur.

		Tâches							
		TI	t_1	t_2	t_3	t_4	t_5	t_6	TO
Opérateurs	P_1	1.1	2	5	1	4	3	3	∞
	P_2	2	3	6.5	∞	5	4.5	∞	3
	P_3	∞	1.5	7	0.0	6	5	1	4

Tableau 4.1. Durée d'exécution des tâches sur les processeurs

A chaque média (bus) de communication est associée une durée de communication ou de transfert de données pour chaque dépendance de données. Cette durée dépend de la quantité de données échangée entre deux tâches et des caractéristiques physiques du médium de communication. Comme l'architecture est hétérogène, cette durée peut être différente d'un médium à un autre.

Dans notre travail nous utilisons un seul bus de communication, donc pour chaque transfert de dépendance de données il y a un seul chemin à prendre, ce qui implique que la durée de communication entre deux tâches correspond uniquement à la quantité de donnée échangée.

Le tableau 4.2 présente les durées de transfert des données entre les tâches du graphe d'algorithme de la figure 4.2 sur le média de communication du graphe d'architecture de la figure 4.3.

	Dépendances de communications									
	TI→ t_1	TI→ t_2	t_1→t_3	t_1→t_4	t_1→t_5	t_3→t_6	t_4→t_6	t_6→TO	t_2→TO	t_5→TO
Bus	1.1	2.25	1.5	3.00	1.00	1.00	1.75	2.00	1.00	3.25

Tableau 4.2. Durée d'exécution des dépendances de données

4.5.4. Optimisation de l'implantation

L'adéquation entre un algorithme et une architecture, tenant compte des contraintes temps réel et d'embarquabilité, est un problème complexe d'optimisation, Le problème d'optimisation de cette méthode AAA consiste à minimiser les ressources matérielles, tout en respectant des contraintes temporelles et technologiques.

L'algorithme, l'architecture et les contraintes sont définis a priori. Ainsi, le problème d'optimisation se réduit à une adéquation entre un algorithme et une architecture, dont les granulantes et la topologie ont été prédéfinies. Le problème reste NP-complet, même s'il a été réduit. Cela signifie que le nombre d'implantations possibles d'un algorithme sur une architecture dans le cas d'une application réelle est tellement important qu'il est impossible de trouver la solution optimale par recherche exhaustive dans tout l'espace de solutions. Il faut donc utiliser des heuristiques capables de nous donner des solutions approximatives [29].

4.5.5. Formalisation de la méthode AAA

L'AAA peut être formalisée [29] de la manière suivante : Soit

- T l'ensemble des tâches,
- D l'ensemble des dépendances entre tâches,
- P l'ensemble des processeurs et
- L l'ensemble des liaisons physiques de communications inter-processeurs.

Le graphe matériel est donc une paire (P, L) et le graphe algorithmique est une paire (T, D). La distribution consiste à partitionner l'ensemble T des tâches du graphe algorithmique en n partitions notées Tp (où n correspond au nombre de processeurs dans l'ensemble P, et p correspond au pième processeur, p ϵ (1, . . ., n) et à ordonnancer les tâches qui lui ont été affectées sur l'unité de contrôle de chaque processeur.

Ce partitionnement entraîne des communications inter-processeurs. Ces communications sont aussi distribuées et ordonnancées sur les liaisons de communications. Même si le graphe matériel est forcément connexe, chaque processeur n'est pas toujours directement connecté à tous les autres. Pour supporter toutes les communications inter-processeurs, on construit l'ensemble R qui dénote tous les chemins (ou routes) du graphe matériel (P, L). Les routes, comme les liaisons de communication, sont bidirectionnelles. La transformation du graphe matériel initial en un graphe matériel routé est appelée routage :

$$(P, L) \xrightarrow{\text{Routage}} (P, R)$$

Et l'on distribue les communications inter-processeurs sur les routes :

$$((T\ ,D\ ),(P\ ,R\ )) \xrightarrow{\text{distrib}_R} \cup_{p\epsilon P}(T_P\ ,D_P),\ \cup_{r\epsilon R} D_r$$

T ⊃Tp, où Tp correspond aux tâches exécutées par le processeur p,
D ⊃ Dp, où Dp correspond aux dépendances entre les tâches exécutées par p,
D ⊃ Dr, où Dr correspond aux dépendances inter-processeurs routées sur r ϵ R.

Pour considérer la limitation du nombre de liaisons de communication inter-processeurs, chaque arc inter-partitions d_r ϵ D_r doit être transformé en une chaîne (graphe linéaire) comportant un sommet pour chaque liaison de la route suivant la transformation formulée ci-dessous :

$$dr \xrightarrow{\text{Com}} (d_p, c_l, d_{p'}, \ldots, d_{p''}, c_{l''}), \forall\ r \epsilon R, \forall\ dr \epsilon Dr$$

Chaque nouveau sommet cl est une opération de communication qui correspond à un transfert de données entre les mémoires de deux processeurs directement connectés par une liaison physique de la route (l ϵ L). On peut considérer que chaque opération de communication est distribuée sur les unités de communication partageant la liaison physique de communication. Les nouveaux arcs dp sont des transferts intra-processeurs inter-unités appartenant à Dp.

4.5.6. Présentation de l’algorithme de distribution et d’ordonnancement de AAA

En regroupant les cl d'un même l ϵ L et les dp d'un même p ϵ P, on obtient les ensembles Cl et Dp et la transformation distribR est modifiée en transformation appelée distrib.

$$((T\ , D\ ), (P\ , L\ )) \xrightarrow{\text{distrib}} \cup_{p\epsilon P}(T_P , D_P),\ \cup_{l\epsilon C} C_l$$

D’une autre façon, on peut formaliser le problème de distribution et d’ordonnancement temps réel en suivant la méthodologie AAA comme suit :

- *Une architecture matérielle hétérogène AMH composée de n composants matériels:*
  AMH $= \{ P_1, \ldots\ldots , P_n \}$
- *Un algorithme ALG composé de m composants logiciels :*
  ALG $= \{T_1 , \ldots\ldots , T_m \}$
- *Des coûts d’exécution* C_{exe} *des composants de ALG sur les composants de AMH*
- *Des contraintes temps réel* C_{tr} *et matérielles* C_{mt}
- *Un ensemble de critères à optimiser*

Il s’agit de trouver une application A qui associe chaque composant T_i de ALG à un composant P_j de AMH , et qui lui assigne un ordre d’exécution O_k sur son composant matériel :

$$A : \quad ALG \rightarrow AMH$$
$$\quad T_i \rightarrow A(T_i) = (P_j\ ,\ O_k)$$

Et qui doit satisfaire C_{mt} et C_{tr} , et optimiser les critères spécifiés.

4.5.6. Présentation de l’algorithme de distribution et d’ordonnancement de AAA

Le but principal de l’heuristique de distribution/ordonnancement AAA est de chercher une allocation spatiale et temporelle du graphe d’algorithme ALG sur le graphe d’architecture AMH, tout en respectant les contraintes temps réel C_{tr} .

Avant de présenter l’heuristique, nous décrirons d’abord des notations qui seront utilisées par cette heuristique et aussi dans le reste de ce travail :

- pred(T_i) : l’ensemble des tâches prédécesseurs de l’opération T_i
- succ(T_i) : l’ensemble des tâches successeurs de la tâche T_i
- $C_{exe}(T_i, P_j)$: le coût d’exécution de T_i sur le processeur P_j
- $X^{(n)}$: n désigne l’étape de l’heuristique, c’est-`a-dire après avoir alloué la n^{ieme} tâche ; donc, X(n) désigne l’ensemble X à l’étape n de l’heuristique
- $T_{card}^{(n)}$: représente la liste des tâches candidates ; une tâche de ALG est dite candidate si elle est implantable, c’est-à-dire que tous ses prédécesseurs sont déjà alloués
- $T_{fin}^{(n)}$: représente la liste des tâches déjà allouées

- $St_{Ti,Pj}^{(n)}$: représente la date de début au-plus-tôt de T_i surP_j , depuis le début [31]
- $st_{Ti,Pj}^{(n)}$: représente la date de début au-plus-tard de T_i sur P_j , depuis le début [31]
- $\overline{st}_{ti}^{(n)}$: représente la date de début au-plus-tard de T_i , depuis la fin [31]

L'heuristique de distribution/ordonnancement est un algorithme glouton [32] de type ordonnancement de liste, basée sur une fonction de coût appelée la pression d'ordonnancement, dont l'objectif est de minimiser la longueur de la distribution/ordonnancement. Toutes les définitions données ci-dessous sont expirées des travaux de H.Kalla [25].

- **Longueur d'une distribution/ordonnancement :** La longueur de la distribution/ordonnancement d'un graphe d'algorithme sur un graphe d'architecture, noté $R^{(n)}$, est la durée d'exécution de ce graphe d'algorithme sur ce graphe d'architecture, c'est-`a-dire la date de terminaison du dernier processeur en exécution.
- **Pression d'ordonnancement :** La pression d'ordonnancement, notée $\sigma_{Ti,Pj}^{(n)}$, est une fonction de coût qui induit des priorités d'ordonnancement entre les opérations de ALG. Elle mesure à la fois la marge d'ordonnancement F(n) et l'allongement P(n) de la longueur de la distribution/ordonnancement. Elle est calculée pour chaque tâche candidate T_i sur chaque processeur P_j par :

$$\sigma_{Ti,Pj}^{(n)} = P_{Ti,Pj}^{(n)} - F_{Ti,Pj}^{(n)} \qquad (4\text{-}1)$$

- **Pénalité d'ordonnancement :** La pénalité d'ordonnancement, notée $P_{Ti,Pj}^{(n)}$, est une fonction qui mesure l'allongement de la longueur de la distribution/ordonnancement $R_{Ti,Pj}^{(n)}$ après avoir alloué T_i sur P_j à la n^{ieme} étape de l'heuristique, en tenant compte des coûts de communication engendrés par l'allocation. Elle est définie par :

$$P_{Ti,Pj}^{(n)} = R_{Ti,Pj}^{(n)} - R^{(n-1)} \qquad (4\text{-}2)$$

- **Flexibilité d'ordonnancement :** La flexibilité d'ordonnancement, notée $F_{Ti,Pj}^{(n)}$, est une fonction qui mesure la marge d'ordonnancement de T_i sur P_j à la n^{ieme} étape de l'heuristique. Elle est définie par :

$$F_{Ti,Pj}^{(n)} = st_{Ti,Pj}^{(n)} - St_{Ti,Pj}^{(n)} \qquad (4\text{-}3)$$

En utilisant l'équation suivante [31] :

$$st_{Ti,Pj}^{(n)} = R_{Ti,Pj}^{(n)} - \overline{st}_{Ti}^{(n)} \qquad (4\text{-}4)$$

Et les équations (2.1), (2.2) et (2.3), la pression d'ordonnancement est définie par :

$$\sigma_{Ti,Pj}^{(n)} = St_{Ti,Pj}^{(n)} + \overline{st}_{Ti}^{(n)} - R^{(n-1)} \qquad (4\text{-}5)$$

L'heuristique de distribution/ordonnancement a donc la forme qui suit :

4.5.6. Présentation de l'algorithme de distribution et d'ordonnancement de AAA

ALGORITHME
- **Entrées** = ALG, AHM, C_{exe}, C_{mtr} et C_{tr} ;
- **Sortie** = Distribution/ordonnancement statique de ALG sur AMH en fonction de C_{exe} et et C_{mtr} qui satisfait C_{tr}, ou un message d'´echec ;

INITIALISATION

Initialiser la liste des tâches candidates, et la liste des tâches déjà allouées :

$T^{(1)}_{cand}$:={tâches de ALG sans prédécesseurs} ;

$T^{(1)}_{fin} := \emptyset$ faire

BOUCLE DE DISTRIBUTION ET D'ORDONNANCEMENT

Tant que $T^{(n)}_{cand} \neq \emptyset$ **faire**

SELECTION
- Calculer pour chaque candidate $T_{cand} \in T^{(n)}_{cand}$ et chaque processeur $P_j \in$ AHM la pression d'ordonnancement (équation (2.5)) ;
- Sélectionner pour chaque candidate T_{cand} le processeur Pbest qui minimise la pression d'ordonnancement ;
- Sélectionner le meilleur couple (Tbest;Pbest) qui maximise la pression d'ordonnancement ;

DISTRIBUTION ET ORDONNANCEMENT
- Placer cette candidate Tbest sur le processeur Pbest (allocation spatiale) ;
- Placer et ordonnancer toutes les communications nécessaires à ce placement : $(T_i \rightarrow \text{Tbest})\ \forall T_i \in \text{pred(Tbest)}$;
- Ordonnancer Tbest sur le processeur Pbest (allocation temporelle) ;

VERIFICATION DES CONTRAINTES TEMPORELLES
- si ($R^{(n)}_{Ti,Pj} > C_{tr}$) alors terminer et répondre « échec » ;

MISE A JOUR
- Mettre à jour la liste des tâches candidates et déjà placées :
  $T^{(n+1)}_{fin} := T^{(n)}_{fin} \cup \{\text{Tbest}\}$;
  $T^{(n+1)}_{cand} := T^{(n)}_{cand} - \{\text{Tbest}\} \cup \left\{T \in \text{succ(Tbest)} \middle| \text{pred}(T) \subseteq T^{(n+1)}_{fin}\right\}$;

Fin tant que

FIN DE L'ALGORITHME

Figure 4.6. L'heuristique de distribution/ordonnancement AAA

Les grandes lignes de l'heuristique sont alors :
- L'exécution de l'algorithme est effectuée par plusieurs itérations ; à chaque itération, une liste des tâches candidates $T^{(n)}_{cand}$ est établie.
- Pour chaque tâche T_i de $T^{(n)}_{cand}$, une pression d'ordonnancement $\sigma^{(n)}_{Ti,Pj}$ sur chaque processeur P_j de AMH est calculée, puis le processeur (pbest) qui minimise cette pression d'ordonnancement est sélectionné .
- Parmi les couples (T_i, Pbest), celui qui maximise la pression d'ordonnancement est sélectionné (tbest , Pbest). C'est-à-dire la tâche Tbest sera placée et ordonnancée sur le processeur Pbest.
- Ce processus d'allocation est répété pour toutes les tâches restantes, jusqu'à ce qu'il n'en reste plus.

4.6. Conclusion

Dans ce chapitre, nous avons présenté le problème d'ordonnancement qui met en relation des tâches à exécuter, des machines pour les exécuter et le temps dans le contexte des systèmes distribués temps réel et embarqués ; exactement, nous avons présenté ses : typologies, propriétés et complexité. Parmi la multitude des solutions d'ordonnancement statique hors-ligne proposées dans la littérature, et pour des raisons de simplicité et d'efficacité nous avons décrit l'algorithme AAA de Syndex que nos solutions en y se basent. Dans le chapitre suivant nous allons présenter un état de l'art sur les différentes solutions proposées dans la littérature pour assurer la tolérance aux fautes dans les systèmes temps réel embarqués.

Chapitre 5

Etat de l'art sur l'ordonnancement temps réel tolérant aux fautes

5.1. Introduction

Pour assurer la sûreté de fonctionnement des systèmes temps réel embarqués stricts et en particulier garantir la tolérance aux fautes, et vu les contraintes d'embarquabilités imposées par ces systèmes : coût, énergie, taille, … ; nous avons opté pour des solutions logicielles pour tolérer les fautes matérielles d'un seul processeur. La redondance, appelée aussi réplication, des composants logiciels (tâches) est le principe utilisé dans les systèmes distribués, La réplication consiste à la création des copies multiples des processus, appelée dans notre thèse tâches ou composants logiciels, sur des processeurs différents. On trouve dans la littérature trois techniques de base de réplication : la réplication active, passive et hybride (semi-active).

Dans ce chapitre, nous présentons quelques résultats scientifiques liés à la problématique de l'ordonnancement temps réel tolérant aux fautes utilisant la technique de redondance active, passive et hybride. Avant de présenter les travaux pour chaque technique, nous fournissons d'abord une définition spécifique à elle. Nous rappelons ainsi que le nombre n de répliques de chaque composant dépend directement du nombre k de fautes à tolérer ainsi que de type de ces fautes. Par exemple, pour tolérer au plus k fautes permanentes de processeurs, chaque composant logiciel est répliqué en une copie primaire et en k copies secondaires, d'où n = k+1.

5.2. Algorithmes tolérants aux fautes basés sur la redondance logicielle

5.2.1. Réplication active

La réplication active (active replication ou state machine approach en anglais) se définie par la symétrie des comportements des copies d'un composant répliqué. Chaque copie joue un rôle identique à celui des autres [38] [47].

a) Principe

La réplication active est définie ainsi [38] :

- **Réception des requêtes :** toutes les copies reçoivent la même séquence de requêtes ;
- **Traitement des requêtes :** toutes les copies traitent les requêtes de manière déterministe ;
- **Emission des réponses :** toutes les copies émettent la même séquence de réponses. Cette technique permet en particulier de mettre en œuvre un vote sur les sorties afin de se prémunir contre les défaillances byzantines.

b) Tolérance aux fautes des processeurs

Ici, chaque copie est répliquée et exécutée simultanément sur n processeurs distincts. Les répliques doivent synchroniser leurs exécutions à chaque fois qu'un message est reçu ou envoyé afin d'assurer que toute réplique réalisant un même calcul reçoive les messages provenant des autres tâches dans le même ordre. Quand un processeur se défaille, toutes les tâches implantées sur ce processeur deviennent elles-mêmes défaillantes (inactives). La tolérance aux fautes est assurée par masquage d'erreur, c.-à-d. la défaillance d'une copie est masquée par le comportement des copies non défaillantes implantées dans les autres processeurs. Comme chaque copie joue un rôle identique, la défaillance de l'une d'entre elle ne perturbe pas le service fourni par le composant.

La figure 5.1 montre un exemple des tâches répliquées activement pour assurer la tolérance aux fautes d'un seul processeur sur une architecture matérielle composée de trois processeurs : P_1, P_2 et P_3 reliés par un bus de communication. L'architecture logicielle est composée de deux tâches : T_1 et T_2 , leurs répliques T_1' et T_2' sont placées activement sur deux processeurs distincts.

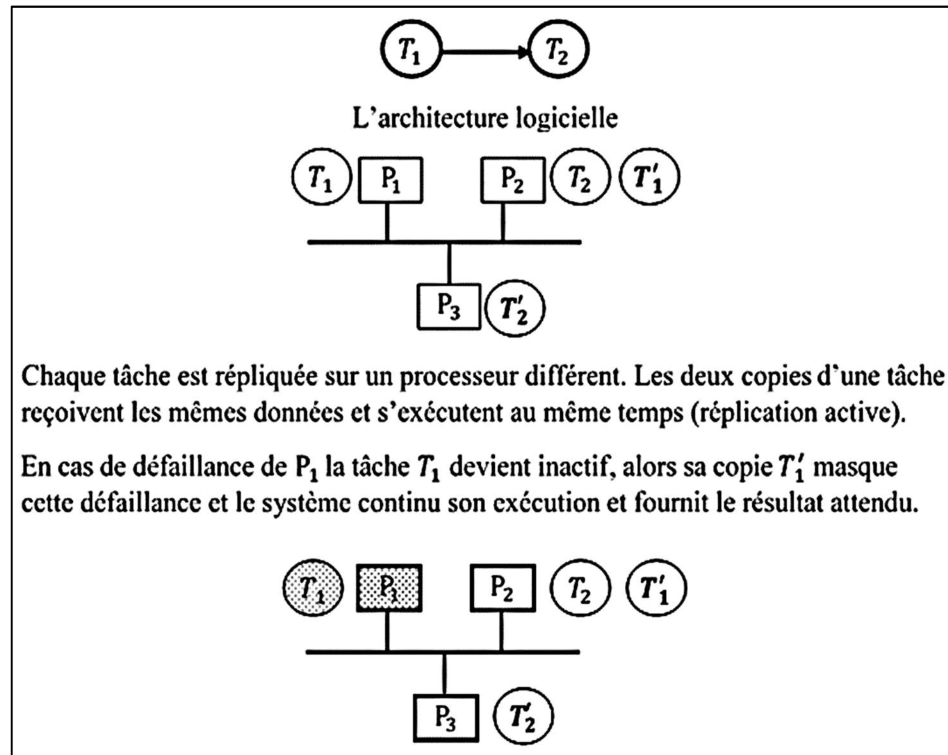


Figure 5.1. Exemple de la réplication active

c) Présentation de quelques algorithmes basés sur la redondance active

L'exigence croissant des systèmes critiques sûrs de fonctionnement dans les différents domaines de la vie (aéronautique, médecine, avionique, …) prouve la multitude des travaux réalisés sans cesse pour assurer la tolérance aux fautes dans ces systèmes.

