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

THÈSE

pour obtenir le titre de

Docteur en Sciences

Spécialité : Informatique

présentée par

Chafik ARAR

Redondance logicielle pour la tolérance aux fautes des communications

Thèse dirigée par
Pr. Mohamed Salah KHIREDDINE

Soutenue le 08 Décembre 2016
devant le jury composé de :

Directeur :	Pr. Mohamed Salah KHIREDDINE	Prof	Université de Batna 2
Président :	Dr. Samir ZIDAT	MCA	Université de Batna 2
Examinateurs :	Pr. Abdelmadjid ZIDANI	Prof	Université de Batna 2
	Pr. Mohamed Ridda LAOUAR	Prof	Université de Tebessa
	Pr. Hamid SERIDI	Prof	Université de Guelma
	Dr. Salima OUADFEL	MCA	Université de Constantine 2

A ma mère, ma sœur et mes frères
A ma femme et mes deux filles, ranim et lina

Remerciement

En premier lieu, je tiens à remercier Monsieur Mohamed Salah KHIREDDINE, Mon Encadreur, d'avoir accepter de diriger mes travaux de recherche. Je le remercie pour ses conseils, son encadrement et pour la confiance qu'il m'a accordé durant ces trois années de thèse pour mener à bien ces travaux de recherche.

Je tiens également à remercier l'ensemble des membres de mon jury d'avoir accepter de juger et d'évaluer ces travaux de thèse. Je remercie sincèrement Monsieur Samir ZIDAT, Maître de conférences "A" à l'université de BATNA 2, de m'avoir fait l'honneur de présider mon jury de soutenance et l'attention qu'il à accordée à la lecture de ce manuscrit.

Je remercie également Monsieur Abdelmadjid ZIDANI, Professeur à l'université de Batna 2, Monsieur Mohamed Ridda LAOUAR, Professeur à l'université de Tebessa, Monsieur Hamid SERIDI, Professeur à l'université de Guelma ansi que Madame Salima OUADFEL, Maître de conférences "A" à l'université de Constantine 2, d'avoir pris de leurs temps et d'avoir participé au jury en tant qu'examinateurs.

Je tiens également à remercier sincèrement toute ma famille, sûrtout ma mère, ma sœur et mes frères pour leur soutient constant durant toutes ces années d'études. Pour finir, je ne pourrai clore ces remerciements sans remercier du fond du cœur ma moitié pour son soutien, sa patience et son écoute pendant ces longues années de thèse. Je ne la remercierai jamais assez pour m'avoir encouragé et soutenu en permanence ...

Table des matières

Remerciement iii

1 Introduction 1
- 1.1 Contexte et problématique de la thèse . . . 1
- 1.2 Objectifs et contributions . . . 2
- 1.3 Plan général de la thèse . . . 3

I Concepts de base et État de l'art 7

2 Ordonnancement des systèmes temps réel embarqués 9
- 2.1 Introduction . . . 9
- 2.2 Systèmes temps réel embarqués . . . 10
  - 2.2.1 Systèmes embarqués . . . 10
  - 2.2.2 Systèmes temps réel . . . 11
- 2.3 Caractéristiques des Systèmes temps réel embarqués . . . 12
- 2.4 Classement des Systèmes temps réel embarqués . . . 12
- 2.5 Modèles des Systèmes temps réel embarqués . . . 13
  - 2.5.1 Modèle de Tâches . . . 14
  - 2.5.2 Modèle d'architecture . . . 14
  - 2.5.3 Modèle de fautes de base . . . 16
- 2.6 Contraintes des Systèmes temps réel embarqués . . . 16
  - 2.6.1 Contraintes temporelles . . . 16
  - 2.6.2 Contraintes de placements . . . 17
  - 2.6.3 Contraintes matérielles . . . 18
- 2.7 Problème d'ordonnancement temps réel . . . 18
  - 2.7.1 Terminologies . . . 18
  - 2.7.2 Présentation du problème . . . 19
  - 2.7.3 Algorithmes d'ordonnancement temps réel . . . 20
- 2.8 Conclusion . . . 21

3 Sûreté de fonctionnement 23
- 3.1 Introduction . . . 23
- 3.2 Qu'est-ce que la sûreté de fonctionnement ? . . . 24
- 3.3 Concepts de base . . . 25
  - 3.3.1 Fonction et service d'un système . . . 25
  - 3.3.2 Attributs, Entraves et moyens . . . 25
- 3.4 Tolérance aux fautes . . . 33
  - 3.4.1 Techniques de la tolérance aux fautes . . . 33

3.4.2 Tolérance aux fautes logicielles et matérielles 36
3.4.3 Techniques logicielles de tolérance aux fautes matérielles . . . 37
3.4.4 Techniques matérielle de tolérance aux fautes matérielles . . . 41
3.5 Mise en œuvre de la tolérance aux fautes 42
3.6 Conclusion 43

4 Etat de l'art sur la tolérance des fautes des processeur et des communication temps réel 45
4.1 Introduction 45
4.2 Tolérance aux fautes 46
4.2.1 Les algorithmes d'ordonnancement basés sur la redondance active 46
4.2.2 Les algorithmes d'ordonnancement basés sur la redondance passive 50
4.2.3 Les algorithmes d'ordonnancement basés sur la redondance hybride 53
4.3 Fiabilité 54
4.3.1 Approche uni-objectif 54
4.3.2 Approche multi-objectifs 55
4.4 Conclusion 56

II Approches Proposées 57

5 Approche d'ordonnancement fiable de communication basée sur la fragmentation variable de données 59
5.1 Introduction 60
5.2 Présentation du problème 61
5.3 RBF-VDF : Un algorithme d'ordonnancement tolérant aux fautes des bus de communications basé sur la fragmentation variable des données. 63
5.3.1 Taux Global de défaillance du système (TGDS) 63
5.3.2 Tolérance aux fautes des processeurs par la redondance PP(Passive-Passive) : 64
5.3.3 Tolérance aux fautes des bus de communication par redondance Passive : 69
5.3.4 Fragmentation variable des données : 70
5.3.5 Ordonnacement tolérant aux fautes des processeurs et des bus de communication 77
5.4 Réduction de la Consommation d'énergie 80
5.5 Exemple et Simulations 81
5.5.1 Exemple 81
5.5.2 Simulations 83
5.6 Conclusion 84

6 Approche d'ordonnancement fiable de communication basée sur la désallocation des données 87
6.1 Introduction 87
6.2 Définition du problème 88
6.2.1 Les modèles du système 88
6.2.2 Les copies de sauvegarde 89
6.3 Exemple illustratif 90
6.4 Approche proposée. 94
6.5 Simulations, résultats et discussion 99
6.6 Conclusion 103

7 Approche d'ordonnancement fiable de communication temps réel basée sur les chemins de sauvegarde fragmentés dans les réseaux multi-sauts 105
7.1 Introduction 106
7.2 Modèle de fautes 108
7.3 Présentation du problème 108
7.4 RBF-FB : Un algorithme d'ordonnancement tolérant aux fautes des chemins de communications 109
7.4.1 La sauvegarde fragmentée 110
7.4.2 Fonctionnement de la sauvegarde fragmentée 112
7.4.3 Algorithme de la sauvegarde fragmentée 114
7.4.4 Fonction de sélection des segments de sauvegarde avec pondération de l'apport temps réel 117
7.4.5 Un exemple de la sauvegarde fragmentée 118
7.5 Algorithme d'ordonnancement 120
7.6 Recouvrement des fautes(Reprise sur faute) 121
7.7 Délai et Scalabilité 123
7.7.1 Délai 123
7.7.2 Scalabilité 124
7.8 RSVP 124
7.9 Conclusion 125

8 Conclusion Générale 127

Bibliographie 129

Liste des publications et communications 139

Table des figures

2.1 Systèmes temps réel embarqués 12
2.2 Exemple d'un modèle de tâches G_{Task}. 14
2.3 Exemple d'un modèle d'architecture G_{Arch} de type point à point. . . 15
2.4 Exemple d'un modèle d'architecture G_{Arch} de type bus. 15
2.5 Modèle de tâches : G_{Task} 17

3.1 Arbre de la sûreté de fonctionnement 26
3.2 Entraves de la sûreté de fonctionnement 28
3.3 Fautes solides, furtives et intermittentes 30
3.4 Groupements des moyens pour la sûreté de fonctionnement 33
3.5 Technique de tolérance aux fautes 34
3.6 Processus Communicant Reprise 38
3.7 Redondance Modulaire Triple 42

4.1 Modèles de tâches et d'architecture 47
4.2 Tolérance aux fautes des processeurs par redondance active 48
4.3 Tolérance aux fautes des médias de communication par redondance active 48
4.4 Tolérance aux fautes des processeurs par redondance passive 51
4.5 Tolérance aux fautes des média de communication par redondance passive 51
4.6 Tolérance aux fautes de processeurs et des médias de communication par redondance 53

5.1 Approche Proposée. 60
5.2 Transformation du graphe du modèle de tâches pour $Nf_{Proc} = 1$. . . . 65
5.3 Redondance passive des tâches t_1 et t_2 pour $Nf_{Proc} = 1$. 66
5.4 Partie de l'ordonnancement pour les tâches t_1 et t_2 (Nf_{Proc}=1). . . . 66
5.5 Décalage de la copie primaire $t_{1(p)}^{(s)1}$. 67
5.6 Principe de préemption avec l'approche PP(Passive-Passive). 68
5.7 Réplication et fragmentation des données de communication. 69
5.8 Tolérance au fautes du Bus B_2. 70
5.9 Fragmentation variable des données. 71
5.10 Décalages des dates d'arrivées des fragments de données. 71
5.11 Minimisation des écarts des dates d'arrivées des données. 72
5.12 Exemple de minimisation des dates d'arrivées des fragments. 72
5.13 Taille minimale de d'un fragment. 73
5.14 Exemple : retard de l'envoi des données sur le bus B_2. 74
5.15 Calcul des charges des bus. 74

5.16 Charge moyenne des bus et seuil de chargement des bus. 75
5.17 Adaptation des tailles des fragments aux charges des bus. 75
5.18 Changement de fréquence d'exécution de $t_j^{(s)1}$. 80
5.19 Modèle de tâches. 81
5.20 Modèle d'Architecture. 82
5.21 Ordonnancement non-fiable SynDEx. 82
5.22 Ordonnancement tolérant aux fautes pour Nf_{Bus}=1. 83
5.23 Ordonnancement tolérant aux fautes pour Nf_{Bus}=2. 83
5.24 Impact du CCR sur la longueur d'ordonnancement pour $Nf_{Bus} = 1$ et $Nf_{Bus} = 2$. 84
5.25 Impact du CCR sur TGDS pour $Nf_{Bus} = 1$ et $Nf_{Bus} = 2$. 85

6.1 Modèle d'architecture G_{Arch}. 90
6.2 Modèle de tâches G_{Task}. 91
6.3 Modèle de données G_{Data}. 91
6.4 ordonnancement de données non-tolérant aux fautes. 92
6.5 Ordonnancement de données tolérant aux fautes et optimal avec réplication et désallocation. 92
6.6 Ordonnancement de données tolérant aux fautes et optimal avec réplication. 93
6.7 Ordonnancement de données tolérant aux fautes et optimal avec désallocation. 93
6.8 Nœud fictif ajouter au DAG 95
6.9 Le nouveau DAG pour $m_i \rightarrow m_j$. 96
6.10 Le DAG complet pour le modèle de données de notre exemple. . . . 97
6.11 Le DAG complet après choix du type des copies de sauvegarde. . . . 97
6.12 Contraintes d'Exécution. 98
6.13 Modèle de tâches et affectation des bus. 101
6.14 Longueur d'ordonnancement de : *FTA-RD* vs *FTA-R*. 102
6.15 Longueur d'ordonnancement de : *FTA-RD* vs *FTA-D*. 102
6.16 Longueur d'ordonnancement de : *FTA-RD* vs La solution de la programmation linéaire 103

7.1 Le chemin de sauvegarde de bout-en-bout 107
7.2 Les chemins de sauvegarde fragmentés 107
7.3 Fragments primaires du chemin primaire 111
7.4 Fragments de la sauvegarde des fragments primaires 111
7.5 Faute d'un lien de communication sur le chemin primaire 112
7.6 Avantage de l'approche fragmentée 113
7.7 Graphe G_{net}. 114
7.8 Sélection d'un fragment de sauvegarde 118
7.9 La topologie du réseau de communications. 118
7.10 Graphe G_{net}. 119

7.11 Graphe G^*_{net}. 119
7.12 La Sauvegarde Segmentée 120
7.13 Recouvrement des fautes : cas de la sauvegarde fragmentée 123
7.14 Recouvrement des fautes : cas de la sauvegarde de bout en bout . . . 123

Liste des tableaux

2.1 $Ext(t_2)$ sur les différents processeurs. 17
2.2 $Extc$ pour la dépendance $A \to B$. 17

3.1 Classes de fautes 29
3.2 Modes de défaillance 31
3.3 Comparaison entre les trois approches de redondance. 41

6.1 Informations des benchmarks 101

Chapitre 1

Introduction

Sommaire

1.1 Contexte et problématique de la thèse 1
1.2 Objectifs et contributions 2
1.3 Plan général de la thèse 3

Les systèmes distribués temps-réel embarqués trouvent place de plus en plus dans des domaines applicatifs importants tels que l'automobile, l'avionique, les systèmes de contrôle de processus industriels, etc. Ces systèmes ont pour mission la réalisation de tâches complexes et surtout critiques, et sont soumis à des contraintes très strictes en terme de temps et de ressources. Etant ainsi, et vu les conséquences catastrophiques (perte d'argent, de temps, ou pire de pertes humaines) que pourrait entraîner une panne, ces systèmes doivent être extrêmement fiables. Cette robustesse est mise en œuvre dans la plupart des cas par des mécanismes de tolérance aux fautes. Ces mécanismes constituent l'objectif principal de notre travail.

La tolérance aux fautes matérielles constitue le domaine le plus étendu et développé de la tolérance aux fautes en général. Différentes techniques ont été élaborées et utilisées dans des domaines variés allant du la téléphonie aux missions spatiales. La tolérance des fautes peut être matérielle ou logicielle. La solution matérielle consiste à répliquer des composants matériels, et même si les performances des processeurs ne cessent de croitre et leurs coûts de décroitre, cette solution augmente la consommation d'énergie, c'est pourquoi on opte généralement pour les solutions basées sur la redondance logicielle.

1.1 Contexte et problématique de la thèse

Le travail présenté dans le cadre de cette thèse, se place dans le cadre général de la conception des systèmes sûrs de fonctionnement. Plus particulièrement, il s'intéresse aux systèmes réactifs, embarqués et temps réels. Il traite essentiellement l'aspect de la communication dont le rôle conditionne largement la capacité d'un système à respecter la ponctualité et les exigences de la sûreté de fonctionnement. La proposition, la définition et la caractérisation d'algorithmes d'ordonnancement temps réel tolérants aux fautes adaptés aux architectures matérielles distribuées basées sur les communications par bus ou par réseaux non fortement couplés, sont au

centre de notre travail. Dans ce dernier, on a essayer de contourner la difficulté de proposer des algorithmes d'ordonnancement tolérants aux fautes des processeurs et des communications temps réel offrant à la fois une optimalité théorique du point de vue de l'utilisation des ressources de calcul et de communications, mais également une efficacité pratique traduite par une sûreté de fonctionnement maximisée.

Nous étudions plus particulièrement, le problème de l'ordonnancement temps réel, à la fois tolérant aux fautes et aussi fiable. Ce problème est NP difficile ; et comme la solution optimale et exacte à ce problème 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. Étant donné que nous visons des systèmes embarqués, nos solutions sont uniquement des solutions logicielles qui essayent d'utiliser au mieux la redondance matérielle existante.

1.2 Objectifs et contributions

L'objectif principal de cette thèse, est de proposer des approches basées sur les techniques logicielles. Ce choix est motivé par les contraintes d'embarquabilité et de consommation d'énergie imposées par la nature non encombrante des systèmes embarqués temps réels.

Les solutions que nous proposons permettant de générer des ordonnancement des composants logiciels d'un algorithme sur les composants matériels d'une architecture d'un système donné. L'ordonnancement en question doit :

1. être tolérant aux fautes, (le nombre maximal de fautes des processeurs et de fautes des médias de communication que le système doit tolérer, est défini par les hypothèses.)
2. respecter les contraintes de distribution (ces contraintes définissent une relation d'exclusion entre certains composants logiciels et certains composants matériels),
3. respecter les contraintes temps réel (ces contraintes définissent un seuil maximal pour l'exécution de l'algorithme sur l'architecture).

Les trois approches de conception proposées sont :

1. La première approche consiste à générer des ordonnancements tolérantes aux fautes, adaptées aux architectures matérielles distribuées et hétérogènes munies d'un réseau de communication multi-bus. Elle permet de tolérer les fautes des processeurs en utilisant une nouvelle version de la redondance passive dite passive-passive reposant sur un principe de préemption qui permet le reclassement des tâches à ordonnancer. Notre approche permet aussi la tolérance des fautes des bus de communication en utilisant la fragmentation variable

des données, ce qui permet de réduire considérablement la retransmission des données après une faute. Notre approche se base sur une heuristique d'ordonnancement statique qui permet d'affecter les composants logiciels, redondants (tâches) et fragmentés (données), de l'algorithme sur les composants de l'architecture, tout en respectant les contraintes de distribution et de temps réel.

2. La deuxième approche proposée dans cette thèse, considère seulement une seule faute de bus de communication dans une architecture multi-bus et hétérogène. Cette faute est causée par des défauts de matériel et compensée par des solutions de redondance logicielle. L'approche proposée est une approche hybride qui se base sur deux types de copies de sauvegarde, répliquées et désallouées. L'objectif principal est de minimiser la longueur des ordonnancements des données sur les bus de communications, par un mécanisme de chevauchement des copies de sauvegarde désallouées, ce qui permet une utilisation plus optimale des bus de communications.

3. La troisième approche proposée dans cette thèse est adaptée aux architectures qui utilisent des réseaux de communication non fortement couplés et multi-sauts. Elle consiste à générer des ordonnancements fiables en se basent sur une heuristique qui permet la création pré-définie de plusieurs chemins de sauvegarde fragmentés, chacun couvrant une partie contigüe du chemin primaire de communication, ce qui permet de traiter localement et de tolérer une ou plusieurs fautes temporelles de communication.

   Par l'utilisation de cette nouvelle approche nous visons une fiabilité de communication supérieure : cela est dû à des chemins primaires qui ont une sauvegarde fragmentée et non une sauvegarde de bout en bout. Nous visons aussi l'amélioration de l'utilisation des ressources du réseau et ça parceque les sauvegardes fragmentées sont généralement plus courtes et ont besoin de mois de ressources. Notre dernier objectif est d'avoir une meilleure et une paramétrable QoS : Une sauvegarde fragmentée peut comprendre plusieurs sauvegardes, chacune dédiée à un objectif de Qos bien défini.

1.3 Plan général de la thèse

La thèse est composée de deux parties :

- La première partie est constituée de trois chapitres :
  - Le premier chapitre donne une présentation générale des concepts de bases et des terminologies liés aux systèmes temps réel, embarqués et temps réel. Il définit les modèles de tâches et d'architectures, ainsi que les contraintes temporelles et matérielles, il aborde le problème de la conception de ces systèmes, plus spécialement les techniques basées sur la théorie de l'ordonnancement.

  - Dans le deuxième chapitre, la sûreté de fonctionnement, est définie comme étant la science qui propose des moyens et des techniques qui permettent la réalisation de systèmes sûrs. Il présente les moyens de la sûreté de fonctionnement mis à la disposition de ces systèmes, plus particulièrement la tolérance aux fautes.
  - Le troisième chapitre comprend un état de l'art très détaillé, il recense tous les moyens et les techniques de la tolérance aux fautes utilisées jusqu'à aujourd'hui. Il présente aussi une mesure de probabilité, appelée fiabilité, qui est employée par plusieurs méthodes pour concevoir des systèmes fiables, c'est-à-dire des systèmes garantissant un niveau maximum de probabilité de bon fonctionnement.

- La deuxième partie constituée de trois chapitres, présente les trois approches proposées dans ce travail, à savoir la première technique basée sur la fragmentation variable des données, la deuxième basée sur les copies de sauvegarde désallouées et la troisième basée sur les sauvegardes fragmentées des chemins des communications dans les réseaux multi-sauts.

  - Le quatrième chapitre présente la première approche que nous proposons pour résoudre le problème de la génération d'ordonnancements fiables et tolérants aux fautes, adaptée aux architectures matérielles munies d'un réseau de communication multi-bus. Nous commençons par la présentation du modèle de fautes adopté par notre approche, ensuite nous présentons les trois techniques de bases sur lesquels repose notre approche, à savoir, la redondance passive, le TGDS et la fragmentation variable des données.

  - Le cinquième chapitre présente notre deuxième approche proposée dans le cadre de ce travail. Elle assure un recouvrement local rapide et permet la tolérance d'une faute de bus de communication. L'approche proposée est une approche hybride, qui se base sur deux types de copies de sauvegarde : des copies de sauvegarde répliquées activement et copies de sauvegardes désallouées puis réallouées passivement, deux des copies de sauvegarde désallouées peuvent se chevauchées sur le même bus en même temps, ce qui permet une utilisation plus optimale des bus de communications. Pour évaluer les résultats obtenus par cette technique, nous définissons le problème d'ordonnancement comme un problème d'optimisation et nous utilisons la programmation linéaire pour trouver l'ordonnancement optimal et de longueur minimale, par rapport auquel, les résultats de la nouvelle approche seront évalués.

  - Le sixième chapitre présente notre troisième approche proposée dans le cadre de ce travail, elle est adaptée aux systèmes à base de réseaux de communications multi-sauts non fortement couplés, où la transmission de

données entre deux processeurs n'est pas directe et peut faire le passage par un ou plusieurs autres processeurs avant d'arriver à destination. Cette approche assure un recouvrement local rapide et garanti une ou de plusieurs fautes temporelles de communication. Elle se base sur la création d'un chemin de sauvegarde qui est routé sur un chemin disjoint du chemin principal. Les ressources de rechange réservées sur ce chemin ne sont activées que lorsque le chemin principal est cible de faute, pour assurer un recouvrement local rapide et garanti d'une ou de plusieurs fautes temporelles de communication.

Enfin, une conclusion décrit nos contributions, et les perspectives de recherche associées.

Première partie

Concepts de base et État de l'art

Chapitre 2

Ordonnancement des systèmes temps réel embarqués

Sommaire

2.1 Introduction **9**
2.2 Systèmes temps réel embarqués **10**
2.2.1 Systèmes embarqués 10
2.2.2 Systèmes temps réel 11
2.3 Caractéristiques des Systèmes temps réel embarqués **12**
2.4 Classement des Systèmes temps réel embarqués **12**
2.5 Modèles des Systèmes temps réel embarqués **13**
2.5.1 Modèle de Tâches 14
2.5.2 Modèle d'architecture 14
2.5.3 Modèle de fautes de base 16
2.6 Contraintes des Systèmes temps réel embarqués **16**
2.6.1 Contraintes temporelles 16
2.6.2 Contraintes de placements 17
2.6.3 Contraintes matérielles 18
2.7 Problème d'ordonnancement temps réel **18**
2.7.1 Terminologies 18
2.7.2 Présentation du problème 19
2.7.3 Algorithmes d'ordonnancement temps réel 20
2.8 Conclusion **21**

2.1 Introduction

Les systèmes temps réel embarqués prennent de plus en plus de place dans notre vie, ce type de systèmes permet de réaliser des tâches complexes et dans la plus part du temps critiques. La conception de tels systèmes passe d'abord par la spécification des modèles de tâches, d'architecture et de l'ensemble des contraintes temps réel, matérielles et de sûreté de fonctionnement. Dans un deuxième temps, il faut trouver une affectation optimale et valide des composants du modèle de

tâches sur les composants de l'architecture tout en respectant les contraintes déjà définies. Les algorithmes d'ordonnancement sont la meilleure façon de réaliser cette affectation.

2.2 Systèmes temps réel embarqués

Un système embarqué ("embedded system") se définit comme étant un système autonome, intégré dans un système plus large avec lequel il interagit, et pour lequel il réalise des fonctions bien déterminées. Les systèmes embarqués sont souvent temps réel car d'un part ils sont en constante interaction avec leurs environnement, et d'autre part ils doivent réaliser des calculs et prendre des décisions en respectant des contraintes temporelles.

2.2.1 Systèmes embarqués

Un système embarqué est un système dédié qui ne réalise q'une tâche bien précise. Ses ressources sont généralement limitées (poids et consommation d'energie limités). Le premier système embarqué reconnu comme tel est le système de guidage des missions lunaires Apollo (Apollo Guidance Computer ou AGC) dès 1967 (première mission Apollo), développé par Charles Stark Draper du MIT, chargé du système de guidage inertiel du module. L'AGC était un ordinateur multitâches réalisant des traitements en temps réel. L'unité comportait 64 ko de mémoire morte, contenant l'ensemble des programmes, et 4 ko de mémoire vive. Le processeur était constitué de plus de 5000 portes logiques réalisées à l'aide de circuits intégrés. L'ensemble pesait environ 35 kg. Actuellement, les systèmes embarqués se retrouvent dans de nombreux domaines.

Les systèmes embarqués sont soumis à des contraintes :

— de coût, le plus faible possible surtout si le système est produit en grande série ;

— de taille, généralement elle doit être réduite ;

— d'empreinte mémoire, ou l'espace mémoire est généralement limité ;

— de consommation énergétique la plus faible possible, due à l'utilisation de batteries comme unique source d'alimentation ;

— temporelles, les temps d'exécution et l'échéance temporelle d'un service sont connus à priori. Ce qui fait que de tels systèmes ont des propriétés temps réel ;

— de sûreté de fonctionnement, un système critique ne doit jamais faillir (même si le zéro défaillance est impossible) ;

— de sécurité, puisque certaines informations peuvent êtres confidentielles.

2.2.2 Systèmes temps réel

A l'égard de certains systèmes, la validité d'une action dépend du temps qui s'écoule entre la fin de la réalisation de cette action et l'évènement qui l'a déclenchée. Autrement dit, dans certains systèmes, on qualifie une action de valable si le temps qui s'écoule entre la fin de la réalisation de cette action et l'évènement qui l'a déclenchée est inférieur à une borne maximale. Ces systèmes sont dites de temps réel et les bornes maximales sont appelées contraintes temps réel.

Définition : Un système temps réel est un système dont l'exactitude des résultats ne dépend pas seulement de l'exactitude logique des calculs mais aussi de la date à laquelle les résultats sont produits. Si les contraintes temporelles ne sont pas satisfaites, on dit qu'une défaillance système s'est produite.

Quand les interactions d'un système temps réel avec son environnement ont lieu à des moments déterminés par l'horloge du système, on parle alors de système temps réel basé sur l'horloge, à la différence d'un système basé sur les évènements où ces moments ne sont pas déterminés par le système mais par son environnement.

On distingue deux types de systèmes temps réel, le temps réel strict ou dur (hard real-time) et le temps réel souple ou mou (soft real-time), cette distinction est faite par rapport à l'importance accordée aux contraintes temporelles [1], [2], [3].

Ainsi, on peut distinguer trois grandes catégories des systèmes temps réel :

- **Systèmes temps réel à contraintes strictes/dures :** le temps réel strict ne tolère aucun dépassement de ses contraintes, ce qui est souvent le cas lorsque de tels dépassements peuvent conduire à des situations critiques, voir catastrophiques. Les systèmes de contrôle de vol, les systèmes de contrôle de station nucléaire et les systèmes de contrôle de voies ferrées en sont des exemples.
- **Systèmes temps réel à contraintes relatives/souples :** le temps réel souple autorise certains dépassements limités des contraintes temporelles au delà desquelles le système devient inutilisable. Les applications multimédias sont un bon exemple de ces systèmes.
- **Systèmes temps réel à contraintes mixtes :** sont composés des tâches à contraintes strictes et à contraintes souples.

Une bonne façon de comprendre la relation entre les systèmes temps réel et les systèmes embarqués est de les considérer comme deux cercles entrecroisés, comme le montre la Figure 2.1. On peut voir que tous les systèmes embarqués présentant des comportements temps réel, ne sont pas tous des systèmes temps réel embarqués.

Cependant, les deux systèmes ne sont pas mutuellement exclusives, et la zone dans laquelle ils se chevauchent crée la combinaison des systèmes connus comme les systèmes embarqués temps réel.

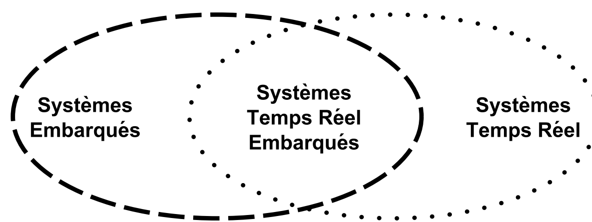


Figure 2.1 – Systèmes temps réel embarqués

2.3 Caractéristiques des Systèmes temps réel embarqués

Le développement d'un système temps réel et embarqué doit respecter un certain nombre de caractéristiques :

1. La production en masse : la plus part des systèmes temps réel embarqués sont conçus pour un marché de masse. Cela signifie que le coût d'une seule unité de production doit être aussi faible que possible.
2. Structure statique : il n'est pas nécessaire de disposer de structures et de mécanismes dynamiques (le dynamisme augmente les besoins en ressources et conduit à une complexité inutile de la mise en œuvre).
3. Interface : elle est spécialement conçu pour répondre à l'objectif déclaré et elle est facile à utiliser.
4. La minimisation du sous-système mécanique : pour une fiabilité maximale et un coût minimal.
5. Fonctionnalité : le logiciel du système implanté dans la ROM doit déterminer les fonctionnalités en détails. Les normes de qualité pour ce logiciel sont élevés, du moment où on ne peut pas le modifier.
6. Entretien : du moment où le partitionnement du système en unités remplaçables est trop cher, ces systèmes sont conçus pour être non maintenable, Une interface de diagnostic et une excellente stratégie d'entretien.
7. Communication : les connexions avec des systèmes plus grands sont prises en charge, et le thème de la sécurité est de la plus haute préoccupation.
8. Consommation d'énergie limitée : de nombreux dispositifs embarqués et mobiles sont alimentés par des batteries dont la duré de vie est limité.

2.4 Classement des Systèmes temps réel embarqués

Un système embarqué et temps réel gère deux parties, une partie matérielle composée d'un ensemble d'éléments physiques : processeur(s), mémoire(s) et média

de communication, une partie logicielle qui consiste en des programmes (tâches ou processus), et une source d'énergie.

Un premier classement des architectures pour système embarqué dépend du nombre de processeurs :

— Architecture mono-processeur.

— Architecture multi-processeurs.

Dans le cas des architectures multi-processeurs, Il existe différents classements :

— homogène/hétérogène selon la nature des processeurs de l'architecture :
  - Identiques : les processeurs ont la même puissance de calcul, et sont interchangeables,
  - Uniformes : à chaque processeur sa puissance de calcul qui se calcul par nombre d'opérations par unité de temps
  - Indépendants : à la différence des processeurs uniformes, un taux d'exécution est associé à chaque couple (Tâche, Processeur) ; Donc la vitesse de progression sur un même processeur varie d'une tâche à une autre.

— Homogène ou 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 ou 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, ou via un réseau de communication.

2.5 Modèles des Systèmes temps réel embarqués

Les systèmes distribués embarqués temps réel peuvent être décrits par deux Modèles :

— Un Modèle de tâches : un graphe représentant les fonctionnalités à implanter ainsi que leurs dépendances sous la forme d'un graphe orienté.

— Un Modèle d'architecture matérielle : un graphe représentant les processeurs et les connexions.

2.5.1 Modèle de Tâches

Le modèle de tâches est composé d'un ensemble d'entités appelées composants logiciels ; chacun d'eux assure une objectif fonctionnel bien déterminé du système. Dans la mesure où nous visons des systèmes distribués, le modèle de tâches G_{Task} d'un système temps-réel embarqué est modélisé par un graphe orienté acyclique (DAG). Les sommets de ce graphe sont les opérations de calcul (tâches) de l'algorithme et les arrêtes sont les dépendances de données entre ces opérations [4] [5] [6].

Définition 1 : Une opération (tâche) est une séquence finie de code. elle peut être de calcul ou de communication (entrée/sortie).

Définition 2 : Une dépendance de données ($t_i \to t_j$) définie à la fois une relation de précédence et un échange de données entre l'opération prédécesseur t_i et son opération successeur t_j. Dans le cas où une opération possède plusieurs opérations consommatrices, on parle alors d'une dépendance de donnée de diffusion.

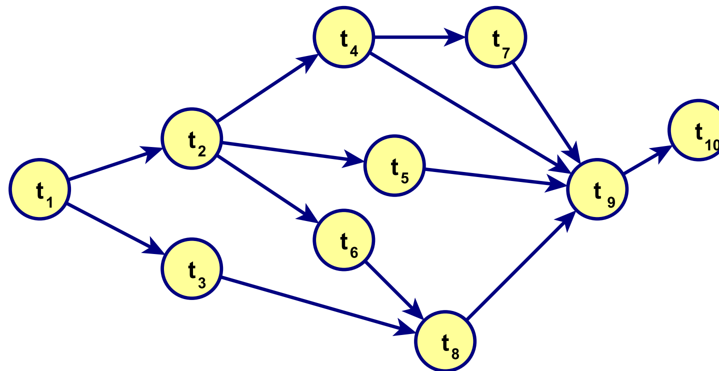


FIGURE 2.2 – Exemple d'un modèle de tâches G_{Task}.

La Figure 2.2 est un exemple d'un modèle de tâches G_{Task} avec dix tâches $t_1, t_2, t_3, t_4, t_5, t_6, t_7, t_8, t_9$ et t_{10}.

2.5.2 Modèle d'architecture

Le modèle d'architecture G_{Arch} d'un système distribué temps-réel est modélisé par un graphe non orienté, où les sommets désignent les processeurs et les médias (lien ou bus) de communication, et les arêtes désignent les liaisons physiques entre processeurs et média de communication. Dans le cas de communication par passage de messages on distingue deux modèles de communication : les communications point-a-point et les communications multi-point dites par bus [7].

La Figure 2.3 est un autre exemple d'un modèle d'architecture, composé de quatre processeurs P_1, P_2, P_3 et P_4 et de cinq liens de communication de type point à point $L_{P_1,P_2}, L_{P_1,P_3}, L_{P_2,P_4}, L_{P_3,P_4}$ et L_{P_2,P_3}.

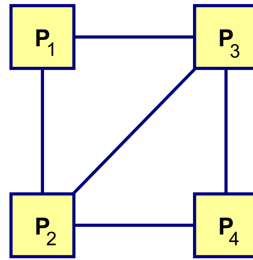


Figure 2.3 – Exemple d'un modèle d'architecture G_{Arch} de type point à point.

La Figure 2.4 est un exemple d'un modèle d'architecture, composé de quatre processeurs P_1, P_2, P_3 et P_4 chacun de ces processeurs est connecté aux quatre bus de communication B_1, B_2, B_3 et B_4.

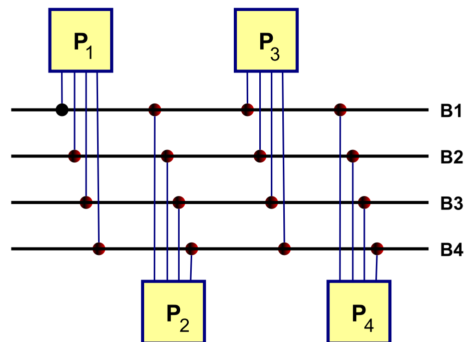


Figure 2.4 – Exemple d'un modèle d'architecture G_{Arch} de type bus.

Les tâches et les dépendances du modèle de tâches devront être ordonnancées sur les processeurs et liens de communication de l'architecture. Pour faire cela les sommets du graphe du modèle de tâches doivent être étiquetés par des caractéristiques temporelles indépendantes de l'architecture comme la période, la période minimale, l'échéance, et des caractéristiques temporelles dépendantes de l'architecture comme le $WCET$ [1], la quantité de mémoire, etc. De la même manière les dépendances de données doivent être étiquetées par des caractéristiques temporelles dépendantes de l'architecture $WCCT$ [2], quantité de mémoire, etc. Les périodes des communications sont déduites de celles des opérations dépendantes. Quand les sommets et les dépendances sont étiquetés le modèle de tâches est un graphe de tâches temps réel [8].

1. $WCET$ = Worst-Case Execution Time.
2. $WCCT$ = Worst-Case Execution Time.

2.5.3 Modèle de fautes de base

Dans le cadre de cette thèse, nous ne considérons dans notre modèle de fautes que les fautes des processeurs et des médias de communications. Les fautes à prendre en considération sont les fautes permanentes ou intermittentes (ces fautes se produisent sporadiquement et chaque composant en faute peut revenir à son état actif, ce qui n'est pas le cas pour les fautes permanentes. Notre modèle de fautes de base suppose aussi que chaque composant est de type silence sur défaillance (le composent en faute ne produit aucun résultat en sortie).

2.6 Contraintes des Systèmes temps réel embarqués

Nous distinguons deux types de contraintes : les contraintes temporelles et les contraintes matérielles.

2.6.1 Contraintes temporelles

Les systèmes temps réel peuvent avoir deux types de contraintes temporelles : contraintes strictes 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 dit à contraintes souples quand le non respect des contraintes temporelles est acceptable. Il est alors acceptable de ne pas respecter certain pourcentage d'échéances ; On parle alors de qualité de service (QoS). Dans les applications de contrôle et de 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 ou plus qu'une faute de calcul [8].

Les systèmes temps-réel sont soumis à des contraintes strictes en termes de temps de réponse. Les contraintes temps-réel C_{Rt} peuvent être de différents types. Dans notre modèle, la seule contrainte est de minimiser le temps global d'exécution d'une itération de G_{Task}. Suivant les caractéristiques temporelles d'un système temps-réel strict, certains paramètres temporels sont attribués à chaque tâche du système.

A chaque tâche t_i de G_{Task} on associer un temps d'exécution $Ext(t_i, P_j)$ sur chaque processeur P_k de G_{Arch}. Un seuil supérieur du temps d'exécution d'une opération t_i sur chaque processeur P_k est noté $WCET$ (Worst-Case Execution Time), il est déterminé par le concepteur du système en utilisant des méthodes statiques ou dynamiques [9]. L'exemple du tableu 2.1 défini les temps d'exécutions Ext d'une

opération A du graphe G_{Task} (Figure 2.5) sur les processeurs P_1, P_2, P_3 et P_4 de G_{Arch}.

Processeur	P_1	P_2	P_3	P_4
Temps d'exécution	1.2	2.3	1.6	∞

TABLE 2.1 – $Ext(t_2)$ sur les différents processeurs.

A chaque dépendance de données $t_i \rightarrow t_j$ on associe un temps de communication $Extc(t_i \rightarrow t_j)$ à travers un média de communication (lien, bus ou un chemin de réseau) de l'architecture. Par exemple, le Tableau 2.2 donne les temps des communications $Extc$de la dépendance $A \rightarrow B$ du graphe G_{Task} sur les média de communication L_k) de G_{Arch} de la figure 2.5.

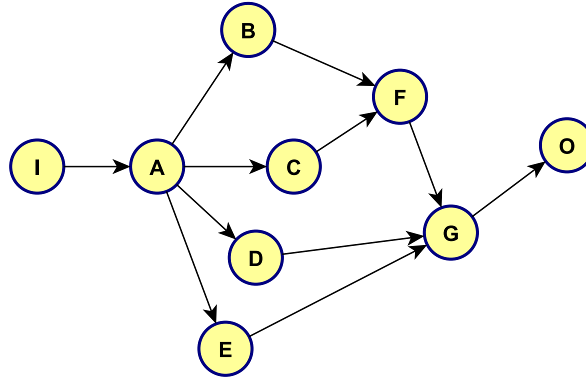


FIGURE 2.5 – Modèle de tâches : G_{Task}

Media de communication	L_{P_1,P_2}	L_{P_1,P_3}	L_{P_2,P_4}	L_{P_3,P_4}	L_{P_2,P_3}
Temps de communication	1.2	1.5	1.6	2.0	2.1

TABLE 2.2 – $Extc$ pour la dépendance $A \rightarrow B$.

2.6.2 Contraintes de placements

Les contraintes de placements (ou de distribution) C_{Sch} définissent la possibilité d'affecter ou de placer une opération (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.

2.6.3 Contraintes matérielles

L'origine de ces contraintes est la restriction des ressources dans les systèmes embarqués. Comme résultat aux limitations de poids, de volume, de consommation d'énergie, ou de prix, on se retrouve avec des puissances de calcul et de stockage limités. Parmi ces différentes contraintes, la gestion de la mémoire et la consommation sont les principales contraintes étudiées dans la littérature.

Par rapport au vitesse des processeur 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 [10].

Beaucoup de techniques ont pour objectif la minimisation de la consommation d'énergie, ces techniques sont basées essentiellement sur la minimisation du "makespan" qui correspond au temps total d'exécution des tâches. L'idée est d'utiliser des processeurs moins puissants pour avoir les mêmes résultats obtenus avec des processeurs puissants et qui consomment plus d'énergie [11] [12] [13] [14] [10].

2.7 Problème d'ordonnancement temps réel

Après avoir spécifié les modèles de tâhes et d'architecture, et déterminer les contraintes temporelles et matérielles. Le problème d'ordonnancement peut être formulé précisément comme étant la recherche d'une implantation des tâches de G_{Task} sur le modèle d'architecture G_{Arch} qui satisfasse les contraintes temporelles et matérielles.

2.7.1 Terminologies

Nous définissons dans cette section quelques concepts qui seront utilisés tout au long de cette thèse.

Ordonnancement temps réel : L'ordonnancement est le mécanisme qui permet d'affecter un composant choisi de l'algorithme pour être exécuter sur un composant de l'architecture de manière à respecter les contraintes de temps réel et à optimiser un ou plusieurs critères. Le processus qui réalise une telle sélection, et donc qui définit un ordre d'exécution entre les composants de G_{Task} est appelé algorithme d'ordonnancement.

Ordonnancement statique et dynamique : Un ordonnancement statique est un ordonnancement ou l'ordre d'exécution des tâches d'un modèle de tâches ne change pas pendant l'exploitation du système. Si, cet ordre change on parle alors d'un ordonnancement dynamique.

Fonction de coût : Le rôle d'une fonction de coût est classer les différents composants de l'algorithme pour l'objectif d'ordonner leurs exécution, ce classement est réalisé par l'affectation d'un poids à chaque composant logiciel.

Algorithme d'ordonnancement temps réel Ce type d'algorithme est nécessaire surtout pour les architecture distribuées. il permet en plus de la sélection du composant de l'algorithme, la sélection du composant de l'architecture qui peut l'exécuter.

2.7.2 Présentation du problème

Le problème d'ordonnancement temps réel se définit comme le problème qui cherche à trouver une affectation des composants logiciels du modèle de tâches sur les composants matériels de l'architecture distribuée, en respectent les contraintes temps réel [15]. La solution à ce problème consiste à chercher l'existence d'une allocation possible (qui satisfasse les contraintes temporelles et matérielles).

Ce problème peut devenir un problème d'optimisation lorsqu'il s'agit de rechercher une affectation possible et optimale qui doit en plus optimiser un certain critère. Un tel problème d'optimisation et soit polynômial où NP-difficile [4].

Enfin, le problème d'ordonnancement des composants logiciels d'un modèle de tâches sur les composants matériels d'une architecture est formalisé comme suit :

Les Données du problème :

- Une architecture matérielle hétérogène G_{Arch} composée d'un ensemble E_{Proc} d'opérateurs de calcul (processeurs) et d'un ensemble E_{Bus} de bus de communication :
$$E_{Proc} = \{\ldots, P_i, \ldots\}, E_{Bus} = \{\ldots, B_j, \ldots\}$$
- Un modèle de tâches G_{Task} composé d'un ensemble E_{Task} d'opérations (tâches) et d'un ensemble E_{Dd} de dépendances de données :
$$E_{Task} = \{\ldots, t_i, \ldots, t_j, \ldots\}, E_{Dd} = \{\ldots, (t_i \triangleright t_j), \ldots\}$$
- Des caractéristiques d'exécution C_{EExe} et C_{CExe} des composants de G_{Task} sur les composants de G_{Arch},
- Un ensemble de contraintes matérielles C_{Had} et temps réel C_{Rt},
- Un ensemble de critères à optimiser,

Il faut trouver une application SH_{Ft} qui place chaque tâche (resp. dépendance de données) de G_{Task} sur un processeur (resp. un lien ou un bus) de G_{Arch} et qui lui assigne un ordre d'exécution $Ordre_k$ sur son processeur (resp. un bus) :

$$\begin{array}{rccl} SH_{Ft}: & G_{Task} & \longrightarrow & G_{Arch} \\ & G_{Task\,i} & \longmapsto & SH_{Ft}(G_{Task\,i}) = (G_{Arch\,j}, Ordre_k) \end{array}$$

qui respecte C_{Had} et optimise les critères spécifiés.

2.7.3 Algorithmes d'ordonnancement temps réel

Les algorithmes d'ordonnancement ont comme mission, de trouver à tout instant donné de l'exécution du système, un composant matériel pour le composant logiciel le plus prioritaire. La façon de réaliser cette affectation permet de classer ces algorithmes en :

2.7.3.1 Algorithmes hors-ligne et en-ligne

Un algorithme d'ordonnancement est dit hors ligne s'il construit la séquence d'ordonnancement complète à partir des paramètres temporels de l'ensemble des tâches avant l'exécution de l'application. Cela convient bien aux systèmes de tâches périodiques. Ces algorithmes permettent de concevoir des systèmes prédictifs, du moment où les contraintes temporelles peuvent être vérifiées et validées même avant la mise en exploitation du système.

Un algorithme d'ordonnancement est dit en ligne s'il construit la séquence d'ordonnancement à partir des paramètres temporels de l'ensemble des tâches pendant l'exécution de l'application. Les algorithmes en ligne sont plus robustes vis-à-vis des dépassements des WCETs. Cela convient bien aux systèmes de tâches sporadiques et apériodiques citemarouf2012ordonnancement.

2.7.3.2 Algorithmes exacts et approchés

Les algorithmes hors-ligne et en-ligne qui trouvent toujours une solution optimale pour le problème d'ordonnancement temps réel, bien sûr si cette solution existe, font partie de la classe des algorithmes exacts. Cependant, dans le cas général ce problème est NP-difficile et de complexité exponentielle, et pour le résoudre dans un temps polynômial, on adopte des algorithmes heuristiques qui cherchent des solutions approchées le plus possible de la solution optimale.

2.8 Conclusion

Dans ce chapitre nous avons présenté le problème d'ordonnancement temps réel pour des systèmes distribués réactifs embarqués. Nous avons commencé par donner des définitions concernant les architectures des systèmes embarqués ainsi que les contraintes temporelles et matérielles, puis nous avons présenté les modèles qui définissent de tels systèmes. à savoir les modèles de tâches et d'architecture. A la fin nous avons vu que ce problème peut être résolu par des algorithmes d'ordonnancement temps réel, et qu'il en existe deux classifications : d'une part les algorithmes hors-ligne ou en-ligne, et d'autre part les algorithmes exacts ou approchés.

Chapitre 3

Sûreté de fonctionnement

Sommaire

3.1 Introduction **23**
3.2 Qu'est-ce que la sûreté de fonctionnement ? **24**
3.3 Concepts de base **25**
3.3.1 Fonction et service d'un système 25
3.3.2 Attributs, Entraves et moyens 25
3.4 Tolérance aux fautes **33**
3.4.1 Techniques de la tolérance aux fautes 33
3.4.2 Tolérance aux fautes logicielles et matérielles 36
3.4.3 Techniques logicielles de tolérance aux fautes matérielles . . . 37
3.4.4 Techniques matérielle de tolérance aux fautes matérielles . . . 41
3.5 Mise en œuvre de la tolérance aux fautes **42**
3.6 Conclusion **43**

3.1 Introduction

La nécessité de la sûreté de fonctionnement n'est apparue que au cours du XXème siècle, suite à la révolution industrielle. Le terme "dependability" est apparu dans une publicité sur des moteurs Dodge Brothers dans les années 30. La sûreté de fonctionnement peut se définir par son objectif principale qui vise d'atteindre l'idéal de la conception des systèmes : 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 et impossible dans le contexte industriel.

La sûreté de fonctionnement, c'est la science qui propose des moyens et des techniques qui permettent la réalisation des système sûrs et aussi fiable que possible dans des délais et avec des coûts raisonnables. L'importance et la vitalité de la sûreté de fonctionnement vient du fait que les conséquences que pourrait entrainer une faute dans un système réactif critique sont catastrophiques (perte d'argent, de temps, ou pire de vies humaines). Les moyens qui permettent la conception de systèmes sûrs de fonctionnement sont les techniques de la sûreté de fonctionnement [16] [17] [18] [19] [20] [21].

Dans la littérature, on trouve que plusieurs méthodes utilisent la tolérance aux fautes comme le moyen le plus important parmi plusieurs pour la conception des systèmes sûrs de fonctionnement, que nous appelons dans la suite "systèmes tolérants aux fautes". La tolérance aux fautes permet à un système de continuer à fonctionner et à délivrer un service conforme à sa spécification en présence de fautes. Ils existe d'autres méthodes utilisant une mesure de probabilité, appelée fiabilité, pour concevoir aussi des systèmes sûrs de fonctionnement, que nous appelons "systèmes fiables". A la différence de la la tolérance aux fautes, la fiabilité permet d'évaluer aléatoirement le bon fonctionnement d'un système.

Ce chapitre reprend et actualise les points principaux liés à la sûreté de fonctionnement. Nous commençons tout d'abord par la définition de la sûreté de fonctionnement, ensuite on met l'accent sur les concepts de base de cette dernière. Les spécificités relatives à la tolérance aux fautes et ces techniques, plus spécifiquement la redondance logicielle sont aussi exposés et enfin nous identifions quelques défis pour la mise en œuvre de la tolérance aux fautes.

3.2 Qu'est-ce que la sûreté de fonctionnement ?

La sûreté de fonctionnement ou science des défaillances comme elle est souvent appelée, inclut la connaissance des défaillances, leur évaluation, leur prévision, leurs mesures et leur maîtrise. Dans la plupart du temps l'étude de la sûreté de fonctionnement, relève d'un domaine transverse et souvent multidisciplinaire qui nécessite une connaissance globale du système comme les architectures fonctionnelles et matérielles, les conditions d'utilisation, les risques extérieurs. Beaucoup d'avancées dans ce domaine sont le résultat d'un feed-back d'expérience et des rapports d'analyse d'accidents.

Définition 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."[22]

Cette définition met en évidence la notion de "justification de la confiance", qui est une dépendance acceptée (explicitement ou implicitement). La dépendance d'un système par rapport à un autre système est l'influence, réelle ou potentielle, de la sûreté de fonctionnement de ce dernier sur la sûreté de fonctionnement du système considéré.

Définition 2 (SdF) : "La sûreté de fonctionnement (Dependability) consiste à évaluer les risques potentiels, prévoir l'occurrence des défaillances et tenter de minimiser les conséquences des situations catastrophiques lorsqu'elles se présentent."

Il existe beaucoup de définitions et de standards de la sûreté de fonctionnement qui peuvent varier selon les domaines d'application. Le Technical Committee 56 Dependability de l'International Electrotechnical Commission (IEC) développe

et maintient des standards internationaux reconnus dans le domaine de la sûreté de fonctionnement. Ces standards fournissent les méthodes et les outils d'analyse, d'évaluation, de gestion des équipements, services et systèmes tout au long du cycle de développement [23] [24] [25].

3.3 Concepts de base

Dans cette section nous introduisons les concepts de base de la sûreté de fonctionnement [23] [25]. Nous donnons des définitions précises caractérisant les principes qui influencent la sûreté de fonctionnement des systèmes informatiques. Les définitions données sont de caractère générales pour couvrir tout le spectre des systèmes informatiques .

3.3.1 Fonction et service d'un système

Un *système* est une entité qui est en constante interaction avec d'autres systèmes, qui constituent son environnement. La frontière du système est la limite commune entre le système et son environnement.

Un système est dit cohérent si :

— La panne de tous les composants entraîne la panne du système,

— Le fonctionnement de tous les composants entraîne le fonctionnement du système,

— Lorsque le système est en panne, aucune défaillance supplémentaire ne rétablit le fonctionnement du système,

— Lorsque le système est en fonctionnement, aucune réparation ne conduit à la panne du système.

La *fonction* d'un système est ce à quoi il est destiné. Elle est décrite par la spécification fonctionnelle, qui inclut les performances attendues du système.

Le *service* délivré par un système est son comportement tel que perçu par ses utilisateurs. Le comportement est ce que le système fait pour accomplir sa fonction, il est représenté par une séquence d'états externes. Un utilisateur est un autre système, éventuellement humain, qui interagit avec le système considéré. Un service correct est délivré par un système lorsqu'il accomplit sa fonction.

3.3.2 Attributs, Entraves et moyens

La sûreté de fonctionnement manipule principalement trois concepts [25] [23] [24] :

— Les Attributs : par quoi la sûreté de fonctionnement est évaluée ;

— Les Entraves : par quoi la sûreté de fonctionnement est affectée ;

— Les Moyens : par quoi la sûreté de fonctionnement est améliorée.

La Figure 3.1 résume les principales notions de la sûreté de fonctionnement.

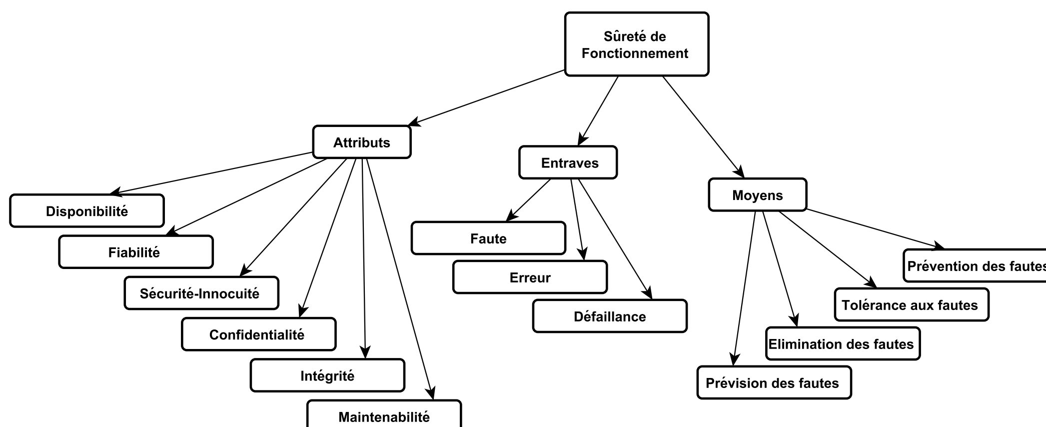


FIGURE 3.1 – Arbre de la sûreté de fonctionnement

3.3.2.1 Les attributs de sûreté de fonctionnement

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

— Disponibilité(Availability) : c'est le fait d'être prêt pour l'utilisation, c'est l'aptitude d'une entité à être en état d'accomplir une fonction requise dans des conditions bien déterminées, à un instant donnée ou pendant un intervalle de temps donné, en supposant que la fourniture des moyens extérieurs nécessaires soit assurée.

— Fiabilité (Reliability) : qui est la continuité du service, c'est l'aptitude d'un dispositif à accomplir une fonction requise dans des conditions données pendant une durée donnée.

— Sécurité-innocuité (Safety) : qui est l'absence de conséquences catastrophiques pour l'environnement, c'est l'aptitude d'une entité à éviter de conduire, dans des conditions données à des évènements critiques ou catastrophiques.

— Confidentialité : qui est l'absence de divulgations non autorisées de l'information.

— Intégrité : c'est l'absence d'altérations inappropriées de l'information.

— Maintenabilité (Maintainability) : permettre les réparations et les évolutions, c'est l'aptitude d'une entité à être maintenue ou rétablie, sur un intervalle de temps donnée dans un état dans lequel elle peut accomplir une fonction requise.

C'est l'application à laquelle est destiné le système informatique, qui définit l'importance des attributs de la sûreté de fonctionnement : disponibilité, intégrité, et maintenabilité sont généralement requises (à des degrés variables), alors que fiabilité, sécurité-innocuité, confidentialité peuvent ou non êtres requises. La mesure dans laquelle un système possède les attributs de la sûreté de fonctionnement doit être considérée de façon relative, probabiliste, et non de façon absolue, déterministe, et ça parceque un système n'est jamais totalement disponible, fiable ou sûr du fait de l'inévitable présence ou occurrence de fautes.

En plus des attributs déjà définis, et qui sont dans la plupart des références qualifiés d'attributs primaires, d'autres attributs peuvent être définis en tant qu'attributs secondaires, dont le rôle est l'affinement ou la spécialisation des attributs primaires. Un exemple d'attribut secondaire spécialisé est :

— Robustesse : la sûreté de fonctionnement par rapport aux fautes externes.
— Responsabilité : la disponibilité et l'intégrité de l'identité de la personne qui a effectuée une opération.
— Authenticité : l'intégrité du contenu et de l'origine d'un message.
— Non-réfutabilité : la disponibilité et l'intégrité de l'identité de l'émetteur d'un message, ou du destinataire.
— Testabilité : le degré d'un composant ou d'un système à fournir des informations sur son état et ses performances.
— Diagnosticabilité : la capacité d'un système à exhiber des symptômes pour des situations d'erreur.
— Survivabilité : la capacité d'un système à continuer sa mission après perturbation humaine ou environnementale.

L'importance donnée à chacun des attributs de la sûreté de fonctionnement influence directement sur le seuil des techniques à mettre en œuvre pour que le système résultant soit sûr de fonctionnement. Ceci est un problème délicat, surtout que certains attributs sont antagonistes (par exemple, disponibilité et sécurité-innocuité, disponibilité et sécurité-immunité), d'où la nécessité de négocier des compromis.

3.3.2.2 Les entraves à la sûreté de fonctionnement

Les entraves qui peuvent affecter le fonctionnement d'un système et dégrader la sûreté de fonctionnement sont classées en trois grandes classes [25], [24] : les fautes, les erreurs et les défaillances, qui s'enchaînent comme illustrées dans la Figure 3.2.

Figure 3.2 – Entraves de la sûreté de fonctionnement

Les flèches de la chaîne expriment la relation de causabilité entre fautes, erreurs et défaillances. Elles doivent être interprétées de façon générique : par propagation, plusieurs erreurs sont généralement générées avant qu'une défaillance ne survienne.

Faute (Fault) : Une faute dans un système [26] représente un défaut d'un composant matériel ou logiciel de ce système, elle est générée d'une manière intentionnelle ou accidentelle. Durant l'exécution du système, la faute reste inactive jusqu'à ce qu'un évènement intentionnel ou accidentel provoque son activation.

Quand une faute est active elle produit automatiquement une erreur. Une faute active est soit une faute interne qui était là mais inactive, et c'est le processus de traitement qui l'a activée, soit une faute externe qui a profité d'une vulnérabilité. Les fautes susceptibles d'affecter un système (de nature et de source extrêmement diverses), peuvent être classées selon sept points de vue, permettant de définir les classes de fautes élémentaires, comme indiqué sur le Tableau 3.1.

On combinant ces sept classes de fautes élémentaires, on se trouve avec des classes de fautes combinées, on peut noter que le reclassement de toutes les combinaisons des classes de fautes élémentaires est possible (et il y aurait exactement 192 classes de fautes combinées) ; la plus part de ces combinaisons n'étant pas pertinentes, donc les classes de fautes combinées peuvent être regroupées en trois grandes classes non exclusives :

— Fautes de développement.
— Fautes physiques ou fautes du matériel.
— Fautes d'interaction ou externes.

Reproduire l'activation d'une faute est par définition la possibilité d'identifier les conditions et l'environnement de l'activation d'une faute (source d'une ou plusieurs erreurs). La Figure 3.3 définit les fautes solides ou furtives selon que leurs conditions d'activation sont reproductibles ou non ; elle définit aussi la notion de faute intermittente, en raison de la similitude de manifestation des fautes furtives et des fautes temporaires.

Erreur (Defect) : La présence d'un état interne erroné pendent l'activation d'un système conduit dans des circonstances particulières à une erreur, c'est-à-dire à un résultat incorrect ou imprécis. Donc l'erreur est le résultat d'une faute et la cause

Faute	Création ou occurrence	Fautes de développement : (imperfections de développement ou de maintenance).
		Fautes opérationnelles : au moment de l'exploitation du système.
	Frontière du système	Fautes internes : activées par les traitements.
		Fautes externes : suite aux interactions du système avec son environnement extérieur (physique ou humain).
	Cause	Fautes naturelles : sans intervention humaine.
		Fautes dues à l'homme : suite aux actions et imperfections humaines.
	Dimension	Fautes matérielles : elles affectent le matériel.
		Faute logicielle : elles affectent le logiciel (programmes ou données).
	Intention	Fautes malveillantes : elles sont humaines et faites exprès.
		Fautes non malveillantes : elles sont sans intentions malveillantes.
	Capacité	Fautes accidentelles : elles sont créées de manière fortuite.
		Fautes délibérées : elles sont créées délibérément avec une décision.
		Fautes d'incompétence : elles sont le résultat d'un manque de compétence professionnelle.
	Persistance	Fautes permanentes : ce type de fautes est continue dans le temps (jusqu'a réparation).
		Fautes temporaires : ces fautes sont présentées pour une durée limitée.

TABLE 3.1 – Classes de fautes

d'une défaillance. Une erreur peut être latente tant qu'elle n'a pas été reconnue en tant que telle ou détectée par un algorithme ou un mécanisme de détection qui permet de la reconnaître comme telle. Une erreur peut disparaître avant d'être détectée. Une erreur peut aussi crée des nouvelles erreurs par propagation [24].

Défaillance : Quand un service correct est délivré par un système, on dit que ce système a accompli sa fonction. Une défaillance du service ou simplement défaillance, est un évènement qui survient lorsque le service délivré n'est plus correct (parce qu'il ne respecte pas la spécification fonctionnelle ou parceque la spécification

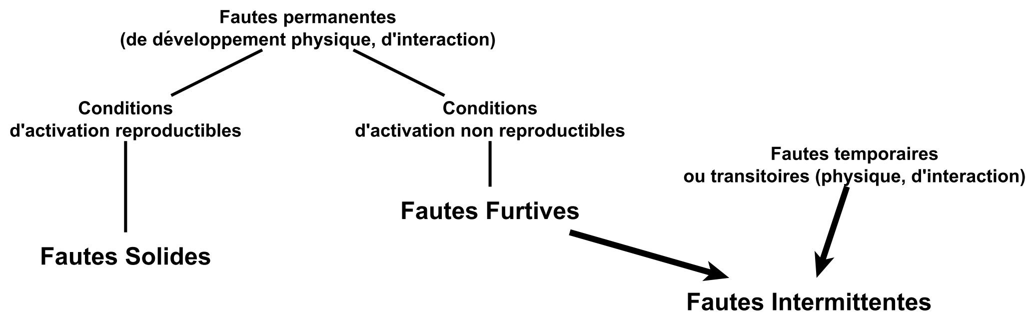


Figure 3.3 – Fautes solides, furtives et intermittentes

fonctionnelle ne décrivait pas de manière adéquate la fonction du système).

Une défaillance est par définition le passage du service de l'état correct à un état incorrect, c'est le résultat d'une erreur qui à changé le comportement du système et provoque le non respect de sa spécification. Le résultat d'une défaillance (d'un composant) est une faute non seulement pour le système qui le contient mais aussi pour tous les les composants qui interagissent avec lui. La délivrance d'un service incorrect est une panne du service, et la restauration du service est par définition le passage de l'état de service incorrect à l'état d'un service correct [26].

Mode de défaillance (Failure mode) : Un système ne défaille généralement pas toujours de la même façon, ce qui conduit à la notion de mode de défaillance. Un mode de défaillance spécifie l'effet par lequel une défaillance est observée. Un mode de défaillance peut être caractérisé selon quatre points de vue, comme indiqué dans le Tableau 3.2, ce qui défini la classification des défaillances du service.

On notes les trois commentaires suivants sur les modes de défaillance :

— Une défaillance non signalée est le résultat direct d'une défaillance des mécanismes de détection d'une délivrance d'un service incorrect ; un autre cas peut se produire lorsque ces mécanismes détectent et signalent une défaillance inexistante (donc émettent une fausse alarme).
— Un système dont toutes les défaillances sont acceptables avec des défaillances par arrêt est un système à arrêt sur défaillance ; un système dont toutes les défaillances sont acceptables mais avec des défaillances bénignes est un système sûr en présence de défaillance.
— Le déni de service est l'exemple des malveillances qui peuvent affecter la disponibilité du service.

La gravité des défaillances instaure un classement des conséquences des défaillances sur l'environnement du système. Lorsque la fonction du système comporte un ensemble de fonctions élémentaires, la défaillance d'un ou de plusieurs services

<table>
<tr><td rowspan="9">Défaillance</td><td rowspan="2">Domaine</td><td>Défaillance en valeur : l'information délivrée ne cordonne pas avec la fonction du système.</td></tr>
<tr><td>Défaillance temporelles : le moment de la délivrance de l'information ne permet pas l'accomplissement de la fonction du système.</td></tr>
<tr><td rowspan="2">Détectabilité</td><td>Défaillance signalées : la possibilité de détecter et de signaler que le service délivré est incorrect.</td></tr>
<tr><td>Défaillance non signalées : La délivrance d'un service incorrect n'est pas détectée.</td></tr>
<tr><td rowspan="2">Cohérence</td><td>Défaillances cohérentes : tous les utilisateurs aperçoivent le service incorrect identiquement.</td></tr>
<tr><td>Défaillances incohérentes ou défaillances byzantines : certains ou tous les utilisateurs n'aperçoivent pas le service incorrect de la même manière.</td></tr>
<tr><td rowspan="3">Conséquences</td><td>Défaillances bénignes : le dommage causé au système ou à son environnement est négligeable et sans présenter de risque pour l'homme.</td></tr>
<tr><td>Défaillance critique (hasardeuse) : elle entraîne la perte d'une ou des fonctions importantes du système et cause des dommages importants au système.</td></tr>
<tr><td>Défaillances catastrophiques</td></tr>
</table>

TABLE 3.2 – Modes de défaillance

remplissant ces fonctions peut laisser le système dans un mode dégradé, qui offre encore un sous-ensemble de services à l'utilisateur. Plusieurs de ces modes dégradés peuvent être identifiés, tels que service ralenti, service restreint, service d'urgence, etc. Dans ce cas, le système est dit avoir souffert des défaillances partielles.

Le service délivré étant une séquence d'états externes, une défaillance du service signifie qu'au moins un état externe dévie du service correct. La déviation est une erreur. La cause adjugée ou supposée d'une erreur est une faute. Les fautes peuvent être internes ou externes au système. La présence antérieure d'une vulnérabilité, c-à-d d'une faute interne qui permet à une faute externe de causer des dommages au système, est nécessaire pour qu'une faute externe entraîne une erreur, et, éventuellement, une défaillance.

Généralement, une faute cause d'abord une erreur dans l'état interne d'un composant, l'état externe du système n'étant pas immédiatement affecté. Il s'ensuit la définition d'une erreur : partie de l'état total du système qui est susceptible d'entraîner sa défaillance, qui survient lorsque l'erreur affecte le service délivré.

3.3.2.3 Les moyens de la sûreté de fonctionnement

Toute solution développée dans le cadre d'un système sûr de fonctionnement doit utilisée une ou une combinaison d'un ensemble de méthodes qui sont des solutions éprouvées pour casser les enchaînements Faute $\longmapsto$ Erreur $\longmapsto$ Défaillance et donc améliorer la fiabilité du système. Ces méthodes sont les moyens de la sûreté de fonctionnement [24] [8] et peuvent être classées en :

- Méthode de prévention des fautes : le principe de ces méthodes est d'empêcher l'occurrence ou l'introduction de fautes qui auraient pu être introduites pendant le développement du système. Cela peut être réaliser par des bonnes techniques d'implantation et de développement.
- Tolérance aux fautes : elle consiste à mettre en place des mécanismes qui maintiennent le service fourni par le système, même en présence de fautes et on peut même accepter un fonctionnement dégradé. les outils de base de la tolérance aux fautes sont les mécanismes de redondance, l'idée est de réaliser la même fonction par des moyens différents.

On distingue plusieurs types de redondance :

1. Redondance homogène : on réplique plusieurs composants identiques.
2. Redondance avec dissemblance : les sous-systèmes réalisent les mêmes fonctions mais sont différents.
3. Redondance froide : les composants sont activés quand ceux déjà actifs tombent en panne.
4. Redondance chaude : les composants tournent en parallèle avec une politique de prise en main.
5. Redondance tiède : les composants sont idle (non actif, tourner au ralenti) avant de prendre la main.

A la base de toutes les solutions existantes, la récupération de plusieurs valeurs calculées par redondance est de déterminer laquelle est la plus proche de la réalité. c'est l'idée de base des mécanismes comme les comparateurs ou les voteurs.

1. Élimination des fautes : elle consiste à réduire la présence (nombre, sévérité) des fautes, l'élimination de faute peut être divisée en 2 catégories :
   - Elimination pendant le développement.
   - Elimination pendant l'utilisation.
2. Prévision des fautes : elle consiste à estimer la présence, le taux futur, et les possibles conséquences des fautes (leur impact sur le système).

La Figure 3.4 suivante illustre les différents groupements des moyens pour la sûreté de fonctionnement,

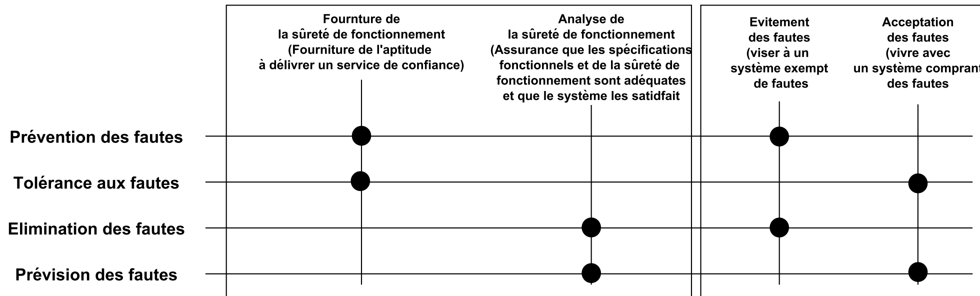


Figure 3.4 – Groupements des moyens pour la sûreté de fonctionnement

3.4 Tolérance aux fautes

Un système dont les programmes peuvent être exécutés correctement même en présence de fautes est un système tolérant aux fautes, dans la littérature plusieurs travaux traitent le concept de la tolérance aux fautes [27] [28].