Plusieurs approches utilisent la redondance active [25, 33, 34, 35, 36] comme solution au problème de tolérance aux fautes. Par exemple, Alain Girault et al dans [36] présentent une heuristique basée sur la réplication active pour tolérer un nombre arbitraire de défaillances des processeurs et des liens de communication, dont les défaillances sont assumées d'être silence sur défaillance, dans les système distribués temps réel et embarqués. La tolérance aux fautes est obtenue hors-ligne en deux phases, la première s'appuie sur un formalisme de transformation d'une spécification d'un graphe non redondant en une spécification avec redondance des composants logiciels pour tolérer les fautes des processeurs et des liens de communication. La deuxième phase, consiste à allouer spatialement et temporellement les composants logiciels de ce nouveau graphe redondant sur l'architecture matériel par une heuristique de distribution et d'ordonnancement temps réel.

Un autre mécanisme de tolérance aux fautes est présenté par Koji Hashimoto et al dans [35], ce mécanisme permet de tolérer les fautes d'un seul processeur en utilisant la duplication active des composants logiciels. Leurs algorithme, appelé HBP[3], se base sur des algorithmes présentés dans leurs travaux précédents (DSH , RSR_k, GRD , PHS^l). Dans un premier temps, l'algorithme HBP partitionne l'ensembles des tâches

[3] Height – Based Partitionning

en groupes de tâches selon leur taille, ou hauteur calculée par une formule, afin d'améliorer le temps d'exécution du système dans le cas de défaillance d'un processeur et dans un deuxième temps, il appelle un algorithme de base pour chaque groupe. L'algorithme de base s'exécute en deux étapes ; dans la première étape, les tâches de chaque groupe sont ordonnancées suivant leurs priorités[4] en permettant à ses prédécesseurs de s'ordonnancer en premier sur le même processeur, et dans la deuxième étape toutes les tâches de chaque groupe sont dupliquées et ordonnancées de la même façon que l'étape1 à condition que chaque instance d'une tâche est ordonnancée sur un processeur différent. Ce mécanisme permet la tolérance aux fautes et l'élimination, dans la plupart du temps, des couts de communication.

P.Ramanathan et K.Shin ont proposé dans [33] une approche basée sur la réplication active pour résoudre le problème de délivrance des messages critiques dans leur date d'échéance (deadline) dans le cas de défaillance des processeurs ou des liens de communication en un moindre coût. Le modèle utilisé est basé sur une architecture distribuée à liaison point-à-point (maille hexagonale et une topologie hypercube), et les fautes considérées sont des fautes transitoires. L'heuristique consiste donc à dupliquer chaque message au moins en deux copies, en fonction de sa criticité et du nombre des processeurs et des liens de communication qu'elle doit traverser, puis les diffuser sur des routes disjointes pour réduire le coût de la retransmission des messages. Le même principe est proposé dans [34] par Molina, Kao et Barbara et dans [37] par Kandasamy, Hayes et Murray.

5.2.2. Réplication passive

La réplication passive distingue deux comportements d'un composant répliqué : la copie primaire (primary copy) et les copies secondaires(backups). La copie primaire est la seule à effectuer tous les traitements. Les copies secondaires ne traitent pas de messages, elles surveillent uniquement la copie primaire. En cas de défaillance de la copie primaire, la copie secondaire doit détecter cette défaillance et devient la nouvelle copie primaire. Cette technique nécessite un mécanisme de détection d'erreur [38].

a) Principe

La réplication passive est définie ainsi [38] :

- **Réception des requêtes :** la copie primaire est la seule à recevoir les requêtes;
- **Traitement des requêtes :** la copie primaire est la seule à traiter les requêtes ;
- **Emission des réponses :** la copie primaire est la seule à émettre les réponses ;

b) Tolérance aux fautes des processeurs

Dans ce cas, un composant logiciel est répliqué en n exemplaires (voir la remarque ci-dessus), mais une seule des n répliques effectue le calcul. Les n-1 autres répliques sont passives et ne prennent la relève que si la réplique active est défaillante (son processeur est en état de défaillance). Pour que cette stratégie fonctionne, il est

[4] La tâche ayant la plus grande hauteur est considérée la plus prioritaire

nécessaire que la réplique active transmette aux répliques passives à intervalles réguliers son état d'exécution pour la mettre à jour. Si la réplique active est défaillante, une des répliques passive est activée et reprend l'exécution du calcul à partir du dernier point de reprise enregistré. On dit que le processus effectue un retour arrière.

Nous reprenons l'exemple précédant, mais cette fois nous appliquons la redondance passive. Donc, les deux répliques T_1' et T_2' sont placées passivement (oisivement) sur deux processeurs distincts, lorsqu'une défaillance d'un processeur se produit, la copie secondaire, de secours, de la copie primaire défaillante sera réveillée (activée).

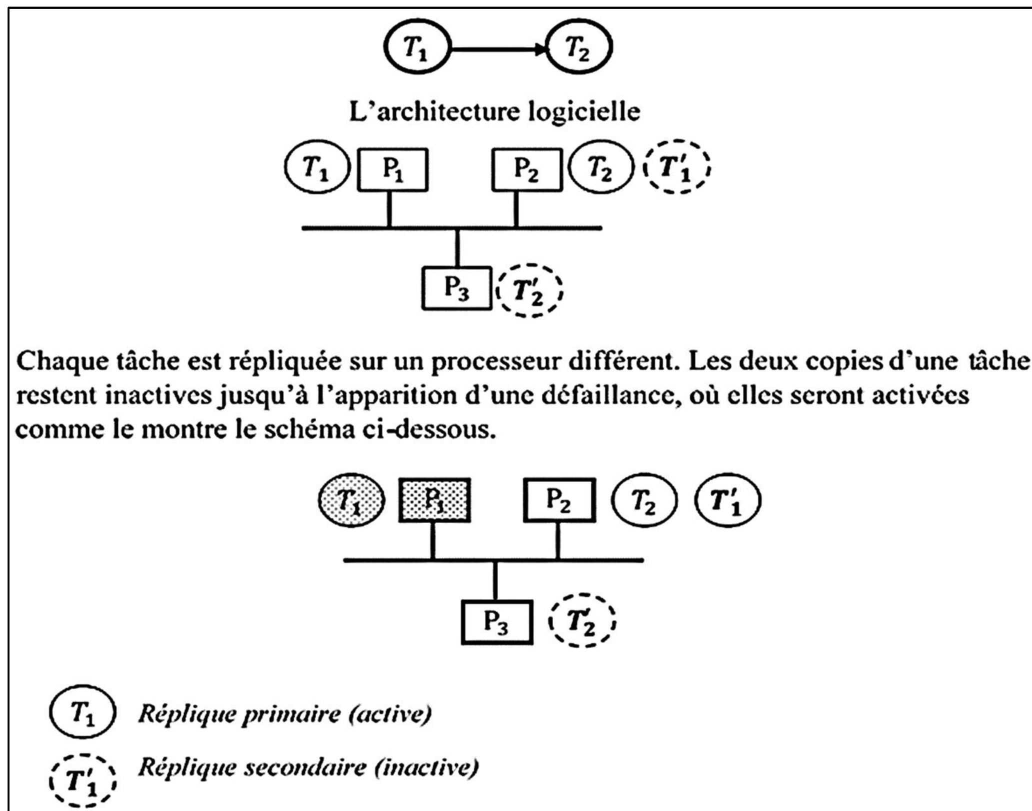


Figure 5.2. Exemple de la réplication passive

d) Présentation de quelques algorithmes basés sur la redondance passive

Plusieurs travaux dans la littérature s'intéressent à la duplication passive pour atteindre des systèmes sûrs de fonctionnement en moindre coût. Par exemple Xiao Qin et al ont présenté dans [39] une heuristique d'ordonnancement tolérante aux fautes dans les systèmes temps réel distribués et hétérogènes en se basant sur la redondance passive. Les tâches sont supposées indépendantes (absence de dépendances de données) et non-préemptives, elles doivent satisfaire leur date d'échéance même en présence de fautes d'un seul processeur. Pour arriver à ce type de système, ils ont choisi de répliquer chaque tâche en deux copies, l'une primaire et l'autre de sauvegarde qui ne s'exécute qu'à l'apparition d'une défaillance d'un processeur. Ces deux copies sont nécessairement placées sur deux processeurs distincts.

Même principe était suivi par Y. Oh et S.H.Son dans [40], ils ont proposé une heuristique d'ordonnancement tolérante aux fautes dans les systèmes temps réel distribués et hétérogènes, où les tâches sont indépendantes et non-préemptives. Leur but est de réaliser des systèmes dont les tâches (qui sont strictes) doivent finir leur exécution le plus tôt possible avant ou dans leur date d'échéance. Alors l'heuristique proposé est un ordonnancement hors ligne qui assure la satisfaction des dates limites de chaque tâche même à la présence de défaillances arbitraires d'un seul processeur, elle est appelée 1TFT[5]. La redondance des tâches spécifiées dans ce cas correspond à la redondance passive où chaque tâche est répliquée en une copie primaire et une copie de sauvegarde qui doivent être ordonnancer de façon séquentielle (pas de chevauchement) sur deux processeurs distincts.

Alain Girault et al dans [55] ont proposé une solution qui se base sur la réplication passive des opérations du graphe d'algorithme pour tolérer les fautes permanentes de k processeurs. Dans cette approche, chaque tâche est répliquée sur K+1 processeur ; le processeur exécutant la tâche primaire est appelé processeur principal, les autres sont appelés processeurs backup.

Une réplication passive est appliquée dans [59] pour tolérer les défaillances des réseaux de capteur. Il utilise la méthodologie de chevauchement des copies de sauvegarde passive pour surveiller l'exécution des copies de sauvegarde de manière adaptative via la planification de copies principales à l'avance et les copies de sauvegarde retardées.

La réplication passive est utilisée ainsi pour tolérer les fautes des processeurs utilisés pour le routage dans une route de communication pour assurer la qualité des services de communication (comme le délai d'arrivée des messages et le taux d'erreur). S.Han et K.G.Shin ont proposé dans [41] une heuristique pour reconstituer rapidement les routes temps réel (real-time channels)[6] dans le cas de défaillances des processeurs dans les réseaux de communication. Pour cela, des routes de sauvegardes sont installées a priori en plus de chaque route primaire. Les routes de sauvegarde ne portent aucun message dans le cas normal, et ils sont activés en cas de défaillance d'un ou plusieurs processeurs.

5.2.3. Réplication semi-active (hybride)

La réplication semi-active (semi-active replication) est une technique hybride entre la réplication active et la réplication passive. Contrairement à la réplication passive, les copies secondaires ne sont pas oisives. La copie primaire est appelée leader et les copies secondaires sont appelées suiveuse [38].

[5] 1-Timely-Fault-tolerance : est définie par l'ordonnancement dans lequel aucun deadline de tâches est manqué malgré les fautes arbitraires d'un seul processeur.

[6] Routes temps réel : est un circuit virtuel point-à-point unidirectionnel avec la possibilité d'opportunité garantie (timeliness-guaranteed en anglais).

c) Présentation de quelques algorithmes basés sur la redondance semi-active

a) Principe

La réplication semi-active est définie ainsi [38] :

- **Réception des requêtes :** toutes les copies reçoivent le même ensemble de requêtes.
- **Traitement des requêtes :** une seule réplique (leader) traite toutes les requêtes dès qu'elle les reçoit. Par contre, les copies secondaire (suiveuses) doivent attendre une notification de la copie primaire pour pourvoir traiter les requêtes;
- **Emission des réponses :** la copie primaire est la seule à émettre les réponses.

b) Tolérance aux fautes des processeurs

La tolérance aux fautes est réalisée par détection et compensation de l'erreur, comme dans le cas de la réplication passive. Deux situations peuvent se présenter suivant la défaillance de la copie primaire détectée par les copies secondaires avant ou après la notification.

- Si la défaillance de la copie primaire est détectée avant la réception de la notification, la nouvelle copie primaire envoie une notification concernant la première requête présente dans sa queue et la traite normalement.
- Si la défaillance de la copie primaire est détectée après la réception de la notification, la nouvelle copie primaire traite la requête correspondante sans envoyer de notification et envoie la réponse dans le bus.

c) Présentation de quelques algorithmes basés sur la redondance semi-active

H.Kalla dans [25] présente une heuristique appelée AAA-TB, basée sur la redondance hybride pour tolérer les fautes arbitraires des processeurs et des bus de communication dans un système réactif. La redondance hybride consiste, dans ce cas, d'une part à répliquer les opérations de l'architecture logicielle en plusieurs copies placées activement sur plusieurs processeurs distincts, et d'autre part à répliquer les dépendances de données en plusieurs répliques dont une seul s'exécute et les autres restent inactives jusqu'à l'apparition de défaillances de bus ou processeur implantant la copie primaire. Cela nécessite un mécanisme de recouvrement d'erreurs qui peut augmenter la longueur de la distribution/ordonnancement. Pour accélérer le recouvrement d'erreurs, il a proposé de fragmenter les données de communication.

Une autre approche présentée par P.Chevochot et I.Puaut dans [42] consiste à tolérer les fautes transitoires et permanentes d'un composant physique dans un système distribué temps réel dur. Dans cette approche, un site qui est composé d'un processeur, d'une mémoire et d'une horloge est supposé de type silence sur défaillance, il a un accès à des capteurs et à des actionneurs ; les capteurs sont exopsés à des erreurs temporelles et fonctionnelles tandis que les actionneurs sont supposés fiables. La tolérance aux fautes est achevée par le développement d'un outil de réplication appelé HYDRA, ce dernier permet d'intégrer la réplication active, passive ou hybride dans l'algorithme d'ordonnancement.

C. Arar et M.S Khireddine dans [54] ont présenté une approche basée sur la redondance active et passive pour tolérer les fautes transitoires ou permanentes d'un

seul bus de communication, cette approche nécessite un mécanisme de détection d'erreur. Le même principe est suivi dans [58] pour tolérer les fautes des processeurs.

A la fin, on peut conclure que la réplication active permet de traiter tout type de fautes. Notamment, elle est la seule à pouvoir se prémunir des défaillances arbitraires en mettant en œuvre un vote sur les sorties des copies. L'avantage principal de la réplication active est que le recouvrement de fautes est quasi-instantané puisque toutes les copies sont maintenues dans le même état, donc on n'a pas besoin de détecter les défaillances. Cependant, cette technique nécessite en permanence d'importantes ressources de traitement (surcoût élevé). Elle exige, en effet, la création de plusieurs processus sur différentes machines ainsi qu'une utilisation intensive du réseau. De plus, la réplication active n'est applicable qu'à des processus au comportement déterministe, dans le cas contraire, les copies risqueraient de diverger. En raison de sa propriété de recouvrement rapide, ce type de réplication est particulièrement adapté à un environnement exigeant des temps de réponses bornés.

La réplication passive est reconnue comme étant plus performante que la réplication active en absence de fautes (meilleur temps de réponse). En effet, les sites contenant une copie ne participent pas au traitement ce qui implique une surcharge de calcul plus faible. De plus, cette approche n'exige pas un comportement déterministe de l'application. La réplication passive est souvent préférée dans des environnements où les fautes sont rares et lorsqu'il n'y a pas de forte contrainte temporelle. Ces environnements correspondent essentiellement aux réseaux d'ordinateurs faiblement couplés.

Dans la réplication hybride, le surcoût et le temps de réponse dépend du niveau de la réplication active par apport à la réplication passive. A la différence de la réplication active, les réplications passive et hybride nécessitent un mécanisme spécial de détection d'erreur coûteux et souvent compliqué.

Le choix de la technique de réplication est délicat, il se fait en fonction des contraintes et des besoins applicatifs, par exemple le surcoût en traitement, le surcoût en communication, les types de défaillances à traiter, …etc. Nos solutions dans ce mémoire se base sur la réplication active, passive et hybride.

5.3. Conclusion

La plupart des stratégies traitant la tolérance aux fautes des composants matériels ou logiciels sont basées sur la redondance. Dans le cas de la redondance logicielle pour tolérer des fautes matérielles, nous avons présenté des algorithmes de distribution/ordonnancement tolérants aux fautes en utilisant les techniques de réplication les plus utilisées à savoir la réplication active, la réplication passive et la réplication hybride. Quand la réplication active est utilisée, toutes les copies d'un composant logiciel sont exécutées en parallèle sur les différents processeurs ; en cas d'erreur d'une copie, les autres répliques masquent cette erreur. Quand la réplication passive est utilisée, une seule copie dite primaire est exécutée tandis que les autres copies dites secondaires mémorisent l'état d'exécution. Dans la réplication hybride, les deux répliques précédentes sont coexistées dans le même algorithme d'ordonnancement. Le choix d'une stratégie de redondance se fait en fonction des contraintes et des besoins applicatifs. Nous allons présenter dans le chapitre suivant trois méthodologies de tolérance aux fautes matérielles basées respectivement sur la redondance, passive, hybride et active.

Chapitre 6

Présentation des méthodologies AAA-FAULT^{DT} et FT-TDEP

6.1. Introduction

Dans ce travail, nous étudions l'intégration de la tolérance aux fautes dans les systèmes temps réel embarqués en partant d'un algorithme de flux de données et d'une architecture hétérogène distribuée connectée par des liaisons multipoints (bus) ou point à point, notre objectif est de générer automatiquement une distribution/ordonnancement temps réel tolérante aux fautes de l'algorithme sur l'architecture. Les fautes considérées sont des fautes permanentes d'un seul processeur avec le comportement arrêt sur défaillance (fail-stop). Pour ce faire, nous présentons dans ce chapitre deux nouvelles heuristiques basées sur la redondance logicielle [46] qui génèrent un ordonnancement statique tolérant aux fautes. En prenant en compte les durées d'exécution de toutes les tâches sur tous les processeurs et les durées de communication de toutes les dépendances de données sur les liens de communication, nous sommes en mesure de calculer la longueur de distribution/ordonnancement obtenu (le temps d'exécution totale de l'algorithme sur l'architecture), à la fois en présence et en absence de fautes. L'heuristiques proposées, qui sont une extension de la méthodologie Adéquation Algorithme Architecture (AAA) de l'outil SynDEx, essayent de trouver des solutions qui répondent à des contraintes de temps réel, de distribution et de tolérance aux fautes.

La tolérance aux fautes est obtenue hors-ligne en trois phases :

1 La première consiste à transformer le graphe d'algorithme non redondant à un nouveau graphe redondant, puis d'appliquer la réplication passive, active ou hybride pour assurer la tolérance aux fautes.

2 La deuxième phase consiste à une allocation spatiale et temporelle de ce nouveau graphe d'algorithme sur le graphe d'architecture en utilisant l'algorithme de distribution et d'ordonnancement tolérant aux fautes.
3 Enfin, la dernière phase consiste à la détection et à la tolérance de fautes.

Nous distinguons deux types de graphes d'architecture logicielle :

- **Graphe avec tâches indépendantes :** les tâches s'exécutent d'une manière que chacune est indépendante de l'exécution des autres, c.-à-d. il n'existe pas de dépendances de données entre les différentes tâches du système.
- **Graphes avec tâches dépendantes :** les tâches sont reliées par des dépendances de précédence où chacune ne peut être exécutée que si elle reçoit les données de ses prédécesseurs.

Les méthodologies AAA-FAULT^{DT} et FT-TDEP résolvent le problème dans des graphes d'architectures avec tâches dépendante.

Dans la suite de ce chapitre, nous présentons en premier lieu la méthodologie AAA-FAULT^{DT} ; et en deuxième lieu la méthodologie FT-TDEP.