3.4.1 Techniques de la tolérance aux fautes

La tolérance aux fautes à comme objectif principal d'éviter les défaillances. Sa mise en œuvre est réaliser par la détection des erreurs et le rétablissement du système [24]. La Figure 3.5 liste les techniques de tolérance aux fautes. Habituellement le traitement de fautes est complété par des opérations de maintenance corrective, afin d'éliminer les composants passives.

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

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

Il est à noter que :

— Reprise et poursuite ne sont pas exclusives : une reprise peut d'abord être tentée ; si l'erreur persiste, une poursuite peut alors être entreprise ;
— Les fautes intermittentes ne nécessitent ni passivation, ni reconfiguration. Le traitement d'erreur est le moyen le plus utilisé pour identifier si une faute est

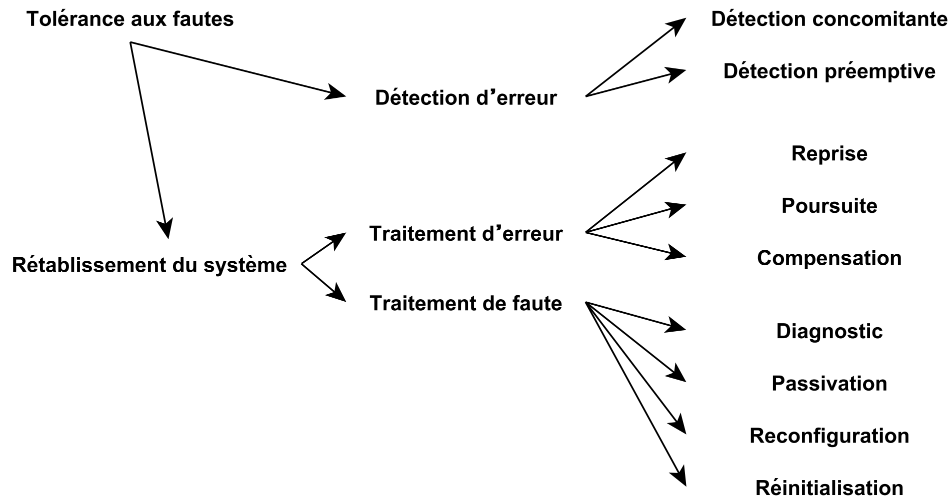


FIGURE 3.5 – Technique de tolérance aux fautes

intermittente ou non (la récurrence d'une erreur indique que la faute n'est pas intermittente) ou par diagnostic de faute dans le cas de la poursuite.

3.4.1.1 Détection d'erreur

La redondance permet la détection d'erreurs, elle peut prendre plusieurs formes : redondance au niveau information ou composant, redondance temporelle ou algorithmique. La forme la plus sophistiquée de la détection d'erreur consiste à construire des composants auto testables 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 [28] [29].

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.

3.4.1.2 Rétablissement du système

Trois formes de rétablissement du système ont été identifiées, cette identification se fait par rapport au moyen utilisé pour reconstruire un état correct. Les trois formes

sont : la reprise, la poursuite et la compensation d'erreur.

Reprise : La reprise est la technique la plus utilisée pour le rétablissement du service d'un système. Elle consiste en la sauvegarde périodique de l'état du système de façon à pouvoir, après avoir détecté une erreur, ramener le système vers un état dit 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 [30] [31].

La sauvegarde opère par des mécanismes matériels ou logiciels permettant de sauvegarder automatiquement les données modifiées entre deux points de reprise. Si la couverture de détection n'est pas totale, les points de reprise peuvent être eux aussi sujet d'une erreur avant même qu'elle ne soit détectée. Dans ce cas, la reprise ne pourra être efficace que s'il est possible de restituer un état exempt d'erreur, ce qui implique qu'il doit exister plusieurs points de reprise successifs ou que la structure de l'application permette de conserver des points de reprise emboîtés.

Les inconvénients des techniques de reprise sont : premièrement, elles sont généralement incompatibles avec les applications ayant des contraintes temps réel strictes. Deuxièmement, la taille des points de reprise et le surcoût temporel nécessaire à leur établissement imposent souvent des contraintes structurelles qui doivent être prises en compte pendent le développement de l'application, avec un support spécifique du système d'exploitation. Cela interdit généralement l'usage de systèmes d'exploitation et de tout logiciel qui n'aurait pas été développé spécialement pour l'architecture en question.

Poursuite : La poursuite se définie comme 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 même en mode dégradé. L'une des plus simple réalisation de la poursuite consiste à réinitialiser le système et à acquérir un nouveau contexte à partir de l'environnement lui même (comme par exemple la relecture des capteurs). Une autre façon de faire est le traitements d'exceptions, en se basant sur des primitives offertes par certains langages de programmation [30].

Compensation : Avec la compensation d'erreur, le système doit comporter suffisamment de redondance, pour permettre sa transformation en un état exempt d'erreur, en dépit des erreurs qui pourraient l'affecter [24]. La compensation ne nécessite ni la ré-exécution d'une partie de l'application (comme dans le cas de la reprise), ni l'exécution d'une procédure dédiée (comme dans le cas de la poursuite), pour permettre la continuation du service. Ce type de recouvrement est donc relativement

transparent par rapport à l'application elle même, ce qui permet l'utilisation de systèmes d'exploitation et de progiciels standard (il n'est pas nécessaire de structurer l'application en vue d'un éventuel traitement d'erreur).

La compensation d'erreur peut être par détection d'erreur (détection et compensation), ou systématique (dans le cas du masquage). Enfin, elle peut être aussi fournie par les codes correcteurs d'erreur, spécialement pour la transmission et ou le stockage de l'information.

3.4.2 Tolérance aux fautes logicielles et matérielles

Un système peut défaillir à cause d'une faute logicielle ou matérielle.

3.4.2.1 Tolérance aux fautes logicielles

Le problème de conception et développement (écriture) de logiciels est très difficile [32]. Il y a beaucoup de difficultés essentielles ou accidentelles pour produire un logiciel correct et sans erreurs. A l'origine de difficultés essentielles, le défi d'essayer de comprendre l'application et l'environnement de fonctionnement qu'ils sont souvent complexes. Pour le cas des difficultés accidentelles, l'origine est qu'on ne peut pas produire un bon logiciel du moment ou les programmeurs peuvent faire des erreurs dans les programmes. Un exemple intéressant de faute logicielle est l'explosion de la fusée Ariane 5 en juin 1996 à cause des erreurs de conception du logiciel.

Il existe deux techniques de base pour la tolérance aux fautes logicielles :

— **Uni-version du logiciel :** l'objectif principal est de tolérer les fautes logicielles en utilisant une seule version du logiciel, ou d'un de ses composants. Pour tolérer la faute de chaque uni-version du logiciel, le concepteur doit modifier cette version en lui ajoutant des mécanismes de détection et de traitement d'erreurs [16] [17] [18] [19]. Des exemlpes des approches qui utilisant cette technique : le traitement d'exceptions, la détection d'erreur, le point de reprise (check-point and restart),

— **Multi-version du logiciel :** cette technique est basée sur le principe de la redondance logicielle, où chaque composant logiciel est répliqué pas identiquement mais en plusieurs versions programmées différemment. Le premier avantage par rapport à l'approche uni-version est que ces versions logicielles peuvent être exécutées en séquence ou en parallèle pour tolérer les fautes de certaines versions. Un autre avantage est que ces différentes versions logicielles peuvent être, développés par différents développeurs sur des outils différents. Parmi les approches utilisant cette technique, on trouve : N-version de programmation, et c'est la cas des commandes de vol des avions Airbus et Boeing.

3.4.2.2 Tolérance aux fautes matérielles

La tolérance aux fautes matérielles est le domaine le plus développé de la tolérance aux fautes générale. Beaucoup de techniques ont été développées et utilisées. On peut tolérer les fautes matérielles soit par logiciel soit par matériel. La solution matérielle consiste à répliquer des composants matériels, et même si le coût des processeurs ne cesse de décroître, cette solution a un notre inconvénient qu'est l'augmentation de la consommation de l'énergie, c'est pourquoi on opte généralemnt pour les solutions basées sur la redondance logicielle. Dans la suite du manuscrit, on désigne par faute une faute matérielle. Nous ne ferons pas un état de l'art sur la tolérance aux fautes logicielles qui est un domaine de recherche en soi. On fait donc l'hypothèse que le logiciel est sans fautes et donc dans la suite on ne considèrera que la tolérance aux fautes matérielles.

3.4.3 Techniques logicielles de tolérance aux fautes matérielles

3.4.3.1 Reprise locale

Minimiser l'impact négatif de la défaillance temporaire d'un processus par un redémarrage rapide, est la forme la plus simple de la tolérance aux fautes dans un système réparti. Cela peut être facile à mettre en œuvre si ce processus ne sauvegarde aucun état entre deux requêtes. Pas d'état interne implique la non nécessité de la restauration des données lors d'une reprise. Cela implique aussi qu'aucune information n'est conservée [24]. C'est cette stratégie qui permet, par exemple, la conception du système de fichier réseau de Sun (NFS). Si au contraire un processus serveur comporte un état interne, il faut que celui-ci soit sauvegardé sur une mémoire stable en tant que point de reprise. Une reprise locale est effectuer par un processus lorsque il n'a pas interagi avec d'autres processus depuis le dernier point de reprise, ou si d'éventuelles interactions peuvent être rejouées. Si cela n'est pas le cas, une reprise répartie de plusieurs processus est nécessaire.

3.4.3.2 Reprise répartie

Une reprise répartie est faite si une communication interne provoque à la suite d'une reprise d'un processeur, la reprise de d'autres processeurs. Les processus doivent reprendre leurs exécutions depuis un ensemble de points de reprise qui constituent un état global cohérent.

Un effet domino peut survenir lorsque le seul état global cohérent antérieur est l'état initial du système. La définition d'un état global cohérent dépend du positionnement du protocole de reprise par rapport au protocole assurant la fiabilité des communications (Figure 3.6) [33].

Lorsque le protocole de reprise est situé au-dessus d'un protocole de communication fiable (Figure 3.6(a)), un ensemble de points de reprise ne constituent un état global cohérent que si la « ligne de reprise » qui représente cet ensemble n'est traversée par aucun message (A et B sur la Figure3.6(b)). Dans le cas contraire (Figure 3.6 (c)), une ligne de reprise traversée éventuellement par des messages de gauche à droite (C sur la Figure 3.6 (b)) représente aussi un état global cohérent, car ces messages « émis » mais pas encore reçus, seront réémis plus tard par le protocole de communication fiable (dont l'état fait parti de l'état sauvegardé par les points de reprise).

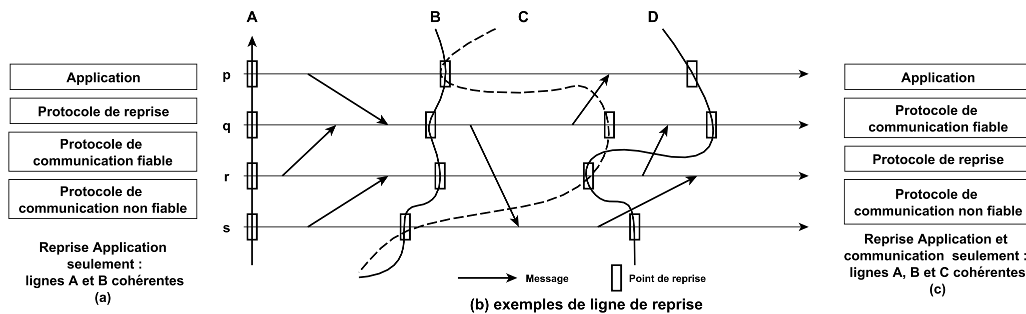


FIGURE 3.6 – Processus Communicant Reprise

La ligne D de la Figure 3.6(b) ne correspond à aucun état global cohérent, car elle est traversée par un message de droite à gauche, ce qui correspondrait à un message reçu mais pas encore émis.

On distingue généralement trois approches pour la reprise répartie :

— La création indépendante des points de reprise : chaque processus est autorisé à créer des points de reprise au moment opportun pour lui ;

— La création coordonnée des points de reprise permet de créer uniquement des lignes de reprise cohérentes ;

— La création de points de reprise induite par la communication est un compromis entre les deux approches précédentes.

Les techniques de reprise considérées jusqu'à maintenant peuvent êtres utilisées pour tolérer des fautes permanentes si les points de reprise sont enregistrés sur une mémoire stable accessible depuis le réseau de communication. En effet, cela autorise la création de nouveaux processus sur des processeurs non défaillants qui peuvent être initialisés à partir de ces points de reprise.

3.4.3.3 Données répliquées

Du point de vue des données, l'utilisation de la redondance permet d'assurer la disponibilité de données, puisque une donnée dupliquée est accessible même si certaines copies résident sur des nœuds défaillants ou inaccessibles. D'autre part, il est logiquement plus rapide de lire une copie proche qu'une copie distante. Cependant, garder la cohérence des données par des opérations d'écriture sur des données dupliquées peut ralentir le traitement, car elle impliquent potentiellement toutes les copies.

Les protocoles de gestion de données dupliquées sont qualifiés d'optimistes ou de pessimistes selon s'ils négligent ou non l'accès conflictuels en cas de partitionnement des copies [24].

Les protocoles pessimistes permettent à tout utilisateur de perçoir la donnée comme si elle n'existait qu'en une seule copie) en forçant l'exclusion mutuelle entre les opérations d'écriture et de lecture. Le protocole le plus simple s'appelle « lire une, écrire toutes » : un processus utilisateur peut lire n'importe quelle copie, mais il doit effectuer toute écriture sur l'ensemble des copies. Cette technique permet de fournir une excellente performance en lecture, mais en contre partie elle fournit une mauvaise performance en écriture, et même par fois les écritures sont bloquées si une seule copie de la donnée n'est pas accessible.

Les protocoles optimistes améliorent la disponibilité des données en sacrifient leurs cohérence. Avec ces protocoles l'écriture sur les copies présente dans des composants distincts est autorisée. Par exemple, le protocole à copies disponibles est une variante optimiste du protocole « lire une, écrire toutes » par laquelle seules les copies accessibles sont modifiées lors d'une opération d'écriture. Mais tout conflit résultant d'écritures effectuées dans des composants différents doit être détecté et résolu. cette résolution de conflit ne peut être automatique que dans des cas très particuliers parceque elle dépend directement de la sémantique des données.

3.4.3.4 Processus répliqués (redondance logicielle)

Les méthodes de tolérance aux fautes matérielles diffèrent selon l'origine et le type de fautes pris en compte [24]. Un service tolérant aux fautes peut être mis en œuvre en coordonnant un ensemble de processus répliqués sur deux ou plusieurs processeurs.

Avec la redondance logicielle, la nécessité d'avoir une architecture distribuée est obligatoire, pour pouvoir attribuer les répliques aux différents processeurs. 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 [34] [34] [35]. L'ensemble des processus (tâches) doit être géré de façon à présenter le service correct comme étant fournit par un seul processus, malgré la défaillance d'un sous ensemble des membres de l'ensemble.

On distingue généralement trois classes d'algorithmes d'ordonnancement temps réel et tolérants aux fautes (voir chapiter 4) :

1. Les algorithmes basés sur la redondance active,
2. Les algorithmes basés sur la redondance passive et
3. Les algorithmes basés sur la redondance hybride.

Ces algorithmes s'appliquent à deux types de composants matériels : les processeurs et les médias de communication.

Redondance passive : Avec la Redondance passive, une seule copie dite primaire traite tous les messages reçus, met à jour son état interne et effectue l'envoi de messages de sortie. La copie primaire met à jour une copie de son état interne, pour créer un point de reprise sur une mémoire stable accessible par les copies de secours, ces dernières ne font rien tant que la copie primaire fonctionne correctement (Redondance passive froide), soit, par les copies de secours elles mêmes (redondance passive tiède). Lorsque la copie primaire est cible de faute, une des copies de secours est élue pour la remplacer [36] [37] [38] [39] [40].

Redondance Active : La réplication active est basée sur le même traitement équitable de toutes les copies de tâches. Chacune traite tous les messages reçus, met à jour son état interne de la même manière, et génère les messages de sortie. Le choix des messages de sortie effectifs est fait par le moyen d'une fonction de décision qui dépend des hypothèses de défaillance. Par exemple et pour des arrêts simples, la fonction de décision peut être de choisir le premier message disponible, alors que pour des défaillances arbitraires, le choix peut être fait au moyen d'une fonction de décision à vote majoritaire [26] [41] [42] [43].

Redondance hybride Dans la redondance hybride [44] [45] [16] tout comme dans la réplication active toutes les copies reçoivent les messages d'entrée et peuvent ainsi les traiter. Cependant, comme pour la réplication passive, le traitement n'est pas le même car seulement la copie privilégiée (primaire) assume la responsabilité de certaines décisions (par exemple, sur l'acceptation des messages, ou sur la préemption du traitement en cours).

Comparaison entre les trois types de redondance : Le Tableau 3.3 compare les différentes approches de tolérance aux pannes basées sur la redondance active, passive ou hybride des tâches et/ou des communications.

On note ici que avec la redondance active une faute n'augmente pas la latence du système temps réel, ce qui n'est pas le cas de la redondance passive, où une faute de la réplique primaire peut augmenter la latence de manière significative. Cependant, la redondance passive permet une meilleure utilisation des ressources matérielles

	Approche		
	Redondance active	Redondance passive	Redondance hybride
Temps de réponse	un temps de réponse prévisible, et rapide (architectures avec un taux élevé de parallélisme)	temps de réponse meilleur en cas d'absence de fautes ; la panne de la réplique primaire augmente le temps de réponse	temps de réponse meilleur en cas de présence de fautes
Détection de pannes	pas de mécanisme	mécanisme dédie	mécanisme dédie
Traitement des erreurs	par compensation	par recouvrement	par compensation et par recouvrement
Reprise après panne	immédiate	non immédiate	immédiate (compensation) et non immédiate (recouvrement)

Table 3.3 – Comparaison entre les trois approches de redondance.

offertes par l'architecture en réduisent aux maximum le surcharge sur les processeurs et sur le réseau de communication. Le choix d'une stratégie de réplication est guider par les contraintes et les besoins applicatifs. Par exemple, il est plus logique d'utiliser la réplication active en cas de défaillances fréquentes des composants matériels, et la réplication passive lorsque le nombre de communications est élevé.

3.4.4 Techniques matérielle de tolérance aux fautes matérielles

Dans la plupart des systèmes critiques, comme les commandes de vol ou les circuits hydrauliques des avions, certains actionneurs et capteurs sont doublés et même triplés par fois. Une défaillance d'un composant peut donc être compensée par les autres copies, et comme la défaillance de chaque composant est un évènement complètement indépendant de celle des deux autres, et que les composants sont supposés de bonne fiabilité, alors il est extrêmement improbable que les trois composants soient fautifs.

Supposons que nous appliquons les mêmes entrées à deux circuits logiques et que nous comparons les sorties. Si nous obtenons le même résultat, nous pouvons conclure que les deux composants sont fonctionnels, ou qu'ils présentent une défaillance tous les deux. Le problème de devient délicat si l'un seulement des deux défaille, et dans ce cas on ne peut pas savoir lequel exactement juste en comparant leurs sorties. C'est la principale limite des systèmes DMR (Dual Modural Redundant). En dupliquant un composant à la redondance la situation s'améliore : avoir les mêmes sorties implique que les trois composants sont fonctionnels (avec une grande probabilité) ou qu'ils défaillent simultanément (cas très peu probable), et si on a deux des trois sorties qui sont identiques, alors il est plus probable que le composant ayant une sortie différente est le composant défaillant. Cette redondance est dite redondance modulaire triple TMR (Triple Modular Redundancy) [46] [47].

Redondance modulaire triple : La Figure 3.7 montre un circuit TMR [48] [49] avec trois circuits logiques identiques A,B et C ayant la même entrée . Les sorties sont comparées à l'aide du composant voteur qui donne un résultat correct à la majorité des votes. Si aucun ou l'un des composant défaille le résultat délivré est correct, mais si les deux ou les trois composants défaillent, le résultat sera erroné. Certains circuits défaillants ont une sortie mise à 1 ou à 0, dans ce cas, cette information sera prise en compte par le voteur.

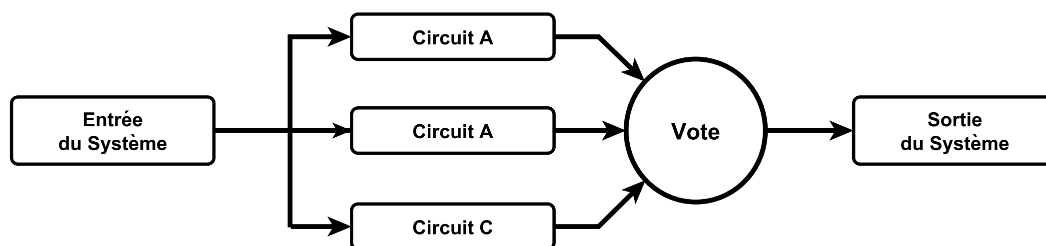


FIGURE 3.7 – Redondance Modulaire Triple

3.5 Mise en œuvre de la tolérance aux fautes

Si on augmente la capacité de traitement fonctionnel d'un composant par un mécanisme de détection d'erreur on se trouve avec un composant auto testable ; le principal intérêt est la possibilité de définir clairement des zones de confinement d'erreur.

Éviter que la propagation d'erreurs n'affecte le travail des composants non défaillants est la mission principale de la coordination des activités de composants multiples. Cet aspect devient particulièrement important lorsqu'un composant donné doit intéragir et communiquer avec d'autres composants (les informations communiquées peuvent êtres des données locales de capteurs ou la valeur d'une horloge locale) [24]..

La tolérance aux fautes est aussi récursive : on doit aussi protéger les mécanismes destinés à mettre en œuvre la tolérance aux fautes puisque eux aussi doivent être protégés contre les fautes susceptibles de les affecter. Des exemples sont fournis dans [31], ils utilisent la rédondance des voteurs, par les contrôleurs auto-testables, où la notion de mémoire stable est utilisée pour sauvegarder les programmes et les données de la reprise .

3.6 Conclusion

Dans ce chapitre nous avons présenté les terminologies et les principes de la sûreté de fonctionnement. En s'appuyant sur la notion de tolérance aux fautes matérielles. Les principales méthodes ont été détaillées, d'une part pour la redondance matérielle et surtout pour la redondance logicielle.

Enfin, les principales tendances et les défis posés à l'informatique, en ce qui concerne la tolérance aux fautes, ont été brièvement exposées. Le lecteur est encouragé à approfondir l'ensemble de ces aspects à partir de la bibliographie.

Chapitre 4

Etat de l'art sur la tolérance des fautes des processeur et des communication temps réel

Sommaire

4.1 Introduction **45**

4.2 Tolérance aux fautes **46**

4.2.1 Les algorithmes d'ordonnancement basés sur la redondance active 46

4.2.2 Les algorithmes d'ordonnancement basés sur la redondance passive 50

4.2.3 Les algorithmes d'ordonnancement basés sur la redondance hybride 53

4.3 Fiabilité **54**

4.3.1 Approche uni-objectif 54

4.3.2 Approche multi-objectifs 55

4.4 Conclusion **56**

4.1 Introduction

La fiabilité et la tolérance aux fautes ont donné naissance à des algorithmes d'ordonnancement temps réel à objectif de tolérance aux fautes et/ou de fiabilité. Plusieurs approches logicielles basées sur la théorie d'ordonnancement qui intègrent des mécanismes de tolérance aux fautes matérielles des processeurs et des médias de communication ont été proposées dans la littérature pour concevoir des systèmes communicants fiables.

Ce chapitre présente un état de l'art sur les travaux destinés au développement de ces algorithmes, qui permettent de résoudre le problème de la fiabilité en prenant en considération les taux de défaillance des composants matériels durant leur processus d'allocation spatiale pour tolérer des fautes matérielles des processeurs et des médias de communications.

4.2 Tolérance aux fautes

La tolérance aux fautes matérielles constituent le domaine le plus développé de la tolérance aux fautes en général. Différentes techniques ont été développées et utilisées dans des domaines allant du téléphone aux missions spatiales ; on peut tolérer les fautes soit par logiciel soit par matériel. La solution matérielle consiste à répliquer des composants matériels, et même si le coût des processeurs ne cesse de décroitre, cette solution augmente la consommation d'énergie, c'est pourquoi on opte souvent pour la solution logicielle basée sur la redondance logicielle.

Plusieurs techniques logicielles ont été développées ces dernières années pour intégrer des mécanismes de tolérance aux fautes matérielles des processeurs et média de communication dans les systèmes distribués temps-réel embarqués, elles diffèrent sur leurs hypothèses de défaillance, les mécanisme utilisés, le service fourni, la disponibilité et la sécurité.

La technique de la redondance logicielle [34] [50], est à la base de la plupart des algorithmes d'ordonnancement temps réel développés dans la littérature pour tolérer des fautes matérielles des processeurs et des média de communication. Elle nécessite une architecture distribuée afin de pouvoir allouer les répliques sur des processeurs différents. La redondance logicielle est la technique de tolérance aux fautes matérielles la plus adaptée aux systèmes où le nombre de composants matériels ne peut pas être facilement augmenté et où l'exigence sur les contraintes d'embarquabilité est vraiment forte.

On présente dans ce qui suit trois grandes classes d'algorithmes d'ordonnancement temps réel et tolérants aux fautes, à savoir :

1. Les algorithmes basés sur la redondance active,
2. Les algorithmes basés sur la redondance passive,
3. Les algorithmes basés sur la redondance hybride.

Ces algorithmes s'appliquent aux deux types de composants matériels : les processeurs et les médias de communication. Beaucoup de nos travaux passés [51] [52] [53] [54] [55] [56] [57] [58], se basant sur la stratégie d'ordonnancement utilisant la redondance logicielle, pour générer des ordonnancements tolérants aux fautes.

4.2.1 Les algorithmes d'ordonnancement basés sur la redondance active

4.2.1.1 Redondance Active :

La redondance active [26] [41] [42] [43] est utilisée pour masquer la panne de plusieurs composants matériels (processeurs et média de communication), elle se base sur la réplication des composants logiciels (tâches) de l'algorithme. Toutes les répliques sont exécutées de sorte à ce qu'en présence de fautes il y a toujours au moins

une qui soit exécutée. La redondance active est adaptée à toutes les hypothèses de défaillance : pannes temporelles, pannes byzantines, pannes transitoires et silence sur défaillances ; elle permet de masquer N pannes de processeurs en répliquant activement chaque tâche A sur $N+1$ processeurs distincts.

La redondance active traite toutes les répliques sur un pied d'égalité : chacune traite tous les messages reçus, met à jour son état interne de façon autonome, et génère les messages de sortie. Pour cela, chaque réplique A_i de A doit recevoir ses données d'entrées en $N+1$ exemplaires.

4.2.1.2 Exemple

Afin d'illustrer le mécanisme de la redondance active, nous considérons le modèle de tâches et d'architecture de la Figure 4.1. Le modèle de tâches (Figure 4.1(a)) est composé de deux tâches A et B et d'une dépendance de données *data*. Le modèle d'architecture (Figure 4.1(b)) est composé de cinq processeurs P_1, P_2, P_3, P_4 et P_5, et de six médias de communication (liens) l_{P_1,P_2}, l_{P_1,P_3}, l_{P_1,P_5}, l_{P_3,P_5}, l_{P_2,P_4} et l_{P_4,P_5}.

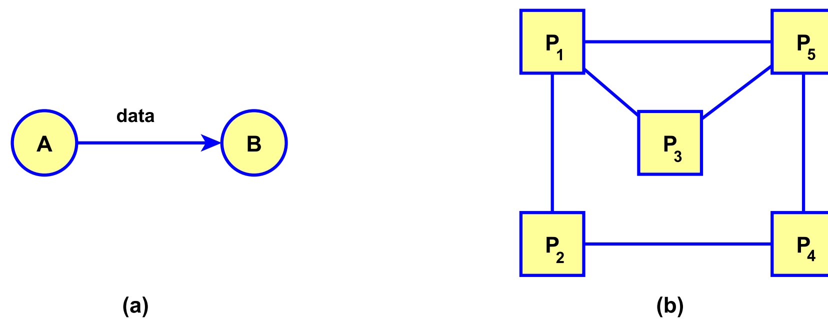


FIGURE 4.1 – Modèles de tâches et d'architecture

Tolérance aux fautes des processeurs : Une faute d'un processeur implique l'inactivité de toutes les tâches implantés sur ce processeur. La redondance active consiste à répliquer chaque tâche de l'algorithme sur $N+1$ processeurs pour tolérer au plus N fautes de processeurs.

Dans la Figure 4.2(a) les deux tâche A et B ont été allouées aux processeurs P_1 et P_2. La tâche A (resp. B) du modèle de tâches est répliquée en deux répliques A_1, A_2 (resp. B_1, B_2), qui sont implantées sur deux processeurs distincts P_1 et P_2 (resp. P_2 et P_5) (voir Figure 4.2(b)), afin de tolérer une faute permanente d'un seul processeur (à savoir P_1).

Tolérance aux fautes des médias de communication : Les fautes des médias de communication engendrent la perte de messages dans les réseaux de communi-

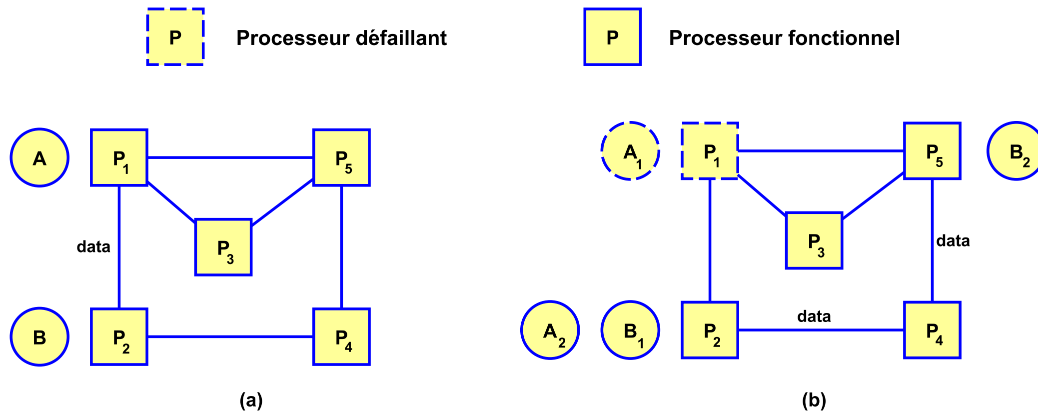


FIGURE 4.2 – Tolérance aux fautes des processeurs par redondance active

cation. Ces fautes peuvent être tolérées par la transmission de ces mêmes messages via plusieurs routes disjointes.

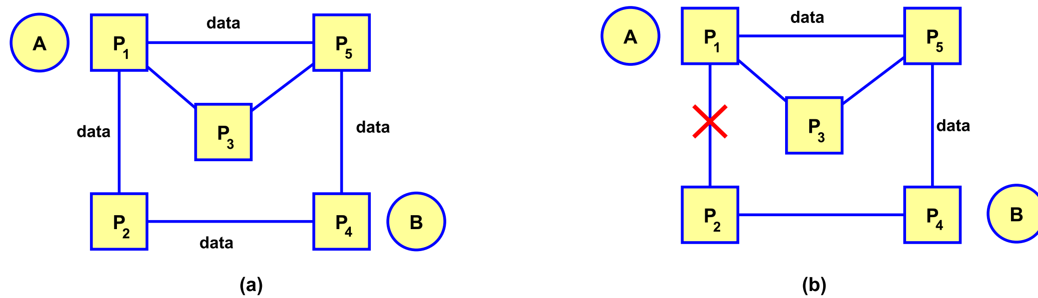


FIGURE 4.3 – Tolérance aux fautes des médias de communication par redondance active

Dans la Figure 4.3(a) la communication $A \to B$ $(data)$ est implantée sur deux routes disjointes reliant le processeur P_1 à P_4. Le processeur P_1, implantant l'opération A, envoie au processeur P_4, implantant l'opération B, les données $data$ via les deux routes disjointes : $Route_1 = (l_{P_1,P_2}, l_{P_2,P_4})$ et $Route_2 = (l_{P_1,P_5}, l_{P_5,P_4})$.

4.2.1.3 État de l'art de la tolérance des fautes des processeurs

Les auteurs de l'article [59] proposent une heuristique de distribution et d'ordonnancement pour une architecture complètement connectée, qui se base sur la réplication active des tâches et passive des communications. Ils supposent qu'au plus un processeur peut être défaillant, donc chaque tâche est répliquée activement sur deux processeurs distincts.

Les auteurs dans [42] proposent une heuristique d'ordonnancement tolérante aux fautes des processeurs. Elle est basée sur la redondance active des opérations et des communications, où les tâches composant l'algorithme sont répliquées activement sur au moins $N+1$ processeurs distincts afin de tolérer N fautes de processeurs. En plus, chaque opération reçoit ses données d'entrées via $N+1$ routes disjointes.

4.2.1.4 État de l'art de la tolérance des fautes des communications

Dans [41], les auteurs proposent une technique basée sur la redondance active des messages, leur technique permet de réduire le coût de la retransmission des messages en présence de pannes dans un système distribué temps-réel. Pour réduire le coût de la retransmission des messages, chaque message est envoyé en parallèle via au moins deux routes disjointes. Le nombre des routes disjointes est défini par la criticité du message et le nombre de liens de communication dans chaque route.

Dans [60] un nouveau mécanisme de redondance active des communications est proposé par les auteurs, tout comme dans [61], le mécanisme se base sur le codage de messages en FEC (Forward Error Correction) et utilise les routes disjointes pour tolérer plusieurs pannes des liens de communication. La technique proposée opère en deux phases, dans un premier temps elle code chaque message M par FEC en plusieurs paquets avec redondance, ensuite dans un deuxième temps, chaque paquet d'un message est envoyé vers sa destination via des routes disjointes. Si une partie des données d'un message est perdue à cause d'une panne des liens de communication, le message peut être reconstruit grâce à des informations redondantes envoyées via les routes disjointes.

Dans [62], les auteurs présentent une méthode basée sur la redondance active des messages pour réduire le temps de transmission/retransmission des messages. La méthode proposée est adaptée aux messages courts. c'est à l'opposé de la méthode proposée dans [41], où le nombre des routes disjointes est différent d'un message à un autre. Cette même méthode est utilisée dans [63], mais pour tolérer M pannes de bus dans une architecture multi-bus.

Dans [64], les auteurs utilisent aussi le même principe des routes disjointes pour tolérer les pannes de plusieurs processeurs/liens dans un réseau en étoile, et ont adopté le même modèle de fautes utilisé dans [64] [63] [41].