6.2. Présentation de l'algorithme de distribution/ordonnancement tolérant aux fautes des tâches dépendantes AAA-FAULT^{DT}

Le but de ce chapitre est de résoudre le problème de la recherche d'un ordonnancement des composants logiciels sur les composants matériels qui doit tolérer des fautes matérielles des processeurs, tout en minimisant la longueur de l'ordonnancement dans le but de satisfaire la contrainte temps réel en absence et en présence de défaillances.

Notre solution est liée aux hypothèses de défaillances définies par le modèle de fautes suivant :

6.2.1. Modèle de fautes

Toute solution aux problèmes de distribution/ordonnancement tolérant aux fautes est dépendante des hypothèses de défaillance, alors celles supposées dans notre modèle sont :

- L'algorithme est fiable et sans fautes, (c'est à dire que le logiciel a été supposé validé par des techniques de tolérance aux fautes logiciels [48] [53].
- Les fautes matérielles sont des fautes des processeurs et plus particulièrement le système accepte les fautes d'un seul processeur durant tous les cycles d'exécution du graphe d'algorithme sur le graphe d'architecture.
- Nous considérons que les fautes permanentes.
- Le bus de communication est supposé fiable.
- Le type de défaillance du processeur est arrêt sur défaillance (fail stop).
- Les tâches constituants l'architecture algorithmique sont dépendantes.

Avant d'aborder le principe général de la méthodologie proposée, un ensemble de données doit être définit dans ce qui suit :

6.2.2. Données du problème

- Une architecture matérielle hétérogène AMH composée d'un ensemble P de processeurs et d'un ensemble L qui contient un bus B de communication (liaison à bus).
  $P = \{P_1, P_2, \ldots\ldots P_m\}$, $L = \{B\}$

Exemple :

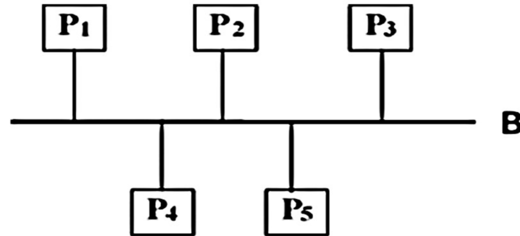


Figure 6.1. Architecture liaison à bus

- Un algorithme ALG, composé d'un ensemble T de tâches et d'un ensemble D de dépendances de données.
  $T = \{t_1, t_2, \ldots\ldots t_n\}$, $D = \{\ldots\ldots, (t_i \rightarrow t_j), \ldots\ldots\}$

Exemple :

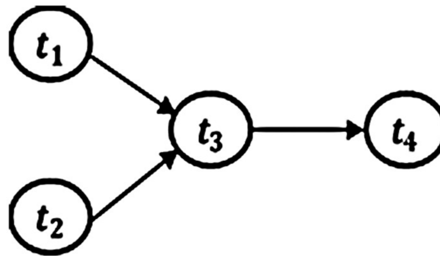


Figure 6.2. Exemple d'un graphe d'algorithme

- Des caractéristiques d'exécution C_{exe} des composants de ALG sur les composants de AMH.
- Un ensemble de contraintes matérielles C_{mt} .
- Une contrainte temps réel C_{tr} .
- Les fautes d'un seul processeur qui peuvent causer la défaillance du système.

Il s'agit de trouver une application A qui place chaque tâche T_i de ALG sur un processeurP_j de AMH et qui lui assigne un ordre d'exécution O_k sur son processeur. L'application A doit respecter les contraintes matérielles, minimiser la longueur de la distribution/ordonnancement afin de satisfaire les contraintes temps réel, et tolérer les fautes d'un seul processeur.

$$A: \quad ALG \rightarrow AMH$$
$$T_i \quad \rightarrow A(T_i) = (P_j\ ,\ O_k)$$

6.2.3. Principe général de la méthodologie AAA-FAULT^{DT}

Le problème de distribution/ordonnancement tolérant aux fautes des tâches dépendantes peut être vu comme étant un problème à deux parties ; un problème de temps réel et un problème de tolérance aux fautes. C'est un problème d'optimisation NP-difficile. Pour le résoudre en temps polynomial, nous proposons une méthodologie appelée AAA-FAULT^{DT} qui essaye de trouver une solution proche de la solution optimale. A la différence de la méthodologie AAA-FAULT^{IDT}, AAA-FAULT^{DT} est basée sur la redondance passive des tâches dépendantes du système et par conséquence sur un mécanisme de détection d'erreurs.

AAA-FAULT^{DT} : Est une nouvelle méthodologie pour optimiser d'une part la génération automatique de distribution/ordonnancement et d'autre part pour tolérer les fautes permanentes d'un seul processeur durant un cycle d'exécution de l'architecture logicielle sur l'architecture matérielle. Dans cette méthodologie, la tolérance aux pannes est basée sur la redondance passive qui nécessite un mécanisme de détection des erreurs [45]. Par conséquent, nous insérons dans ALG une nouvelle tâche appelée watchdog, notée w.

Principe 1 : redondance logicielle

La redondance logicielle est une technique utilisée dans les systèmes répartis pour tolérer les fautes matérielles (ici processeurs), elle consiste à répliquer les processus (tâches) sur différents processeurs.

Principe 2 : Redondance passive

Pour ne pas trop charger les processeurs en exécutant toutes les répliques des tâches et pour réduire le temps d'exécution en absence et en présence de défaillances, nous avons proposé une solution basée sur la redondance passive qui consiste à exécuter les copies secondaires d'une tâche uniquement à la détection de la défaillance de son processeur.

Principe 3 : Tolérance aux fautes permanentes

La méthodologie AAA-FAULT^{DT} sert à générer automatiquement une distribution/ordonnancement tolérante aux fautes permanentes d'un seul processeur à n'importe quelle étape dans chaque cycle d'exécution de l'algorithme. Elle est composée de trois phases : la phase de transformation, la phase d'adéquation et la phase de tolérance aux fautes détectées.

- **La phase de transformation** consiste à transformer le graphe d'algorithme non redondant ALG à un nouveau graphe ALG^n redondant dont :
  - Chaque tâche est répliquée en deux copies, primaire t_i^p et secondaire (backup)t_i^b ;
  - La tâche W_i est ajoutée entre chaque tâche t_i^p et sa réplique t_i^b ;
  - Trois nouvelles dépendances sont créées : ($t_i^p \rightarrow W_i$), ($W_i \rightarrow t_i^b$) et ($t_i^b \rightarrow t_j$), la tâche t_j est le successeur de la tâche t_i .

Les nouvelles dépendances sont de deux type :

- Deux dépendances de données, l'une d'elle transfère un message de la tâche primaire t_i^p indiquant son exécution à la tâche W et l'autre transfère le résultat de la tâche secondaire t_i^b à son successeur (t_j),
- Et une dépendance conditionnée qui transfère un message de réveil depuis la tâche W vers la réplique de chaque tâche. Elle est appelée dépendance conditionnée puisque son exécution dépend de l'état des processeurs (défaillants ou non).

La figure suivante montre un exemple de transformation d'un graphe composé de deux tâches : t_i^i et t_j^i liés par une dépendance de données $t_i \rightarrow t_j$

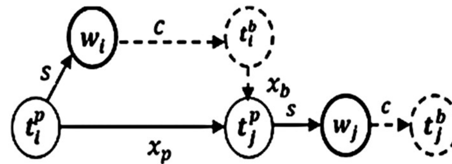


Figure 6.3. Exemple de transformation d'un graphe d'algorithme

- **Dans la phase d'adéquation,** l'heuristique proposée est basée sur une heuristique de distribution/ordonnancement hors-ligne qui consiste à mettre en correspondance de manière efficace le nouveau graphe d'algorithme ALG^n sur le graphe d'architecture AMH.
- **La phase de tolérance aux fautes** consiste à détecter la défaillance d'un processeur puis mettre à jour l'ordonnancement des tâches en y implantées.

Principe 4 : Détection d'erreurs

La redondance passive nécessite un mécanisme de détection d'erreurs. Sachant que nous travaillons sur une architecture à liaison bus où les processeurs sont complètement connectés et ils reçoivent les mêmes données, donc le mécanisme de détection d'erreur est simple et facile. Pour cela, nous avons proposé de créer une nouvelle tâche appelée watchdog « W» ajoutée comme successeur de chaque tâche dans le graphe d'architecture.

Le rôle de la tâche W est de recevoir un signal indiquant l'exécution de son prédécesseur qui est la tâche t_i^p. Ainsi, après un certain temps Δ[7] si elle ne reçoit aucun signal, elle détecte alors que le processeur P_i qui exécute la tache t_i^p est en panne, par suite elle réveille (active) la copie secondaire t_i^b implantée sur un autre processeur, et cela par l'envoie d'un message de réveil à travers le bus ou via une communication intra-processeur.

[7] L'intervalle de temps Δ appelé timeout est déterminé en fonction de l'application à réaliser. Il est défini par le réalisateur du système à l'entrée de l'algorithme.

Le corps de la tâche W a la forme suivante :

Si (W_i ne reçoit pas un signal de t_i^p après le temps Δ) **alors** / *Pi est en panne /*

- Envoyer un message de réveil à sa réplique t_i^b pour mettre à jour l'ordonnancement. Ce message concerne aussi les répliques des autres tâches implantées sur P_i .

Fin Si

La figure 6.4 schématise une représentation de la méthodologie AAA-FAULT^{DT}.

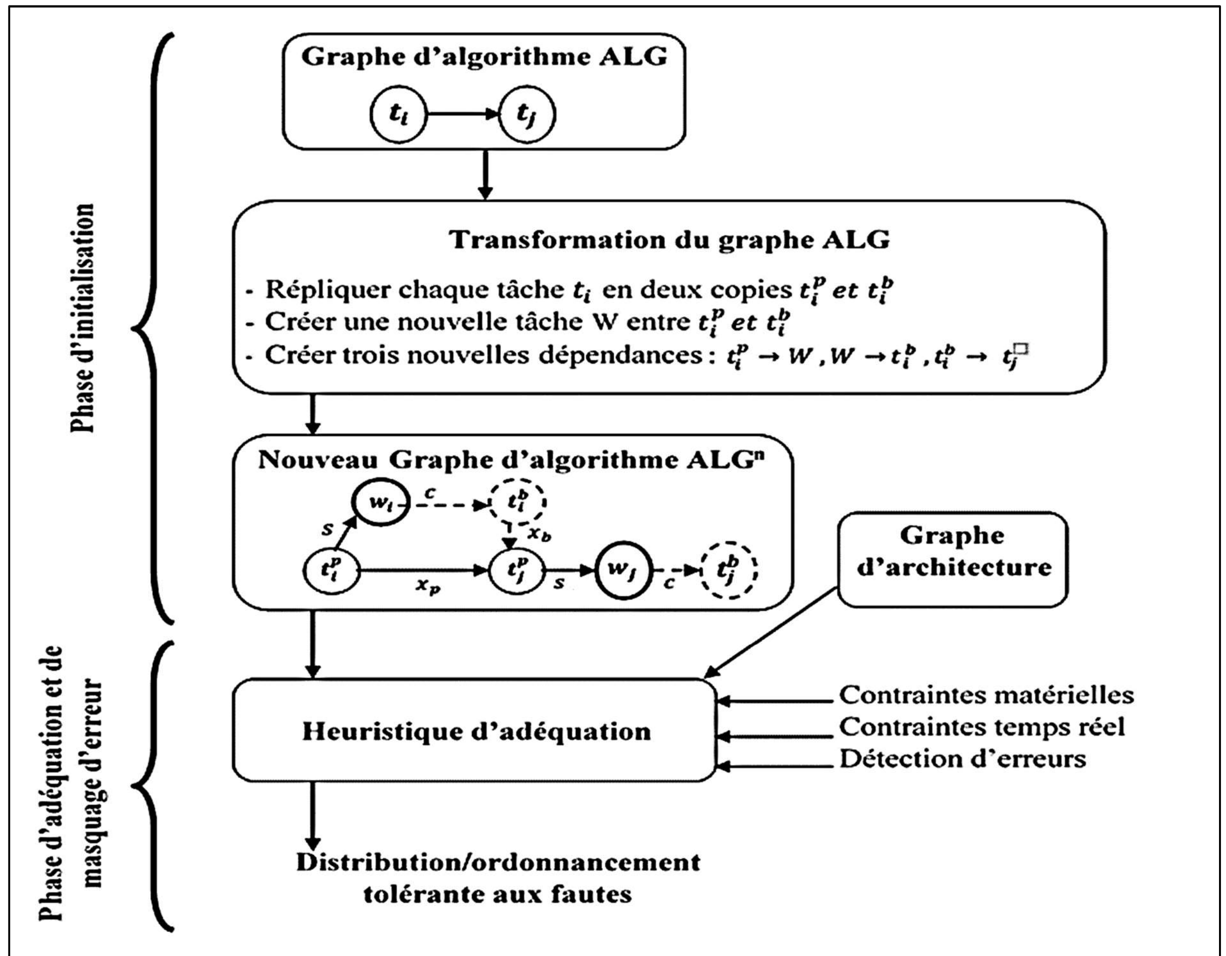


Figure 6.4. Méthodologie AAA-FAULT^{DT}

Dans ce qui suit, nous donnons quelques terminologies utilisées :

Graphe ALG^n : Le nouveau graphe est aussi bien un graphe orienté. Dans ce nouveau graphe, deux nouvelles tâches sont ajoutées pour chaque tâche t_i , La première c'est sa copie secondaire, et la deuxième appelée W surveille le bon fonctionnement de son processeur. Trois nouvelles dépendances sont en effet créées comme l'explique le principe3.

Tâche watchdog : Une tache W est une tâche dont le rôle est de détecter la panne d'un processeur puis d'activer les répliques de ses tâches, son temps d'exécution est presque nul.

Dépendance conditionnée : Une dépendance est dite conditionnée si son exécution dépend de la panne d'un processeur, c.-à-d. elle s'exécute si un processeur donné est défaillant si non elle ne s'exécute jamais.

Un exemple de transformation est donné par la figure 6.14 pour tolérer les fautes d'un seul processeur. L'exemple ci-dessous montre une distribution/ordonnancement d'une architecture logicielle sur une architecture matérielle, dont lequel les tâches (resp. Communications) sont représentées par des boites dont la longueur est proportionnelle à leur temps d'exécution (resp. à leur durée de communication).

Soit l'architecture matérielle hétérogène suivante : P = {P1, P2, P3}, L = {Bus}, et l'architecture logicielle ALG transformée en ALG^{N} dans laquelle chaque nœud est associé à ses temps d'exécution sur différents processeurs et chaque arc est associé à la durée de communication entre les nœuds qu'il relie.

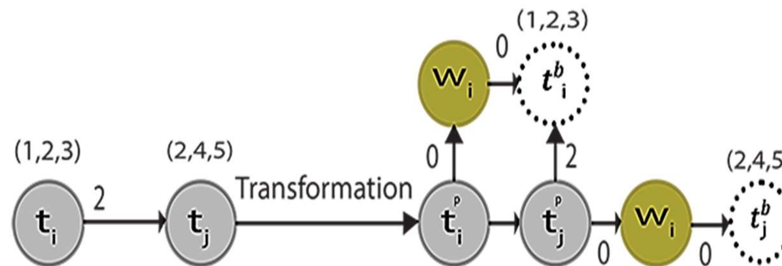


Figure 6.5. Schéma de transformation de ALG

La distribution/ordonnancement tolérante aux fautes est décrite dans la figure 6.6, d'où p_i^v (processeur virtuel) indique le placement passif des répliques des tâches en attente d'activation sur le processeur pi, et L est la longueur de la distribution/ordonnancement avec absence de défaillance.

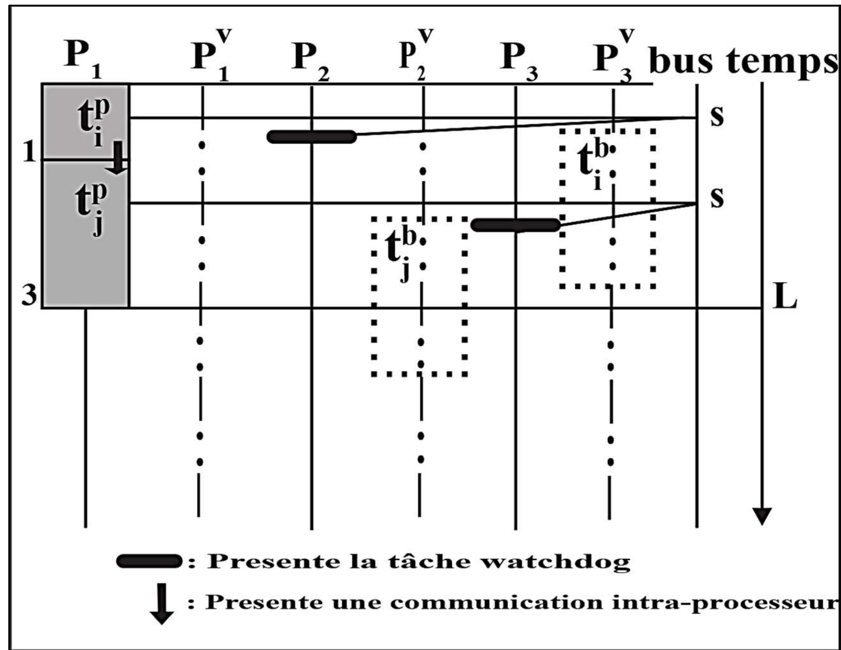


Figure 6.6. Distribution/ordonnancement tolérante aux fautes avec absence de défaillance

Dans le cas de défaillance d'un processeur, p_1 par exemple, la tâche w_i ne reçoie pas un message de t_i^p ; donc après un temps Δ, la tâche w_i réveille la copie backup t_i^b implantée sur p_3 en exécutant la dépendance conditionnée C, de même ce message de réveille est reçu par la tâche t_j^P. La défaillance est donc masquée comme indiqué dans la figure 6.7

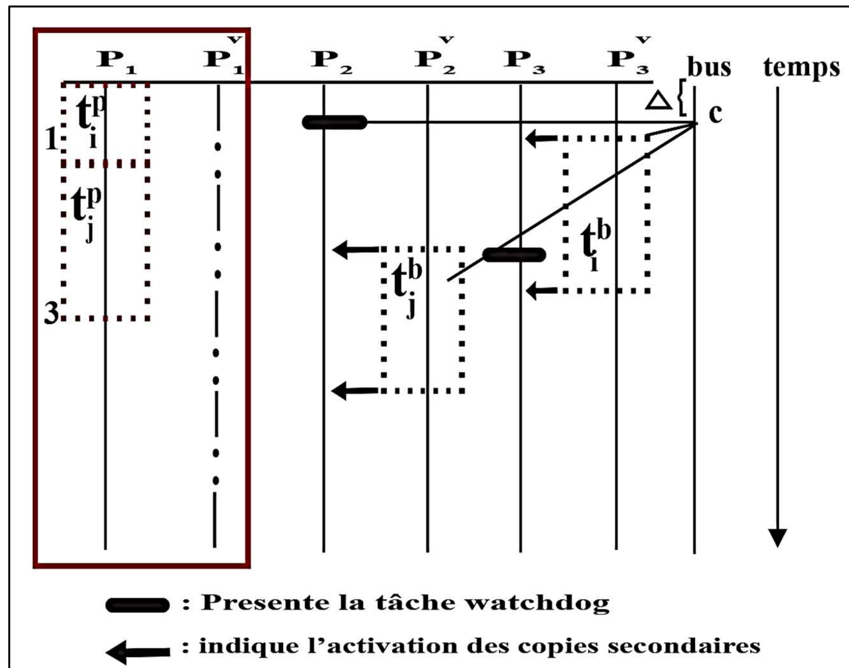


Figure 6.7. Distribution/ordonnancement tolérante aux fautes avec présence de défaillance

Alors t_i^b sera activée et exécutée sur p_3, elle envoie le résultat x_b à t_j^b par une communication inter-processeur pour être exécuter et fournir les résultats. Le processeur p_1 sera exclu de l'architecture matérielle (jusqu'à réparation si possible), et finalement l'exécution du système lors des cycles suivants suit cette nouvelle distribution/ordonnancement (figure 6.8).

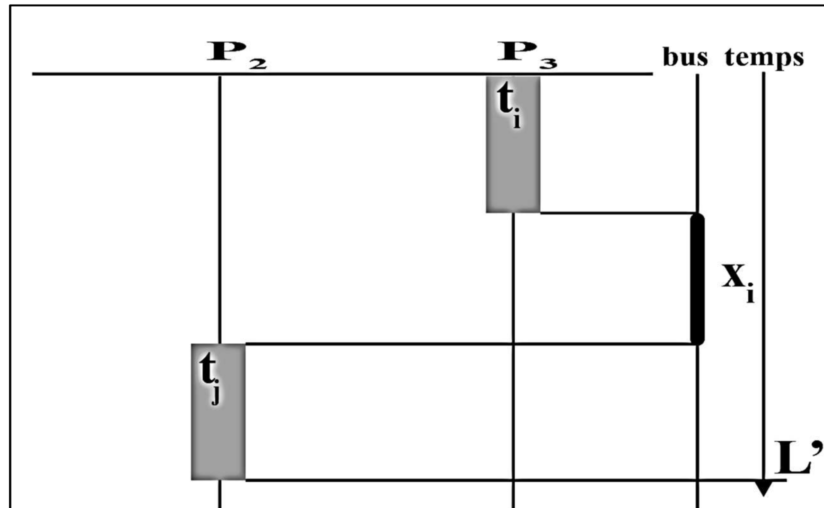


Figure 6.8. Nouvelle distribution/ordonnancement après défaillance

Notes :

1. Si les tâches ayant des dépendances de précédence entre elles sont implantées sur le même processeur, alors le transfert de données dans ce cas est réalisé par une communication intra-processeur d'où la durée de communication est nulle.
2. Les répliques de toutes les tâches placées sur le processeur en panne et qui ne sont pas encore exécutées au moment de la défaillance auront toutes reçu le message de réveil envoyé par la tâche W (pour éviter la perte de temps).
3. La tâche n'ayant pas de successeur est aussi suivie dans le graphe d'algorithme par la tâche W pour s'assurer que le résultat se sera fourni dans le cas de la défaillance du processeur à ce niveau.
4. Les durées des communications correspondent uniquement à la quantité de données échangée, cependant celles des dépendances conditionnées et les dépendances qui transfert les signales vers les tâches W sont nulles.

6.2.3.1. Tâches principales de l'heuristique d'adéquation

- Calcul des dates de début de toutes les tâches du nouveau graphe d'algorithme en utilisant la fonction de la pression d'ordonnancement [25]. Chaque tâche contient deux dates de début : T_{best} et T_{worst} .

  $\boldsymbol{T_{best}}$ **:** c'est la date de début des copies primaires (en absence de défaillance).

  $\boldsymbol{T_{worst}}$ **:** c'est la date de début des copies backups (à la présence d'une défaillance).
- Placement actif des copies primaires : au démarrage du système, seules les copies primaires de chaque tâche sont à exécuter.
- Placement passif des copies secondaires : pour ne pas surcharger les processeurs, les copies backups ne s'exécutent qu'à la détection d'une défaillance.

- Prise en compte de toutes les contraintes : le coût d'exécution, la contrainte temps réel et les contraintes matérielles.
- Prévision du comportement temps réel (prédictibilité)

Nous avons considéré que chaque tâche contient deux dates de début, puisque en cas de défaillance c'est la réplique backup d'une tâche qui va s'exécuter alors sa date de début est considérée comme une mauvaise date d'exécution de la copie primaire.

Nous utilisons les mêmes notations que celles données dans la section 4.5.6.

6.2.3.2. Principe de l'heuristique

L'heuristique est basée sur une fonction de coût appelée la pression d'ordonnancement notée $\sigma^{n}_{(Ti,Pj)}$, donnée par l'équation 4-5 (page 47), dont l'objectif global est de minimiser la longueur de la distribution/ordonnancement en absence et en présence de défaillance d'un seul processeur. Les principes de cette heuristique sont :

- Chaque tâche envoie ces données de dépendances dans le bus, seules les tâches concernées reçoivent le message.
- En cas de la non réception d'un signale qui signifie la bonne exécution de chaque tâche primaire par la tâche W après un temps « timeout », W réveille les copies secondaires par une communication intra-processeur ou via le bus.
- Après la détection de la défaillance et la mise à jour de l'ordonnancement des tâches défaillantes, le système suit dans son exécution ce nouvel ordonnancement.

6.2.3.3. Présentation de l'heuristique

L'heuristique est composée de cinq phases :

ALGORITHME
- **Entrées** = ALG, AHM, C_{exe}, C_{mt} et C_{tr} ;
- **Sortie** = Distribution/ordonnancement statique de ALG sur AMH en fonction de C_{exe} et C_{mt} qui satisfait C_{tr} , ou un message d'échec ;

INITIALISATION
Initialiser la liste des tâches candidates, et la liste des tâches déjà allouées :
$T_{cand}^{(1)}$: = {tâches de ALG sans prédécesseurs} ;
$T_{fin}^{(1)}$: = $\emptyset$ faire

BOUCLE DE DISTRIBUTION ET D'ORDONNANCEMENT
Tant que $T_{cand}^{(n)} \neq \emptyset$ **faire**

SELECTION
- Calculer pour chaque candidate t_i de $T_{cand}^{(n)}$ et chaque processeur $P_j \in$ AHM la pression d'ordonnancement ;
- Sélectionner pour chaque candidate t_i le processeur p_{best} qui minimise la pression d'ordonnancement, le processeur où la copie secondaire est ordonnancée doit être différent de celui de la copie primaire ;
- Sélectionner parmi les couples $(t_i\ ,\ p_{best})$ le meilleur couple (t_{best}, p_{best}) qui maximise la pression d'ordonnancement ;

DISTRIBUTION ET ORDONNANCEMENT
- Placer et ordonnancer la tâche T_{best} sur le processeur Pbest (allocation spatiale et temporelle) ;
- Placer et ordonnancer les communications nécessaires à ce placement.

VERIFICATION DES CONTRAINTES TEMPORELLES
- **si** ($R_{Ti,Pj}^{(n)} > C_{tr}$) **alors terminer et répondre « échec » ;**

MISE A JOUR
- Mettre à jour la liste des tâches candidates et déjà placées :
  $T_{fin}^{(n+1)} := T_{fin}^{(n)} \cup \{t_{best}\}$;
  $T_{cand}^{(n+1)} := T_{cand}^{(n)} - \{t_{best}^{p}\} \cup \{T \in succ(t_{best}^{p}) | pred(t_{best}^{p}) \subseteq T_{fin}^{(n+1)}\}$;

Fin tant que
FIN DE L'ALGORITHME

Figure 6.9. L'algorithme AAA-FAULT^{DT}

1. **Phase d'initialisation :** dans laquelle :
   - Les tâches qui n'ont pas de prédécesseurs sont placées dans une liste appelée T_{cand} ;
   - Une autre liste appelée T_{fin} est initialisée à $\emptyset$, cette liste contient les tâches déjà placées.
2. **Phase de sélection :** dans cette phase, un choix de la tâche la plus urgente qui va s'implanter sur le meilleur processeur est effectué par le calcul de la pression d'ordonnancement pour chaque tâche candidate sur tous les processeurs. Dans le cas où la meilleure tâche sélectionnée est une copie secondaire, alors l'euristique doit lui choisir un deuxième processeur distinct de celui de sa copie primaire.
3. **Phase de distribution et d'ordonnancement :** Après avoir sélectionner la meilleur candidate t_{best} et son meilleur processeur p_{best} pendant la phase de sélection, il ne

reste plus qu'à la placer/ordonnancer, la première copie de chaque tâche sélectionnée est allouée activement sur son processeur (à exécuter), par contre sa deuxième copie est allouée passivement sur le deuxième processeur choisi en attendant son activation.

4. **Phase de vérification des contraintes temporelles :** à chaque fois, l'algorithme doit vérifier le respect de la contrainte de temps, deadline, spécifiée par le système.
5. **Phase de mise à jour :** l'objectif de cette phase est de mettre à jour :
   - La liste des tâches déjà allouées T_{fin} : dans laquelle les tâches nouvellement placées sont ajoutées ;

Dans la liste des tâches candidates T_{cand}, les tâches déjà placées doivent être supprimées de cette liste et ses nouvelles tâches ajoutées sont celles qui ont leurs prédécesseurs dans la liste des tâches déjà placées.

6.3. Evaluation de la méthodologie AAA-FAULT^{DT}

Il existe deux manières d'évaluer une méthodologie, relativement à d'autres méthodologies du même type, ou bien de façon absolue. Par ailleurs, il faut définir les critères utilisés pour cette évaluation. Il nous semble ici intéressant d'évaluer le surcoût de la longueur de la distribution/ordonnancement introduit par la méthodologie AAA-FAULT^{DT} en présence et en absence d'une défaillance d'un processeur unique.

6.3.1. Paramètres d'évaluation

Nous avons appliqué l'heuristique AAA-FAULT^{DT} à un ensemble de graphes d'algorithmes aléatoires avec un ensemble de paramètres qui affectent nos résultats. Le nombre de processeurs P, le nombre de tâches N et le facteur CCR (rapport entre le temps de communication moyen et le temps d'exécution moyen) sont les paramètres que nous avons modifiés pour tester l'efficacité de notre méthodologie en présence et absence de fautes.

Suivant le schéma de Kalla [25], un graphe d'algorithme aléatoire est généré (Figure 6.10) par plusieurs niveaux (≥ 2). Chaque niveau i est composé de plusieurs nœuds (tâches) et chaque nœud de niveau i a au moins un prédécesseur de niveau inférieur j tel que i> j. La figure 6.10 montre un exemple sur le processus de génération de graphe aléatoires ALG. Ce générateur est basé sur trois paramètres :

- **T** - la taille du graphe t ; le nombre de nœuds dans le graphe (chaque nœud correspond à une tâche).
- **H** - la hauteur du graphe ; le nombre maximum de niveaux dans le graphe.
- **G** - la largeur du graphe ; le nombre maximal de nœuds indépendants dans un niveau du graphe.

Ce processus est réalisé en deux phases complémentaires : une phase de génération de nœuds et une phase de génération d'arcs.

La phase de génération de nœuds elle-même est réalisée en deux étapes :

- Premièrement, nous tirons aléatoirement t nœuds dans une matrice de dimension (G*H) ; G et H agissent sur la forme du graphe.

- Ensuite, nous construisons les niveaux du graphe ; chaque niveau k est composé des nœuds situés sur une même ligne horizontale dans la matrice.

Dans la phase de génération des arcs, nous générons les arcs aussi en deux étapes.

- Premièrement, nous choisissons de manière aléatoire pour chaque nœud de niveau i un ou plusieurs nœuds comme prédécesseurs du niveau j, avec j = i - 1.
- Ensuite, nous choisissons de manière aléatoire pour chaque nœud de niveau i zéro ou plusieurs nœuds comme prédécesseurs du niveau j, avec j < i - 1.

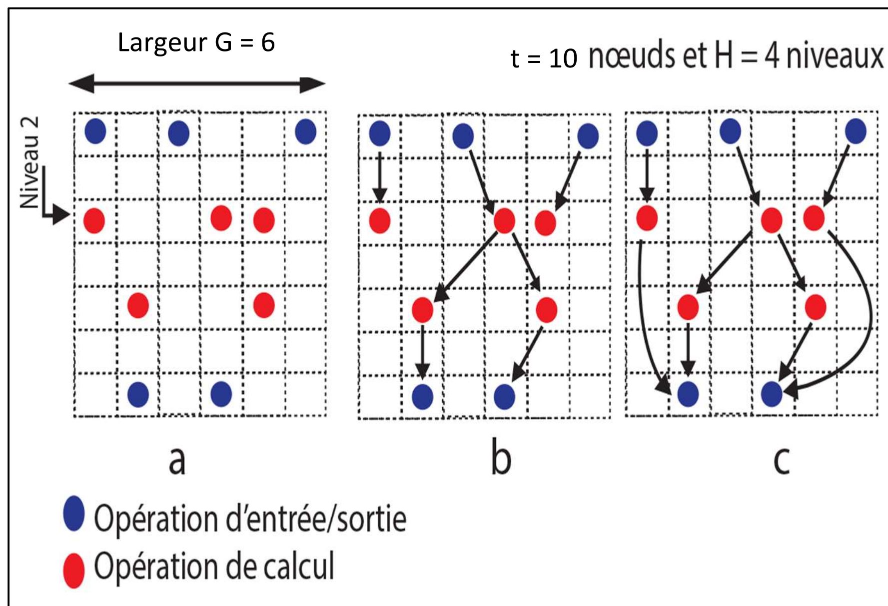


Figure 6.10. Etapes de génération aléatoire d'un graphe d'algorithme

6.3.2. Les résultats

Nous avons tracé sur les figures 6.11, 6.12 et 6.13 la longueur de distribution/ordonnancement **L** en fonction de **N**, **P** et **CCR**. L'axe orthogonal de toutes les figures montre cette longueur obtenue par l'exécution de notre méthodologie sur un ordinateur de bureau à capacités moyenne, ainsi que les valeurs des coûts d'exécution des tâches sur les processeurs et les durées de communication sont prises de manière aléatoire à partir d'un intervalle d'entier. Les valeurs comprises entre 1 et 10 ne reflètent pas les valeurs réelles du temps d'exécution du microprocesseur, ni le temps de communication entre eux. En réalité, ils sont incomparables. bien sûr, beaucoup mieux.

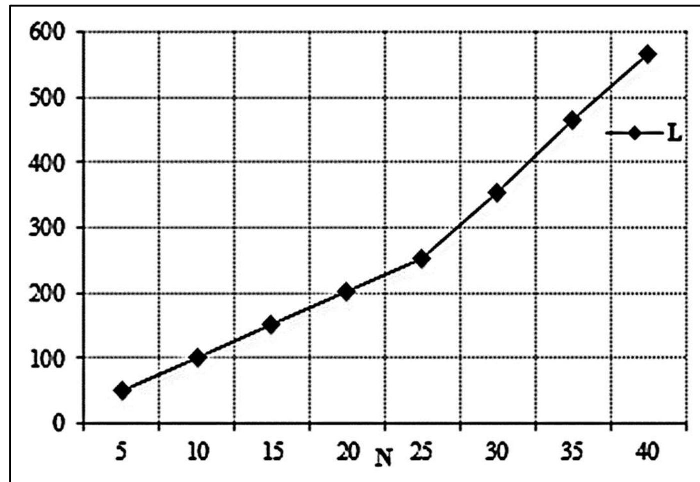


Figure 6.11. Effet de N sur AAA-FAULT^{DT} pour p = 5 et CCR = 2

La figure 6.11 montre que la longueur de distribution/ordonnancement s'augmente avec N parce qu'elles sont répliquées sur des processeurs distincts où leurs durées d'exécution peuvent être plus élevées, leurs durées de communication aussi.

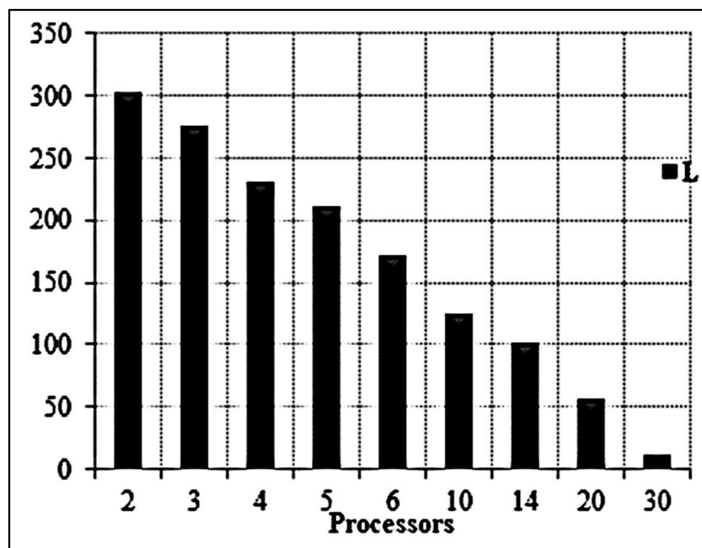


Figure 6.12. Effet du P sur AAA-FAULT^{DT} pour N = 40 et CCR = 1

La figure 6.12 montre que la longueur de distribution/ordonnancement se baisse avec P. cela est dû au nombre de processeurs disponibles pour tolérer les fautes.

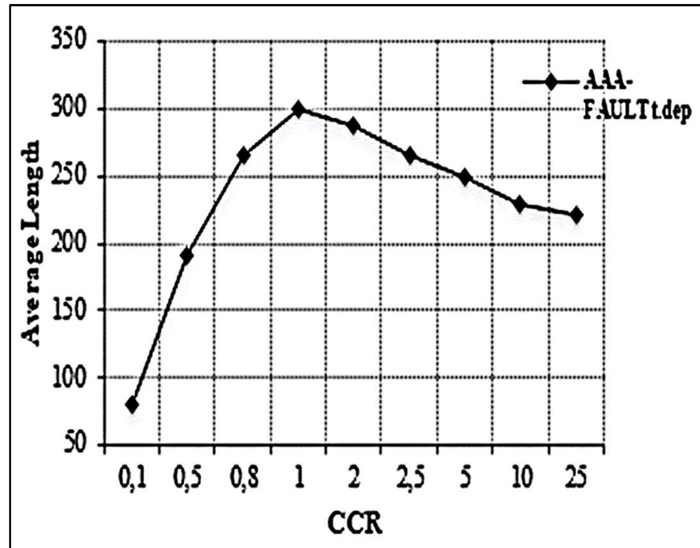


Figure 6.13. Effet du CCR sur AAA-FAULT^{DT} pour P = 6 et N = 50

La figure 6.13 montre que la longueur de distribution/ordonnancement s'augmente pour CCR <1 et se diminue pour CCR> 1. Ceci est dû aux performances de l'heuristique lorsque le coût de la communication est important.

Quels que soient le nombre de tâches, les durées d'exécutions et les durées de communication qui augmentent la longueur de distribution/ordonnancement, les résultats obtenus prouvent l'efficacité de notre méthodologie en cas de défaillance du processeur. En fait, même la longueur de distribution/ordonnancement s'augmente, elle ne dépasse pas la date limite (deadline). Ainsi, le service attendu du système est fourni à temps dans tous les cas.

Dans ce qui suit nous présentons une nouvelle méthodologie appelée **FT-TDEP**, pas encore publiée, qui est une optimisation de la méthodologie AAA-FAULT^{DT}. C'est une nouvelle méthodologie permettant d'optimiser d'abord la génération automatique hors ligne de la distribution/ordonnancement des tâches de l'architecture logicielle ALG sur les processeurs et les médias de communication de l'architecture matérielle AHM, ensuite de tolérer les défauts permanents d'un processeur unique lors du cycle d'exécution de l'architecture logicielle sur l'architecture matérielle.

6.4. Présentation de la méthodologie FT-TDEP

Dans cette méthodologie, la tolérance aux fautes est basée sur la redondance passive et active des tâches dépendantes liées par des dépendances de priorité, c'est-à-dire que chaque tâche ne peut pas être exécutée à moins que ses prédécesseurs envoient les résultats après exécution. Pour satisfaire la propriété qui permet au système de continuer à fournir le service attendu même en présence de fautes, FT-TDEP est basés sur la réplication de chaque tâche en copies primaire et secondaire. En premier, les deux sont distribuées et ordonnancées pour être exécutés en parallèle (réplication active), à la suite et lorsque l'une des copies, **primaire ou secondaire**, termine son exécution, elle envoie un signal de blocage de l'exécution de l'autre copie (réplication passive). A la fin de l'exécution de chaque tâche, les résultats obtenus sont envoyés à

ses successeurs ainsi qu'à sa réplique. Cette technique permet de gagner un temps considérable en termes de la longueur de distribution/ordonnancement, notamment parce que notre solution ne nécessite pas un mécanisme de détection d'erreur.