Les auteurs de [65] ont proposé une heuristique basée sur la redondance active pour tolérer uniquement les fautes permanentes des processeurs et de média de communication. L'heuristique commance par la génération d'un ordonnancement des tâches d'un algorithme sur une architecture ; Ensuite, pour chaque configuration de fautes, elle génère tout d'abord une architecture réduite, puis, elle génère un ordonnancement des tâches du même algorithme sur la nouvelle architecture réduite, qui est le résultat de l'exclusion des composants de la configuration de fautes dans l'architecture globale. A la fin, l'heuristique génère un ordonnancement globale qui est la combinaison de toutes les ordonnancements obtenue précédemment.

Dans [66], les fautes des bus de communication sont tolérés en utilisant le protocole de communication TDMA (Time Division Multiple Access).

Enfin, les solutions proposées dans [41] [60] [61] [62] [63] [64] ne tolèrent que des fautes de processeurs et de média de communication composant une route de communication, et la solution proposée dans [65] est dédiée à certaines configurations de fautes.

4.2.2 Les algorithmes d'ordonnancement basés sur la redondance passive

4.2.2.1 Redondance passive

La redondance passive [36] [37] [38] [39] [40] est basée aussi sur la réplication des tâches de l'algorithme. A la différence de la redondance active une seule copie (la copie primaire) de chaque tâche est exécutée, alors que les autres répliques (copies de sauvegardes), ne seront exécutées que si une faute provoque une erreur puis une défaillance du processeur ou média de communication implantant la réplique primaire. Pour tolérer au plus K fautes permanentes de processeurs ou de média de communication, chaque tâche ou dépendance de données est répliqué en une réplique primaire et en K répliques de sauvegardes, d'où $N = K + 1$.

La copie primaire met à jour régulièrement une copie de son état interne, qui constitue ainsi un point de reprise. Lorsque la copie primaire estdéfaillante, une des copies de secours est élue pour prendre sa place.

4.2.2.2 Exemple

Tolérance aux fautes des processeurs Dans la Figure 4.4, la tâche A (resp. B) est répliquée en deux exemplaires A_p et A_s (resp. B_p et B_s), qui sont implantés sur deux processeurs distincts P_1 et P_2 (resp. P_2 et P_5) afin de tolérer une panne de processeur. Les répliques A_s et B_s sont des répliques de sauvegarde qui ne seront exécutées qu'en cas de défaillance, alors que les répliques primaires A_p et B_p sont toujours exécutées.

Tolérance aux fautes des média de communication La perte d'un message dans le cas de la redondance passive des communications, due aux fautes des médias de communication, peut être tolérée par la retransmission de ce message. Cette technique de redondance passive nécessite un mécanisme particulier de détection de pannes.

Dans l'exemple de la figure 4.1, afin de tolérer une panne de lien de communication, le processeur P_1, implantant la copie primaire A_s, envoi au processeur P_2, implantant la copie primaire B_s, les données *data* via une seule route

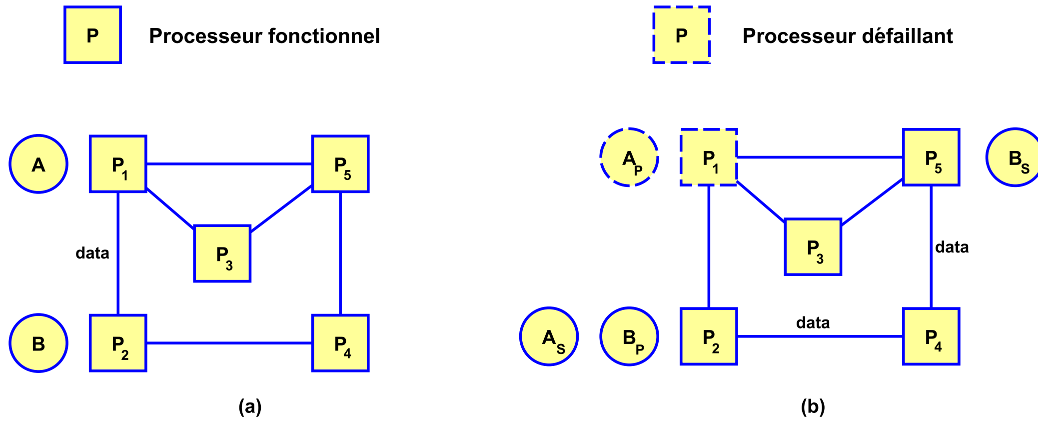


FIGURE 4.4 – Tolérance aux fautes des processeurs par redondance passive

$Route_1 = (l_{P_1,P_2}, l_{P_2,P_4})$. Dans la Figure 4.5(b), si les média de communication de la route $Route_1 = (l_{P_1,P_2}, l_{P_2,P_4})$ est défaillant, alors il faut détecter sur le processeur P_1 cette défaillance et envoyer les données *data* à P_4 via une autre route $Route_2 = (l_{P_1,P_5}, l_{P_5,P_4})$, même si cette nouvelle route est aussi sujet d'une autre faute (Figure 4.5(c)), une troisième route est activée : $Route_3 = (l_{P_1,P_3}, l_{P_3,P_5} et, l_{P_5,P_4})$

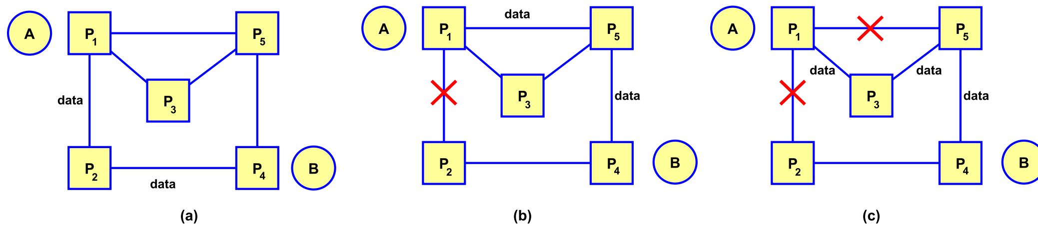


FIGURE 4.5 – Tolérance aux fautes des média de communication par redondance passive

4.2.2.3 État d'art de la tolérance des fautes des processeurs

Dans [38] une nouvelle heuristique d'ordonnancement tolérante aux fautes est proposée, elle se base sur la redondance passive des tâches. Le modèle de fautes adopté suppose que les processeurs sont de type silence sur défaillances, et que la faute d'un processeur peut être détectée par les autres processeurs. Pour tolérer une seule faute d'un processeur, on réplique chaque tâche en deux copies identiques, une copie primaire et une copie de sauvegarde. Ces deux copies sont allouées temporellement de façon séquentielle sur deux processeurs distincts, et seulement la copie

primaire est exécutée. En cas de faute d'un processeur, la copie de sauvegarde, de chaque copie primaire implantée sur ce processeur en question, est exécutée pour tolérer la faute du primaire.

Sur le même principe que la méthode proposée dans [38], les auteurs de [67] proposent une nouvelle heuristique d'ordonnancement tolérante aux fautes basée sur la redondance passive des tâches, la seule différence est que dans leur modèles, les tâches sont dépendantes, et qu'ils supposent que plusieurs processeurs peuvent défaillir.

4.2.2.4 État d'art de la tolérance des fautes des communications

Dans [68] une méthode basée sur la redondance passive des messages est développée pour tolérer une faute arbitraire d'un lien de communication. Chaque message est répliqué en deux copies : une primaire et l'autre de sauvegarde, seule la copie primaire de chaque message est envoyée tandis que la copie de sauvegarde est en attente au cas où la copie primaire serait défaillante. La méthode proposée tolère une seule faute de lien de communication.

Ce même principe est utilisé dans [69], sauf que cette méthode tolère plusieurs pannes des liens de communication. Comme les routes primaires et les routes de sauvegardes pour chaque message sont pré-calculées, la méthode proposée utilise le multiplexage des routes de sauvegarde de plusieurs messages pour réduire le sur-coût en communications en présence de fautes.

Il reste à noter que les deux solutions proposées dans [68] et [69] ne tolèrent que les fautes des médias de communication et des processeurs intermédiaire (utilisés pour le routage des communications), mais pas les fautes des processeurs émetteurs et récepteurs des communications.

Dans [70] les auteurs traitent le problème du délai de recouvrement de panne d'un seul lien de communication et se basent sur une technique de redondance passive de messages. La méthode utilise une route primaire, composée de plusieurs sous-routes primaires, et une route de sauvegarde, composée de plusieurs sous-routes de sauvegarde, où chaque sous-route de sauvegarde consiste à recouvrir une panne d'un lien dans une sous-route primaire. Seule la copie primaire de chaque message est envoyée tandis que les autres sous-copies de sauvegardes sont en attente au cas où la copie primaire serait défaillante dans une sous-route primaire.

Dans [71], les auteurs proposent de gérer le problème de la tolérance aux pannes par l'intermédiare des algorithmes de routages.

Dans [72], les auteurs proposent un algorithme qui permet de sélectionner le meilleur processeur accélérant le recouvrement de pannes et minimisant l'utilisation des ressources. Avec la même philosophie, une stratégie de recouvrement de pannes basée sur des routes non complètement disjointes est proposée dans [73].

Enfin, la reconstruction de la route à chaque panne est utilisée comme une nouvelle méthode dans [74] pour tolèrer plusieurs pannes des liens de communication.

4.2.3 Les algorithmes d'ordonnancement basés sur la redondance hybride

4.2.3.1 Redondance hybride

La redondance hybride [44] [45] [16] est une combinaison de la redondance active et passive. Par exemple, pour tolérer une fautes permanente d'un processeur ou d'un médium de communication, on utilise la redondance active pour les tâches de l'algorithme et la redondance passive pour les communications.

Dans la figure 4.6(a), les deux tâches A et B sont répliquées activement, tandis que les communications ont des répliques passives. La tâche A (resp. B) du modèle de tâches est répliqué en deux répliques actives A_1, A_2 (resp. B_1, B_2), qui sont implantées sur deux processeurs distincts P_1 et P_2 (resp. P_2 et P_4) afin de tolérer une faute permanente d'un seul processeur. Puisque les communications ont des répliques passives, le message *data* ne sera envoyé par la réplique A_1 via la route $Route = (l_{P_1,P_5})$.

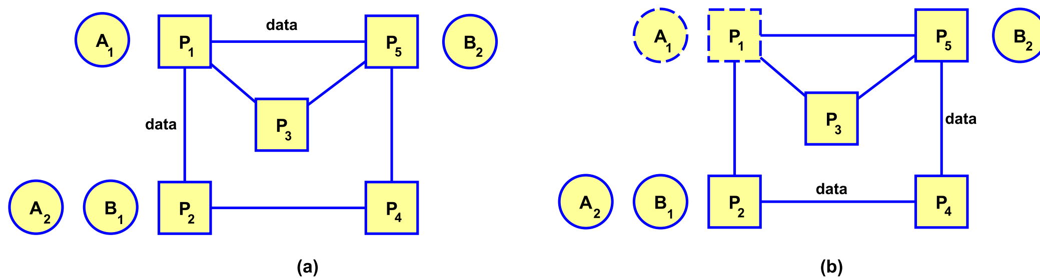


FIGURE 4.6 – Tolérance aux fautes de processeurs et des médias de communication par redondance

Dans la figure 4.6(b), si le média de communication l_{P_1,P_5} est défaillant, alors il faut que le processeur P_2 détecte cette défaillance et renvoie la donnée *data* à P_5 via une nouvelle route $Route_2 = (l_{P_2,P_4}, l_{P_4,P_5})$

4.2.3.2 État d'art de la tolérance des fautes des processeurs

L'article [16] propose une nouvelle approche de tolérance aux fautes des processeurs. Le modèle de faute utilisé suppose que les processeurs sont de type silence sur défaillance, et que les fautes peuvent être permanentes ou transitoires. Un outil,

dit *Hydra*, qui implante un algorithme de transformation de graphe et une heuristique d'ordonnancement tolérante aux fautes est proposé. L'algorithme permet la transformation de graphe en transformant chaque tâche non répliquée en une nouvelle tâche avec redondance active, passive, et/ou hybride. Ensuite, il génère un ordonnancement de la nouvelle architecture sur les processeurs de l'architecture.

Les auteurs dans [75], proposent une nouvelle méthode basée sur la reprise transparente à grain fin, où la propriété de la transparence peut être appliquée de manière sélective sur les processus et les médias de communication.

4.2.3.3 État de l'art de la tolérance des fautes des communications

La réplication active des tâches et passive des communications est à la base de la solution proposée dans [45]. Les auteurs supposent que l'architecture est complètement connectée et qu'au plus une faute permanente d'un processeur peut affecter le système, ensuite ils répliquent activement chaque tâche de l'algorithme sur deux processeurs distincts.

Une méthode hybride des communications est proposée dans [76]. Elle définie une date d'échéance et un niveau de tolérance pour chaque message, ensuite chaque message est répliqué activement ou passivement suivant la date d'échéance et le niveau de tolérance du message. Le seul inconvénient de cette technique est qu'elle ne tolère que les fautes transitoire des routes de communication.

Dans [77], les auteurs traitent le problème de l'ordonnancement des tâches dans l'objectif de garder les débits dans des intervalles raisonnable en débit des fautes des processeurs et des médias de communication, quelque soit leurs natures, ils traitent même les fautes d'origine logiciel.

4.3 Fiabilité

Évaluer la fiabilité d'un système est l'une des préoccupations principales des concepteurs des systèmes sûrs de fonctionnement. Dans ce but, plusieurs approches ont été proposées dans la littérature. Elles diffèrent sur plusieurs critères, tel que le modèle de fautes pris en compte et la méthode de calcul utilisée pour évaluer la fiabilité. Dans le cadre de notre travail, nous ne nous intéressons qu'aux travaux basés sur la théorie de l'ordonnancement. Suivant le nombre d'objectifs visés par leurs algorithmes d'ordonnancement, on peut classer ces approches en deux classes : approche uni-objectif et approche multi-objectifs.

4.3.1 Approche uni-objectif

Les approches uni-objectif [78] [79] [80] [81] [82] [83] [84] [85] s'intéressent exclusivement au problème de la fiabilité des systèmes. Leurs algorithmes d'ordonnance-

ment ne vise qu'à maximiser la fiabilité de l'allocation des tâches de l'algorithme sur les composants de l'architecture. Donc, le seul objectif pris en compte est l'objectif de fiabilité.

Deux algorithmes optimal et sub-optimal pour le problème d'ordonnancement temps réel sont proposés dans [86]. Leurs modèles supposent que les fautes sont des fautes permanentes des processeurs et des médias de communication. Ils supposent aussi que la défaillance des composants matériels (processeurs et média de communication) suit une loi exponentielle à taux de défaillance constant, la mesure de fiabilité calculée par cette heuristique représente la probabilité que chaque tâche fonctionne correctement durant son exécution.

Pour les systèmes hétérogènes, les auteurs de [79] proposent deux heuristiques d'ordonnancement temps réel MCMS (Minimum Cost Match Schedule) et PRMS (Progressive Reliability Maximization Schedule). Ils utilisent le même modèle de fautes que [86], et ils ont pour objectif la génération des ordonnancements les plus fiables que possible des tâches de l'algorithme sur les composants matériels de l'architecture distribuée, tout en respectant des contraintes temps réel.

4.3.2 Approche multi-objectifs

À la différence des approches uni-objectif, les approches multi-objectifs [87] [67] [88] [89] s'intéressent au problème de la fiabilité et aussi au problème de la tolérance aux fautes. Donc, ces approches visent deux objectifs qui sont la maximisation de la fiabilité et la tolérance aux fautes matérielles.

Les deux objectifs visés par l'heuristique d'ordonnancement temps réel proposée dans [67] sont : La génération d'une allocation tolérante à une seule faute permanente d'un processeur, et la maximisation de la fiabilité de cette allocation. L'heuristique d'ordonnancement prposée suppose que le modèle de la défaillance des processeurs suit une loi exponentielle à taux de défaillance constant et la tolérance aux fautes des processeurs est obtenue par l'utilisation de la redondance passive des tâches.

Quatre autres algorithmes hors-ligne d'ordonnancement pour la génération des ordonnancement fiables et tolérantes aux fautes sont proposés dans [89]. Ces quatre algorithmes adoptes le même modèle de fautes que dans [67] et ils supposent en plus que chaque composant matériel défaillant est remplacée immédiatement après sa défaillance.

L'heuristique proposée dans [90] est une heuristique glouton de type ordonnancement de liste, elle se base sur une fonction de coût à deux objectifs (temps réel et fiabilité) pour trier les tâches à ordonnancer. Au contraire de [89], la mesure de fiabilité calculée par cette heuristique représente la probabilité que chaque tâche fonctionne correctement durant uniquement son exécution, ce qui ne nécessite donc pas d'estimer la durée totale de la mission du système.

RDRTPSA (Reliability Driven Real Time Periodic Scheduling Algorithms) est présentée dans [87]. C'est une heuristique qui utilise le même modèle de fautes et la même fonction d'évaluation de la fiabilité que [89], à l'exception que les média de communication sont supposées sans fautes et que la mesure de fiabilité calculée par leur fonction d'évaluation représente la probabilité que le système fonctionne correctement durant un cycle d'exécution du système. Enfin, la tolérance aux fautes des processeurs est obtenue par l'utilisation de la redondance passive des composants logiciels.

4.4 Conclusion

Nous avons présenté dans ce chapitre différents algorithmes d'ordonnancement, existants dans la littérature, pour générer des ordonnancements tolérants aux fautes et fiables. Les algorithmes que nous avons présentés sont tous basés sur la redondance logicielle des composants matériels avec ces trois types : active, passive et hybride. Enfin nous avons présenté la fiabilité d'un système comme étant l'une des préoccupations les plus importantes des concepteurs des systèmes sûrs de fonctionnement.

Deuxième partie

Approches Proposées

Chapitre 5

Approche d'ordonnancement fiable de communication basée sur la fragmentation variable de données

Sommaire

5.1 Introduction 60
5.2 Présentation du problème 61
5.3 RBF-VDF : Un algorithme d'ordonnancement tolérant aux fautes des bus de communications basé sur la fragmentation variable des données. 63
5.3.1 Taux Global de défaillance du système (TGDS) 63
5.3.2 Tolérance aux fautes des processeurs par la redondance PP(Passive-Passive) : 64
5.3.3 Tolérance aux fautes des bus de communication par redondance Passive : 69
5.3.4 Fragmentation variable des données : 70
5.3.5 Ordonnancement tolérant aux fautes des processeurs et des bus de communication 77
5.4 Réduction de la Consommation d'énergie 80
5.5 Exemple et Simulations 81
5.5.1 Exemple 81
5.5.2 Simulations 83
5.6 Conclusion 84

Dans ce chapitre nous présentons une nouvelle technique d'ordonnancement tolérant aux fautes pour les systèmes temps réel embarqués, l'approche d'ordonnancement que nous proposons est dédiée aux architectures hétérogènes multi-bus, qui prennent en entrée une description donnée du système et un ensemble d'hypothèses de fautes.

Notre solution est basée sur la fragmentation variable des données [52] [51], la redondance passive des communications et la redondance PP (passive-passive) des

tâches, ce qui permet une détection/retransmission rapide des fautes et donc une utilisation efficace de l'architecture multi-bus.

Au cœur de notre solution, une heuristique de liste d'ordonnancement basée sur le *TGDS*[1] [91] : Taux Global de défaillance du système et la fragmentation variable des données. La taille de chaque fragment de données dépend du *TGDS* et donc indirectement des taux de défaillance de chaque bus λ_{B_i}, ce qui permet une communication tolérante aux fautes pour un système à fiabilité maximalisée.

5.1 Introduction

Les techniques de tolérance aux fautes sont nécessaires pour s'assurer que le système continue à fournir un service correct en dépit de fautes [92] [93]. Une faute peut affecter soit le matériel soit le logiciel du système, nous avons choisi de se concentrer sur les pannes matérielles, plus particulièrement, nous considérons les pannes des bus de communications. Un bus est une connexion multi-point, caractérisée par un support physique qui connecte tous les processeurs de l'architecture. Comme nous ciblons les systèmes embarqués avec des ressources limitées (pour des raisons de poids, encombrement, consommation d'énergie, ou les contraintes de prix), nous nous intéressons uniquement à des solutions de redondance logicielle.

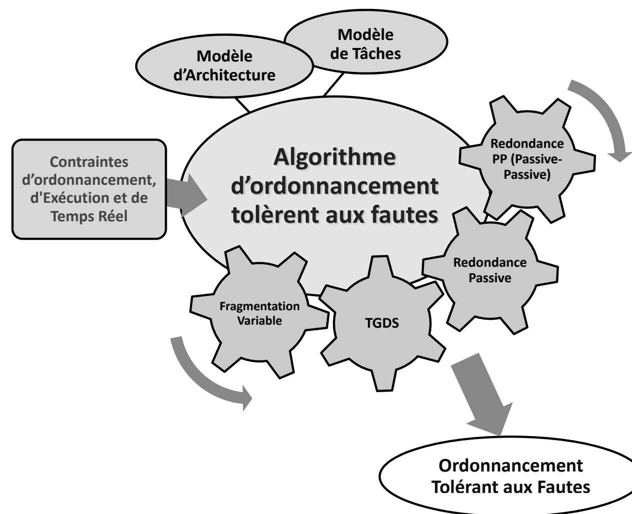


FIGURE 5.1 – Approche Proposée.

L'approche que nous proposons (Figure 5.1) est basée sur la fragmentation variable des données, où la taille de chaque fragment est calculer en fonction des états

1. *TGDS* = Taux Global de défaillance du système : *Global System Failure Rate (GSFR).*

des bus a l'aide des deux fonctions de coût $TGDS$ et σ (schedule pressure), elle permet de tolérer une ou plusieurs fautes temporelles des processeurs et des bus de communication. Il s'agit de résoudre le problème de la recherche d'un ordonnancement des composants logiciels du modèle de tâches G_{Task} sur les composants matériels du modèle d'architecture G_{Arch}, qui tolère Nf_{Proc} [2] fautes de processeurs et Nf_{Bus} [3] fautes de bus de communication, tout en minimisant la longueur de l'ordonnancement afin de satisfaire la contrainte temps réel C_{Rt}(La prédictibilité consiste à vérifier hors-ligne que les contraintes temporelles sont respectées en absence et en présence de défaillances.)

Afin de bien présenter notre solution, voici tout d'abord le modèle de fautes que nous considérons dans ce chapitre.

5.2 Présentation du problème

Le but de ce chapitre est de résoudre le problème de la recherche d'un ordonnancement des composants logiciels du modèle de tâches G_{Task} sur les composants matériels du modèle d'architecture G_{Arch} qui doit tolérer des fautes matérielles des processeurs et des bus de communication, tout en minimisant la longueur de l'ordonnancement dans le but de satisfaire la contrainte temps réel C_{Rt} en absence et en présence de défaillances.

Nous ne nous intéressons dans ce travail qu'aux techniques de tolérance aux fautes matérielles basées sur des solutions logicielles, plus spécifiquement aux fautes des bus de communication. Notre solution est liée aux hypothèses de défaillances définies par le modèle de fautes suivant (c'est le raffinement du modèle présenté dans 2.5.3), dans lequel nous supposons que :

1. L'algorithme est fiable et sans fautes (c'est à dire que le logiciel à été validé par des techniques de tolérance aux fautes logiciels [94] [95] : model-cheking, démonstration de théorèmes, traitement des exceptions, ...)
2. Les fautes matérielles sont des fautes des processeurs et des fautes des bus de communication. La défaillance d'un bus de communication peut être partielle ou complète. La défaillance complète d'un bus est la conséquence de la défaillance de tous ses communications, tandis que la défaillance partielles est la conséquence d'une ou plusieurs défaillance des ses communications à condition qu'au moins deux de ses communications restent actifs.
3. Le système accepte au plus Nf_{Proc} fautes des processeurs et Nf_{Bus} fautes de bus de communications dans un cycle d'exécution de son modèle de tâches sur son architecture, et que l'architecture comprend plus de Nf_{Bus} bus.

2. Nf_{Proc}= Nombre de fautes de processeurs tolérées.
3. Nf_{Bus}= Nombre de fautes de bus tolérées.

4. Nous considérons seulement les fautes de bus transitoires [96], qui persistent pendant une durée limitée, ce type de fautes est beaucoup plus fréquent que d'autres. Les fautes permanentes sont un cas particulier de fautes transitoires.
5. Les processeurs et les bus de communication sont à défaillances temporelles, c'est-à-dire que les valeurs calculées par les opérateurs de calcul sont soit correctes et délivrées à temps, soit correctes et délivrées trop tôt, trop tard ou infiniment tard.

Enfin, notre modèle de fautes en ce qui concerne les défaillances temporelles couvre les deux hypothèses de défaillances les plus utilisées, a savoir : hypothèse de défaillances par omission et hypothèse de silence sur défaillances [97].

Le problème de la recherche d'un ordonnancement peut être formalisé comme suit :

Les Données du problème :

- Une architecture matérielle hétérogène G_{Arch} composée d'un ensemble E_{Proc} d'opérateurs de calcul (processeurs) et d'un ensemble E_{Bus} de bus de communication :
$$E_{Proc} = \{\ldots, P_i, \ldots\}, E_{Bus} = \{\ldots, B_j, \ldots\}$$
- Un modèle de tâche G_{Task} composé d'un ensemble E_{Task} d'opérations (tâches) et d'un ensemble E_{Dd} de dépendances de données :
$$E_{Task} = \{\ldots, t_i, \ldots, t_j, \ldots\}, E_{Dd} = \{\ldots, (t_i \to t_j), \ldots\}$$
- Des caractéristiques d'exécution Ext des composants de G_{Task} sur les composants de G_{Arch},
- Un ensemble de contraintes matérielles C_{Had},
- Une contrainte temps réel C_{Rt},
- Un critère de minimisation de la longueur de l'ordonnancement,
- Un nombre Nf_{Proc} de fautes de processeurs et un nombre Nf_{Bus} de fautes de bus de communication qui peuvent causer la défaillance du système,

L'objectif : C'est la définition d'une application App dont le rôle est d'ordonnancer chaque tâche et chaque dépendance de données de G_{Task} sur les processeurs et les bus de G_{Arch} l'ordonnancement est réaliser par l'affectation d'un ordre d'exécution ORD_k d'une tâche sur un processeur, ou d'une dépendance de donnée sur un bus.

$$\begin{array}{llll} App : & G_{Task} & \longrightarrow & G_{Arch} \\ & G_{Task\,i} & \longmapsto & A(G_{Task\,i}) = (G_{Arch\,j}, ORD_k) \end{array}$$

App respecte C_{Had} , minimise la longueur de l'ordonnancement afin de satisfaire C_{Rt} et tolérer $Nf_{Proc}+Nf_{Bus}$ fautes de processeur et de bus de communication.

5.3 RBF-VDF : Un algorithme d'ordonnancement tolérant aux fautes des bus de communications basé sur la fragmentation variable des données.

Le problème que nous venons d'exposer est un problème NP-difficile. Pour le résoudre en un temps polynomial on propose une nouvelle technique basée sur une heuristique dite *RBF-VDF*[4] qui essaye de trouver une solution proche de la solution optimale, rappelons que notre objectif n'est pas nécessairement d'avoir un ordonnancement de longueur minimale mais plutôt un ordonnancement qui respecte la contrainte temps réel C_{Rt}.

RBF-VDF est basée sur la fragmentation variable des données. son but est de maximiser la fiabilité du système tout en minimisent la longueur de l'ordonnancement généré et cela dans les deux cas : avec ou sans défaillances.

Notre problème peut être résolu par l'utilisation des techniques logicielles, et plus spécifiquement les techniques de redondance puisque nous visons les systèmes embarqués. L'approche que nous proposons permet d'avoir une grande fiabilité et une tolérance aux pannes en exploitant les quatre techniques suivantes :

❶ TGDS.

❷ Redondance PP (Passive-Passive) des opérations.

❸ Redondance Passive des communications.

❹ Fragmentation variable des données.

Afin de bien expliquer cette phase de transformation du modèle de tâches pour tolérer les fautes des processeurs et des bus de communication, nous avons choisi de présenter tout d'abord l'approche de la **redondance passive passive**, utilisée pour tolérer uniquement les fautes des processeurs. Parmi les points les plus importants de cette étape, il y a le calcul du TGDS et de la pression de l'ordonnancement pour chaque tâche à ordonnancer, ce qui optimise le choix du processeur. Puis nous présentons l'approche de la redondance passive des communications avec fragmentation variable des données pour tolérer uniquement les fautes des bus de communication et enfin l'approche proposée pour tolérer les deux.

5.3.1 Taux Global de défaillance du système (TGDS)

Le Taux Global de Défaillance du Système (TGDS) (en anglais : Global System Failure Rate : GSFR) [91] : est le taux de défaillance par unité de temps de l'ordonnancement multiprocesseur obtenu. l'utilisation du TGDS est très satisfaisante dans le domaine des ordonnancements à exécutions périodiques, et c'est le cas de la

4. RBF-VDF = *R*eliable *B*us *F*ault- tolerant based *V*ariable *D*ata *F*ragmentation

plupart des systèmes temps réel embarqués, qui sont des systèmes périodiquement échantillonnés .

Dans de tels systèmes, l'application directe du modèle de fiabilité exponentielle engendre une fiabilité de très faible niveau en raison des délais d'exécution très longs (la même remarque s'applique également à des ordonnancement très longs). Par conséquent, il faut calculer à l'avance la fiabilité souhaitée dans une seule itération par rapport à la fiabilité globale du système au cours de toute sa mission ; mais ce calcul dépend de la durée totale de la mission (qui est connue) et de la durée d'une seule itération (qui peut ne pas être connue, car elle dépend de la durée du programme en cours d'exécution). En revanche, le TGDS reste constant au cours de la totalité de la durée du système : le TGDS par définition, au cours d'une seule itération est identique au TGDS pendant la totalité de l'ordonnancement.

L'utilisation du TGDS par notre approche, permet d'avoir un contrôle totale sur l'ordonnancement de chaque tâche et de chaque fragment de données et cela du début jusqu'à la fin de l'ordonnancement. Dans [91], le TGDS de l'ordonnancement d'une tâche t_i, noté $\Lambda(S_n)$, est calculé par l'équation :

$$\Lambda(S_n) = \frac{-\log \prod_i e^{-\lambda_k exe(t_i,p_k)+\sum_k \sum_j \lambda_c exe(dpd_j^k,b_c)}}{\sum_i^j exe(t_i,p_j) + \sum_k^m exe(dpd_k,b_m)} \tag{5.1}$$

Où S_n est l'ordonnancement statique du modèle de tâches à l'étape n, et la valeur de $\sum_i^j exe(t_i,p_j) + \sum_k^m exe(dpd_k,b_m)$ représente l'utilisation totale des ressources matérielles.

$\prod_i e^{-\lambda_k exe(t_i,p_k)+\sum_k \sum_j \lambda_c exe(dpd_j^k,b_c)}$ est calculée pour chaque tâche t_i sur chaque processeur P_j, cette valeur définit la fiabilité suivante :

$$R(S_n) = \prod_i e^{-\lambda_k exe(t_i,p_k)+\sum_k \sum_j \lambda_c exe(dpd_j^k,b_c)} \tag{5.2}$$

5.3.2 Tolérance aux fautes des processeurs par la redondance PP(Passive-Passive) :

Nous supposons ici que les communications sont fiables, c'est-à-dire que le système est sans fautes des bus de communication. Alors, notre objectif est de générer un ordonnancement d'un modèle de tâches G_{Task} sur un modèle d'architecture G_{Arch} tolérant uniquement aux fautes des processeurs. pour cela on doit transformer le graphe du modèle de tâches G_{Task} en un nouveau graphe ${G_{Task}}^*$ avec redondances passives pour tolérer Nf_{Proc} fautes de processeurs. Cette transformation se fait en deux étapes.

Dans un premier temps, pour tolérer au plus Nf_{Proc} fautes des processeurs, il est nécessaire de placer chaque tâche ou opération de G_{Task} sur $Nf_{Proc}+1$ processeurs distincts de G_{Arch}. Donc :

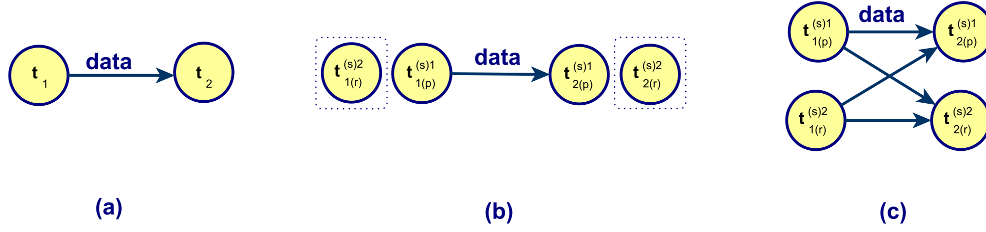


FIGURE 5.2 – Transformation du graphe du modèle de tâches pour $Nf_{Proc} = 1$.

1. Nous répliquons chaque tâche t_i du modèle G_{Task} en Nf_{Proc}+1 copies de sauvegarde exclusives $t_i^{(s)1}$, $t_i^{(s)2}$, . . . , $t_i^{(s)Nf_{Proc}+1}$; l'ensemble de ces répliques noté $Rep(t_i)$ est ordonnancé sur Nf_{Proc}+1 différents processeurs, il comprend une copie de sauvegarde primaire $t_{i(p)}^{(s)x}$ et Nf_{Proc} copies de sauvegarde répliquées $t_{i(r)}^{(s)x}$ (lors de l'ordonnancement de ces copies, plus d'importance est donnée à copie primaire, celle ci est généralement ordonnancée sur le meilleur processeur, celui qui minimise $\sigma^{(n)}$ et la valeur de $\Lambda(S_n)$), elles sont ensuite ordonnées en ordre croissant suivant leurs date de fin d'exécution $Et(t_i^{(s)x})$.

   Par exemple, dans la Figure 5.2(b), afin de tolérer une seule faute d'un processeur (i.e., Nf_{Proc}=1), les deux tâches t_1 et t_2, de la Figure 5.2(a), sont répliquées dans $G_{Task}{}^*$ en deux copies chacune.

2. Ensuite, puisque les tâches répliquées $Rep(t_i)$ d'une tâche t_i doivent envoyer en parallèle leurs données de sortie ($t_i^{(s)x} \rightarrow t_j^{(s)k}$) à chaque tâche successeur $t_j^{(s)k}$, celle ci doit tout d'abord vérifier la validité temporelle de chaque copie, ensuite sélectionner parmi ces répliques la meilleure réplique.

Dans un deuxième temps, chaque copie de sauvegarde $t_i^{(s)x}$ de la tâche t_i doit envoyer à toutes les copies de sauvegardes $Rep(t_j)$ de t_j, successeur de t_i, les données ($t_i \rightarrow t_j$) par l'intermédiaire des bus de communications. Donc : La dépendance de données ($data = t_i \rightarrow t_j$) de G_{Task} est répliquée dans $G_{Task}{}^*$ entre chaque tâche de $Rep(t_i)$ et chaque tâche de $Rep(t_j)$. Ce qui ajoute à $G_{Task}{}^*$ un ensemble de Nf_{Proc}+1 dépendances exclusives de données pour chaque opération de $Rep(t_j)$, le nombre total des dépendances de données ajoutées est de : $(Nf_{Bus} + 1)^2 - 1$.

Avec la redondance passive une seule copie, appelée copie primaire, de chaque tâche est exécutée, et les autres copies, appelées copies de sauvegardes répliquées, ne seront exécutées que si une faute provoque une erreur, puis une défaillance du composant matériel implantant la copie primaire. Cela nécessite un mécanisme spécial de détection d'erreurs. on parle du schéma maître-esclave (primary-backup scheme)

Par exemple, dans la Figure 5.3, pour tolérer une seule faute de processeur ($Nf_{Proc} = 1$), les deux tâches t_1 et t_2 de la Figure 5.2(a), sont répliquées en deux copies, une copie primaire $t_{1(p)}^{(s)1}$ et une copie de sauvegarde répliquée $t_{1(r)}^{(s)2}$ pour t_1,

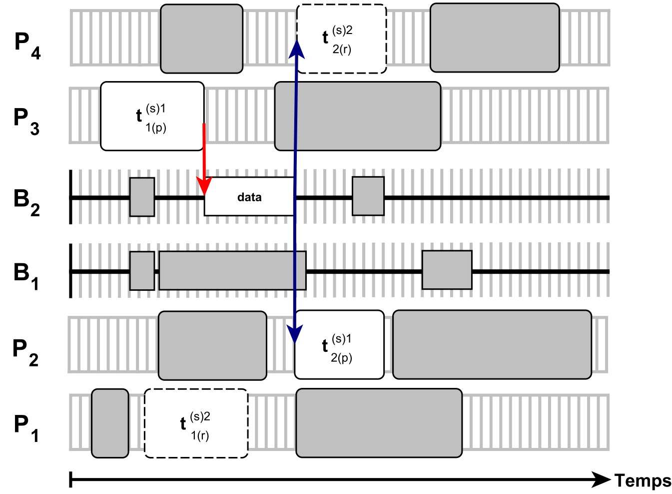


FIGURE 5.3 – Redondance passive des tâches t_1 et t_2 pour $Nf_{Proc} = 1$.

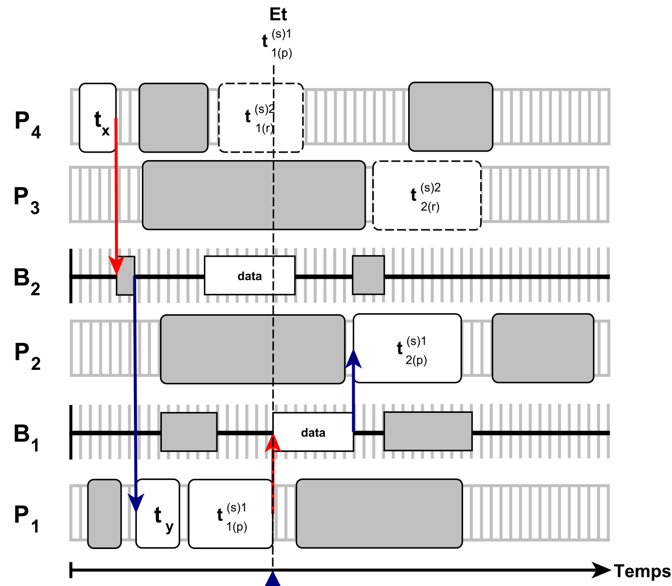


FIGURE 5.4 – Partie de l'ordonnancement pour les tâches t_1 et t_2 (Nf_{Proc}=1).

$t_{2(p)}^{(s)1}$ et $t_{2(r)}^{(s)2}$ pour t_2, une partie de l'ordonnancement est représentée par la Figure 5.4.

Le problème principale de cette technique est que parfois certaines des copies de sauvegarde d'une tâche donnée ont un temps de fin d'exécution Et plus proche que la copie primaire de cette tâche. l'approche de redondance passive ne prévoie le passage à une copie de sauvegarde répliquée qu'en cas de faute de la copie de sauvegarde primaire, la Figure 5.5 présente un cas d'exécution hors ligne d'un ordonnancement

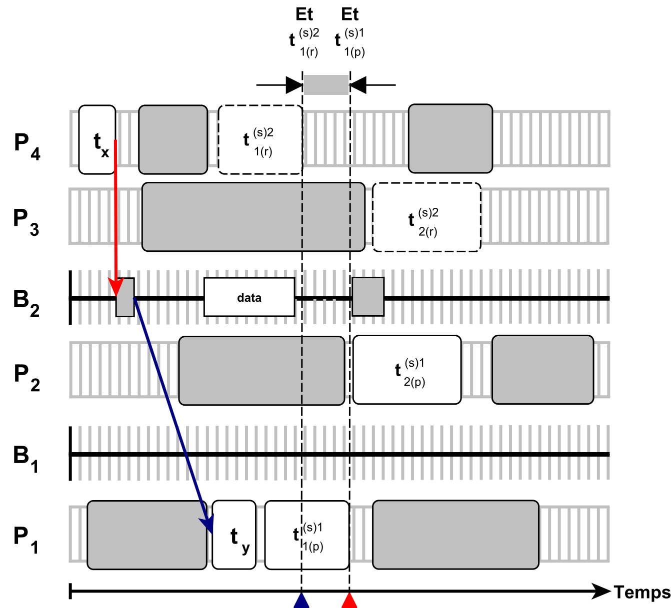


FIGURE 5.5 – Décalage de la copie primaire $t_{1(p)}^{(s)1}$.

ou les tâches de sauvegarde risquent de finir plutôt que la copie primaire.

Dans cet exemple, la tâche $t_{1(p)}^{(s)1}$ peut avoir du retard à cause d'un retard accumulé par une autre tâche t_x due par exemple à des données non reçues à temps, le résultat est que le temps de fin d'exécution de $t_{1(r)}^{(s)2}$ est plus proche que le temps de fin d'exécution de $t_{1(p)}^{(s)1}$ $(Et(t_{1(r)}^{(s)2}) \leq \; Et(t_{1(p)}^{(s)1}))$, aussi le temps de fin d'exécution de $t_{1(r)}^{(s)3}$ est plus proche que le temps de fin d'exécution $t_{1(p)}^{(s)1}$ $(Et(t_{1(r)}^{(s)3}) \leq \; Et(t_{1(p)}^{(s)1})$.

Notre solution à ce problème est l'approche **PP(Passive-Passive)** avec la quelle on peut attribuer dynamiquement le label **primaire** à toute copie d'une tâche qui à le temps de fin d'exécution le plus proche, on parle d'un mécanisme de **Préemption** qui permet de définir une copie de sauvegarde répliquée comme étant une copie de sauvegarde primaire au détriment de la copie de sauvegarde primaire d'origine, qui devient une simple copie de sauvegarde répliquée. Dans l'exemple de la Figure 5.6, la copie de sauvegarde répliquée $t_{1(r)}^{(s)2}$ devient la copie de sauvegarde primaire $t_{1(p)}^{(s)2}$ de la tâche t_1 et l'encienne copie de sauvegarde primaire $t_{1(p)}^{(s)1}$ redevient une copie de sauvegarde répliquée $t_{1(r)}^{(s)2}$.

Afin de rendre la phase de la tolérance aux fautes transparente aux utilisateurs, le mécanisme de préemption qui permet l'affectation du label **primaire** à l'une des copies de sauvegarde d'une tâche est réalisé par un nouveau composant logiciel (Voir Algorithme 1), dont la mission est : avant d'exécuter l'occurrence d'une tâche donnée, de vérifier les dates de fin d'exécution de chacun de ces copies de sauvegarde (date de fin d'exécution = date prévu de début d'exécution + durée d'exécution de

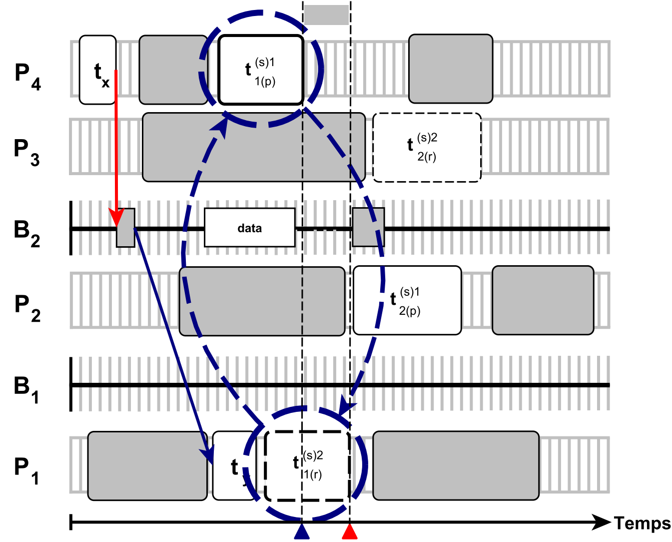


FIGURE 5.6 – Principe de préemption avec l'approche PP(Passive-Passive).

la tâche sur son processeur), puis choisir celle avec la date de fin d'exécution la plus proche, cette copie de sauvegarde est la copie primaire, les autres sont les copies de sauvegardes répliquées. Dans le cas de défaillance de cette copie l'une des copies répliquées (probablement, celle avec la date de fin d'exécution la plus proche) est choisie pour la remplacer.

Algorithme 1 Algorithme de Préemption

Entrées : $Rep(t_i)$, $St(t^{(s)k}_{i(x)}$, $Ext(t^{(s)k}_{i(x)}, B_j)$.

Sorties : La nouvelle copie de sauvegarde primaire : $t^{(s)k}_{i(p)}$;

Début

1. Calculer les dates de fin d'exécutions prévues des Nf_{Proc}+1 copies de sauvegarde de la tâche t_i sur chacun des processeurs sur lesquels elles sont ordonnancées.

   Pour $k = 1$ **à** $Nf_{Bus} + 1$

   $Et(t^{(s)k}_{i(x)}) = \left\{ St(t^{(s)k}_{i(x)}) + Ext(t^{(s)k}_{i(x)}, P_j) \right\}$

   Fin

2. Ordonner ces dates dans un ordre croissant, la copie de sauvegarde avec le temps de fin d'exécution le plus proche est la copie primaire, le reste c'est les copies de sauvegarde répliquées.

   $t^{(s)y}_{i(p)} = t^{(s)m}_{i(x)} \ / \ Et(t^{(s)m}_{i(x)}) = min\{Et(t_y), t_y \in Rep(t_i)\}$

Fin

5.3.3 Tolérance aux fautes des bus de communication par redondance Passive :

Nous supposons que les processeurs sont exempts de fautes, pour tolérer Nf_{Bus} fautes de bus de communication, chaque dépendance de données est répliquée en $Nf_{Bus}+1$ copies et chaque copie est fragmentée en $Nf_{Bus}+1$ fragments ordonnancés sur $Nf_{Bus}+1$ bus distincts. Nous appelons copie primaire d'un fragment donnée $data_{i(p)}$, la copie avec le Eet : temps d'arrivée le plus proche. Les autres copies sont des copies de sauvegarde $data_{i(s)}$. Seule la copie ou la réplique primaire (ses $Nf_{Bus}+1$ fragments) est exécutée (voir Figure 5.7).

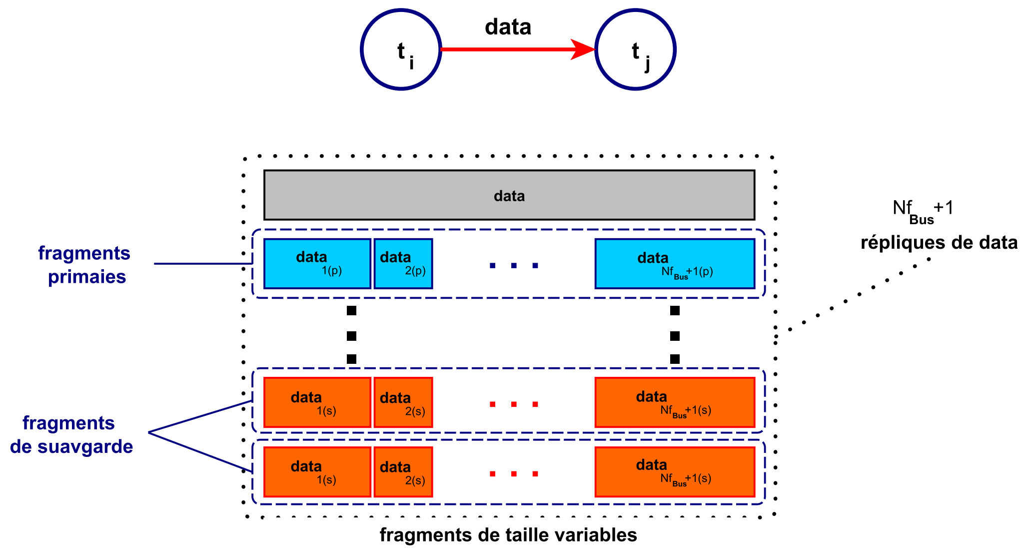


FIGURE 5.7 – Réplication et fragmentation des données de communication.

Si l'un de ses fragments échoue (en cas de défaillance du bus sur le quel ce fragment est envoyé ou de son processeur émetteur), l'un des fragments de sauvegarde est choisi pour devenir le nouveau fragment primaire.

Par exemple, dans l'ordonnancement de la Figure 5.8, afin de tolérer une seule faute de bus de communication, la dépendance de données ($t_1 \rightarrow t_2$) est fragmentée en 2 paquets ou fragments de données primaires : $data_{1(p)}$ et $data_{2(p)}$, ce qui donne $data = data_{1(p)} \circ data_{2(p)}$, et deux paquets de sauvegarde sont crées $data_{1(s)}$ et $data_{2(s)}$. Ensuite, l'opération t_1 envoie ces deux paquets de données primaires $data_{1(p)}$ et $data_{2(p)}$ à l'opération t_2 via les deux bus B_1 et B_2. Après un certain temps ($Time_{out}$), le paquet $data_{2(p)}$ n'arrive pas à destination puisque le bus B_2 est défaillant, t_1 envoie le paquet de sauvegarde $data_{2(s)}$ sur un troisième bus B_3.

Pour calculer les tailles des fragments de données, nous proposons une nouvelle technique dite **Fragmentation Variable des Données**, elle permet d'ajuster dy-

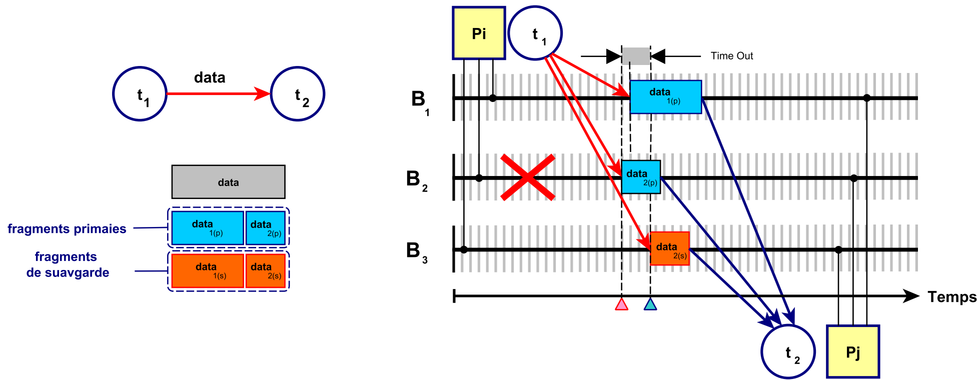


FIGURE 5.8 – Tolérance au fautes du Bus B_2.

namiquement la taille de chaque fragment à l'état et la charge du bus sur lequel il est ordonnancé.

5.3.4 Fragmentation variable des données :

Afin de bien exploiter la redondance matérielle offerte par les architectures multi-bus, la stratégie de communication que nous proposons est basée sur la fragmentation variable des données de communication en plusieurs paquets de données. La fragmentation variable nous permet de détecter et de recouvrir rapidement les fautes des bus de communication.

Pour Tolérer Nf_{Bus} fautes de bus de communications, les données de chaque dépendance de données sont fragmentées en $Nf_{Bus}+1$ paquets de données, envoyer par chaque tâche ou opération source en parallèle sur $Nf_{Bus}+1$ bus différents à destination de chaque opération destination, comme nous utilisons une fragmentation variable, la taille de chaque fragment $Tl(data_i)$ transmis sur le bus B_j dépend du $\Lambda(S_n)$ donc indirectement du λ_{B_j}.

$$Tl(data_i/B_j) = F(\Lambda(S_n), \lambda_{B_j}) \tag{5.3}$$

Dans la Figure 5.9, les données $data$ de la dépendance de données $(t_i \to t_j)$ sont fragmentées en $Nf_{Bus}+1$ paquets de données : $data_1$ $\ldots$,$data_{Nf_{Bus}+1}$, ce qui donne $data = data_1 \circ \ldots \circ data_{Nf_{Bus}+1}$. L'opération « $\circ$ » sert à concaténer deux paquets de données ; elle est associative. Chaque paquet $data_i$ est émis sur le bus B_i.

La fragmentation variable des données se fait en trois étapes. Dans un premier temps, pour tolérer au plus Nf_{Bus} fautes de bus de communication chaque dépendance de données $(t_i \to t_j)$ est fragmentée en $Nf_{Bus}+1$ fragments de tailles égales. Cette taille initiale est calculée pour chaque fragment par la formule (5.4) :

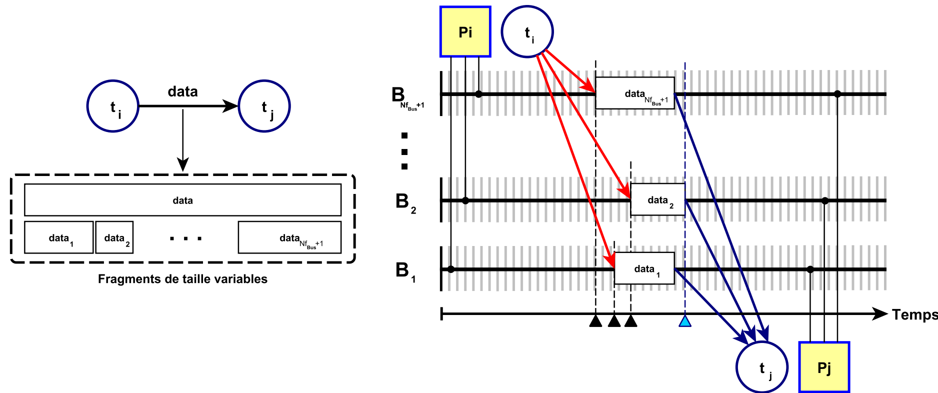


FIGURE 5.9 – Fragmentation variable des données.

$$Tl(data_i) = \frac{Tl(data)}{Nf_{Bus} + 1} \tag{5.4}$$

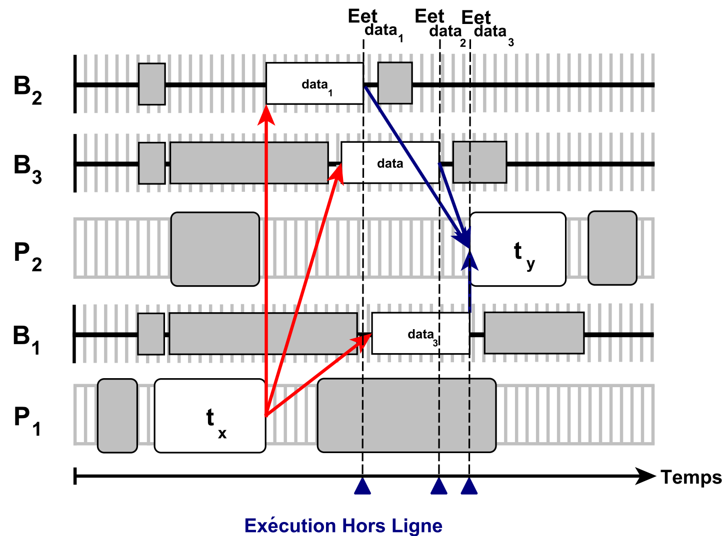


FIGURE 5.10 – Décalages des dates d'arrivées des fragments de données.

Le problème principal avec la fragmentation fixe vient du décalage des dates d'arrivées des différents fragments ; en faite l'opération destination doit attendre l'arriver de la totalité des fragments de données de la dépendance pour commencer son exécution. Dans la Figure 5.10, l'opération t_y doit attendre l'arrivée du fragment $data_3$ pour commencer son exécution malgré quelle peut commencer son exécution plutôt sur le processeur P_2 puisque ce dernier est libre.

L'idéal avec notre exemple est de transmettre la plus grande quantité de données avec les bus les moins chargés de telle manière que les dates d'arrivées des données

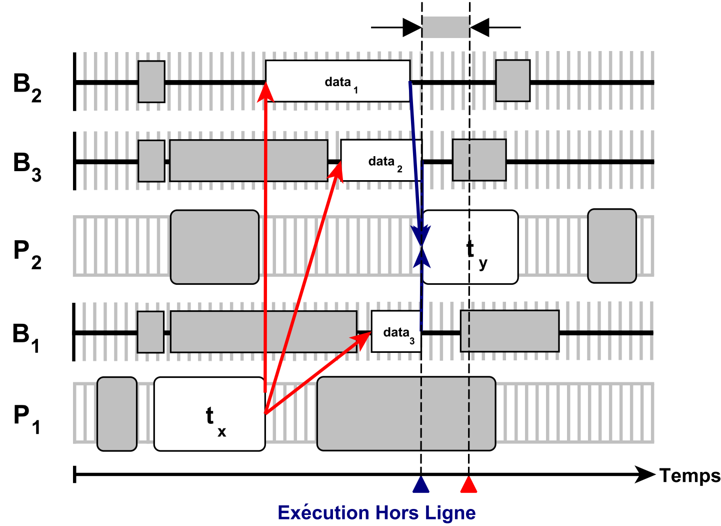


FIGURE 5.11 – Minimisation des écarts des dates d'arrivées des données.

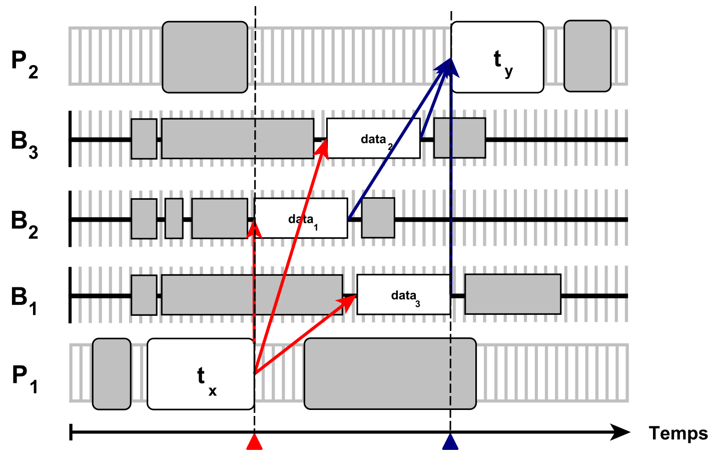


FIGURE 5.12 – Exemple de minimisation des dates d'arrivées des fragments.

ne soient plus très éloignées ; donc, dans notre exemple , cela revient à transmettre plus de données avec le bus B_3 qu'avec le bus B_2, la Figure 5.11 présente un gain de temps d'exécution égale à : $Eet_{data_3} - Eet_{data_2}$

L'astuce dans le passage de la fragmentation fixe à la fragmentation variable est que les dates d'arrivées des données ne soit plus très éloignées, en d'autres termes cela revient à réduire au maximum les écarts de dates d'arrivées.

Soit la dépendance de données $data = t_i \rightarrow t_j$, fragmenter en $Nf_{Bus}+1$ fragments de taille fixe $data = data_1 \circ \ldots \circ data_{Nf_{Bus}+1}$, la taille de chaque fragment est donnée par l'équation (5.4). Soit $Eet_{data_1}, Eet_{data_2}, \ldots, Eet_{data_{Nf_{Bus}+1}}$, les dates d'arrivée des

fragements $data_1, data_2, \ldots, data_{Nf_{Bus}+1}$ sur les bus correspondants. Ces dates sont ordonnées dans un ordre croisant, $Eet_1, Eet_2, \ldots, Eet_{Nf_{Bus}+1}$, Eet_1 corresponde à la date d'arrivée du premier fragment arrivant de la dépendance de données. L'objectif est de réduire au maximum les écarts des dates d'arrivées comment le montre la formule (5.5) :

$$\forall_{i \in \{1,..,Nf_{Bus}\}}, Minimiser\,\{Eet_{i+1} - Eet_i\} \tag{5.5}$$

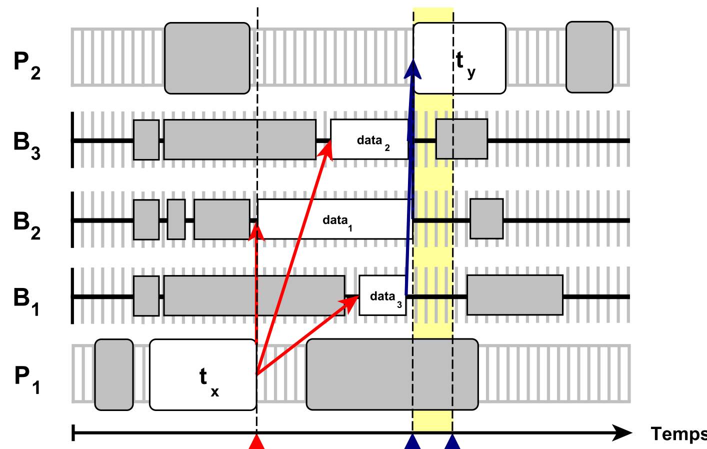


FIGURE 5.13 – Taille minimale de d'un fragment.

Puisque nous utilisons la redondance passive des communications pour la tolérance de Nf_{Bus} fautes de bus, la minimisation des écarts de dates d'arrivées doit prendre en considération le faite que le nombre de fragments doit être égales à $Nf_{Bus}+1$, pour cela chaque fragment a une taille minimale $Tl_{min}(data_i) > 0$ associée à chaque bus, elle dépend de son taux de défaillance. les deux figures Figure 5.12 et Figure 5.13 montrant un exemple de minimisation des dates d'arrivées tout en respectant la contrainte de taille minimale de chaque fragment.

Avec la fragmentation variable des données basée sur la minimisation des écarts des dates d'arrivées un autre problème peut arriver et pousser les temps d'exécution à l'extrême, le bus ou les bus de données qui accumulent plus de données peuvent eux aussi tomber en panne, et dans ce cas, la quantité de données à retransmettre par les autres bus est plus importante.

La Figure 5.14 montre le cas où le bus B_2 accumule un retard sur l'envoi d'une donnée d'une autre opération pré-ordonnancée, et puisqu'il est chargé de transmettre le plus grand fragment $data_1$, donc ce retard va engendrer le décalage de l'opération t_y sur le processeur P_2. La solution à ce problème est la définition d'un compromis entre la charge de chaque bus et la quantité d'informations maximale à transmettre sur ce bus.

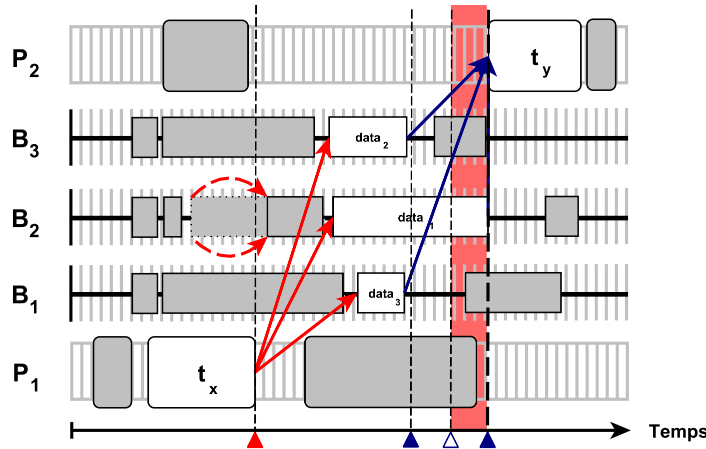


FIGURE 5.14 – Exemple : retard de l'envoi des données sur le bus B_2.

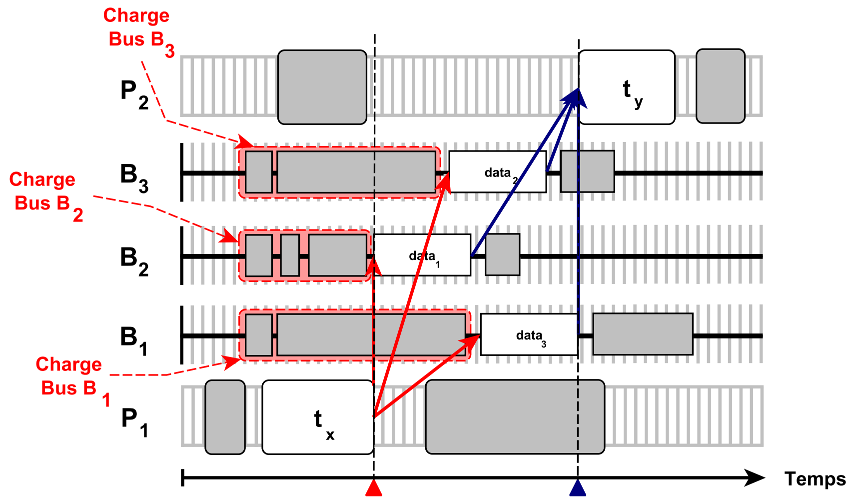


FIGURE 5.15 – Calcul des charges des bus.

La troisième étape de la fragmentation variable calcule la moyenne pondérée des charges des $Nf_{Bus}+1$ bus en fonction de leurs taux de défaillance et de leurs charge, comme l'indique l'équation (5.6)

$$Charge_{moy} = \frac{\sum_{i\in\{1,..,Nf_{Bus}\}}(\lambda_{B_i} * Charge_{B_i})}{\sum_{i\in\{1,..,Nf_{Bus}\}}\lambda_{B_i}} \quad (5.6)$$

$Charge_{B_i}(t)$ représente la charge d'un bus B_i à l'instant t, cette charge est calculée en fonction des deux paramètres suivants : le nombre total des fragments ordonnancés sur ce bus pour transmission et la taille globale de ces fragments, généralement le nombre de fragments a plus d'influence sur le comportement du bus

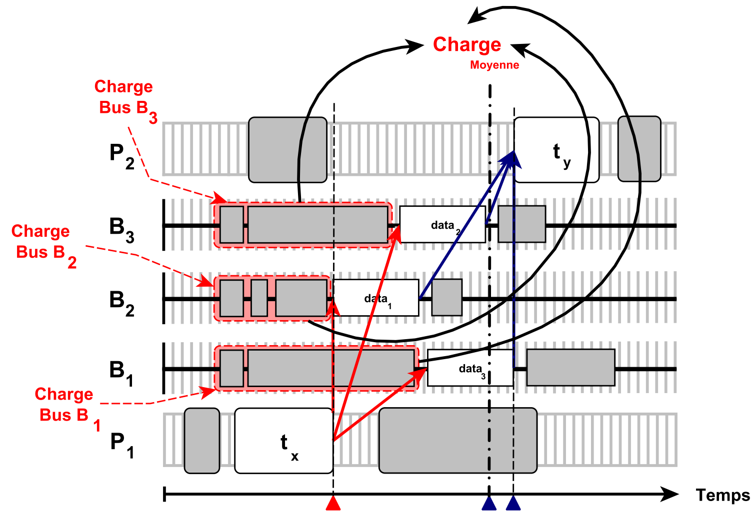


FIGURE 5.16 – Charge moyenne des bus et seuil de chargement des bus.

que la charge elle même, puisque plus le nombre de fragments est important plus la probabilité d'avoir un retard dû à la non délivrance d'un fragment au bus par le processeur où l'opération source est implémentée (à cause de la défaillance du processeur par exemple), est importante. Donc la formule utilisée pour le calcul de $Charge_{B_i}(t)$ doit attribuer plus de poids au nombre de fragments qu'à leur taille globale.

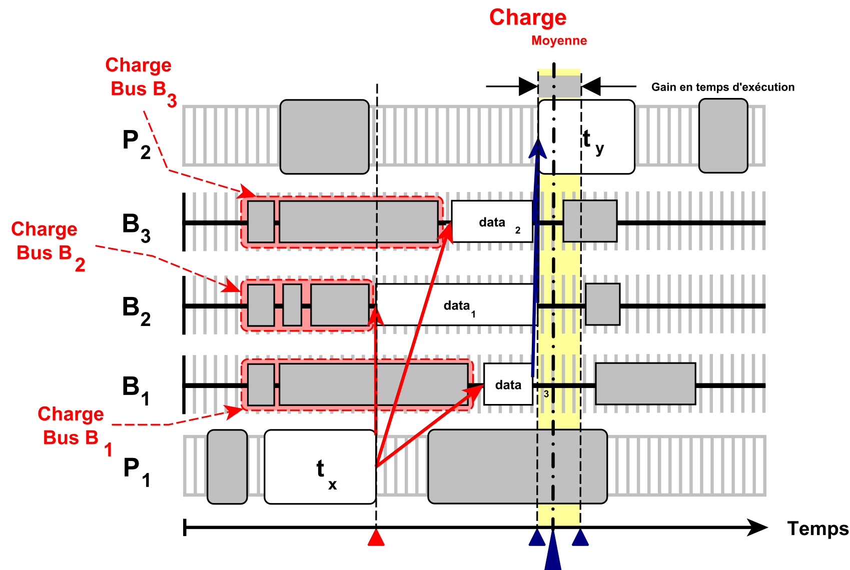


FIGURE 5.17 – Adaptation des tailles des fragments aux charges des bus.

Soit $FRAG_{B_i}(t) = \{frag_1, frag_2, \ldots, frag_n\}$ l'ensemble des fragments de don-

nées à transmettre par le bus B_i à l'instant t.

$$Charge_{B_i}(t) = G(Card(FRAG_{B_i}(t)), \sum_{i=n}^{i=1} Tl(frag_i)) \tag{5.7}$$

La Figure 5.16 définie la valeur $Charge_{moy}$ pour notre exemple, l'algorithme de la fragmentation variable ne doit pas dépasser cette valeur lors de la définition des nouvelles tailles des fragments. Le gain en temps lors de l'ordonnancement est représenté par la Figure 5.17. L'algorithme 2 permet la fragmentation variable des données.

Algorithme 2 ALGORITHME DE FRAGMENTATION VARIABLE DES DONNÉES.