Avant de décrire le principe de cette méthodologie, nous rappelons d'abord qu'elle est dépendante des mêmes hypothèses de défaillance de la méthodologie AAA-FAULT^{DT}, sauf que les médias de communication sont des **liaisons point à point** ; en bref son modèle de faute et comme suit :

- L'algorithme est fiable et sans fautes,
- Fautes permanentes d'un seul processeur
- La liaison de communication est une liaison point à point, elle est supposée fiable.
- Le type de défaillance du processeur est arrêt sur défaillance (fail stop).
- Les tâches constituants l'architecture algorithmique sont dépendantes.

Pour commencer, nous transformons d'abord le graphe d'algorithme ALG en un nouveau graphe appelé ALG^N, dans lequel :

- Chaque tâche est répliquée en deux exemplaires, primaire t_i^p et secondaire t_i^b ;
- Deux nouvelles dépendances sont ajoutées au nouveau graphe ALG^N : ($t_i^p \leftrightarrow t_i^b$) et ($t_i^b \rightarrow t_j$) ; où t_j est le successeur de t_i

Les nouvelles dépendances sont de deux types : une dépendance de données et une dépendance conditionnelle. Cette dernière est une dépendance à double flèche, c'est-à-dire qu'elle transfère le signal de blocage de l'exécution des tâches dans les deux sens, du primaire au secondaire ou inversement, selon le premier terminant son exécution ; sa durée d'exécution est nulle. La dépendance de données transfère le résultat de la tâche secondaire t_i^b à son successeur t_j .

Ensuite, l'heuristique proposée mappe le nouveau graphe ALG^N et l'architecture matérielle AHM pour obtenir une distribution/ordonnancement optimale.

Dans des conditions normales (pas de défaillance d'un processeur), le corps de chaque tâche (principale ou secondaire) est le suivant :

Si (t_i termine son exécution) **Alors**

Envoyer un message de blocage à sa réplique pour minimiser la longueur de distribution/ordonnancement.

Si (t_i reçoit le message de blocage) **Alors**

Se bloque et donne la main à la prochaine tâche à s'exécuter

En cas de défaillance, la tolérance aux fautes consiste à poursuivre sans délai l'exécution normale des tâches en privilégiant la réplication active qui masque la défaillance du processeur en question. La réplique, d'une tâche défaillante, en cours d'exécution sur un autre processeur couvre cette défaillance et envoie son résultat à

son successeur (tâche t_j) via la connexion point à point (inter-processeur) ou via une communication intra-processeur.

Pour comprendre ce principe, commençons par l'exemple ci-dessous qui montre un algorithme de distribution/ordonnancement d'un graphe d'algorithme sur un graphe d'architecture dans lequel les tâches (resp. Communications) sont représentées par des cases de hauteur proportionnelle à leur temps d'exécution (resp. De communications).

Soit l'architecture matérielle hétérogène suivante appelée AHM composée de trois processeurs : P1, P2 et P3 reliés par les liaisons : l12, l13 et l23 et l'architecture de graphe d'algorithme ALG transformée en ALG^N dans laquelle chaque nœud est associé à ses temps d'exécution sur des processeurs différents, et chaque arc est associé à la durée de communication entre les nœuds qui relie.

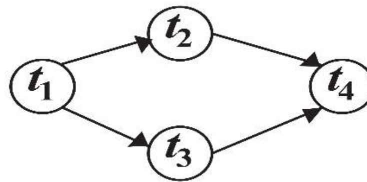


Figure 6.14. Example d'un graphe d'algorithme

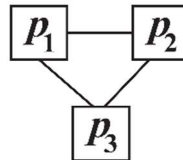


Figure 6.15. Example d'une connexion point à point

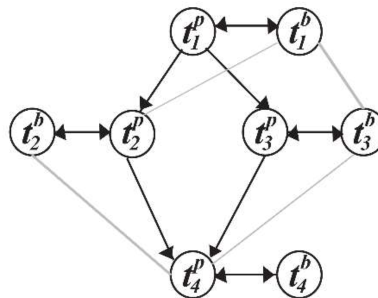


Figure 6.16. Schéma de transformation du graphe d'algorithme

Les processeurs étant hétérogènes, alors les temps d'exécution de chaque tâche sur chaque processeur sont indiqués dans le tableau suivant :

Tâches / Processeurs	T1	T2	T3	T4
P1	1	2	3	3
P2	2	1	1	2
P3	3	3	2	2

Tab1eau 6.1. Coût d'exécution des tâches sur différents processeurs

Nous supposons que les temps de transfert de données de chaque tâche sur la connexion point à point sont identiques sur toutes les liaisons. Le tableau 2 présente donc un exemple :

	T1	T2	T3
T2	2	/	/
T3	1	/	/
T4	/	3	2

Tab1eau 6.2. Coût de transfert de données entre tâches

Après adéquation de l'algorithme ALG^N sur l'architecture AHM, nous avons obtenu les résultats montrés aux figures 6.17 et 6.18

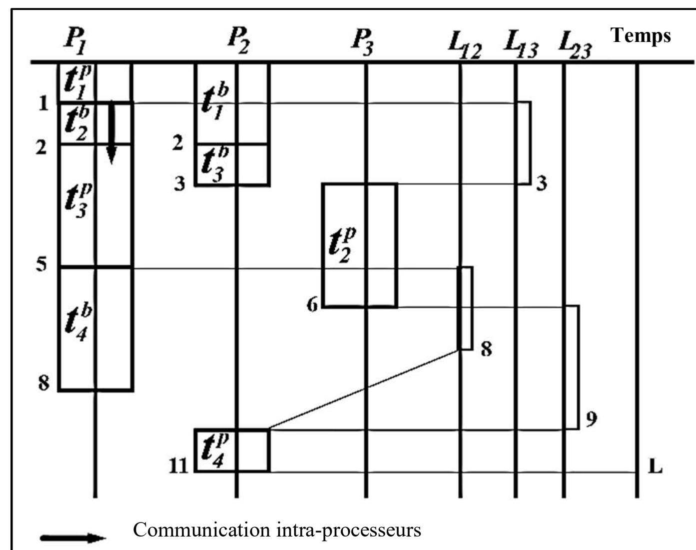


Figure 6.17. Réplication Active des tâches

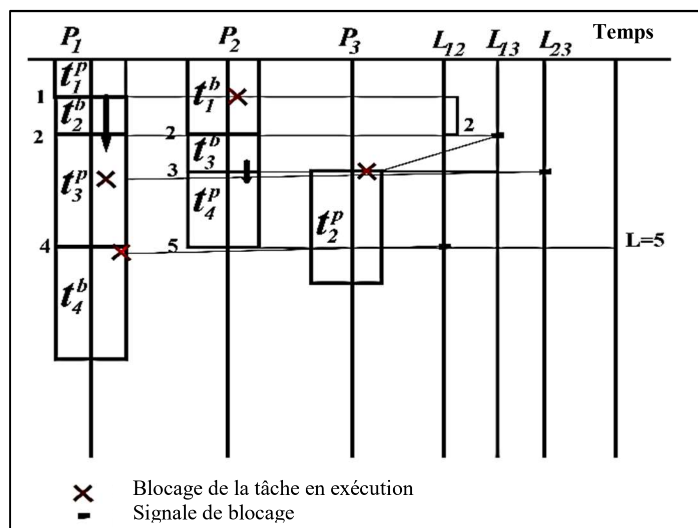


Figure 6.18. Réplication passive des tâches

En cas de défaillance du processeur, P1 par exemple lors de l'exécution de la tâche t_1^p, la réplication active de ses tâches sur P2 et P3 masque cette défaillance de la manière suivante :

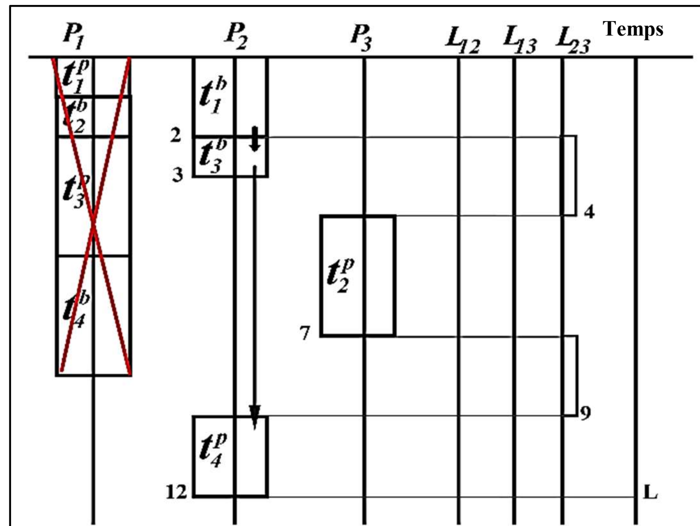


Figure 6.19. Distribution/ordonnancement après défaillance de P1

Pour résumer, Comme cette méthodologie est basée sur la redondance hybride et que nous tolérons les défaillances d'un processeur, alors :

- Chaque tâche est dupliquée seulement en deux copies
- Ils sont planifiés et exécutés en parallèle (réplication active)
- Le premier qui termine son exécution bloque l'autre (réplication passive)
- En cas d'échec, les copies qui se trouvent dans des processeurs distincts masquent cet échec et le système suit cette nouvelle planification.

6.5. Evaluation de la méthodologie FT-TDEP

Nous avons appliqué l'heuristique FT-TDEP à un ensemble de graphes d'algorithmes générés aléatoirement avec un ensemble de paramètres qui affectent nos résultats. Le nombre de processeurs P, le nombre de tâches N et le facteur CCR (rapport entre le temps de communication moyen et le temps d'exécution moyen) sont les paramètres que nous avons modifiés pour tester l'efficacité de notre méthodologie en l'absence/présence de défaillance. Nous l'avons donc comparée à la méthodologie AAA-FAULT^{DT}, qui est basée sur la réplication passive des tâches dépendantes dans une architecture de connexion à bus.

Nous avons tracé sur les figures 6.20, 6.21, 6.22, 6.23, 6.24 et 6.25 la longueur de distribution/ordonnancement L en fonction de N, P et CCR.

L : présente la longueur de la distribution/ordonnancement de la méthodologie FT-TDEP.

L': la longueur de la distribution/ordonnancement de la méthodologie AAA-FAULT^{DT}

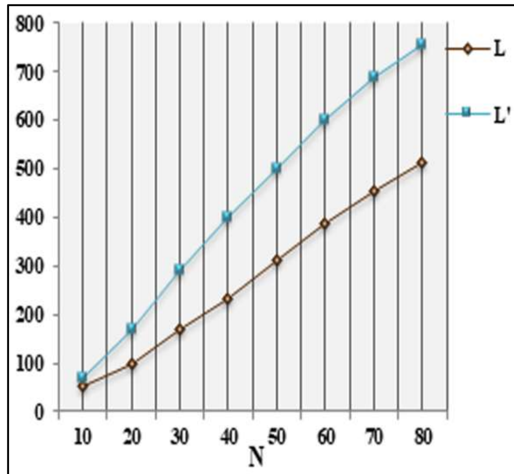


Figure 6.20. Effet de N sur AAA-FAULT^{DT} et FT-TDEP pour p=5 et CCR=2 en absence de fautes

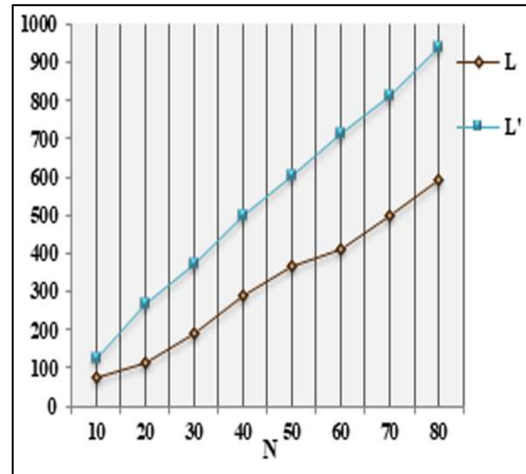


Figure 6.21. Effet de N sur AAA-FAULT^{DT} et FT-TDEP pour p=5 et CCR=2 en présence de fautes

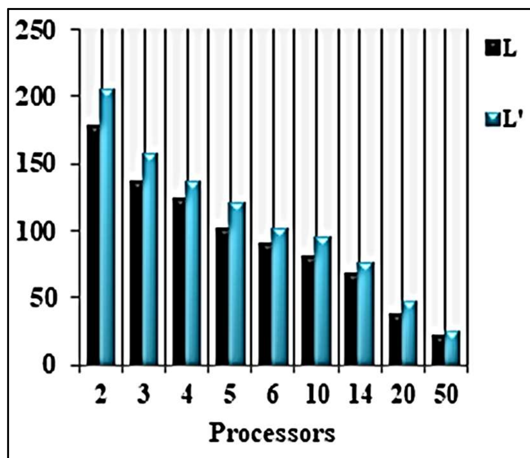


Figure 6.22. Effet de P sur AAA-FAULT^{DT} et FT-TDEP pour N=40 et CCR=1 en absence de fautes

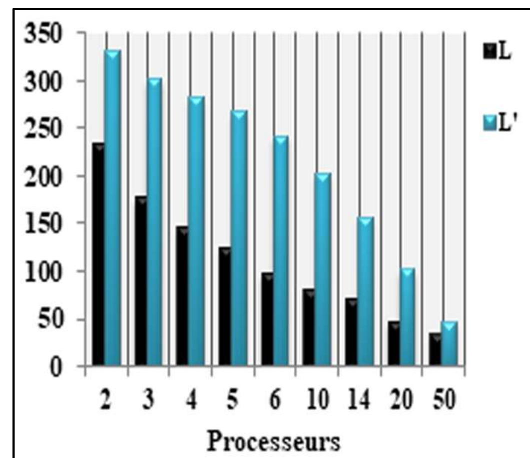


Figure 6.23. Effet de P sur AAA-FAULT^{DT} et FT-TDEP pour N=40 et CCR=1 en présence de fautes

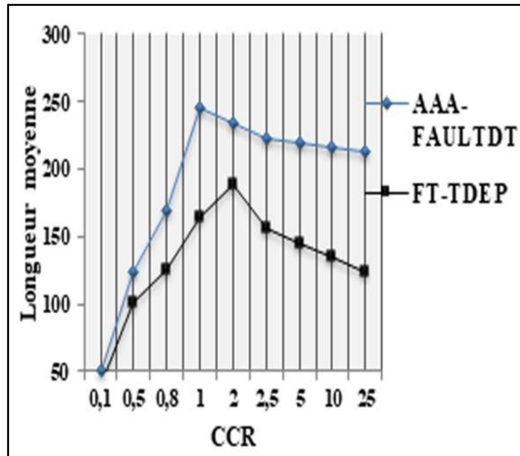


Figure 6.24. Effet de CCR sur AAA-FAULT^{DT} et FT-TDEP pour P = 6 et N =50 en absence de fautes

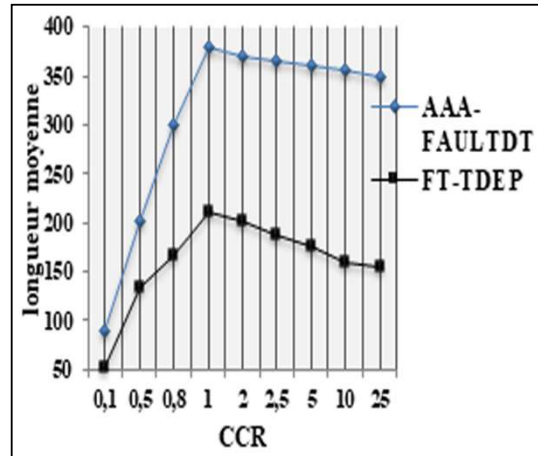


Figure 6.25. Effet de CCR sur AAA-FAULT^{DT} et FT-TDEP pour P = 6 et N =50 en présence de fautes

Les résultats de cette nouvelle méthodologie sont hautement meilleurs à la précédente, principalement dans le cas d'une défaillance du système. Son principe permet de réduire la longueur de la distribution/ordonnancement dans tous les cas, avec ou sans fautes. En l'absence de défaillance, le système gagne beaucoup de temps car il suit le principe du premier qui termine l'exécution, l'autre est bloqué ; cela influe également sur les communications qui deviennent souvent intra-processeur. En cas de défaillance, le système ne perd toujours pas de temps pour masquer cette défaillance car toutes les tâches sont en cours d'exécution.

6.6. Prédiction du comportement temps réel

La seule contrainte à vérifier après l'étape de distribution/ordonnancement est la contrainte temps réel C_{tr}, c'est-à-dire que la durée d'exécution de l'algorithme sur l'architecture doit être inférieure à un seuil défini par C_{tr}. La vérification de cette contrainte temps réel est effectuée hors-ligne :

- **En absence de défaillance** : dans ce cas, il suffit de vérifier si la longueur de la distribution/ordonnancement, qui égal à la date maximale entre les dates de la fin d'exécution de la dernière tâche de chaque processeur p_j, est inférieure à C_{tr}. Sinon l'heuristique échoue à trouver une distribution/ordonnancement qui satisfasse cette contrainte temps réel.
- **En présence de la défaillance d'un seul processeur** : sachant que chaque tâche a une date de début d'exécution en présence de faute d'un processeur, l'heuristique peut donc prédire la date de fin d'exécution de l'algorithme sur l'architecture en présence de défaillances. Cette date est égale à la date maximale de la fin d'exécution de la dernière tâche de chaque processeur p_j. Elle doit être inférieure à C_{tr}, sinon l'heuristique échoue à trouver une distribution/ordonnancement qui satisfasse cette contrainte temps réel.

Dans le cas où la contrainte temps réel n'est pas satisfaite, le concepteur doit soit ajouter des composants matériels à son architecture soit modifier ses contraintes, et ensuite réexécuter l'heuristique.

6.7. Conclusion

Nous avons présenté dans ce chapitre deux nouvelles méthodologies, ce sont des solutions pour assurer la sûreté de fonctionnement des systèmes temps réel embarqués stricts, elle tolère les fautes permanentes d'un seul processeur dans une architecture distribuée hétérogène. La première appelée AAA-FAULT^{DT} tolère les fautes dans une architecture à liaison bus, tandis que la deuxième appelée FT-TDEP assure la tolérance aux fautes dans une architecture point à point. L'architecture logicielle des deux méthodologies est un graphe flots de données avec des tâches ayants des relations de précédence (tâches dépendantes).

La méthodologie AAA-FAULT^{DT}, utilise la redondance passive d'où les tâches primaires s'exécutent sur des processeurs distincts des tâches secondaires ; ces dernières restent sans exécution jusqu'à la défaillance d'un processeur donnée ; dans ce cas elles seront activées par les tâches watchdog pour masquer la défaillance et rendre le service attendu. L'avantage de cette méthodologie s'avère bien dans le cas de la non défaillance, dans lequel il n'y a aucune surcharge sur les processeurs et donc aucun temps d'exécution ou de communication supplémentaires. Au même temps, même il y'a une défaillance et même la longueur de distribution/ordonnancement augmente ; le service attendu est fourni à temps, sans violer les contraints temps réels.

La méthodologie FT-TDEP est optimisation de la méthodologie AAA-FAULT^{DT}, car son principe de la réplication hybride plus la topologie point à point à permet considérablement de réduire la longueur de distribution/ordonnancement en présence et en absence de défaillance.

Chapitre 7

Présentation de l'algorithme de distribution/ordonnancement tolérant aux fautes des tâches indépendantes AAA-FAULT^{IDT}

7.1. Présentation de l'algorithme de distribution/ordonnancement tolérant aux fautes des tâches indépendantes AAA-FAULT^{IDT}

Notre solution est liée aux hypothèses de défaillances définies par le modèle de fautes suivant :

7.1.1. Modèle de fautes

Comme dans le cas de la tolérance aux fautes dans un système avec tâches dépendantes, la méthodologie utilisée dans ce cas est basée sur la redondance des composants logiciels mais la stratégie utilisée est la redondance active. Avant de présenter cette méthodologie, nous rappelons d'abord le modèle de fautes qui est le même pour la méthodologie AAA-FAULT^{DT} sauf que les tâches sont indépendantes.