Entrées : la dépendance de données $data = t_i \rightarrow t_j$, Nf_{Bus}
Sorties : l'ensemble des Nf_{Bus}+1 fragments $data = data_1 \circ \ldots \circ data_{Nf_{Bus}+1}$
Début

1. Fragmenter $data = t_i \rightarrow t_j$ en Nf_{Bus}+1 fragments de tailles égales.
   $Tl(data_1) = Tl(data_2) = \ldots = Tl(data_{Nf_{Bus}+1}) = \frac{Tl(data)}{Nf_{Bus}+1}$
2. Calculer de seuil de chargement des bus :
   $Charge_{moy} = \frac{\sum_{i \in \{1,..,Nf_{Bus}+1\}} (\lambda_{B_i} * Charge_{B_i})}{\sum_{i \in \{1,..,Nf_{Bus}+1\}} \lambda_{B_i}}$
3. Placer les Nf_{Bus}+1 fragments de données sur Nf_{Bus}+1 bus.
4. Ordonner les fragments de données selon leurs dates de fin d'exécution.
   $Eet_1 \leq Eet_2 \leq \ldots \leq Eet_{Nf_{Bus}+1}$
5. Calculer la somme des écarts entre les dates de fin d'exécutions des fragments.
   $Ecart_{Eet} = \sum_{i=1}^{Nf_{Bus}} (Eet_{i+1} - Eet_i)$;
   $Ecart^*_{Eet} = Ecart_{Eet} - 1$;
6. **Tant que** $Ecart^*_{Eet} \leq Ecart_{Eet}$ **faire :**
   - A) Affecter la nouvelle valeur : $Ecart_{Eet} = Ecart^*_{Eet}$.
   - B) Fragmenter le fragment de données dont la date de fin d'exécution est la plus éloignée, en deux fragments ($data(Eet_{Nf_{Bus}+1}) = data^A \circ data^B$), cette fragmentation doit respecter les trois conditions suivantes :
     - i. $Tl(data^A) \geq Tl_{min}(data(Eet_1))$.
     - ii. $Tl(data(Eet_1)) + Tl(data^B) \leq Charge_{moy}$.
     - iii. $Eet_1 + Tl(data^B)_{B_{data_1}} \leq Eet_{(data^B)}$.
   - C) Ordonner les nouvelles valeurs de Eet_i : les temps de fin d'exécutions des nouveaux fragments de données.
   - D) Calculer la nouvelle valeur de la somme des écarts : $Ecart^*_{Eet}$.

   Fin Tant que

Fin

La fragmentation variable des données présente un avantage très important en présence de défaillances, parceque le recouvrement des erreurs sera *plus rapide* et *moins coûteux* en nombre de communications.

5.3.5 Ordonnacement tolérant aux fautes des processeurs et des bus de communication

Dans **RBF-VDF**, afin de tolérer Nf_{Proc} fautes de processeurs et Nf_{Bus} fautes de bus, nous combinons les deux techniques présentées dans la section 5.3.2 (tolérance aux fautes des processeurs) et la section 5.3.3 (tolérance aux fautes des bus). Nous utilisons donc la redondance passive-passive des opérations et passive des communications.

Traiter à la fois les fautes des processeurs et des bus de communication, est plus délicat surtout avec l'utilisation de la redondance passive. L'identification et la distinction des fautes sont plus compliquées, ceci augmente considérablement le temps de recouvrement des erreurs. Afin d'accélérer le processus de recouvrement des erreurs et donc de minimiser le surcoût en longueur de l'ordonnancement de G_{Task} sur G_{Arch} en présence de défaillances, *RBF-VDF* utilise le mécanisme de communication basé sur la fragmentation variable des données déjà présentée dans la section 5.3.4.

Ce mécanisme nous permet :

1. Une distinction rapide entre une faute d'un bus et une faute d'un processeur.
2. Une distinction rapide entre une faute partielle et une faute complète d'un bus.
3. Une identification facile de l'origine d'une faute.
4. Utilisation efficace des ressources de communication.

Algorithme d'ordonnancement : Notre heuristique d'ordonnancement est un algorithme de construction progressive de type glouton [98], qui planifie une seule opération ou tâche à chaque étape n. elle permet de générer un ordonnancement statique distribuée d'un modèle de tâches G_{Task} donné sur un modèle d'architecture G_{Arch} donné, dans le but de minimiser l'exécution du système, et tolérer jusqu'à Nf_{Proc} fautes de processeurs et Nf_{Bus} fautes de bus de communications.

Notre algorithme est utilise deux fonctions de coût, la pression d'ordonnancement et le TGDS. Le *TGDS* est utilisé doublement, d'un côté, il est utilisé pour définir la taille des fragments, et d'un autre côté pour sélectionner le bus le plus fiable pour chaque fragment de données. Alors que la fonction de pression d'ordonnancement notée $\sigma^{(n)}(t_i, p_j)$ est utilisée par notre algorithme comme fonction de coût pour sélectionner la meilleure tâche qui minimise la longueur du chemin critique en prenant en compte la fragmentation variable des données. Elle est calculée pour chaque tâche de la façon suivante :

$$\sigma^{(n)}(t_i, p_j) = S^{(n)}_{t_i,p_j} + \overline{S}^{(n)}_{t_i} - R^{(n-1)}$$

où, $R^{(n-1)}$ est la longueur de la liste d'ordonnancement partielle composée des tâches déjà placées, $S^{(n)}_{t_i,p_j}$ est l'instant le plus proche auquel la tâche t_i peut commencer son exécution sur le processeur P_j, et $\overline{S}^{(n)}_{t_i}$ est le dernier instant de début d'exécution à partir de la fin de t_i, définie comme étant la longueur du chemin le plus longue à partir de t_i jusqu'aux tâches de sortie de lu graphe G_{Task}.

La pression d'ordonnancement mesure de combien l'ordonnancement de la tâche va allonger le chemin critique de l'algorithme. Par conséquent, il introduit une priorité entre les tâches à ordonnancer. Notez que, si toutes les tâches d'une étape donnée n ont toutes la même valeur $R^{(n-1)}$, donc il n'est pas nécessaire de calculer $R^{(n-1)}$.

En résumé, la *RBF-VDF* donne quatre avantages :

① Détection rapide des fautes ;
② Distinction entre fautes de processeurs et fautes de bus de communication ;
③ Communication rapide des données (maximiser la taille du fragment envoyé et de réduire au minimum la probabilité de re-transmission) ;
④ Rapide reprise après incident.

L'heuristique *RBF-VDF* est présentée par l'Algorithme 3, il est composé des quatre phases suivantes :

— Phase d'initialisation : L'ensemble des opérations candidates à l'ordonnancement T_{cand} est initialisé avec les tâches sans prédécesseur. Donc, les seules tâches implémentables à cette première étape de l'heuristique sont les tâcjes d'entrées, l'ensemble des tâches déjà ordonnancées T_{ord} est initialement vide.

— Phase de sélection : Pendant cette étape de sélection, un processeur est choisi parmi l'ensemble des processeurs de G_{Arch} pour ordonnancer une tâche et ses données de communication. La tâche choisie est la candidate la plus urgente t_{urg}, la règle de sélection choisie repose sur la pression d'ordonnancement $\sigma^{(n)}$ et TGDS.

— Phase d'ordonnancement : Après avoir sélectionné la meilleure candidate t_{best}, cette phase consiste en un premier temps à répliquer cette tâche candidate en $Nf_{Proc}+1$ répliques, et en un deuxième temps, à ordonnancer chaque réplique t^k_{best} de t_{best} sur le k^{ieme} processeur p^k_{best} de $E_{Procbest}$.
Avant d'ordonnancer ces répliques, chaque dépendance de données $(t_j \rightarrow t_{best})$ sera fragmentée en $Nf_{Bus}+1$ fragments de données en utilisent l'algorithme de fragmentation variable de données qui seront ordonnancés sur $Nf_{Bus}+1$ bus distincts.

— Phase de mise à jour : La tâche t_{best} est retirée de l'ensemble des tâches candidates pour l'ordonnancement T_{cand}, et les tâches de l'G_{Task} qui ont toutes leurs prédécesseurs dans le nouveau ensemble des tâches déjà ordonnancées sont ajoutées à cet ensemble.

A l'exécution de cet algorithme, trois cas peuvent se présenter :

Algorithme 3 ALGORITHME D'ORDONNANCEMENT RBF-VDF.

Entrées : G_{Task}, G_{Arch}, Nf_{Proc}, Nf_{Bus} ;
Sorties : Ordonnancement des tâches et dépendances de G_{Task} sur G_{Arch} tolérant au plus Nf_{Proc} fautes de processeurs et Nf_{Bus} fautes de bus de communication ;

Début
Initialiser la liste des tâches candidates, et la liste des tâches déjà placées :
n := 0 ;
$T^{(0)}_{cand} := \{t \in E_{Task} \mid pred(t) = \emptyset\}$; (tâche de G_{Task} sans prédécesseurs)
$T^{(0)}_{sched} := \emptyset$;

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

1. Pour chaque tâche candidate t_{cand}, calculer $\sigma^{(n)}$ et $\Lambda(S_n)$ pour chaque processeur p_j.
   $\sigma^{(n)}(t_i, p_j) = S^{(n)}_{t_i,p_j} + \overline{S}^{(n)}_{t_i} - R^{(n-1)}$
   $\Lambda(S_n) = \frac{-\log R(S_n)}{U(S_n)}$
2. Pour chaque tâche candidate t_{cand}, sélectionner le meilleur processeur parmi $p^{t_{cand}}_{best}$ qui minimise $\sigma^{(n)}$ et le TGDS.
3. Sélectionner la tâche candidate la plus urgente t_{urgent}, parmi toutes les opérations t^i_{cand} de $T^{(n)}_{cand}$.
4. Fragmenter chaque dépendance de donnée de t_{urgent} en Nf_{Bus}+1 fragments de tailles différentes en utilisent l'algorithme de fragmentation variable de données (Algorithme : 2).
5. Ordonnancer t_{urgent} et ses fragments de données ;
6. Mettre à jour la liste des tâches candidates et des tâches déjà ordonnancées.
   $T^{(n)}_{sched} := T^{(n-1)}_{sched} \cup \{t_{urgent}\}$;
   $T^{(n+1)}_{cand} := T^{(n)}_{cand} - \{t_{urgent}\} \cup \{t' \in succ(t_{urgent}) \mid pred(t') \subseteq T^{(n)}_{sched}\}$;
7. n := n + 1 ;

Fin Tant que
Fin

1. Tous les paquets $data_1 \circ \ldots \circ \; data_l \circ \ldots \circ data_{Nf_{Bus}+1}$ de $data$ fragmentés et envoyés par t_i sont reçus par l'opération destination t_j. t_j qui rassemble les fragments de $data$ et commence son exécution.
2. Aucun des paquets $data_1 \circ \ldots \circ \; data_l \circ \ldots \circ data_{Nf_{Bus}+1}$ de $data$ n'est reçu par t_j, cela représente Nf_{Bus}+1 paquets et comme par hypothèse le système ne peut avoir plus de Nf_{Bus} fautes de bus de communication, ceci implique forcément la défaillance du processeur P_i implantant la tâche source t_i, on peut utiliser la redondance passive-passive des opérations pour remédier à ce problème.
3. Certains paquets $data_m, ..., data_l$ envoyés par t_i ne sont pas reçus par t_j, l'ensemble des bus de G_{Arch} qui était censé les transmettre est ${E_{Bus}}^* = B_m, ..., B_k$. Étant donné que d'autres paquets ont été reçus, cela signifie que P_1, le processeur exécutant t_1 n'est pas défectueux, donc forcément les bus de ${E_{Bus}}^*$ sont défectueux. Par conséquent, la même opération t_1 ré-envoie les fragments des

données $data_m, ..., data_l$ par l'intermédiaire d'autres bus qui ne font pas partie de l'ensemble $E_{Bus}{}^*$.

5.4 Réduction de la Consommation d'énergie

Dans notre approche, comme les $Nf_{Proc}+1$ répliques de chaque tâche sont ordonnancées sur $Nf_{Proc}+1$ processeurs distincts, l'énergie consommée par le système est maximale. Afin de réduire cette consommation d'énergie, nous proposons d'exécuter ces $Nf_{Proc}+1$ répliques avec différentes fréquences $Freq$. Cependant, comme ces $Nf_{Proc}+1$ répliques ont des temps de fin d'exécution différents(voir Figure 5.18(a)), on choisit de les aligner en changeant la fréquence d'exécution $Freq$ de chaque réplique (Comme c'est montré sur la Figure 5.18(b) pour la tâche $t_j^{(s)1}$).

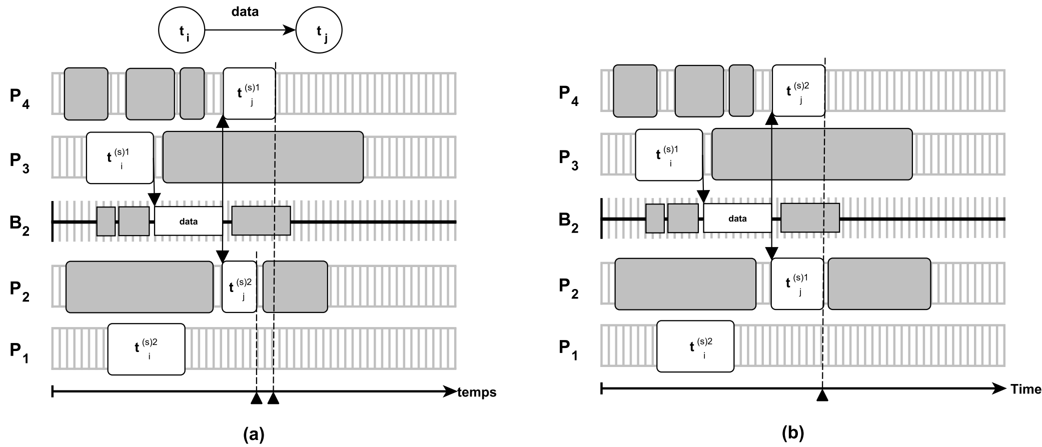


FIGURE 5.18 – Changement de fréquence d'exécution de $t_j^{(s)1}$.

Pour calculer la consommation d'énergie, nous suivons le modèle présenté dans [99]. Pour une tâche ordonnancée un processeur, la consommation d'énergie E est calculée par la formule :

$$E = E_{st} + h(E_{ac} + E_{re}) \tag{5.8}$$

Où E_{st} est la consommation d'énergie statique (c'est la puissance nécessaire pour maintenir les circuits de base et de garder l'horloge en marche), h est égale à 1 lorsque le circuit est actif et 0 quand il est inactif, E_{ac} est la consommation d'énergie active (la partie de la puissance qui est indépendante de la tension et de la fréquence ; il devient 0 lorsque le système est mis en veille) indépendante de la fréquence, P_{re} est la consommation d'énergie relative à la fréquence active, elle est calculée comme suit :

$$E_{re} = Coef_c * V^2 * Freq \tag{5.9}$$

$Coef_c$ est la coefficient de commutation, V est la tension d'alimentation, et $Freq$ est la fréquence de fonctionnement. Pour les architectures multiprocesseurs, ce modèle de calcul de consommation d'énergie est plus raffiné, puisque nos calculons la consommation d'énergie en additionnant la contribution de chaque processeur, en fonction de la tension et la fréquence de chaque tâche ordonnancée sur celui-ci (l'énergie consommée par chaque tâche est le produit de la consommation d'énergie active par son temps d'exécution).

5.5 Exemple et Simulations

5.5.1 Exemple

Nous avons appliqué l'heuristique RBF-VDF à un exemple dont le modèle de tâches est présenté par la Figure 5.19, et le modèle d'architecture par la Figure 5.20. Notre exemple est composé de quatre processeurs et quatre bus, les taux de défaillance des processeurs sont tous égaux à 10^{-5}, et les taux de défaillance des bus B_1, B_2, B_3 et B_4 sont respectivement : $\lambda_{B_1} = 10^{-6}$, $\lambda_{B_2} = 10^{-6}$, $\lambda_{B_3} = 10^{-5}$ et $\lambda_{B_4} = 10^{-4}$.

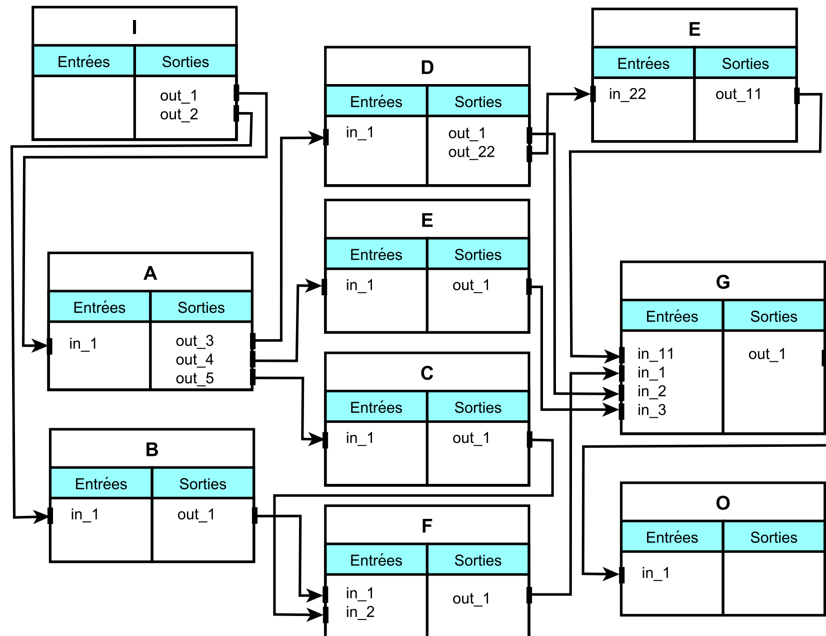


FIGURE 5.19 – Modèle de tâches.

La Figure 5.21 montre l'ordonnancement non fiable de notre exemple avec une heuristique d'ordonnancement de base HTR [5] de SYNDEX [6] [100]. SYNDExest un

5. HTR = Heuristique de distribution/ordonnancement Temps Réel.
6. SYNDEX = **Syn**chronized **D**istributed **Ex**ecutive. http ://www-rocq.inria.fr/syndex.

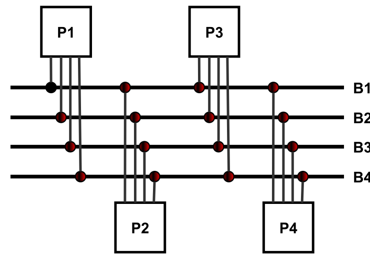


FIGURE 5.20 – Modèle d'Architecture.

outil d'optimisation et de mise en œuvre des applications temps réel embarquées sur l'architecture à composants multiples. La longueur de l'ordonnancement générée par cette heuristique est de 22,2. Le TGDS de l'ordonnancement non fiable Λ_{syndex} est égal à 0.0000246.

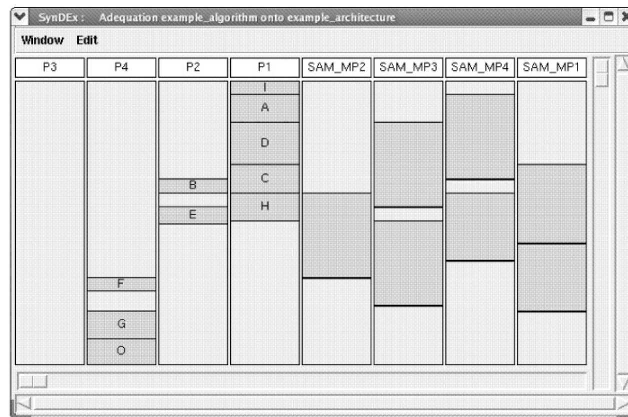


FIGURE 5.21 – Ordonnancement non-fiable SynDEx.

Nous appliquons notre heuristique à l'exemple de la Figure 5.19. Ici le système doit tolérer une seule faute de bus, c'est-à-dire Nf_{Bus}=1. La Figure 5.22 montre l'ordonnancement généré par notre heuristique. La longueur de l'ordonnancement généré est égale à 17,2. Le TGDS de l'ordonnancement non-fiable $\Lambda_{(Nf_{Bus}=1)}$ est égal à $0,00000597$.

La Figure 5.23 montre l'ordonnancement généré par notre heuristique pour Nf_{Bus}=2. La longueur de l'ordonnancement généré est de 15,2. Le TGDS de l'ordonnancement non fiable $\Lambda_{(Nf_{Bus}=2)}$ est égal à $0,00000613$.

La chose la plus importante à noter sur les deux figures Figure 5.22 et Figure 5.23 est que la fragmentation variable de données réduit la longueur de l'ordonnancement et apporte une amélioration considérable à la fiabilité du système.

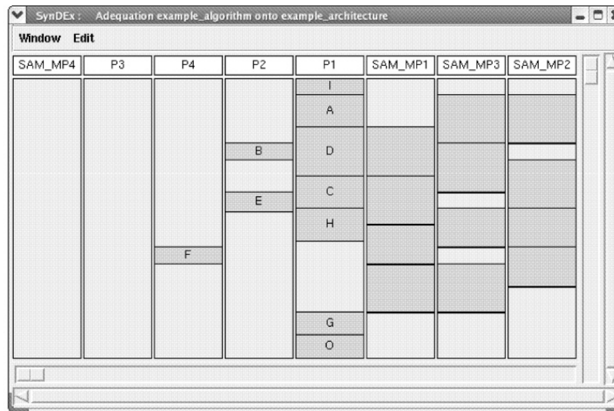


FIGURE 5.22 – Ordonnancement tolérant aux fautes pour Nf_{Bus}=1.

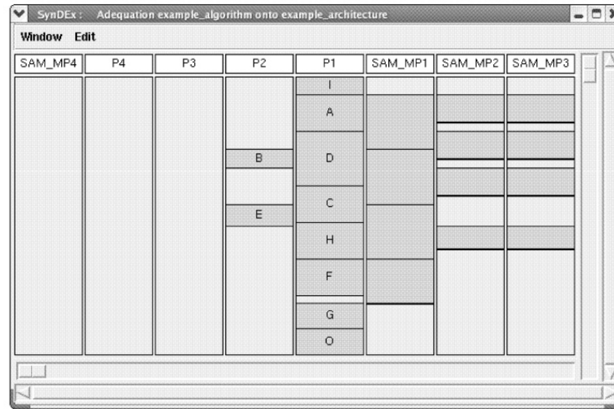


FIGURE 5.23 – Ordonnancement tolérant aux fautes pour Nf_{Bus}=2.

5.5.2 Simulations

Pour évaluer notre heuristique , nous l'avons appliquée RBF-VDF à un ensemble composé d'un modèle de tâches généré aléatoirement et d'un modèle d'architecture hétérogène et complètement connectée composé de E_{Proc}=4 processeurs. Les chiffres suivants ont été obtenus avec un ensemble de valeurs de CCR égales à 1 , 5 , 10 et 20. CCR (Ratio Communication-à-Calcul) est le rapport entre le coût moyen des communications (sur toutes les dépendances de données) et le coût moyen de calcul (pour toutes les tâches).

Un modèle de tâches aléatoire est généré comme suit : étant donné le nombre de N tâches, nous générons aléatoirement un ensemble de niveaux avec un nombre aléatoire de tâches. Ensuite, les tâches à un niveau donné sont reliés de façon aléatoire à des tâches d'un niveau supérieur. Les temps d'exécution de chaque tâche sont choisies au hasard à partir d'une distribution uniforme de moyenne égale à la moyenne des temps d'exécution choisie. De même , le temps de communication de

chaque dépendance de données est sélectionné aléatoirement à partir d'une distribution uniforme de moyenne égale à la durée moyenne des communications choisie.

L'objectif principal de nos simulations est d'étudier l'impact de la fragmentation variable des données et du CCR sur la longueur de l'ordonnancement et la fiabilité introduite par RBF-VDF.

La Figure 5.24 montre l'impact de la longueur d'ordonnancement obtenu par RBF-VDF pour une architecture de E_{Proc}= 4 processeurs, pour Nf_{Bus}=1(a) et Nf_{Bus}=2(b). Comme nous pouvons le voir, la longueur de l'ordonnancement monte presque linéairement quand le CCR augmente de 1 à 20 .

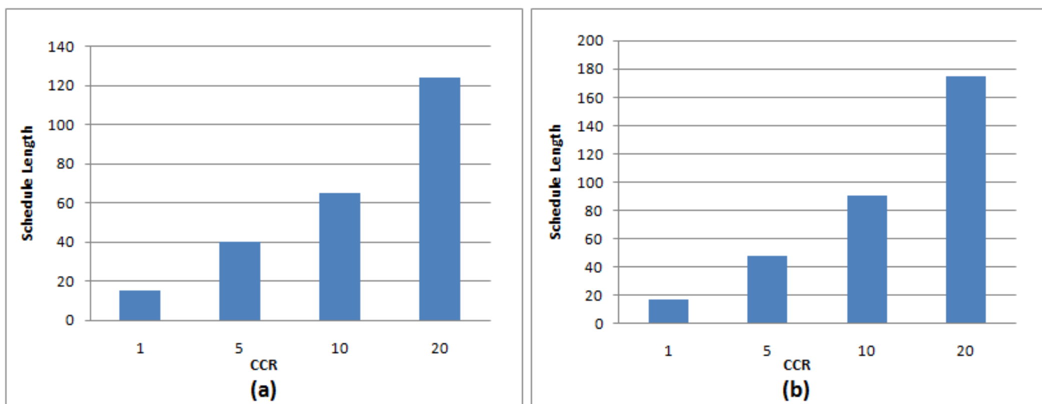


FIGURE 5.24 – Impact du CCR sur la longueur d'ordonnancement pour $Nf_{Bus} = 1$ et $Nf_{Bus} = 2$.

La Figure 5.25 montre l'impact du CCR sur le TGDS obtenu par RBF-VDF, pour une architecture de E_{Proc}= 4 processeurs, pour Nf_{Bus}=1(a) et Nf_{Bus}=2(b). Comme nous pouvons le voir, le TGDS diminue lorsque le CCR augmente de 1 à 20.

5.6 Conclusion

Nous avons proposé dans ce chapitre une nouvelle technique logicielle RBF-VDF de tolérance aux fautes pour des systèmes temps-réel distribués réactifs et embarqués, basée sur le TGDS. L'approche proposée produit automatiquement un ordonnancement statique et distribué tolérant aux fautes d'un modèle de tâche donné G_{Task} sur une architecture multi-bus donnée G_{Arch} . Notre approche implante une solution logicielle basée sur la redondance Passive-Passive des opérations avec un mécanisme de Préemption qui permet la définition dynamique de la tâche primaire et des tâches de sauvegarde répliquées, notre solution est basée aussi sur la redondance passive des communications avec une fragmentation variable des données ;

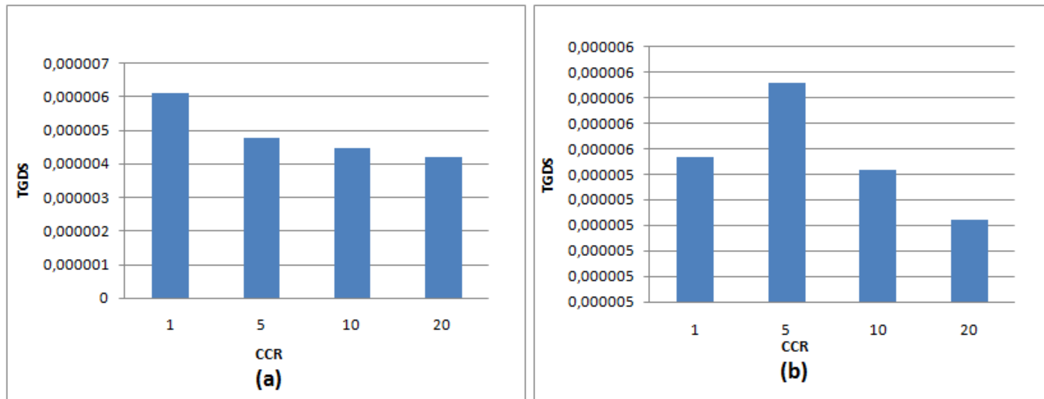


FIGURE 5.25 – Impact du CCR sur TGDS pour $Nf_{Bus} = 1$ et $Nf_{Bus} = 2$.

Seuls les fragments de la copie primaire du message sont envoyés, si un fragment est cible d'une faute, un autre des fragments de sauvegarde sera envoyé. Chaque fragment de donnée est de taille différente, la fragmentation variable permet d'adapter la taille de chaqu'un des fragments à la charge du bus au quel il est attribué pour transmission.

Les objectifs de RBF-VDF sont : la réduction de la latence (délai entre la production et la détection de l'erreur), l'optimisation du chemin critique et la maximisation de la fiabilité de l'algorithme distribué obtenu, Le RBF-VDF par l'utilisation de la fragmentation variable des données permet : une distinction rapide entre une faute d'un bus et une faute d'un processeur, une distinction rapide entre une faute partielle et une faute complète d'un bus de communication, une identification facile de l'origine d'une faute, une détection rapide des erreurs, et enfin un traitement rapide des fautes.

Chapitre 6

Approche d'ordonnancement fiable de communication basée sur la désallocation des données

Sommaire

6.1 Introduction **87**
6.2 Définition du problème **88**
6.2.1 Les modèles du système 88
6.2.2 Les copies de sauvegarde 89
6.3 Exemple illustratif **90**
6.4 Approche proposée. **94**
6.5 Simulations, résultats et discussion **99**
6.6 Conclusion **103**

L'approche d'ordonnancement proposée dans ce chapitre considère uniquement une seule faute de bus de communication dans une architecture multi-bus et hétérogène. Cette faute est causée par des défauts matériels et compensée par des solutions basée sur la redondance logicielle. L'objectif principal de l'approche proposée est de minimiser la longueur de l'ordonnancement tolérant aux fautes des données sur les bus de communications, on se basant sur une technique hybride qui utilise deux types de copies de sauvegarde, répliquées et désallouées.

6.1 Introduction

Dans ce chapitre, nous nous intéressons aux approches fondées sur des algorithmes d'ordonnancement, plus particulièrement celles fondées sur l'ordonnancement statique qui permet d'inclure les dépendances et les coûts d'exécution des tâches et des dépendances de données dans ses décisions d'ordonnancement. L'ordonnancement est déjà calculé au moment de la compilation.

L'objectif principal est de réduire au minimum la longueur de l'ordonnancement des données sur les bus, qui correspond au temps d'émission total des données, sous l'hypothèse que tout au plus un seul bus peut échouer. L'idée de base de notre travail est la combinaison de la redondance active et passive dans le même schéma. Elle a été initialement proposée par [101] pour la tolérance aux fautes des processeurs, ce que nous proposons est son adaptation à la tolérance aux fautes des bus de communication. Pour cela, un grand nombre de transformations et réaménagements sont nécessaires.

Tout d'abord, on définit le problème d'ordonnancement, comme un problème d'optimisation et on utilise la programmation linéaire pour le formuler d'une manière optimale en utilisant deux types de copies de sauvegarde, l'objectif étant de minimiser la longueur d'ordonnancement. L'utilisation de la programmation linéaire permet d'avoir les meilleurs résultats, mais comme c'est un problème NP-difficile, une telle solution prend généralement beaucoup de temps, et dans certains cas on ne peut pas trouver une solution réalisable dans un temps acceptable.

Pour résoudre ce problème, nous proposons notre solution, basée sur un algorithme heuristique appelé *FTA-RD*. Les objectifs de cet algorithme sont de deux ordres, d'un part maximiser la fiabilité du système ; d'autre part, minimiser la longueur de l'ensemble de l'ordonnancement tolérant au fautes généré à la fois en présence et en absence de fautes. Les simulations monteront que notre approche réduit généralement et de manière significative le temps global de l'exécution.

6.2 Définition du problème

6.2.1 Les modèles du système

Dans cette section, nous donnons d'abord quelques définitions qui décrivent notre système et nous définissons le problème d'ordonnancement tolérant aux fautes formellement. La spécification d'un tel système implique de décrire : le modèle de tâches, le modèle d'architecture, le modèle de données et le modèle de fautes.

6.2.1.1 Modèle de tâches

Le modèle de tâches est défini par un graphe orienté acyclique (DAG), qui ne possède pas de circuit (ni simple, ni élémentaire), noté G_{Task}=(E_{Task},E_{Dd},Ext), où $E_{Task} = \{t_1, t_2, ..., t_n\}$ est un ensemble de n tâches, et E_{Dd} est l'ensemble des arêtes orientés représentant les dépendances de données entre les tâches. Chaque arête allant d'une tâche t_i vers une tâche t_j noté : $t_i \rightarrow t_j$, signifie que la tâche t_j dépend des sorties de la tâche t_i. $Ext(t_i)$ est une fonction qui calcule le temps d'exécution de la tâche $t_i \in E_{Task}$. (La Figure 6.2 représente un exemple d'un modèle de tâches).

6.2.1.2 Modèle d'architecture

L'architecture du système est modélisée par un graphe non orienté G_{Arch} noté G_{Arch}=(E_{Proc},E_{Bus}), où chaque nœud est un processeur, et chaque arête est un média de communication (ou bus). Nous supposons que l'architecture est hétérogène et entièrement connectée. (La Figure 6.1 montre un exemple d'un modèle d'architecture).

6.2.1.3 Modèle de données

Le modèle de données est modélisé par un autre graphe orienté acyclique (DAG), noté G_{Data}=(M,E_{Dd},$Extc$). Le graphe G_{Data} est généré à partir du graphe G_{Task} avec une transformation qui respecte la précédence de données. $M = \{m_1, m_2, ..., m_L\}$ représente un ensemble de toutes les données transférées entre les tâches, la cardinalité de l'ensemble M est égale à celle de E_{Dd}. E_{Dd} est l'ensemble des arêtes orientées représentent les dépendances de données, une arête allant d'une données m_i à un données m_j notée $m_i \rightarrow m_j$ signifie que la donnée m_j nécessite la donnée m_i pour être calculée. $Extc(m_i)$ est une fonction représentant le coût de transfert de la donnée $m_i \in M$ en plus du temps requis pour exécuter une routine de détection de faute qui détermine si les données ont été reçus ou non.

6.2.1.4 Modèle de fautes

Dans le cadre de notre travail on traite uniquement les fautes des bus de communications. Chaque bus peut être la cible d'un défaut matériel. Ces fautes peuvent êtres transitoires ou permanentes et sont indépendantes. On suppose que au plus un seul bus ne parviendra pas à exécuter le transfert de données, et qu'il existe des mécanismes pour détecter cette faute, ces mécanismes peuvent se baser sur un signal d'échec ou sur des test d'acceptation.

6.2.2 Les copies de sauvegarde

6.2.2.1 Copie de sauvegarde répliquée

Une copie de sauvegarde répliquée m_i^R d'une donnée m_i, est une copie de sauvegarde active, qui est envoyée indépendamment, peu importe si la copie primaire m_i^P a été reçu avec succès ou non. Dans le cas où la copie primaire ne parvient pas à atteindre sa destination correctement, la copie répliquée est utilisée. Par exemple, dans la Figure 6.5, M_1^R est une copie de sauvegarde répliquée de m_1 ordonnancée sur le bus B_2.

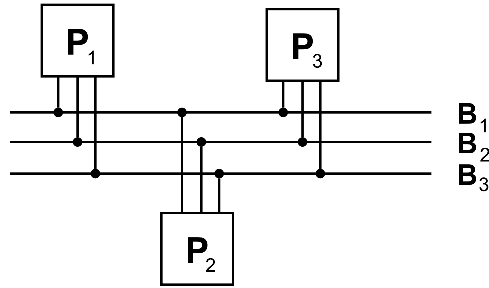


FIGURE 6.1 – Modèle d'architecture G_{Arch}.

6.2.2.2 Copie de sauvegarde désallouée

Une copie de sauvegarde désallouée m_i^D d'une donnée m_i, est une copie de sauvegarde passive, qui est envoyée uniquement si la copie primaire m_i^R échoue. La copie désallouée ne peut pas être exécutée jusqu'à ce que la copie primaire a été entièrement envoyée et le message d'activation a été reçu. Un message d'activation A_{msg} est un message envoyé par une copie primaire à sa copie de sauvegarde désallouée, qui indique si elle a été correctement envoyée ou non. Nous utilisons $|A_{msg}|$ pour désigner le coût (en terme de temps de transfert) d'un message d'activation.

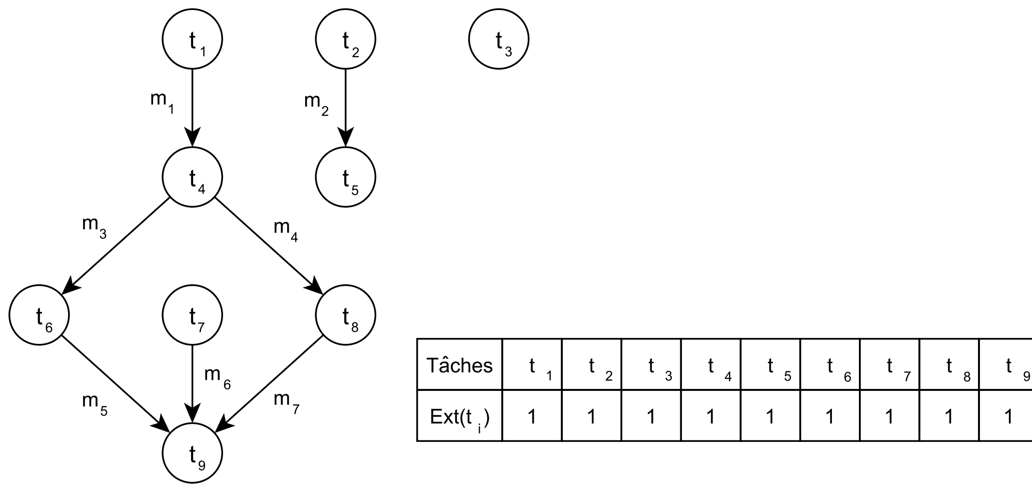
Le fait d'ordonnancer deux copies de sauvegarde désallouées en même temps sur le même bus, est appelé : chevauchement de copies de sauvegarde. La mise en œuvre du chevauchement est sous l'hypothèse qu'au plus une seule faute de bus peut être tolérée de sorte que pas plus d'une copie de sauvegarde désallouée sera exécutée à tout moment.

Par exemple, dans la Figure 6.5, m_2^D est une copie de sauvegarde désallouée de m_2 ordonnancée sur le bus B_2, se chevauchant avec un autre copie de sauvegarde désallouée m_6^D.

6.3 Exemple illustratif

Dans cette section, nous présentons un exemple pour illustrer le problème que nous essayons de résoudre. Le modèle d'architecture G_{Arch} de notre système est composé de trois processeurs qui sont connectés avec trois bus (comme le montre la Figure 6.1), nous supposons qu'il y aura au plus une seule faute de bus de communication.

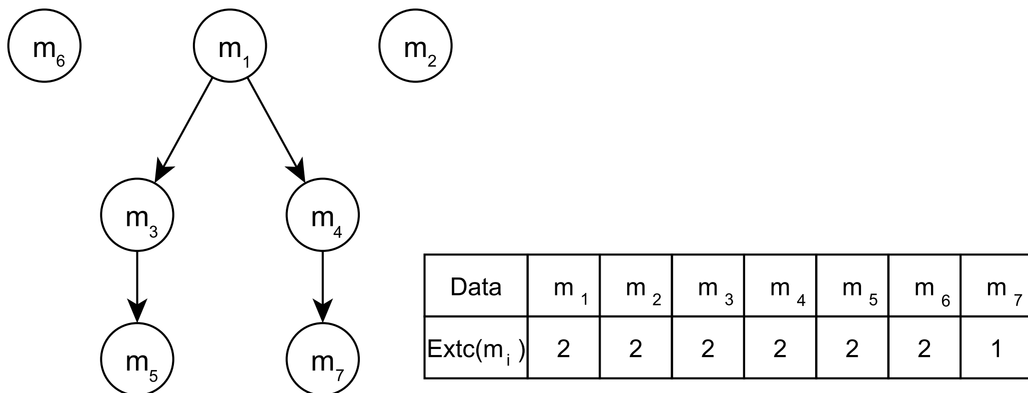
La Figure 6.2 présente le modèle de tâches G_{Task} associé à notre exemple, il se compose de neuf tâches t_1, t_2, t_3, t_4, t_5, t_6, t_7, t_8 et t_9. Les arrêtes montrent les dépendances de données entre les tâches. Par exemple, la tâche t_4 ne peut s'exécutée que lorsque la tâche t_1 a été exécutée correctement.



Tâches	t_1	t_2	t_3	t_4	t_5	t_6	t_7	t_8	t_9
$Ext(t_i)$	1	1	1	1	1	1	1	1	1

FIGURE 6.2 – Modèle de tâches G_{Task}.

La Figure 6.3 présente le modèle de données G_{Data} généré à partir du modèle de tâches G_{Task}, de la Figure 6.2. Le modèle de données reprend et respecte les dépendances des tâches. Par exemple, dans la Figure 6.3, les deux données m_3 et m_4 ne peuvent pas être envoyées jusqu'à ce que la donnée m_1 soit correctement reçu. Parce que dans le modèle de tâches de la Figure 6.2, la tâche t_4 a besoin de m_1 pour calculer m_3 et m_4. Pour notre exemple, nous supposons que le délai nécessaire pour le message d'activation est d'une unité de temps. $|A_{msg}| = 1$.



Data	m_1	m_2	m_3	m_4	m_5	m_6	m_7
$Extc(m_i)$	2	2	2	2	2	2	1

FIGURE 6.3 – Modèle de données G_{Data}.

Le problème à résoudre est de trouver un ordonnancement optimal tolérant aux fautes avec une longueur minimisée, de ces données et leurs copies de sauvegarde, de sorte que toutes les données peuvent être transmises correctement, toute en supposant qu'une seule faute de bus peut se produire.

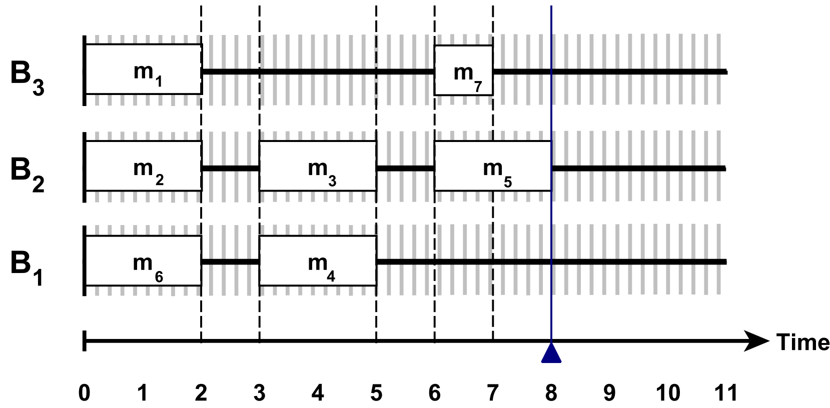


FIGURE 6.4 – ordonnancement de données non-tolérant aux fautes.

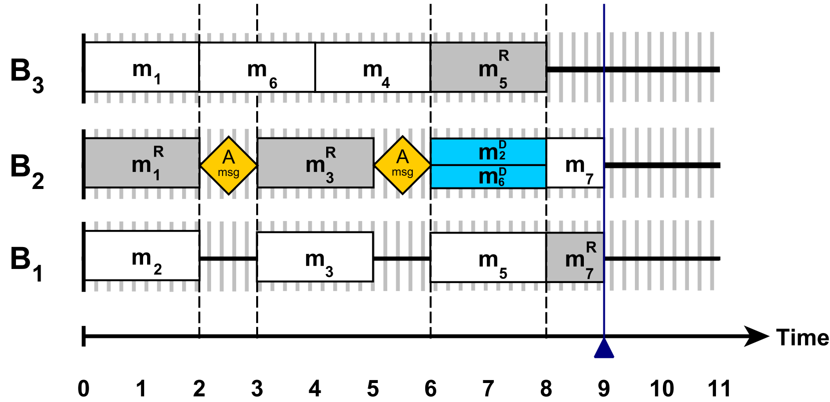


FIGURE 6.5 – Ordonnancement de données tolérant aux fautes et optimal avec réplication et désallocation.

La Figure 6.4 présente un exemple d'ordonnancement non-tolérant aux fautes, la longueur d'ordonnancement pour ce cas est égale à 8.

Dans le cas de notre exemple, l'avantage de combiner, à la fois la réplication et la désallocation dans le même algorithme, est représenté par les trois ordonnancements optimaux suivants, (voir les Figures 6.5, 6.6, 6.7). L'ensemble des résultats optimaux est obtenu par la formulation de la programmation linéaire présentée dans la Section 6.4.

Un ordonnancement tolérant aux fautes optimal, utilisant les deux types de copies de sauvegarde répliquées et désallouées, pour le cas de notre exemple est illustré dans la Figure 6.5. La longueur d'ordonnancement minimale est de 9 unités de temps. Pour cet ordonnancement, nous avons choisi des copies de sauvegarde désallouées pour les données m_2 et m_6 et des copies de sauvegarde répliquées pour le reste des données.

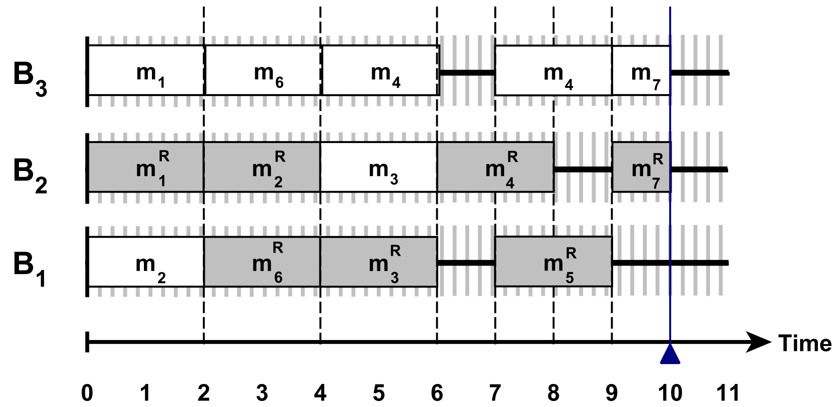


FIGURE 6.6 – Ordonnancement de données tolérant aux fautes et optimal avec réplication.

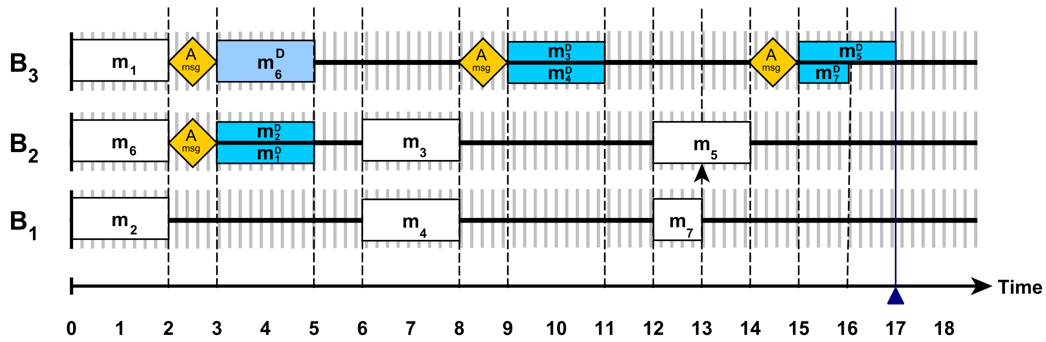


FIGURE 6.7 – Ordonnancement de données tolérant aux fautes et optimal avec désallocation.

Dans cet ordonnancement, a fin de réduire la longueur, la copie de sauvegarde désallouée m_2^D est chevauchée avec la copie de sauvegarde désallouée m_6^D, aux étapes 7 et 8 sur le bus B_2.

Un ordonnancement optimal tolérant aux fautes, avec des copies de sauvegarde uniquement répliquées, est représenté par la Figure 6.6. La longueur de cet ordonnancement est égale à 10 unités de temps.

Un ordonnancement optimal tolérant aux fautes avec des copies de sauvegarde uniquement désallouées est montré dans la Figure 6.7, la longueur d'ordonnancement est de 17 unités de temps. Dans cet ordonnancement les données m_2^D et m_4^D se chevauchent aux étapes 4 et 5 sur le bus B_1, m_3^D et m_4^D se chevauchent aussi à l'étape 6 sur le bus B_1. Il y a au moins un délai d'une unité de temps pour un message d'activation A_{msg} entre une donnée m_i et sa copie de sauvegarde désallouée m_i^D. Si la copie primaire est envoyée correctement, le message d'activation A_{msg} annule sa

copie de sauvegarde désallouée.

Dans cet exemple, l'ordonnancement optimal basé a la fois sur la réplication et la désallocation permet une réduction en longueur de 47.05% par rapport à un ordonnancement optimal qui se base uniquement sur la désallocation, et perd seulement 12,5% par rapport à une solution non tolérante aux fautes. Il permet également une réduction de la longueur d'ordonnancement de 10% par rapport à l'ordonnancement optimal qui utilise uniquement la réplication des données. Les performances sont améliorées de manière significative.

6.4 Approche proposée.

Dans cette section, nous définissons le problème d'ordonnancement comme un problème d'optimisation et nous utilisons la programmation linéaire pour trouver l'ordonnancement optimal et de longueur minimale, des dépendances de données avec leurs copies de sauvegarde afin de tolérer une seule faute de bus.

Comme notre solution est basée sur une approche hybride qui combine à la fois la redondance passive et active, nous utilisons deux types de copies de sauvegarde, répliquées et désallouées. L'ordonnancement des dépendances de données est modélisé par des variables binaires, ce qui permet de déterminer l'ordre des données. $M^{Pr}(i,t,j)$ est une variable binaire, tel que $M^{Pr}(i,t,j) = 1$ si et seulement si la copie primaire m_i^P de la donnée m_i est correctement envoyée sur le bus B_j à l'étape t. De même, les variables binaires $M^{Rep}(i,t,j)$ et $M^{Del}(i,t,j)$ sont utilisées pour les copies de sauvegarde répliquées (m_i^R) et désallouées (m_i^D).

$\forall i \in \{1,\cdots,n\}$, $\forall j \in \{1,\cdots,n_{bus}\}$, $\forall t \in \{1,\cdots,L_{ord}\}$

$$M(i,t,j), M^{Rep}(i,t,j), M^{Del}(i,t,j) \in \{0,1\} \tag{6.1}$$

L_{ord} est une limite supérieure de la longueur d'ordonnancement, n_{bus} est le nombre des bus.

La fonction objectif de notre problème linéaire est la minimisation de la longueur d'ordonnancement, elle peut être mathématiquement formulée comme suit :

$$Minimiser \sum_{t=1}^{L_{ord}} \sum_{j=1}^{n_{bus}} t * M^{Pr}(out,t,j). \tag{6.2}$$

Où m_{out} est un nœud fictif ajouter au modèle de données DAG, pour calculer la longueur totale de l'ordonnancement. (voir la Figure 6.8)

La minimisation de la longueur totale de l'ordonnancement est faite sous les contraintes suivantes :

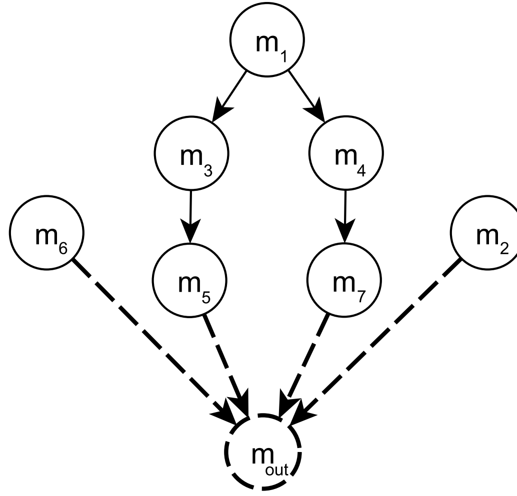


FIGURE 6.8 – Nœud fictif ajouter au DAG

1. **Contraintes de cartographie des données** :

   (a) La copie primaire de chaque donnée est ordonnancée une seule fois, $\forall i \in \{1, \cdots, n\}$,

   $$\sum_{t=1}^{L_{ord}} \sum_{j=1}^{n_{bus}} M^{Pr}(i,t,j) = 1. \quad (6.3)$$

   (b) Une seule copie de sauvegarde de chaque donnée est ordonnancée. $\forall i \in \{1, \cdots, n\}$,

   $$\sum_{t=1}^{L_{ord}} \sum_{j=1}^{n_{bus}} M^{Rep}(i,t,j) + M^{Del}(i,t,j) = 1. \quad (6.4)$$

   A tout moment, un bus B_x est utilisé soit pour la transmission d'une copie primaire d'une donnée ou de sa copie de sauvegarde répliquée. $\forall t \in \{1, \cdots, L_{ord}\}, \forall k \in \{1, \cdots, n_{bus}\}$

   $$\sum_{i=1}^{n} (M^{Pr}(i,t,j) + M^{Rep}(i,t,j)) \leq 1. \quad (6.5)$$

   (c) Dans une architecture multi-bus, pour tolérer une faute de bus, une seule copie de sauvegarde est utilisée. Elle peut être soit répliquée ou désallouée.

2. **Contraintes des dépendances de données**

   (a) Les copies de sauvegarde doivent respecter les mêmes relations de précédence que leurs copies primaires.

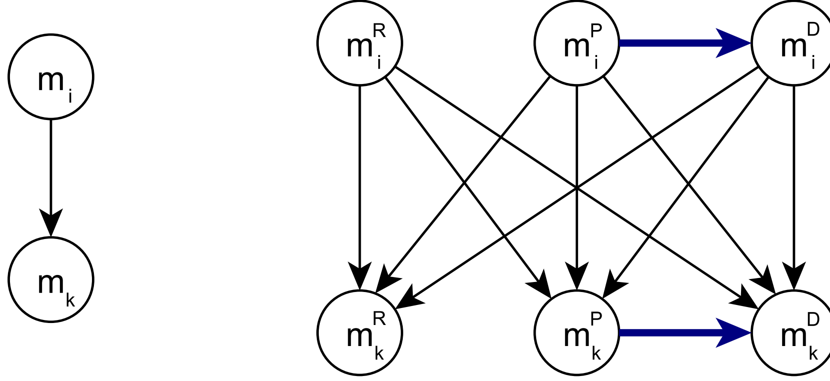


FIGURE 6.9 – Le nouveau DAG pour $m_i \rightarrow m_j$.

(b) $m_i \rightarrow m_k \in E_{Dd}$ signifie que la donnée m_k exiger la donnée m_i pour être calculer ; la donnée m_k ne peut pas être envoyée jusqu'à que la donnée m_i a été reçue et utilisée pour calculer m_k :
$\forall i, k \in \{1, \cdots, n\}$,

$$\sum_{t=1}^{L_{ord}} \sum_{j=1}^{n_{bus}} t * M^{Rep}(i,t,j) + Extc(m_i) \leq \sum_{t=1}^{L_{ord}} \sum_{j=1}^{n_{bus}} t * M^{Rep}(k,t,j). \quad (6.6)$$

Pour chaque $m_i \rightarrow m_k \in E_{Dd}$, un nouveau graphe orienté acyclique qui représente toutes les dépendances possibles entre les copies de sauvegarde et leurs copies primaires est généré. La Figure 6.9 présente la partie du DAG correspondant à la dépendance $m_i \rightarrow m_k$.

La Figure 6.10 présente le DAG complet pour le modèle de données de la Figure 6.3.

Pour notre exemple, nous avons choisi des copies de sauvegarde désallouées pour m_2 et m_6, et des copies de sauvegarde répliquées pour m_1, m_3, m_4, m_5 et m_7, La Figure 6.11 montre le DAG réduit pour ce choix.

3. **Contraintes de tolérance aux fautes :** la copie primaire et sa copie de sauvegarde ne devraient pas, en aucun cas, êtres attribués au même bus.
$\forall i \in \{1, \cdots, n\}, \forall j \in \{1, \cdots, n_{bus}\}$,

$$\sum_{t=1}^{L_{ord}} M^{Pr}(i,t,j) + M^{Rep}(i,t,j) + M^{Del}(i,t,j) \leq 1. \quad (6.7)$$

4. **Contraintes d'Exécution :** Pour chaque dépendance de donnée $t_x \rightarrow t_y \rightarrow t_z$, (comme le montre la Figure 6.12) la donnée m_b ne peut pas être ordonnancée tant que la donnée m_a n'a pas été reçue correctement par la tâche t_y et celle-ci complètement exécutée.

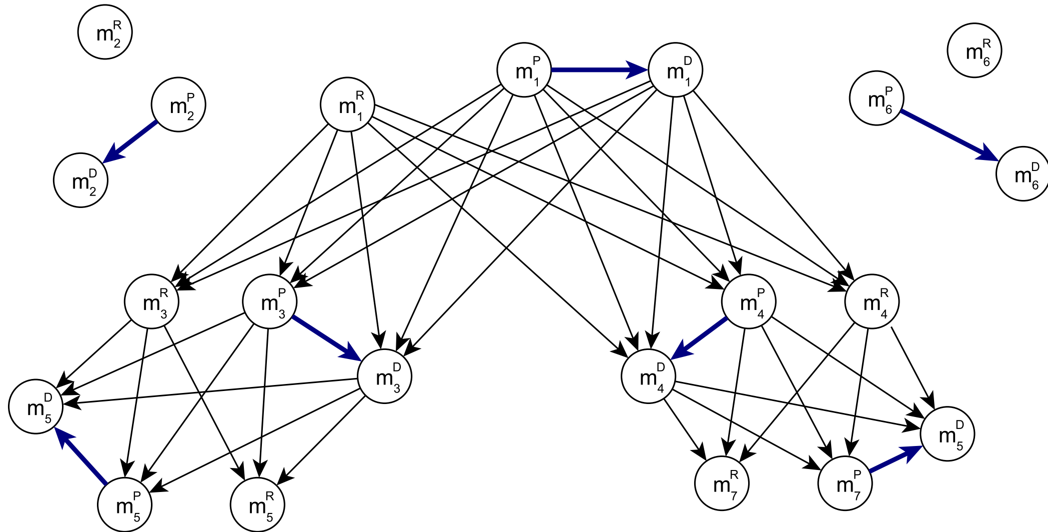


FIGURE 6.10 – Le DAG complet pour le modèle de données de notre exemple.

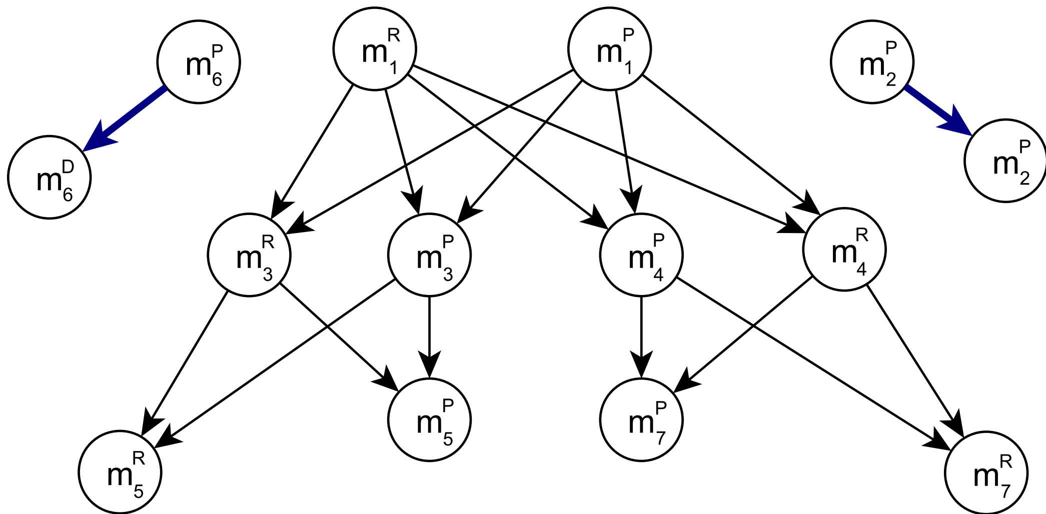


FIGURE 6.11 – Le DAG complet après choix du type des copies de sauvegarde.

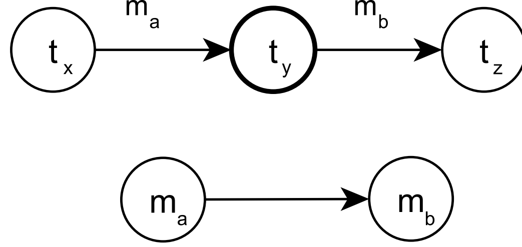


FIGURE 6.12 – Contraintes d'Exécution.

$\forall j \in \{1, \cdots, n_{bus}\}$,

$$t * M^{Pr}(i,t,j) + Ext(t_y) \leq t' * M^{Pr}(i,t',j) \tag{6.8}$$

La même chose peut être dite pour toutes les autres combinaisons, dans notre cas, nous pouvons compter neuf possibilités : (Pr, Pr), (Pr, Rep), (Pr, Del), (Rep, Pr), (Rep, Rep), (Rep, Del), (Del, Pr), (Del, Rep), (Del, Del).

5. **Contraintes de chevauchement et des ressources :**

   (a) Le chevauchement de deux copies de sauvegarde désallouées ou plus, doit respecter les conditions suivantes : leurs copies primaires doivent être affectés à des différents bus et doivent être complètement indépendant, (à savoir, elles n'ont pas de dépendances).

   (b) Seulement, les copies de sauvegarde désallouées, peuvent se chevaucher. S'il y a deux copies de sauvegarde désallouées $M^{Dl}(i,t,j)$ et $M^{Dl}(k,t,j)$ chevauchées sur le même bus B_j, alors leurs copies primaires sont affectées à deux autres bus différents du bus B_j.
   $\forall i, k \in \{1, \cdots, n\}$,

$$M^{Del}(i,t,j) = M^{Del}(k,t,j) = 1 \Longrightarrow \sum_{t=1}^{L_{ord}} M^{Pr}(i,t,j) + M^{Pr}(k,t,j) = 0. \tag{6.9}$$

   et,
   $\forall i, k \in \{1, \cdots, n\}, \forall j \in \{1, \cdots, n_{bus}\}$,

$$\sum_{t=1}^{L_{ord}} M^{Pr}(i,t,j) + M^{Pr}(k,t,j) \leq 1 \tag{6.10}$$

La solution ainsi obtenue est le meilleur résultat, mais comme le problème est NP-hard, cette formulation prend généralement beaucoup de temps, et dans certains cas on ne peut pas trouver une solution réalisable dans un temps acceptable. C'est

pourquoi nous proposons notre deuxième solution, basée sur une heuristique dite *FTA-RD*. Les objectifs de cette heuristique sont de deux ordres, d'abord, maximiser la fiabilité du système ; D'autre part, minimiser la longueur de l'ensemble de l'ordonnancement généré en présence et en absence de fautes.

Notre algorithme d'ordonnancement est une heuristique basée liste d'ordonnancement, qui planifie une opération à chaque étape. L'entrée de notre algorithme est une instance du modèle de tâches G_{Task}=(E_{Task},E_{Dd},Ext) et le modèle d'architecture G_{Arch}=(E_{Proc},E_{Bus}), le délai du message d'activation A_{msg}. Il génère en sortie un ordonnancement statique distribué d'un modèle de tâche donné sur un modèle d'architecture donné, qui minimise l'exécution du système, et tolère au plus une seule faute de bus de communication. Il est évident que la complexité temporelle de l'algorithme *FTA-RD* est polynomiale.

L'heuristique *FTA-RD* (Algorithme 4) fonctionne comme suit :

— Tout d'abord, elle examine les dépendances de données, une par une, pour déterminer quand et sur quel bus chacune d'elles sera envoyée. Elle trouve le premier bus disponible et le moins utilisé (l'utilisation d'un bus est mesurée par le temps total de transmission de toutes les données attribuées à ce bus) ; Par conséquent, cette affectation, ainsi faite, assure l'équilibrage des charges.

— Une fois la copie primaire est ordonnancée, l'heuristique essaye d'ordonnancer la copie de sauvegarde, tout d'abord elle tente de la faire chevaucher avec une copie de sauvegarde désallouée existante, sinon, elle ordonnance une nouvelle copie de sauvegarde répliquée ou désallouée selon la valeur de $BACKuP_{type}$.

— Enfin, afin de réduire la longueur d'ordonnancement, le type de la copie de sauvegarde pour la dernière donnée est redéfini.

6.5 Simulations, résultats et discussion

Dans cette section, nous présentons les résultats des simulations, nous comparons l'algorithme d'ordonnancement proposé *FTA-RD* avec une solution basée sur la formulation de programmation linéaire obtenu avec *lpsolve IDE*, et avec des solutions basées uniquement sur la réplication ou uniquement sur la désallocation, la comparaison est faite par rapport à la longueur d'ordonnancement.

Nous avons appliqué la *FTA-RD* à un exemple d'un graphe d'architecture composé de cinq processeurs et trois bus. Les taux d'échec des processeurs sont respectivement 10^{-5}, 10^{-5}, 10^{-4}, 10^{-6} et 10^{-6}, et le taux d'échec des bus : SAM_{MP1}, SAM_{MP2} et SAM_{MP3} sont respectivement de 10^{-6}, 10^{-5} et 10^{-4}. Les modèles de tâches utilisés sont ceux des benchmark *DSP* de la *DSPstone* [102]. Le nombre de tâches et les dépendances de données dans chaque benchmark est listé dans le tableau 6.1.

Algorithme 4 Algorithme d'ordonnancement *FTA-RD*.

Entrées : G_{Task}=(E_{Task},E_{Dd},Ext), G_{Arch}=(E_{Proc},E_{Bus}), n_{bus}, $|A_{msg}|$.
Sorties : Un ordonnancement fiable et tolérant aux fautes, $BACKuP_{type}(m_i)$.

Début

1. Initialiser la liste des données déjà ordonnancées : $M_{ord} := \emptyset$;
2. $BACKuP \leftarrow DEAL$;
3. Générer le modèle de données G_{Data}=(M,E_{Dd},$Extc$) à partir du modèle de tâches G_{Task} ;
4. Créer une file d'attente M_{cand} des données candidates à l'ordonnancement ;
5. Calculer le nouveau temps de transfert des données ;
6. **Tant Que** ($M_{cand} \neq \emptyset$) **faire :**
   A) $m_i \leftarrow head[M_{cand}]$;
   B) Ordonnancer m_i sur le bus B_j le plutôt possible, B_j est le bus avec le taux d'utilisation le plus petit ;
   C) $M_{ord} := M_{ord} \cup \{m_i\}$;
   D) **Si** (m'_i peut se chevaucher avec une autre copie de sauvegarde désallouée) **Alors**
      i. Ordonnancer et chevaucher la copie de sauvegarde désallouée m'_i de m_i.
      ii. $M_{ord} \leftarrow M_{ord} \cup \{m'_i\}$.
      iii. $BACKuP_{type}(m_i) \leftarrow DEAL$;
      iv. $BACKuP \leftarrow DEAL$;

      Sinon

      Switch ($BACKuP_{type}(m_i)$) **Faire**

      - **case** DEAL
        · Ordonnancer une nouvelle copie de sauvegarde désallouée m'_i de m_i sur le Bus B_k le plutôt possible, B_k est le Bus avec le taux d'utilisation le plus petit et il est différent de B_j ($B_k \neq B_j$)
        · $M_{ord} \leftarrow M_{ord} \cup \{m'_i\}$;
        · $BACKuP_{type}(m_i) \leftarrow DEAL$;
        · $BACKuP \leftarrow REPL$;
        · **break** ;
      - **case** REPL
        · réordonnancer une nouvelle copie de sauvegarde répliquée m'_i de m_i sur le bus B_k le plutôt possible, B_k est le Bus avec le taux d'utilisation le plus petit et il est différent de B_j ($B_k \neq B_j$) ;
        · $M_{ord} \leftarrow M_{ord} \cup \{m'_i\}$;
        · $BACKuP_{type}(m_i) \leftarrow REPL$;
        · $BACKuP \leftarrow DEAL$;
        · **break** ;

      Fin-Switch

      Fin-Si

   Fin Tant que
7. Redéfinir le type de la copie de sauvegarde pour la dernière donnée m_i ;

Fin

Benchmark	Nombre de tâches	Nombre des dépendances de données
FIR filter	12	8
2 - motiv	12	9
2 iir filter	17	13
2 - deq filter	24	20
IIR biquad section	36	31
2 - rls - lat	38	33
8latiir	46	38

TABLE 6.1 – Informations des benchmarks

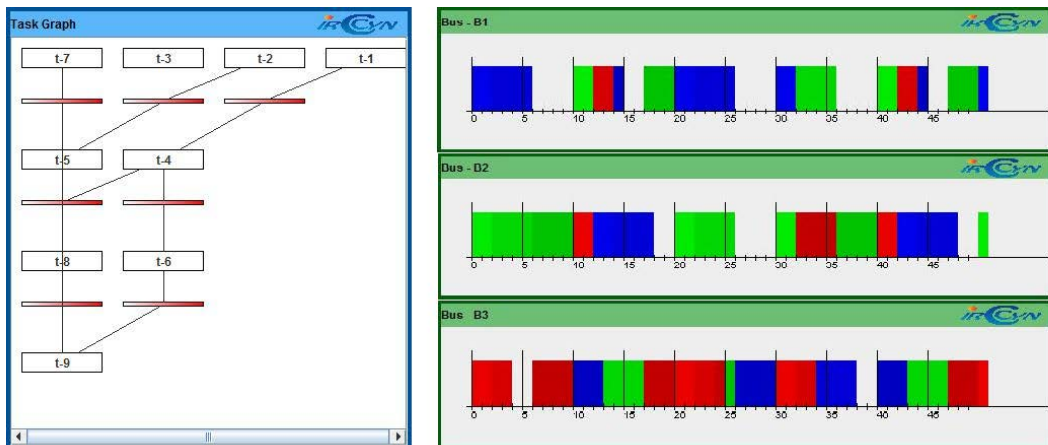


FIGURE 6.13 – Modèle de tâches et affectation des bus.

Nous utilisons *STORM* [103] pour calculer la longueur d'ordonnancement des données, *STORM* est un outil de simulation pour les ordonnancements temps réel multiprocesseur. A partir d'une spécification des caractéristiques : d'une architecture logicielle (les tâches à planifier), d'une architecture matérielle (les ressources pour la mise en œuvre de ces tâches) et le choix d'une politique d'ordonnancement, l'outil simule l'exécution de ces tâches sur ces ressources en fonction des règles de cette politique. La Figure 6.13 montre le modèle de tâche et l'affectation des bus dans l'éditeur de *STORM*.