7.1.2. Données du problème

Le problème peut être formalisé comme suit :

- Une architecture matérielle hétérogène AMH composée d'un ensemble P de processeurs et d'un ensemble L qui contient un bus B de communication (liaison à bus).
  $P = \{P_1, P_2, \ldots\ldots, P_m\}$, $L = \{B\}$

Exemple 1

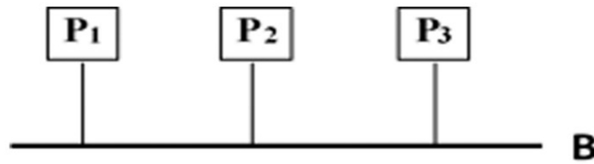


Figure 7.1. Architecture liaison à bus

- Un algorithme ALG, composé d'un ensemble T de tâches indépendantes.
  T= $\{t_1, t_2, \ldots\ldots t_n\}$

Exemple 2

Figure 7.2. Exemple d'un graphe d'algorithme avec tâches indépendantes

- Des caractéristiques d'exécution C_{exe} des composants de ALG sur les composants de AMH.
- Un ensemble de contraintes matérielles C_{mt} .
- Une contrainte temps réel C_{tr} .
- Les fautes d'un seul processeur qui peuvent causer la défaillance du système.

Comme la méthodologie précédente, l'objectif est la définition d'une application A dont le rôle est d'ordonnancer chaque tâche et chaque dépendance de données de ALG sur les processeurs et les bus de AMH ; l'ordonnancement est réalisé par l'affectation d'un ordre d'exécution O d'une tâche sur un processeur, ou d'une dépendance de donnée sur un bus.

7.1.3. Principes généraux de la méthodologie AAA-FAULT^{IDT}

Le problème de distribution/ordonnancement temps réel tolérant aux fautes peut être vu comme un problème à deux parties : Un problème de temps réel noté $\boldsymbol{P_{tr}}$ et un problème de tolérance aux fautes noté $\boldsymbol{P_{tf}}$.

Le problème de temps consiste à minimiser la longueur de distribution/ordonnancement, et le problème de tolérance aux fautes consiste à tolérer les fautes d'un seul processeur. C'est un problème d'optimisation car il s'agit de trouver une solution optimale, il a été démontré comme étant NP-difficile [25].

Théorème 1 (Kalla et al) **:** *Soient un algorithme constitué de plusieurs composants logiciels dépendants, et une architecture matérielle hétérogène constituée de plusieurs processeurs. La distribution/ordonnancement d'un tel algorithme sur une telle architecture visant la minimisation de la longueur de la distribution/ordonnancement est un problème NP-difficile.*

Le problème de tolérance aux fautes P_{tf} est résolu dans la littérature par plusieurs techniques [25, 33, 34, 37, 43, 44]. Etant donné que nous visons des systèmes embarqués, en raison de leurs ressources limitées (espace, poids et coûts) ; il est difficile voire impossible de fournir la redondance matérielle, alors la redondance logicielle est la technique utilisée dans ce travail pour réduire le coût physique et financière exigé par les systèmes embarqués.

En associant les deux problèmes P_{tr} et P_{tf} ensemble, le problème à résoudre est un problème NP-difficile. Afin de résoudre ce problème en temps polynomial, nous proposons une nouvelle méthodologie appelée **AAA-FAULT^{IDT}** dont les principes sont les suivants :

En plus du **principe 1** de la méthodologie précédente, nous présentons ainsi les principes suivants pour cette nouvelle méthodologie :

Principe 2 : Redondance active

Puisque les tâches sont indépendantes, la réplication active peut ne pas augmenter le temps d'exécution des tâches, surtout nous avons supposé que l'exécution des copies secondaires s'arrête si aucune défaillance n'a été détecté.

Principe 3 : Masquage d'erreurs

Si un processeur défaille, les copies secondaires des tâches en y implantées masquent cette erreur. Alors cette technique n'exige pas un mécanisme de détection d'erreurs.

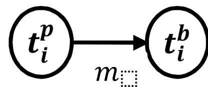
Principe 4 : Tolérance aux fautes permanentes

AAA-FAULT^{IDT} : Est une nouvelle méthodologie pour optimiser d'une part la génération automatique de distribution/ordonnancement des tâches indépendantes et d'autre part pour tolérer les fautes permanentes d'un seul processeur. Dans cette méthodologie, la tolérance aux fautes est basée en premier temps sur la redondance active des tâches indépendantes et en deuxième temps sur le blocage de l'exécution de la copie secondaire de chaque tâche non défaillante. Ce blocage sert à minimiser le surcoût qui est plus élevé dans la redondance active, donc notre méthodologie bénéficie des avantages des deux stratégies de redondance (active et passive). En effet, dans le cas de défaillance la méthodologie n'a pas besoin de la détecter, elle a un temps de réponse prévisible et une reprise immédiate après cette défaillance, et dans le cas où aucun processeur n'est en panne, c'est-à-dire l'exécution des répliques de chaque tâche sera bloquée ; alors la méthodologie AAA-FAULT^{IDT} bénéficie de la réduction du surcoût.

Elle est composée de deux phases : la phase de transformation du graphe et la phase d'adéquation de l'architecture logicielle sur l'architecture matérielle.

1. **La phase de transformation** consiste à transformer le graphe d'algorithme non redondant ALG à un nouveau graphe redondant ALG^n dont :
   - Chaque tâche est répliquée en deux copies, primaire t_i^p et secondaire t_i^b ;
   - Une nouvelle dépendance est ajoutée au nouvel ALG^n entre chaque tâche et sa réplique ($t_i^p \rightarrow t_i^b$), son temps d'exécution est nul.

   Chaque tâche t_i du graphe d'algorithme est transformée comme suit :



2. **Dans la phase d'adéquation**, l'heuristique proposée met en correspondance le nouveau graphe ALG^n et l'architecture matérielle AMH pour réaliser un ordonnancement optimal tolérant aux fautes.

La figure 7.3 schématise une représentation de la méthodologie AAA-FAULT^{IDT}.

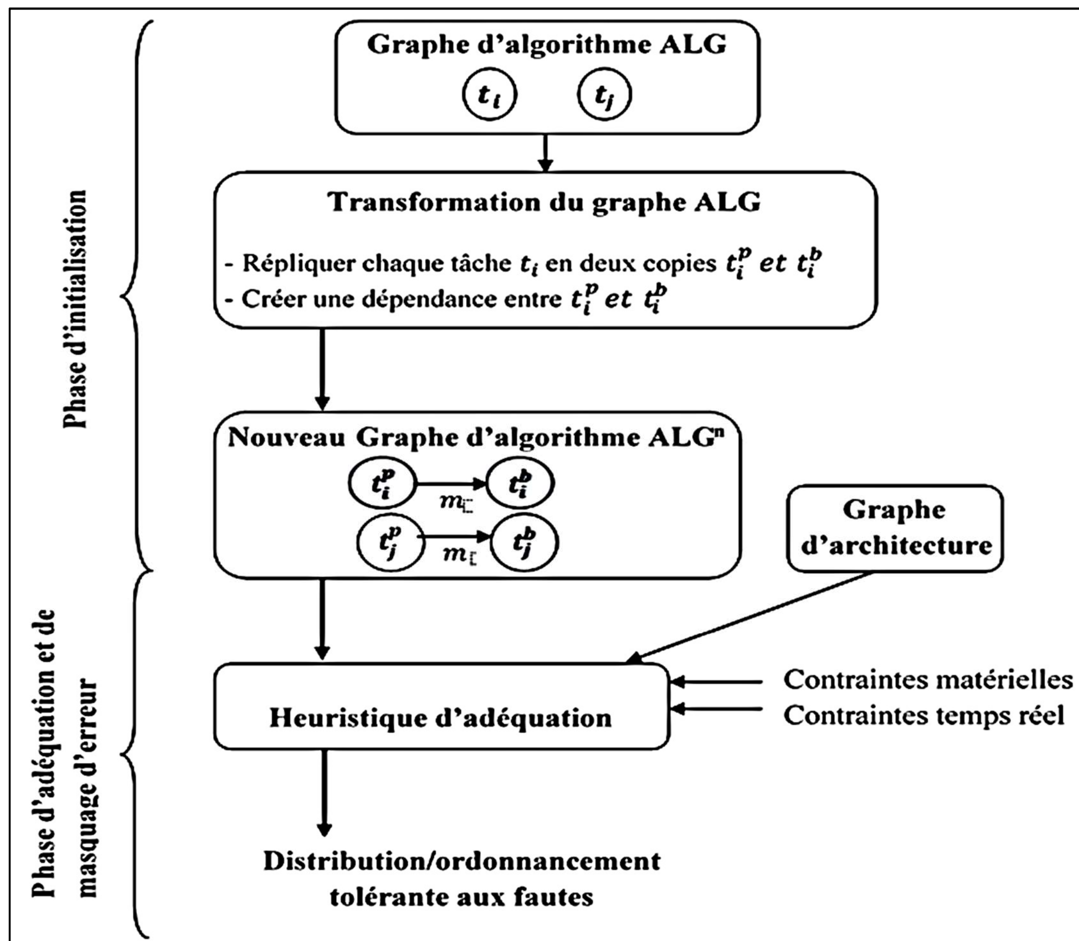


Figure 7.3. Méthodologie AAA-FAULT^{IDT}

La phase de transformation se résulte par un nouveau graphe ALG^n qui est, à la différence du premier graphe, un graphe orienté dans lequel chaque tâche t_i est répliquée en deux copies t_i^p et t_i^b liées par une dépendance conditionnelle.

L'exécution de cette dernière indique le bon déroulement de la tâche primaire, et son temps d'exécution est nul.

Puisque le système est sans faute logicielle, t_i^p et t_i^b sont identiques, c'est-à-dire qu'elles ont les mêmes résultats.

Un exemple de transformation est donné par la figure 7.4.

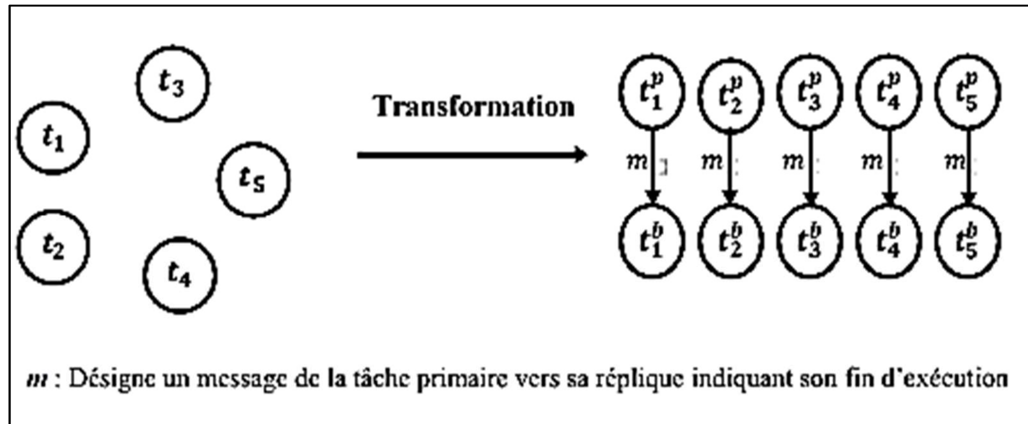


Figure 7.4. Exemple de transformation d'un graphe d'algorithme

Nous constatons que la transformation du graphe d'algorithme permet d'obtenir d'après un graphe sans dépendances de données un graphe avec dépendances orientées.

La phase d'adéquation fait placer l'architecture logicielle ALG^n sur l'architecture matérielle et détermine l'ordre d'exécution des tâches. Elle se base sur une distribution/ordonnancement hors-ligne où l'ordonnancement est statique, c'est-à-dire l'ordonnancement est préparé d'avance avant la mise en exploitation du système et reste le même pour tous les cycles d'exécution (sauf à l'apparition de défaillance). Dans l'ordonnancement statique, toutes les contraintes (matérielles, temps d'exécution, …) sont connues d'avance.

Nous avons supposé que le bus de communication soit fiable, Donc notre solution est basée sur la redondance active des composants logiciels pour tolérer les fautes d'un seul processeur. Pour cela chaque tâche t_i doit être répliquée et les deux répliques soient placées activement sur deux processeurs distincts (P_i , P_j). Si le processeur de la tâche primaire défaille alors sa réplique masque cette défaillance et l'ordonnancement sera reconfiguré, si non, à sa terminaison, elle envoie un message de blocage à sa réplique.

Les tâches principales de l'heuristique de distribution/ordonnancement basé sur la méthodologie AAA sont listées comme suit :

1. Calcule des dates de début de toutes les tâches du graphe d'algorithme en utilisant la pression d'ordonnancement [25]. Chaque tâche contient deux dates de début : T_{best} et T_{worst} .
   - $\boldsymbol{T_{best}}$ **:** c'est la date de début des copies primaires.

- $\boldsymbol{T_{worst}}$ **:** c'est la date de début des copies secondaires.

2. Placement actif des copies primaires et secondaires des tâches : au démarrage du système, les deux copies de chaque tâche sont exécutées.
3. Si chaque tâche t_i^b reçoive un message de sa copie primaire t_i^p, alors son exécution est ignorée jusqu'au prochain cycle.
4. Si une tâche t_i^b ne reçoive pas un message de t_i^p, alors son exécution masque la défaillance de cette dernière.
5. Prise en compte de toutes les contraintes : coût d'exécution, la contrainte temps réel et les contraintes matérielles.
6. Prévision du comportement temps réel (prédictibilité) : le calcul des dates de début d'exécution permet de vérifier hors-ligne si la contrainte temps réel est respectée ou non.

Remarque :

Nous avons considéré que chaque tâche contient deux dates de début, puisque la réplique d'une tâche, qui est placée sur un autre processeur, est ordonnancée après une ou plusieurs tâches primaires, ce qui signifie que $\boldsymbol{T_{best}}$ est inférieur à $\boldsymbol{T_{worst}}$ ($\boldsymbol{T_{best}} < \boldsymbol{T_{worst}}$) .

Principes de l'heuristique

L'heuristique que nous proposons est basée sur une fonction de coût appelée la pression d'ordonnancement notée $\sigma^n_{(Ti,Pj)}$, donnée par l'équation (4.5), dont l'objectif global est de minimiser la longueur de distribution/ordonnancement en absence et en présence de défaillance d'un seul processeur.

Nous présentons dans ce qui suit les principes de l'heuristique :

- Toutes les tâches s'exécutent au même temps, selon leur date de début d'exécution.
- Si aucun processeur n'est en panne, l'exécution des répliques de toutes les tâches sera ignorée.
- A la défaillance d'un processeur, les répliques de ses tâches masquent cette défaillance, et une mise à jour de l'ordonnancement est effectuée en éliminant le processeur défaillant et les tâches répliques.

Nous pouvons alors représenter ces principes comme suit :

Début
Execution active de toutes les taches (primaires et secondaires) par ordre d'execution
Pour chaque tâche primaire & après un temps t
$\boldsymbol{T_i^p}$ envoie un message à son réplique $\boldsymbol{T_i^b}$
$\boldsymbol{T_i^b}$ ignore son exécution et se bloque
Si $\boldsymbol{T_i^b}$ n'a pas reçu un message de $\boldsymbol{T_i^p}$ alors
$\boldsymbol{T_i^b}$ continue son exécution et masque la défaillance de $\boldsymbol{T_i^p}$
Fin

Pour bien assimiler le principe de notre méthodologie, l'exemple ci-dessous présente une distribution/ordonnancement d'un graphe d'algorithme sur un graphe d'architecture dans laquelle les tâches sont représentées par des boites dont la hauteur est proportionnelle à leur durée d'exécution.

Prenant l'architecture logicielle ALG qui consiste en trois tâches indépendantes t_1, t_2 et t_3 et une architecture matérielle AHM composée de trois processeurs hétérogènes P_1, P_2 et P_3 connectés via un bus. Les temps d'exécution des tâches sur ces processeurs sont respectivement : (1,3), (2,1) et (2,4). ALG est transformé en ALG^n dans lequel chaque nœud est répliqué en deux exemplaires avec ses temps d'exécution sur différents processeurs, la durée de communication entre les nœuds est nulle.

Soit l'architecture matérielle suivante :

P= {P_1, P_2, P_3}, L={Bus}

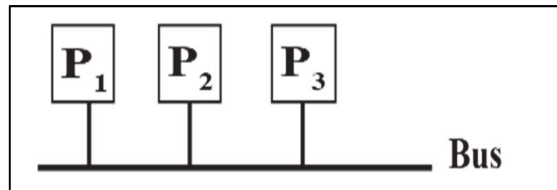


Figure 7.5 : Architecture matérielle

Et soit le graphe d'architecture logicielle ALG transformé en ALG^n :

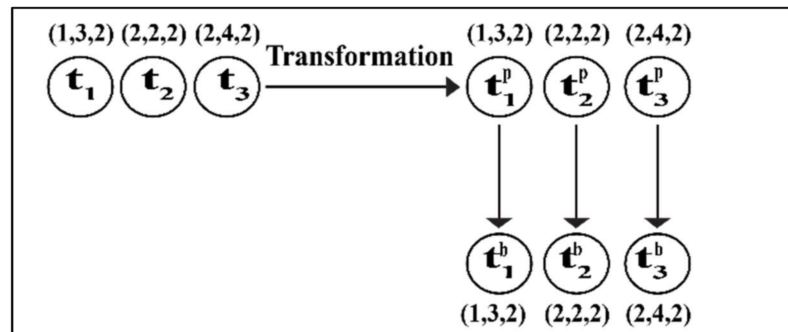


Figure 7.6 Architecture logicielle

Après application de la méthodologie AAA-FAULT^{IDT} sur cette spécification avant et après l'apparition d'une défaillance d'un processeur, nous avons obtenu les résultats résumés dans les figures 7.7 et 7.8

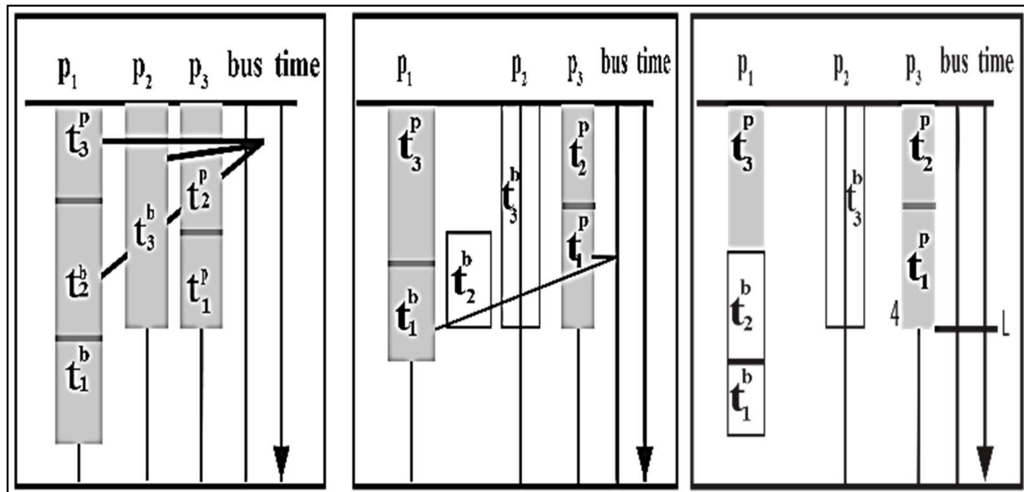


Figure 7.7 Distribution/ordonnancement sans défaillance

En cas de panne, P1 par exemple, la tâche t_3^b ne reçoit pas un message de la tâche t_3^p, alors elle continue à s'exécuter et masquer cette défaillance. Ainsi, la distribution/ordonnancement devient :

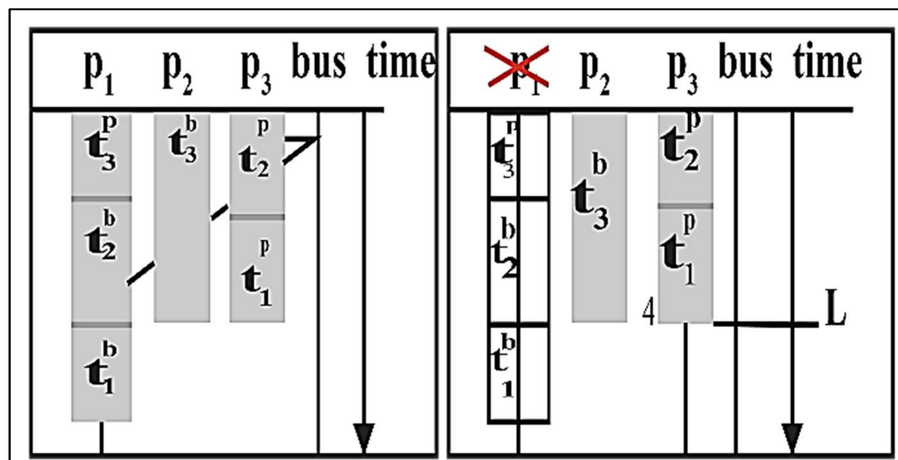


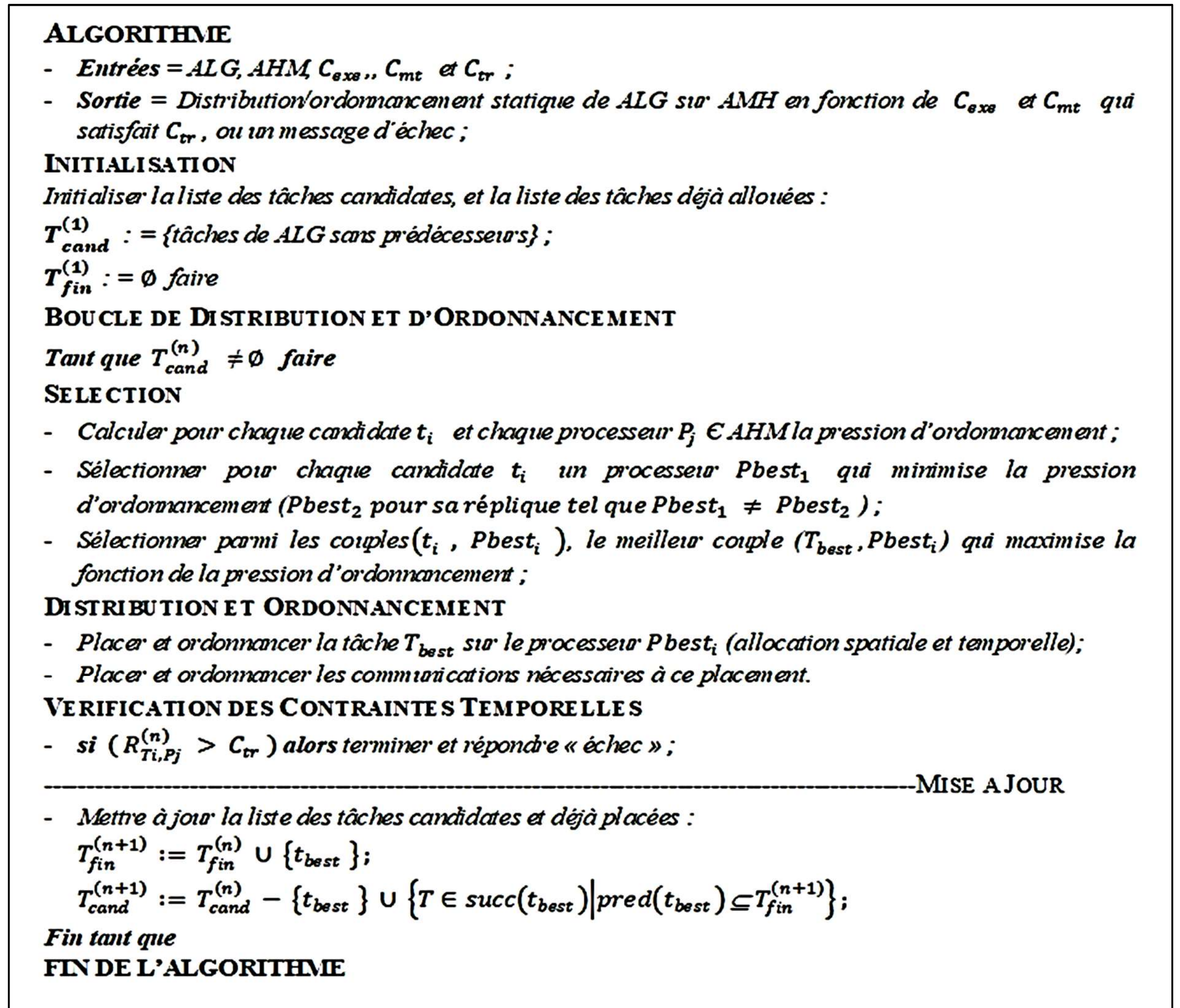
Figure 7.8. Distribution/ordonnancement avec défaillance de P1

Note :

—— : Présente le message de blocage (temps de communication est nul)

Présentation de l'heuristique

L'heuristique proposée consiste à placer et à ordonnancer à chaque étape n (cycle) de l'exécution du système l'ensemble des tâches du nouveau graphe d'algorithme sur le graphe d'architecture. Elle est composée de cinq étapes, l'étape d'initialisation, l'étape de sélection, l'étape de distribution/ordonnancement, l'étape de vérification du respect de la contrainte temps réel et finalement l'étape de mise à jour.



ALGORITHME
- *Entrées = ALG, AHM, C_{exe}, C_{mt} et C_{tr} ;*
- *Sortie = Distribution/ordonnancement statique de ALG sur AMH en fonction de C_{exe} et C_{mt} qui satisfait C_{tr}, ou un message d'échec ;*

INITIALISATION
Initialiser la liste des tâches candidates, et la liste des tâches déjà allouées :
$T^{(1)}_{cand}$: = {tâches de ALG sans prédécesseurs} ;
$T^{(1)}_{fin}$: = $\emptyset$ faire

BOUCLE DE DISTRIBUTION ET D'ORDONNANCEMENT
Tant que $T^{(n)}_{cand} \neq \emptyset$ faire

SELECTION
- *Calculer pour chaque candidate t_i et chaque processeur $P_j \in AHM$ la pression d'ordonnancement ;*
- *Sélectionner pour chaque candidate t_i un processeur $Pbest_1$ qui minimise la pression d'ordonnancement ($Pbest_2$ pour sa réplique tel que $Pbest_1 \neq Pbest_2$) ;*
- *Sélectionner parmi les couples$(t_i\ ,\ Pbest_i\ )$, le meilleur couple $(T_{best}, Pbest_i)$ qui maximise la fonction de la pression d'ordonnancement ;*

DISTRIBUTION ET ORDONNANCEMENT
- *Placer et ordonnancer la tâche T_{best} sur le processeur $Pbest_i$ (allocation spatiale et temporelle);*
- *Placer et ordonnancer les communications nécessaires à ce placement.*

VERIFICATION DES CONTRAINTES TEMPORELLES
- *si ($R^{(n)}_{Ti,Pj} > C_{tr}$) alors terminer et répondre « échec » ;*

--**MISE A JOUR**
- *Mettre à jour la liste des tâches candidates et déjà placées :*

$T^{(n+1)}_{fin} := T^{(n)}_{fin} \cup \{t_{best}\};$

$T^{(n+1)}_{cand} := T^{(n)}_{cand} - \{t_{best}\} \cup \{T \in succ(t_{best}) | pred(t_{best}) \subseteq T^{(n+1)}_{fin}\};$

Fin tant que
FIN DE L'ALGORITHME

Figure 7.9. L'algorithme AAA-FAULT^{IDT}

7.2. Evaluation de la méthodologie AAA-FAULT^{IDT}

Comme l'heuristique précédente, nous évaluons, relativement à la méthodologie AAA-FAULT^{DT}, le surcoût de la longueur de distribution/ordonnancement introduit par la méthodologie AAA-FAULT^{IDT} en présence et en absence d'une défaillance d'un seul processeur. Alors Les paramètres que nous avons modifiés pour tester l'efficacité de cette méthodologie sont les même que AAA-FAULT^{DT}.

Nous avons appliqué AAA-FAULT^{IDT} à un ensemble de graphes d'algorithmes générés aléatoirement par le générateur de graphe aléatoire décrit précédemment (paragraphe 6.3.1). Les résultats obtenus sont discutés ci-dessous.

7.2.1. Les résultats

En raison du principe de cette méthodologie qui combine les avantages de la redondance passive et active, les résultats obtenus sont meilleurs par rapport à la méthodologie AAA-FAULT^{DT}. Nous avons l'appliqué à un ensemble de graphes aléatoires avec un ensemble de paramètres. On note par :

L : Est la longueur de distribution/ordonnancement temps réel des tâches dépendantes (AAA-FAULT^{DT}).

L': Est la longueur de distribution/ordonnancement temps réel des tâches indépendantes (AAA-FAULT^{IDT}).

***** : dénote DT ou IDT

En faisant varier le nombre de tâches sur des graphes d'architecture aléatoires et une architecture matérielle, nous obtenons les résultats suivants (figure 7.10).

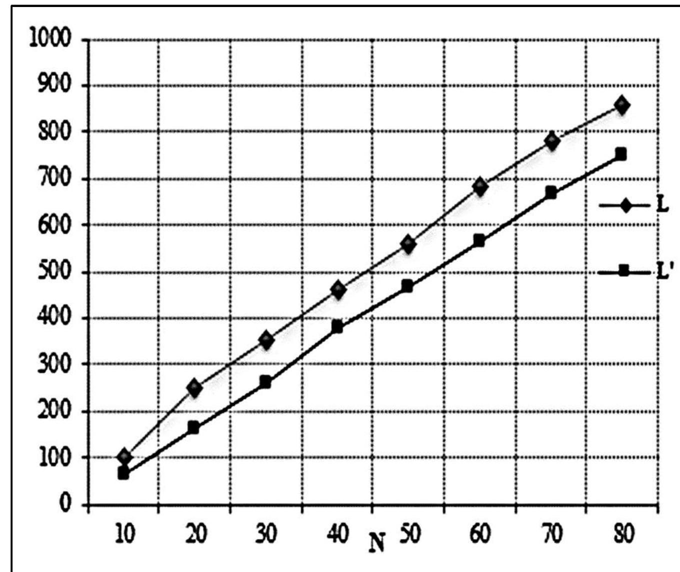


Figure 7.10. Effet de N sur AAA-FAULT* pour P=5 et CCR=2

En variant le nombre de processeurs sur des graphes d'architecture aléatoires de 50 tâches, nous avons obtenus les résultats suivants (figure 7.11).

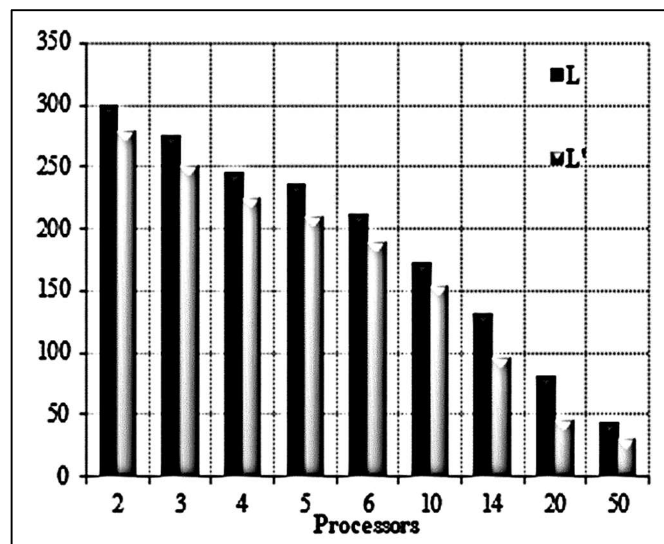


Figure 7.11. Effet de P sur AAA-FAULT* pour N=40 et CCR=1

Les résultats de cette méthodologie montrent qu'elle est meilleure que la précédente. En fait, comme nous l'avons déjà cité, le délai de récupération après défaillances et le délai d'exécution sans défaillance sont faibles.

7.3. Conclusion

Ce chapitre présente la troisième méthodologie appelée AAA-FAULT^{IDT}, qui a pour objectif de tolérer les fautes permanentes d'un processeur unique en utilisant la technique de la redondance active. Avec cette heuristique, chaque tâche et sa réplique s'exécutent à la fois selon la date de début d'exécution de chacune, mais au moment où la tâche primaire termine son exécution ; elle envoie un message à sa réplique, qui est allouée sur un autre processeur, pour bloquer son exécution et éviter la perte du temps d'exécution des autres tâches implantées sur ce même processeur. De cette façon nous pouvons masquer la défaillance des tâches et de plus gagner le temps dans le cas contraire.

Chapitre 8

Conclusion Générale et perspectives

8.1. Conclusion et perspectives

Les systèmes temps-réel embarqués critiques intègrent un nombre croissant de fonctionnalités comme les domaines de l'automobile, la santé, l'aéronautique. Ces systèmes doivent offrir un niveau maximal de sûreté de fonctionnement en disposant des mécanismes pour traiter les défaillances éventuelles, avec le respect de contraintes temps-réel strictes. Ces systèmes sont en outre contraints par leur nature embarquée : les ressources sont limitées, tels que par exemple leur coût et leur taille. Dans cette thèse, nous avons traité le problème de l'ordonnancement tolérant aux fautes de ce type de systèmes.

Comme la sûreté de fonctionnement d'un système temps réel critique est une propriété cruciale qui doit être prise en compte dans le processus de conception, notre travail est basé sur les techniques de tolérance aux fautes ; exactement la tolérance aux fautes logicielles pour tolérer les fautes matérielles des processeurs dans un système distribué hétérogène non préemptif connecté par une liaison point à point ou multipoints toute en respectant les contraints temps réel et les contraintes de distribution. Le problème de distribution/ordonnancement temps réel tolérant aux fautes peut être vu comme un problème à deux parties : Un problème de temps réel et un problème de tolérance aux fautes. Le problème de temps consiste à minimiser la longueur de distribution/ordonnancement, et le problème de tolérance aux fautes consiste à tolérer les fautes d'un seul processeur. C'est un problème d'optimisation car il s'agit de trouver une solution optimale, il a été démontré comme étant NP-difficile. Pour aboutir ces fins, nous avons décrire trois nouvelles heuristiques qui se distinguent par :

- **Tâches dépendantes ou indépendantes**
- **Redondance active, passive ou hybride**
- **Liaison point à point ou multipoints**

La première heuristique appelée AAA-FAULT^{DT}, utilise la technique de redondance passive pour assurer la tolérance aux fautes. Dans ce cas, les copies secondaires placées dans des processeurs différents de leur copies primaires restent inactifs jusqu'à la détection d'une défaillance par les tâches watchdog. Dans le cas de la non défaillance, la méthodologie profite le gain du temps comme effet de la non surcharge des processeurs. Dans le cas échoué, la méthodologie masque la défaillance et rendre le service attendu tout en respectant la contrainte de temps (deadline).

La deuxième heuristique appelée FT-TDEP, est une amélioration de la première méthodologie, elle se distingue par le média de communication qui est dans ce cas une liaison point à point et par le type de la redondance utilisé qui est la redondance hybride. Les résultats montrent l'efficacité de cette technique, par rapport à la précédente, d'une part dans la réduction considérable de la durée de distribution/ordonnancement en l'absence et en présence de défaillance, puis dans la tolérance aux pannes d'autre part.

La troisième heuristique appelée AAA-FAULT^{IDT}, est basée sur la redondance active des composants logiciels afin de satisfaire les contraintes temps réel et d'embarquabilité. Dans cette méthodologie, chaque tâche et son réplique s'exécutent à la fois sur deux processeurs distincts, à la fin d'exécution de la copie primaire, la copie secondaire se bloque et son exécution s'annule pour éviter la perte de temps d'exécution des autres tâches implantées sur son processeur. De cette façon nous pouvons rendre le service attendu en présence de fautes tout en respectant les contraintes temps réel à savoir le deadline, et de plus gagner le temps dans le cas contraire.

Nos solutions permettent de minimiser la durée de distribution/ordonnancement en absence et en présence de défaillance dans le cas de tâches dépendantes ou indépendantes.

Pour conclure, voici quelques perspectives en vue d'améliorer nos heuristiques.

1. Nous avons utilisé la liaison à bus pour relier les processeurs ce qui permet d'augmenter la longueur de la distribution/ordonnancement, utiliser la liaison point à point permet de réduire cette longueur.
2. Dans cette thèse nous nous sommes restreints à l'ordonnancement hors ligne non préemptif des calculs et des communications avec une seule contrainte temporelle de latence. Dans le but d'élargir le champ d'application de notre méthodologie, il faudrait introduire la possibilité de spécifier des applications devant respecter plusieurs contraintes temporelles. Pour cela il sera nécessaire d'introduire de la préemption dans les ordonnancements hors-ligne, en essayant de minimiser son effet, car son coût est loin d'être négligeable.
3. Les fautes matérielles considérées sont les fautes d'un seul processeur, nous proposons que nos heuristiques s'élargir pour tolérer les fautes de K processeurs.

4. Les liaisons de communications sont supposées fiables, alors nous proposons dans un travail future leur tolérance aux fautes probables.
5. Les trois heuristiques doivent être testées sur un outil de simulation plus puissant à savoir l'outils SynDEx afin de mieux comprendre l'influence des paramètres.
6. Tester ces solutions sur des cas réels.
7. Enfin, Les éléments essentiels des systèmes réactifs embarqués sont les capteurs sensibles aux valeurs issues de l'environnement extérieur contrôlé. La question la plus importante qui se pose est surtout comment adapter nos algorithmes d'ordonnancement à un problème spécifique de fautes de capteurs.

Références bibliographiques

[1] N. ABDALLAH. PARTITIONNEMENT TEMPS REEL MULTIPROCESSEUR SOUS CONTRAINTES DE QUALITE DE SERVICE ET D'ENERGIE. SYSTEMES EMBARQUES. THESE DE DOCTORAT DE L'UNIVERSITE DE NANTES, 2014.

[2] C. PAGETTI, SYSTEMES TEMPS REEL : LANGAGES DE PROGRAMMATION DE SYSTEMES TEMP REEL. ENSEEIHT - DEPARTEMENT TELECOMMUNICATION ET RESEAUX, TOULOUSE, 2014.

[3] N. NAVET, INTRODUCTION AUX SYSTEMES TEMPS REEL. HTTP://WWW.LORIA.FR/~NNAVET, 2010/2011.

[4] A. QUEUDET. ORDONNANCEMENT TEMPS REEL AVEC CONTRAINTES DE QUALITE DE SERVICE. THESE DE DOCTORAT DE L'UNIVERSITE DE NANTES, 2006.

[5] M. BOUKHANOUFA, ADAPTABILITE ET RECONFIGURATION DES SYSTEMES TEMPS-REEL EMBARQUES. THESE DE DOCTORAT DE L'UNIVERSITE PARIS-SUD, 2012.

[6] L. BARBE, INFORMATIQUE INDUSTRIELLE – COURS76 : LES SYSTEMES EMBARQUES. INSA STRASBOURG.

[7] OMAR KERMIA , ORDONNANCEMENT TEMPS REEL MULTIPROCESSEUR DE TACHES NON-PREEMPTIVES AVEC CONTRAINTES DE PRECEDENCE ,DE PERIODICITE STRICTE ET DE LATENCE, THESE 2009

[8] N.PERNET, Y.SOREL , SPECIFICATION ET IMPLANTATION DES SYSTEMES DISTRIBUES TEMPS REEL DE CONTROLE ET TRAITEMENT DE DONNEES. PROJET AOSTE - INRIA ROCQUENCOURT , FRANCE, 2014.

[9] M. OULD SASS, LE MODELE BGW POUR LES SYSTEMES TEMPS REEL SURCHARGES, SYSTEMES EMBARQUES. THESE DE DOCTORAT DE L'UNIVERSITE DE NANTES, 2015.

[10] M. MAROUF, ORDONNANCEMENT TEMPS REEL DUR MULTIPROCESSEUR TOLERANT AUX FAUTES APPLIQUEES A LA ROBOTIQUE MOBILE. THESE DE DOCTORAT, ECOLE NATIONALE SUPERIEURE DES MINES DE PARIS, 2012. (CITE EN PAGES 20,21,48,49)

[11] C. ARAR, REDONDANCE LOGICIELLE POUR LA TOLERANCE AUX FAUTES DES COMMUNICATIONS. THESE DE DOCTORAT, UNIVERSITE BATNA 2, 2016. (CITE EN PAGE 18).

[12] C. PAGETTI, MODULE DE SURETE DE FONCTIONNEMENT, ENSEEIHT 3EME TR - OPTION SE. 2012. (CITE EN PAGE 3).

[13] M. SGHAIRI HAOUATI. ARCHITECTURES INNOVANTES DE SYSTEMES DE COMMANDES DE VOL. COMPUTER SCIENCE. INSTITUT NATIONAL POLYTECHNIQUE DE TOULOUSE - INPT, 2010. FRANCE.

[14] X. BESSERON , TOLERANCE AUX FAUTES ET RECONFIGURATION DYNAMIQUE POUR LES APPLICATIONS DISTRIBUEES A GRANDE ECHELLE. THESE DE DOCTORAT, INSTITUT NATIONAL POLYTECHNIQUE DE GRENOBLE - INPG, 2010. FRANÇAIS. (CITE EN PAGE 26, 27, 28).

[15] Jean Arlat, Yves Crouzet, Yves Deswarte, Jean-Charles Fabre, JeanClaude Laprie et David Powell : Encyclopedie de l'Informatique et des Systemes d'Information, chapitre Tolerance aux fautes, pages 241–270. Vuibert, Paris, France, 2006.

[16] J.C. Laprie, "Surete de fonctionnement des systemes informatiques et tolerance aux fautes", Les Techniques de l'Ingenieur, n° H.4.450 (septembre 1989), pp. 1-12.

[17] D. Powell, G. Bonn, D. Seaton, P. Verissimo, F. Waeselynck, "The Delta-4 approach to dependability in open distributed computing systems", Conference: Proceedings of the Eighteenth International Symposium on Fault-Tolerant Computing, FTCS 1988, Tokyo, Japan, 27-30 June, 1988, IEEE, pp. 246-251.

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

[19] A. Avizienis, J.C. Laprie et B. Randell, Fundamental Concepts of Dependability. In Proceedings of the 3rd IEEE Information Survivability Workshop (ISW-2000), Boston, Massachusetts, USA, October 24-26, 2000 pp. 7-12

[20] Avizienis A., Laprie JC., Randell B. Dependability and Its Threats: A Taxonomy. In: Jacquart R. (eds) Building the Information Society. IFIP International Federation for Information Processing, vol 156. Springer, Boston, MA, 2004 pp 91-120

[21] J.C. Laprie, Dependable Computing: Concepts, Limits, Challenges. Invited paper to FTCS -25, the 25th IEEE International Symposium on Fault-Tolerant Computing, Pasadena, California, USA, June 27-30, 1995, Special Issue, pp. 42-54.

[22] S. Krakowiak, Tolerance aux fautes – 1 Introduction, techniques de base. École Doctorale de Grenoble Master 2 Recherche "Systemes et Logiciel", Universite Joseph Fourier Projet Sardes (INRIA et IMAG-LSR). 2003-2004.

[23] A.avizienis, Fault-Tolerant Systems. IEEE transactions on computers, vol. c-25, no. 12, december 1976.

[24] A. Zammali, Approche d'integrite bout en bout pour les communications dans les systemes embarques critiques : application aux systemes de commande de vol d'helicopteres. These de Doctorat, Toulouse III Paul Sabatier, 2016. (Cite en page 20,21,22).

[25] H. Kalla, Generation automatique de distributions/ordonnancements temps reel, fiables et tolerants aux fautes. These de Doctorat, Networking and Internet Architecture ; Institut National Polytechnique de Grenoble - INPG, 2004.

[26] A. Mehiaoui, Techniques d'analyse et d'optimisation pour la synthese architecturale de systemes temps reel embarques distribues : problemes de placement, de partitionnement et d'ordonnancement. These de Doctorat, Computer Aided Engineering. Universite de Bretagne occidentale - Brest, 2014.

[27] M. Abdallah, Ordonnancement temps reel pour l'optimisation de la Qualite de Service dans les systemes autonomes en energie. These de Doctorat, Systemes embarques. Universite de Nantes, 2014. (Cite en page 26).

[28] P. LOPEZ, APPROCHE PAR CONTRAINTES DES PROBLEMES D'ORDONNANCEMENT ET D'AFFECTATION : STRUCTURES TEMPORELLES ET MECANISMES DE PROPAGATION. THESE DE DOCTORAT, INSTITUT NATIONAL POLYTECHNIQUE DE TOULOUSE - INPT, 2003. (CITE EN PAGE 4).

[29] A. FERNANDO DIAS, CONTRIBUTION A L'IMPLANTATION OPTIMISEE D'ALGORITHMESBAS NIVEAU DE TRAITEMENT DU SIGNAL ET DES IMAGES SUR DES ARCHITECTURES MONO-FPGA A L'AIDE D'UNE METHODOLOGIE D'ADEQUATION ALGORITHME ARCHITECTURE. THESE DE DOCTORAT, L'UNIVERSITE PARIS XI ORSAY, 2000. (CITE EN PAGE 56).

[30] C. LAVARENNE, O. SEGHROUCHNI, Y. SOREL, AND M. SORINE. THE SYNDEX SOFTWARE ENVIRONMENT FOR REAL-TIME DISTRIBUTED SYSTEMS DESIGN AND IMPLEMENTATION. IN EUROPEAN CONTROL CONFERENCE, ECC'9, JULY 1991.

[31] A. VICARD. FORMALISATION ET OPTIMISATION DES SYSTEMES INFORMATIQUES DISTRIBUES TEMPS-REEL EMBARQUES. THESE DE DOCTORAT, UNIVERSITE PARIS XIII, JULY 1999.

[32] T. YANG AND A. GERASOULIS. LIST SCHEDULING WITH AND WITHOUT COMMUNICATION DELAYS. PARALLEL COMPUTING, 19(12) :1321–1344, 1993.

[33] PARAMESWARAN RAMANATHAN, KANG G. SHIN. DELIVERY OF TIME-CRITICAL MESSAGES USING A MULTIPLE COPY APPROACH, ACM TRANSACTIONS ON COMPUTER SYSTEMS, VOL 10, NO 2, MAY 1992, PAGES 144-166

[34] HECTOR GARCIA-MOLINA, BEN KAO ET DANIEL BARBARA. AGRESSIVE TRANSMISSIONS OVER REDUNDANT PATHS FOR TIME CRITICAL MESSAGES. UNIVERSITE STANFORD, CA, USA 1993.

[35] KOJI HASHIMOTO, TATSUHIRO TSUCHIYA ET TOHRU KIKUNO. EFFECTIVE SCHEDULING OF DUPLICATED TASKS FOR FAULT TOLERANCE IN MULTIPROCESSOR SYSTEMS. IEICE TRANS.INF.& SYST, VOL.E85-D, NO.3 MARS 2002

[36] ALAIN GIRAULT, HAMOUDI KALLA ET YVES SOREL. AN ACTIVE REPLICATION SCHEME THAT TOLERATE FAILURES IN DISTRIBUTED EMBEDDED REAL-TIME SYSTEMS, PROCESSORS AND COMMUNICATION LINKS FAILURES. IN: KLEINJOHANN B., GAO G.R., KOPETZ H., KLEINJOHANN L., RETTBERG A. (EDS) DESIGN METHODS AND APPLICATIONS FOR DISTRIBUTED EMBEDDED SYSTEMS. DIPES 2004. IFIP INTERNATIONAL FEDERATION FOR INFORMATION PROCESSING, VOL 150. SPRINGER, BOSTON, MA

[37] NAGARAJAN KANDASAMY, JOHN P. HAYES ET BRIAN T. MURRAY. DEPENDABLE COMMUNICATION SYNTHESIS FOR DISTRIBUTE EMBEDDED SYSTEMS, INTERNATIONAL CONFERENCE ON COMPUTER SAFETY, RELIABILITY, AND SECURITY, SEPTEMBER 2003.

[38] L. SONNA MOMO, REPLICATION ET DURABILITE DANS LES SYSTEMES REPARTIS. 19 FEVRIER 2001

[39] X. QIN, Z.F. HAN, H. JIN, L. P. PANG, AND S. L. LI. REAL-TIME FAULT-TOLERANT SCHEDULING IN HETEROGENEOUS DISTRIBUTED SYSTEMS. IN PROCEEDING OF THE INTERNATIONAL WORKSHOP ON CLUSTER COMPUTING-TECHNOLOGIES, ENVIRONMENTS, AND APPLICATIONS (CC-TEA'2000), LAS VEGAS, USA, JUNE 2000.

[40] Y. OH AND S. H. SON. SCHEDULING REAL-TIME TASKS FOR DEPENDABILITY. JOURNAL OF OPERATIONAL RESEARCH SOCIETY, 48(6) :629–639, JUNE 1997.

[41] S. HAN AND K.G. SHIN. FAST RESTORATION OF REAL-TIME COMMUNICATION SERVICE FROM COMPONENT FAILURES IN MULTI-HOP NETWORKS. IN SIGCOMM SYMPOSIUM, SEPTEMBER 1997.

[42] PASCAL. CHEVOCHOT AND ISABELLE. PUAUT. SCHEDULING FAULT-TOLERANT DISTRIBUTED HARD REAL-TIME TASKS INDEPENDENTLY OF THE REPLICATION

STRATEGIE. IN THE 6TH INTERNATIONAL CONFERENCE ON REAL-TIME COMPUTING SYSTEMS AND APPLICATIONS (RTCSA'99), PAGES 356–363, HONGKONG, CHINA, DECEMBER 1999.

[43] A. ZOMAYA. PARALLEL AND DISTRIBUTED COMPUTING HANDBOOK. MCGRAW-HILL, 1996.

[44] HADRIEN CAMBAZARD, PIERRE-EMMANUEL HLADIK, ANNE-MARIE, DEPLANCHE NARENDRA JUSSIEN. UNE CONTRAINTE GLOBALE POUR L'ORDONNANCABILITE DES TACHES TEMPS REEL DUR, CORK CONSTRAINT COMPUTATION CENTRE DEPARTMENT OF COMPUTER SCIENCE, UNIVERSITY COLLEGE CORK, IRELAND, 2007.

[45] 19A. H O L R O Y D, C. B., N. Y E U N G, M. G. H. C O L E S, J. D. C O H E N. A MECHANISM FOR ERROR DETECTION IN SPEEDED RESPONSE TIME TASKS. – JOURNAL OF EXPERIMENTAL PSYCHOLOGY: GENERAL, VOL. 134, 2005, NO 2, PP.163-191.

[46] W. TORRES-POMALES, "SOFTWARE FAULT TOLERANCE : A TUTORIAL," OCTOBER 2000. NATIONAL AERONAUTICS AND SPACE ADMINISTRATION (NASA) LANGLEY RESEARCH CENTER.

[47] P. RAMANATHAN AND K. G. SHIN, "DELIVERY OF TIME-CRITICAL MESSAGES USING A MULTIPLE COPY APPROACH," ACM TRANSACTIONS ON COMPUTER SYSTEMS (TOCS), VOL. 10, NO. 2, PP. 144–166, 1992.

[48] M. HILLER, "SOFTWARE FAULT-TOLERANCE TECHNIQUES FROM A REAL-TIME SYSTEMS POINT OF VIEW," TECH. REP., DEPARTMENT OF COMPUTER ENGINEERING, CHALMERS UNIVERSITY OF TECHNOLOGY, SE-412 96 GÖTEBORG SWEDEN, NOV. 1998.

[49] CEDRIC WILWERT, INFLUENCE DES FAUTES TRANSITOIRES ET DES PERFORMANCES TEMPS REEL SUR LA SURETE DES SYSTEMES X-BY-WIRE. THESE DE DOCTORAT, NETWORKS AND TELECOMMUNICATIONS [CS.NI]. NATIONAL POLYTECHNIC INSTITUTE OF LORRAINE - INPL, FRANCE 2005, 131 P

[50] ALAIN GIRAULT AND HAMOUDI KALLA: A NOVEL BICRITERIA SCHEDULING HEURISTICS PROVIDING A GUARANTEED GLOBAL SYSTEM FAILURE RATE. IEEE TRANS. DEPENDABLE SEC. COMPUT. VOL.6(4), 2009, PP. 241-254.

[51] HTTP://WWW.SYNDEX.ORG (ACCEDE : 3 FEVEIER 2017)

[52] J. C. LAPRIE, EDITOR. DEPENDABILITY: BASIC CONCEPTS AND TERMINOLOGY. IN: LAPRIE J.C. (EDS) DEPENDABILITY: BASIC CONCEPTS AND TERMINOLOGY. DEPENDABLE COMPUTING AND FAULT-TOLERANT SYSTEMS, SPRINGER, VIENNA, VOL 5, 1992, PP 3-245.

[53] M.R. LYU , SOFTWARE FAULT TOLERANCE : JOHN WILEY AND SONS, LTD., 1995.

[54] C.ARAR AND M. S. KHIREDDINE, « AN ALGORITHM BASED ON REPLICATION AND DEALLOCATION EFFICIENT FAULT-TOLERANT MULTI-BUS DATA SCHEDULING ALGORITHM BASED ON REPLICATION AND DEALLOCATION », CYBERNETICS AND INFORMATION TECHNOLOGIES, VOLUME 16, NO 2. 2016, PP. 69-84.

[55] A. GIRAULT, _ C. LAVARENNE, _ M. SIGHIREANU ET_ ET Y. SOREL, FAULT-TOLERANT STATIC SCHEDULING FOR REAL-TIME DISTRIBUTED EMBEDDED SYSTEMS. THE 21ST INTERNATIONAL CONFERENCE ON DISTRIBUTED COMPUTING SYSTEMS, PHOENIX, USA, APRIL 2001.

[56] N. ARYA, A.P. SINGH, FAULT TOLERANT SYSTEM FOR EMBEDDED SYSTEM ARCHITECTURE. INTERNATIONAL JOURNAL OF ENGINEERING AND TECHNOLOGY (IJET), VOL 9 NO 3S JULY 2017.

[57] NAGARAJAN K. JOHN P. HAYES AND BRIAN T. MURRAY «TASK SCHEDULING ALGORITHMS FOR FAULT TOLERANCE IN REAL-TIME EMBEDDED SYSTEMS» AVRESKY D.R. (EDS) DEPENDABLE NETWORK COMPUTING. THE SPRINGER

INTERNATIONAL SERIES IN ENGINEERING AND COMPUTER SCIENCE, VOL. 538. SPRINGER, BOSTON, MA, 2000, PP 395-412.

[58] JUN, Z., E. SHA ET AL. EFFICIENT FAULT-TOLERANT SCHEDULING ON MULTIPROCESSOR SYSTEMS VIA REPLICATION AND DEALLOCATION. – INTERNATIONAL JOURNAL OF EMBEDDED SYSTEMS, VOL. 6, 2014, NO 2-3, PP. 216-224.

[59] PRIYANKA M., ANISHA S. AND SAKTHI PRABHA R. « VLSI DESIGN FOR A PSO-OPTIMIZED REAL-TIME FAULT-TOLERANT TASK ALLOCATION ALGORITHM IN WIRELESS SENSOR NETWORK », ARPN JOURNAL OF ENGINEERING AND APPLIED SCIENCES, VOL. 11, NO. 13, JULY 2016, PP. 8226-8230.

[60] HADRIEN CAMBAZARD, PIERRE-EMMANUEL HLADIK, ANNE-MARIE, DEPLANCHE NARENDRA JUSSIEN. UNE CONTRAINTE GLOBALE POUR L'ORDONNANCABILITE DES TACHES TEMPS REEL DUR, CORK CONSTRAINT COMPUTATION CENTRE DEPARTMENT OF COMPUTER SCIENCE, UNIVERSITY COLLEGE CORK, IRELAND, 2007

[61] A.S. TANENBAUM, M.V. STEEN, DISTRIBUTED SYSTEMS PRINCIPLES AND PARADIGMS. SECOND EDITION, 2007 PEARSON EDUCATION

[62] PIERRE LOPEZ. APPROCHE PAR CONTRAINTES DES PROBLÈMES D'ORDONNANCEMENT ET D'AFFECTATION : STRUCTURES TEMPORELLES ET MÉCANISMES DE PROPAGATION, ELLIPSES MARKETING EDITION S.A.,PARIS, FRANCE, 2004

[63] L. LAMPORT, THE IMPLEMENTATION OF RELIABLE DISTRIBUTED MULTIPROCESS SYSTEMS. COMPUTER SCIENCE LABORATORY. SRI INTERNATIONAL, MENIO PARK, CALIFORNIA 94025, USA. NORTH-ILOHAND PUBLISHING COMPANY COMPUTER NETWORKS (1978)95-114

Résumé : dans ce travail, des algorithmes de distribution et d'ordonnancement temps réel et tolérants aux fautes sont proposés pour mapper les tâches d'un graphe d'algorithme sur les processeurs d'une architecture matérielle distribuée et hétérogène. Les fautes considérées sont des fautes matérielles, exactement des fautes permanentes d'un processeur unique avec le comportement arrêt sur défaillance. Les heuristiques proposées sont basées sur les mécanismes de la redondance logicielle. La première heuristique appelée AAA-FAULT^{DT}, sert à minimiser la longueur de distribution/ordonnancement et tolérer les fautes d'un processeur dans une architecture logicielle avec taches dépendantes en utilisant la redondance passive, dans le cas de la non défaillance, l'heuristique profite le temps dû de la non surcharge des processeurs. Dans le cas échoué, la méthodologie masque la défaillance et rendre le service attendu tout en respectant la contrainte de temps (deadline). La deuxième heuristique appelée FT-TDEP est une optimisation de la première, elle tolère les fautes en utilisant la redondance hybride dans une architecture à liaison point à point. Les résultats obtenus sont extrêmement meilleurs. La troisième appelée AAA-FAULT^{IDT} se base sur la redondance active pour tolérer les fautes d'un processeur dans une architecture algorithmique avec tâches indépendantes. Dans ce travail, nous avons essayé de réduire la difficulté de proposer des algorithmes d'ordonnancement tolérants aux fautes des processeurs ; offrant à la fois une réduction des longueurs de distribution/ordonnancement et une sûreté de fonctionnement optimale.
Mots-clés **:** Systèmes Temps Réel Embarqués, Algorithmes d'ordonnancement, Tolérance aux fautes, Réplication Active, Passive et Hybride.

Abstract : In this work, real-time and fault-tolerant distribution and scheduling algorithms are proposed to map the tasks of an algorithm graph to the processors of a distributed and heterogeneous hardware architecture. The faults considered are material faults, exactly permanent faults of a single processor with fail-stop behavior. The proposed heuristics are based on the mechanisms of software redundancy. The first heuristic called AAA-FAULT^{DT}, is used to minimize the length of distribution/scheduling and to tolerate processor errors in a software architecture with dependent tasks using the passive redundancy, in the case of the non-failure, the heuristic profits the time due to the non-overload of the processors. In the failed case, the methodology masks the failure and return the expected service while respecting the time constraint. The second heuristic called FT-TDEP is an optimization of the first, it tolerates faults using hybrid redundancy in a point-to-point architecture. The results obtained are extremely better. The third called AAA-FAULT^{IDT} is based on active redundancy to tolerate the errors of a processor in an algorithmic architecture with independent tasks. In this work, we have tried to reduce the difficulty of proposing fault tolerant algorithms for processors ; offering both a reduction in distribution/scheduling lengths and optimal dependability.
Keywords **:** Real-time Embedded Systems, Scheduling Algorithms, Fault Tolerance, Active, Passive and Hybride Replication.