Les résultats montrent que l'heuristique *FTA-RD* avec les deux types copies de sauvegarde répliquées et désallouées permet d'avoir des résultats plus performants que l'heuristique utilisant uniquement des copies de sauvegardes répliquées ou uniquement des copies de sauvegardes désallouées, dans tous les benchmarks, nous pouvons voir que l'heuristique *FTA-RD* réduit la longueur d'ordonnancement par 6,19 % en moyenne par rapport à l'heuristique qui utilise seulement la réplication (voir la Figure 6.14), et par 19,29 % en moyenne par rapport à celle utilisant uniquement la désallocation (voir la Figure 6.15). Nous pouvons voir aussi que notre

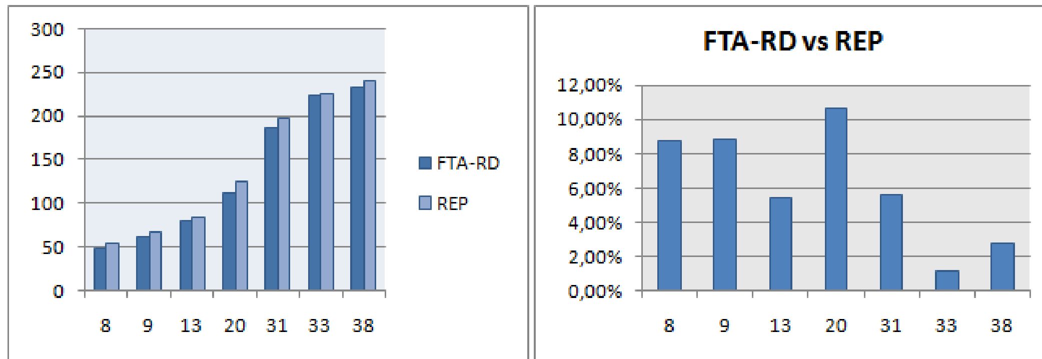


FIGURE 6.14 – Longueur d'ordonnancement de : *FTA-RD* vs *FTA-R*.

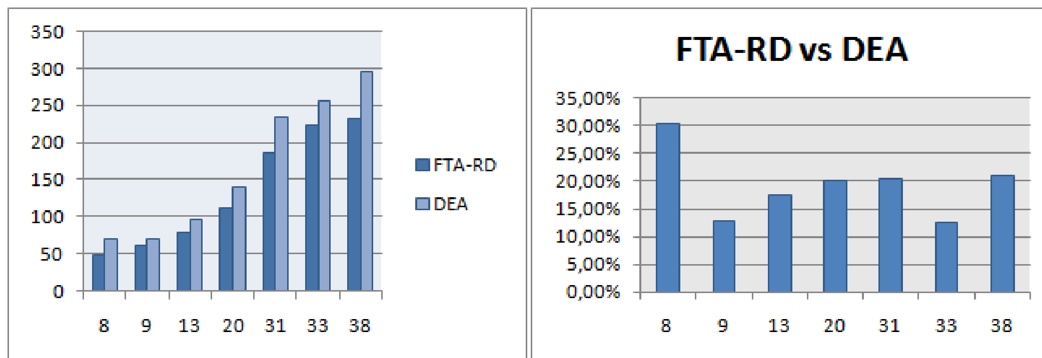


FIGURE 6.15 – Longueur d'ordonnancement de : *FTA-RD* vs *FTA-D*.

heuristique perd 8,41 % en moyenne dans la longueur d'ordonnancement par rapport à la solution optimale obtenue par la formulation par programmation linéaire (voir la Figure 6.16).

D'après les résultats, nous pouvons déduire que :

1. Lorsque le nombre de dépendances de données est petit, l'algorithme qui utilise uniquement les copies de sauvegarde répliquées est plus efficace que celui qui utilise des copies de sauvegarde désallouées. Ceci est expliqué par le fait qu'avec un nombre de dépendances de données faible, la possibilité de chevauchement est également faible.
2. Un nombre de dépendances de données plus important, implique plus de copies de sauvegarde désallouées qui peuvent se chevaucher, ce qui réduit automatiquement la longueur de l'ordonnancement.
3. Également, comme l'activation de la copie de sauvegarde est conditionnée par la réception du message d'activation. Le retard dû à ce message, a impérativement un effet négatif sur la durée de l'ordonnancement.

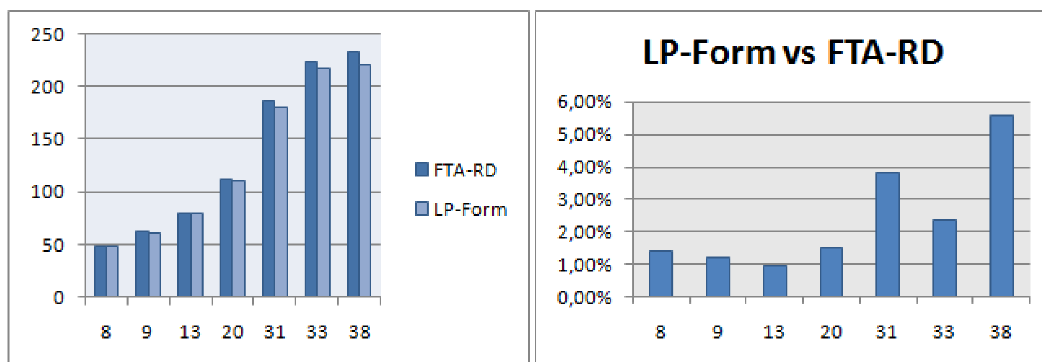


FIGURE 6.16 – Longueur d'ordonnancement de : *FTA-RD* vs La solution de la programmation linéaire .

4. L'utilisation des deux types des copies de sauvegarde répliquées et désallouées permet d'avoir de meilleurs résultats, que dans le cas de l'utilisation d'un seul type des copies de sauvegarde (uniquement répliquées ou uniquement désallouées).
5. Les résultats obtenus en utilisant la formulation par programmation linéaire, mis en œuvre dans *lpsolve IDE* sont meilleurs que ceux obtenus par l'heuristique *FTA-RD*, le problème est que dans de nombreux cas, le temps de calcul de cette formulation est trop long pour produire des résultats. Par exemple, il n'a pas généré de résultat dans un jour, pour les quatre benchmarks suivants : volts, 5-4latiir, 10-4latiir, 20-4latiir. Cependant l'heuristique *FTA-RD* produit des résultats dans un délai polynomial.

6.6 Conclusion

Dans ce chapitre, nous avons étudié le problème de la tolérance aux fautes des communications dans les systèmes embarqués temps réel et nous avons proposé une solution d'ordonnancement mise en œuvre pour les architectures multi-bus basées sur la redondance logicielle.
Nous avons proposé une nouvelle heuristique, appelée *FTA-RD*, qui produit automatiquement un ordonnancement statique tolérant aux fautes ; les dépendances de données sont représentées par des graphes acycliques orientés, avec deux types de copies de sauvegarde répliquées et désallouées sous l'hypothèse qu'il y aura au plus une seule faute de bus de communication. Les simulations montrent une amélioration significative par rapport aux algorithmes avec un seul type de copies de sauvegarde.

Chapitre 7

Approche d'ordonnancement fiable de communication temps réel basée sur les chemins de sauvegarde fragmentés dans les réseaux multi-sauts

Sommaire

7.1 Introduction **106**
7.2 Modèle de fautes **108**
7.3 Présentation du problème **108**
7.4 RBF-FB : Un algorithme d'ordonnancement tolérant aux fautes des chemins de communications **109**
7.4.1 La sauvegarde fragmentée 110
7.4.2 Fonctionnement de la sauvegarde fragmentée 112
7.4.3 Algorithme de la sauvegarde fragmentée 114
7.4.4 Fonction de sélection des segments de sauvegarde avec pondération de l'apport temps réel 117
7.4.5 Un exemple de la sauvegarde fragmentée 118
7.5 Algorithme d'ordonnancement **120**
7.6 Recouvrement des fautes(Reprise sur faute) **121**
7.7 Délai et Scalabilité **123**
7.7.1 Délai 123
7.7.2 Scalabilité 124
7.8 RSVP **124**
7.9 Conclusion **125**

Dans ce chapitre nous nous intéressons au problème d'ordonnancement lié aux architectures matérielles munies d'un réseau de communications multi-sauts, notre solution étant dédiée aux réseaux non fortement couplés, elle se base sur les chemins de sauvegarde fragmentés, ce qui permet une fiabilité maximale et une exploitation efficace des ressources réseaux.

7.1 Introduction

En raison des défaillances de composants réseaux, qui ne peuvent être assurées dans les services de datagramme traditionnels, les applications temps réel distribuées exigent en plus de garder les délais à des niveaux acceptables, la tolérance de fautes, ces applications exigent des garanties précises sur les délais de récupération des données transmises. Beaucoup de solutions ont été proposées pour le cas des systèmes à base de réseaux fortement couplés [76] [104], avec leurs liaisons point-à-point, chaque processeur étant directement connecté aux autres processeurs du système. Dans le cadre de notre travail nous nous intéressons au cas des réseaux non fortement couplés, où la transmission des données entre deux processeurs n'est pas directe et peut se faire en passant par un ou plusieurs autres nœuds avant d'arriver à destination.

Plusieurs systèmes proposés tentent de garantir le recouvrement des fautes de communications en temps opportun et de manière efficace. Les plus élaborées de ces méthodes [68] [105] [106] [74] [107] [108] [109] [105] [110], se basent sur la pré-réservation des ressources réseaux de rechange le long d'un autre chemin dit "canal de sauvegarde", en plus de chaque chemin de communications primaire. Généralement le canal de sauvegarde est routé sur un chemin disjoint du chemin principal. Pour assurer un recouvrement garanti et rapide, les ressources de rechange réservées pour un canal de sauvegarde ne sont activées que lorsque le chemin principal est cible de fautes. Lorsqu'elles ne sont pas utilisées, ces ressources de rechange peuvent être utilisées pour assurer d'autres trafics de moindre priorité (comme le trafic "best effert" ou autre trafic non temps réel).

La mise en œuvre des chemins de sauvegarde peut être réalisée de deux façons différentes. Dans la première, les ressources de sauvegarde dans le voisinage du nœud défaillant sont utilisées pour ré-acheminer le trafic. Cette méthode de détour locale [106], [107] conduit à une utilisation inefficace des ressources, puisque le chemin de sauvegarde obtenu est souvent de longueur plus importante alors que dans le deuxième cas, un chemin de sauvegarde de bout en bout [110], [105], (Figure 7.1), est mis en œuvre par la simple technique de sauvegarde fragmentée qui définit un chemin de sauvegarde complètement séparé du primaire et qui s'étend depuis la source jusqu'à la destination.

Mais cette technique présente aussi certains inconvénients :

1. Comme le chemin de sauvegarde doit être réalisé après le chemin primaire qui lui est totalement disjoint, sa mise en œuvre est délicate, puisque le respect du chemin primaire aux contraintes limite la réalisation des chemins de sauvegarde.
2. Tout comme le chemin primaire le chemin de sauvegarde doit respecter certaines contraintes et satisfaire toutes les exigences de qualité de service (comme

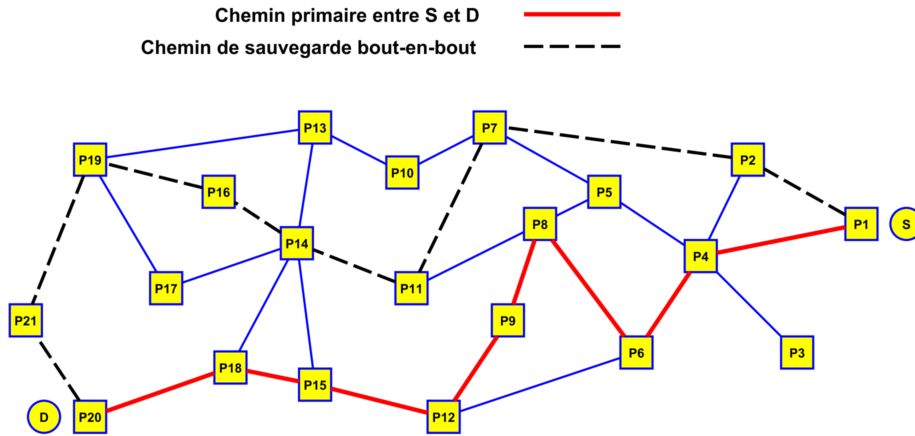


FIGURE 7.1 – Le chemin de sauvegarde de bout-en-bout

par exemple le délai), cela engendre une restriction très difficile, même quand il y a suffisamment de ressources réseau.

Dans ce chapitre, nous proposons une approche d'ordonnancement pour un système temps réel embarqué à base de réseaux de communications multi-sauts, tolérants aux fautes de communications et basée sur les chemins de sauvegarde fragmentés.

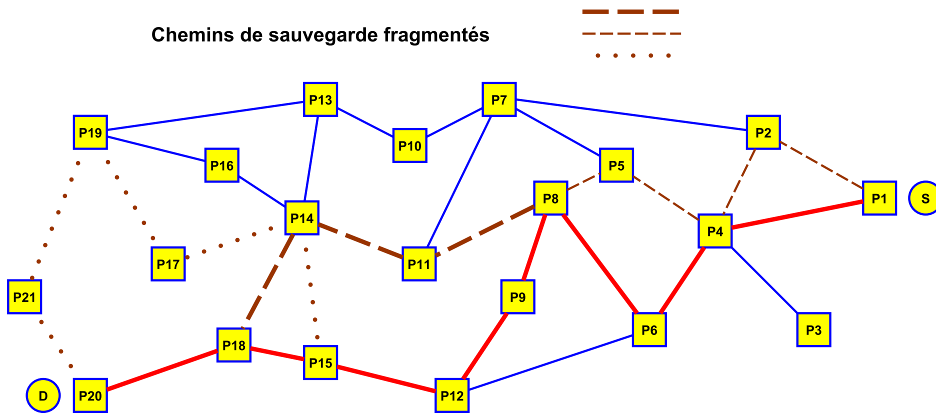


FIGURE 7.2 – Les chemins de sauvegarde fragmentés

L'approche que nous proposons (Figure 7.2) se base sur l'idée présentée dans [111] [70], ce que nous apportons dans ce chapitre est une amélioration adaptée aux systèmes temps réel embarqués pour permettre d'avoir un ordonnancement tolérant aux fautes des communications. La solution est basée sur la création prédéfinie de plusieurs chemins de sauvegarde fragmentés, chacun couvrant une partie contiguë du chemin primaire, elle permet de tolérer une ou plusieurs fautes temporelles des

processeurs et des liens de communications. Il s'agit de résoudre le problème de la recherche d'un ordonnancement des composants logiciels du modèle de tâches G_{Task} sur les composants matériels d'un modèle d'architecture G_{Arch}, qui tolère Nf_{Proc} fautes de processeurs de calcul et Nf_{chm} [1] de chemin de communication, tout en minimisant la longueur de l'ordonnancement afin de satisfaire la contrainte temps réel C_{Rt}.

Notre approche propose aussi, une fonction de sélection de chemins fragmentés qui permet au moment de l'occurrence d'une faute le choix de routage des données par plusieurs chemins fragmentés. Afin de bien présenter notre solution, voici tout d'abord le modèle de fautes que nous considérons dans ce chapitre.

7.2 Modèle de fautes

Dans ce chapitre, nous utilisons les mêmes hypothèses de défaillances que nous avons défini dans la section 1.2 du chapitre 4 à l'exception de :

— Nous remplaçons les fautes de bus de communications par des fautes de chemins de communications. (c-à-d nous remplaçons : le nombre Nf_{Bus} de fautes de bus de communications par le nombre Nf_{chm} de fautes de chemins de communications).

— Une faute d'un lien de communication est la faute de tous les chemins de communications auxquels il appartient.

7.3 Présentation du problème

Le but de ce chapitre est de résoudre le problème de la recherche d'un ordonnancement des composants logiciels du graphe d'algorithme G_{Task} sur les composants matériels du graphe d'architecture G_{Arch}, qui doit tolérer des fautes matérielles des processeurs et des chemins de communications, tout en minimisant la longueur de l'ordonnancement dans le but de satisfaire la contrainte temps réel C_{Rt} en absence et en présence de fautes. Ce problème peut être formalisé comme suit :

Données du problème :

- Une architecture matérielle hétérogène G_{Arch} composée d'un ensemble E_{Proc} d'opérateurs de calcul (processeurs) et d'un ensemble L de liens de communications :

$$E_{Proc} = \{\ldots, P_i, \ldots, P_j, \ldots\}, L = \{\ldots, l_{P_i,P_n}, \ldots\}$$

  Pour chaqu'un des deux processeurs P_i et P_j non directement connectés, on défini un ensemble $SetC_{(P_i,P_j)}$ de chemins relient ces deux processeurs, chaque

1. Nf_{chm}= Nombre de chemins défaillants.

chemin $C \in SetC$ est composé d'un sous ensemble de L reliant les deux processeurs P_i et P_j : $C_{P_i,P_j} \in SetC_{(P_i,P_j)} = \{l_{P_i,P_n} \ldots, l_{P_x,P_y}, \ldots l_{P_m,P_j}\}$
La longueur d'un chemin $Long(C_{P_i,P_j})$ est égale aux nombre de liens qui composent ce chemin. (d'autres définitions peuvent êtres données à cette mesure comme par exemple la somme des coûts des liens qui le composent)

- Un modèle de tâches G_{Task} composé d'un ensemble E_{Task} de tâches et un ensemble E_{Dd} de dépendances de données :

$$E_{Task} = \{\ldots, t_i, \ldots\}, E_{Dd} = \{\ldots, (t_i \rightarrow t_j), \ldots\}$$

- Des caractéristiques d'exécution C_{EExe} et C_{CExe} des composants de G_{Task} sur les composants de G_{Arch},
- Un ensemble de contraintes matérielles C_{Had},
- Une contrainte temps réel C_{Rt},
- Un critère de minimisation de la longueur de l'ordonnancement,
- Un nombre Nf_{Proc} de fautes d'opérateurs de calcul et un nombre Nf_{chm} de fautes de chemins de communications, qui peuvent causer la défaillance du système,

7.4 RBF-FB : Un algorithme d'ordonnancement tolérant aux fautes des chemins de communications

Le problème que nous venons d'exposer est un problème NP-difficile. Pour le résoudre en un temps polynomial on propose une nouvelle technique dite *RBF-FBC*[2] qui essaye de trouver une solution proche de la solution optimale, rappelons que notre objectif n'est pas uniquement, avoir un ordonnancement de longueur minimale mais plutôt un qui respecte la contrainte temps réel C_{Rt}.

Notre problème peut être résolu par l'utilisation des techniques logicielles, et plus spécifiquement les techniques de redondance puisque nous visons les systèmes embarqués. L'approche que nous proposons permet d'avoir une grande fiabilité et une tolérance aux fautes en exploitant les de deux techniques suivantes :

① Redondance Passive-Passive des opérations avec préemption .

② Les Sauvegardes fragmentés des chemins de communications.

L'heuristique *RBF-FB* que nous proposons dans ce chapitre permet de tolérer Nf_{Proc} fautes de processeurs et Nf_{chm} fautes de chemins de communications, nous combinons les deux techniques ; la redondance Passive-Passive des opérations avec

2. RBF-FB = *R*eliable *B*us *F* fault-tolerant based *F*regmented *B*uckup

préemption pour la tolérance des fautes de processeurs et les sauvegardes fragmentées pour la tolérance des fautes du réseau de communications. Le but est de maximiser la fiabilité du système tout en minimisent la longueur de l'ordonnancement généré et ça dans les deux cas, avec ou sans défaillances.

Par l'utilisation de *RBF-FB*, et plus spécifiquement la nouvelle approche proposée pour la mise en œuvre des chemins de sauvegarde fragmentés nous visons les objectifs suivants :

1. Une fiabilité des communications supérieure : Cela est dû à des chemins primaires qui ont une sauvegarde fragmentée et n'ont pas une sauvegarde de bout en bout.
2. Une amélioration de l'utilisation des ressources réseau : c'est parce que les sauvegardes fragmentées sont généralement plus courtes que les sauvegardes de bout en bout et ont besoin de moins de ressources de rechange. Par ailleurs, les chemins de sauvegarde plus courts conduisent à une agrégation de ressources plus efficace grâce à un multiplexage de sauvegarde.
3. Une meilleure garantie de la QoS : Une sauvegarde fragmentée peut comprendre plusieurs sauvegardes, dont chacun couvre une partie du chemin primaire plutôt que sur toute sa longueur. Cela permet une récupération plus rapide après faute et un contrôle plus raffiné de la tolérance de fautes surtout pour les chemins primaires de longueur importante.

Il est possible de construire des sauvegardes fragmentées optimisés pour différents objectifs tels que l'amélioration de l'utilisation des ressources réseaux par rapport à un temps de récupération important après échec. Dans ce chapitre, nous proposons des algorithmes spécifiques pour la tolérance aux fautes des processeurs et du réseau de communications par la construction de sauvegardes fragmentées qui sont optimales en ce qui concerne l'utilisation des ressources réseaux.

7.4.1 La sauvegarde fragmentée

L'approche que nous proposons dans ce chapitre, est basée sur les sauvegardes fragmentées dans un mode entièrement distribué et évolutive. L'idée est que nous trouvons des sauvegardes pour seulement une partie du chemin primaire et non pas pour le chemin entier.

Le chemin primaire est considérée comme composée de petits chemins contigus, que nous appelons fragments primaires comme le montre la Figure 7.3. Nous trouvons un ou plusieurs chemins de sauvegarde indépendants pour chaque fragment primaire (voir Figure 7.4), que nous appelons un fragment de sauvegarde. Par sauvegarde fragmentée nous nous référons à ces fragments de sauvegarde pris ensemble.

Considérons la Figure 7.3, où un réseau de nœuds et de liens sont représentés, et où une connexion fiable est établie entre les nœuds P_1 (marqué Source (S)) et P_{10}

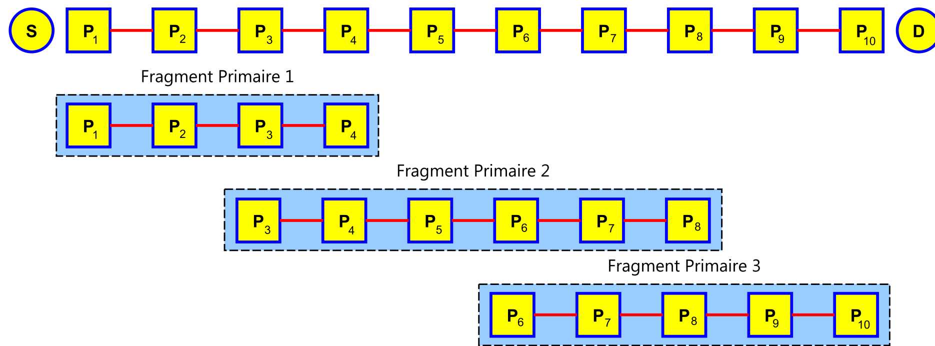


FIGURE 7.3 – Fragments primaires du chemin primaire

(Destination (D)). Le chemin principal s'étend à travers les liens reliant les nœuds : $P_1(S)$, P_2, P_3, P_4, P_5, P_6, P_7, P_8, P_9 et $P_{10}(D)$, sur ce chemin principal on a définit trois fragments primaires :

— $Frag_{(p)1}$=$\{l_{P_1,P_2}, l_{P_2,P_3}, l_{P_3,P_4}\}$.

— $Frag_{(p)2}$=$\{l_{P_3,P_4}, l_{P_4,P_5}, l_{P_5,P_6}, l_{P_6,P_7}, l_{P_7,P_8}\}$.

— $Frag_{(p)3}$=$\{l_{P_6,P_7}, l_{P_7,P_8}, l_{P_8,P_9}, l_{P_9,P_10}\}$.

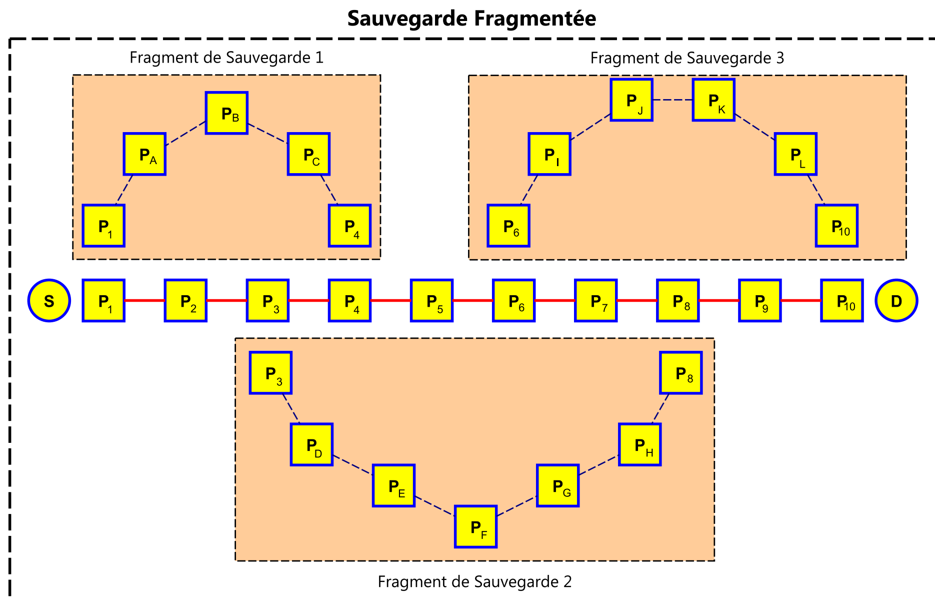


FIGURE 7.4 – Fragments de la sauvegarde des fragments primaires

Il y a trois fragments de sauvegarde impliqués dans cette configuration (voir Figure 7.4) ; Le premier : $P_1 - P_A - P_B - P_C - P_4$, le deuxième : $P_3 - P_D - P_E - P_F - P_G - P_H - P_8$ et le troisième et dernier chemin : $P_7 - P_I - P_J - P_K - P_L - P_{10}$. Ces trois chemins de sauvegarde "sauvegarde" les fragments primaires (donnés par

les nœuds et les liens qui les relient). Notez que nous avons la condition que les fragments primaires devraient toujours se chevaucher d'au moins un lien. (c'est le cas du lien l_{P_3,P_4} pour les deux fragments primaires $Frag_{(p)1}$ et $Frag_{(p)2}$, et les liens l_{P_6,P_7} et $l_{P_7-P_8}$ pour les deux fragments $Frag_{(p)2}$ et $Frag_{(p)3}$.

7.4.2 Fonctionnement de la sauvegarde fragmentée

Supposons qu'il y a une faute sur l'un des liens (une faute de lien est aussi par définition la faute de l'un des deux nœuds qui compose le lien) qui composent le chemin principal. Il faut noter ici, que comme un tel nœud est toujours sur un (ou plusieurs) fragment primaire, donc il y aura toujours un (ou plusieurs) fragment de sauvegarde qui tolère les fautes de ce nœud.

Par exemple, sur la Figure 7.5, lorsque le lien l_{P_2,P_3} est cible de faute, le premier fragment de sauvegarde est activé, et le trafic est acheminé sur ce nouveau chemin (composé du premier fragment de sauvegarde et du reste du chemin primaire). S'il existe plus d'un fragment de sauvegarde pour un même lien (comme c'est le cas des liens : l_{P_3,P_4}, l_{P_6,P_7} et l_{P_7,P_8}), nous pouvons choisir le fragment qui donne les meilleurs performances, pour cela on peut utiliser une nouvelle fonction de sélection des chemins de sauvegarde basée sur la pondération des chemins que nous proposons dans la suite de ce chapitre .

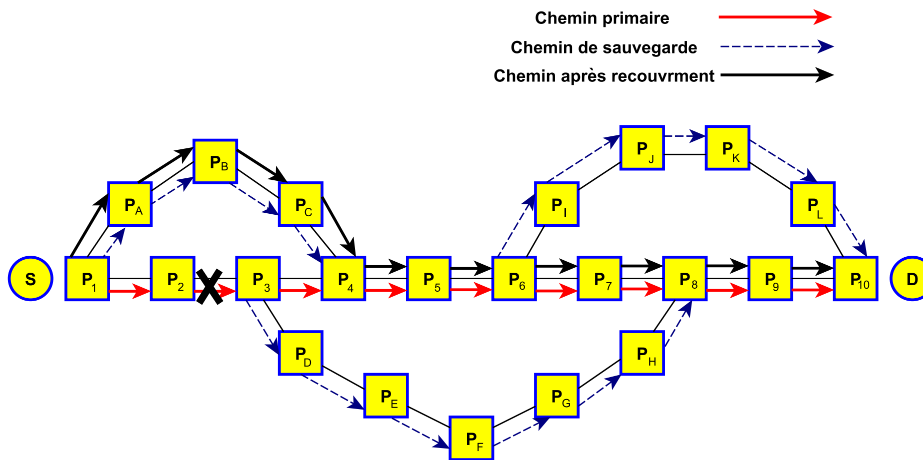


FIGURE 7.5 – Faute d'un lien de communication sur le chemin primaire

Nous allons illustrer le principal avantage des sauvegardes fragmentées avec la Figure 7.6. Ici, une connexion fiable est demandée entre les deux processeurs P_1 et P_{25}, dans une architecture basée sur un réseau maillé (de dimension 4*4). Le chemin primaire comme indiqué sur la figure est composé de trois fragments primaires suivants : $\{l_{P_1,P_2}, l_{P_2,P_3}\}$, $\{l_{P_3,P_8}, l_{P_8,P_{13}}, l_{P_{13},P_{18}}, l_{P_{18},P_{23}}\}$ et $\{l_{P_{23},P_{24}}, l_{P_{24},P_{25}}\}$.

Nous constatons qu'avec ce chemin primaire ainsi établi, la construction d'un chemin de sauvegarde de bout en bout (end-to-end) est impossible (puisque le chemin de sauvegarde doit être complètement disjoint du chemin primaire). Mais l'établissement de chemins de sauvegarde fragmentés est possible. (un bon exemple est donnée dans [70] sur une topologie réelle comme le USANET à 28 nœuds.)

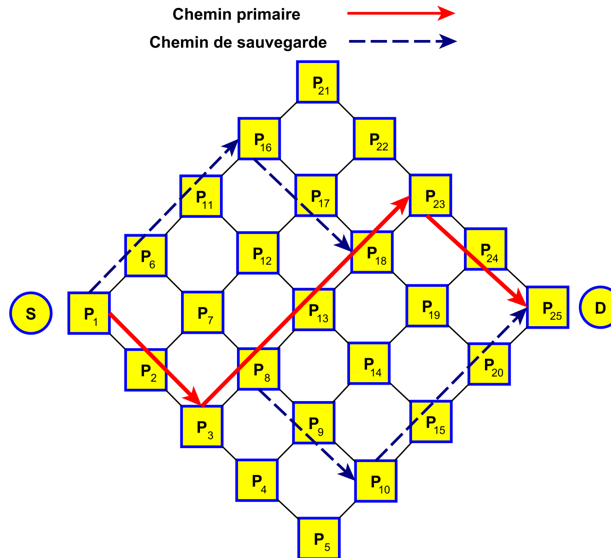


FIGURE 7.6 – Avantage de l'approche fragmentée

Maintenant, nous définissons les critères exacts de sélection de l'ensemble des liens qui constituent les fragments de sauvegarde. Notez que notre choix se fait sur deux choses :

1. Le nombre de fragments de sauvegarde utilisés.
2. Les liens qui constituent chaque fragment de sauvegarde.

Nous définissons la taille d'un fragment donné comme la somme des nombres de sauts ou des liens qui le constituent (ou toute mesure appropriée), ce qui exclut tout lien à partir du chemin principal, et on définit la taille globale d'une sauvegarde fragmentée S_{Frag}, comme la somme des tailles des fragments de sauvegarde qui la constitue :

$$Long(S_{Frag}) = \sum_{i=1}^{i=n} Long(Frag_{(s)i}) \tag{7.1}$$

où n est le nombre de fragments qui constitue la sauvegarde fragmentée. L'objectif principal de notre algorithme est le choix d'une sauvegarde fragmentée avec

la longueur la plus petite. En d'autres termes, nous essayons de minimiser le paramètre : $Long(S_{Frag})$. La minimisation de la longueur de la sauvegarde fragmentée nous permet de minimiser la consommation des ressources réseaux supplémentaires.

Notez que nous avons principalement concentré nos efforts sur la question de réservation en temps réel des ressources réseaux pour assurer les communications et nous ne traitons pas les questions de l'ordonnancement des messages ou de la gestion des fils d'attente des messages au niveau de chaque processeur. Ici, on peut supposer sans risque que le modèle qui précise les fils d'attente et la programmation (supposé dans [106]) peut être aussi appliquer dans notre cas.

Il faut noter que les sauvegardes fragmentées sont généralement plus économiques au niveau de la consommation des ressources que les sauvegardes de bout en bout, qui sont des cas particuliers des sauvegardes fragmentées (avec nombre de fragments primaire égale à 1), cela est expliqué par la fait que les sauvegardes fragmentées ont moins de contraintes par rapport au sauvegardes de bout en bout. Par conséquent, il y a plus de liberté pour choisir les chemins de sauvegarde (qui peut impliquer plusieurs fragments) que dans le cas de la sauvegarde de bout en bout.

7.4.3 Algorithme de la sauvegarde fragmentée

Nous modélisons la topologie réseau par un graphe orienté et pondéré $G_{net}(V, E)$. Chaque noeud n dans le réseau est représenté par un sommet v unique dans l'ensemble des sommets V, tandis que chaque liaison l allant du nœud n_1 (v_1) au nœud n_2 (v_2) avec un coût c est représenté dans le graphe G_{net} par un arc e_1 allant de v_1 vers v_2 avec un poids égale à c. (Une liaison duplex (bi-directionnelles) est remplacé par deux liens, dans les deux sens avec le même coût).(voir Figure 7.7)

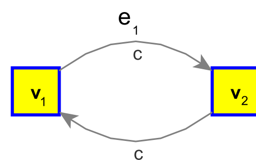


FIGURE 7.7 – Graphe G_{net}.

Les deux sommets S et D désignent les nœuds source et de destination dans le réseau. On note le chemin primaire (déjà établi) dans le graphe G_{net} avec une séquence de sommets $P = \{S, i_1, \cdots, i_n, D\}$. Nous définissons $Succ_P(x, y)$ pour indiquer que sommet x est le successeur du sommet y.

Afin de trouver la sauvegarde fragmentée la plus courte, l'algorithme que nous proposons opère en trois phases. Dans la première phase, on génère un graphe modifié $G^*_{net}(V, E')$ du graphe $G_{net}(V, E)$ sur le même ensemble V de sommets. Dans la deuxième phase, nous trouvons le plus court chemin entre S et D dans le nouveau

graphe G^*_{net}, ce dernier est utiliser au troisième phase pour obtenir les fragments de la sauvegarde fragmentée avec le coût minimum dans le graphe G_{net}.

① Phase 1 : Génération du Graphe modifié $G^*_{net}(V, E')$ à partir de $G_{net}(V, E)$: Les arrêtes du graphe $G_{net}(V, E)$ sont modifiés dans l'ordre indiqué dans l'Algorithme 5.

Algorithme 5 Algorithme de Trasformation du graphe.

Entrées : Le graphe $G_{net}(V, E)$.
Sorties : Un graphe $G^*_{net}(V, E')$.
Début

(a) On associe à chaque lien autre que les liens du chemin primaire P un poids $w_{i,j}$.
(b) Pour chaque liaison du chemin principal, les poids sont attribués comme suit : Les liens dirigés à partir d'un sommet dans la séquence P à son sommet successeur ont un poids égale à l'infini (on d'autres termes ces liens sont supprimés). Les liens dirigées à partir d'un sommet dans la séquence P à son sommet prédécesseur se voient attribuer un poids égal à zéro.
(c) Pour chaque lien $e(v_1, v_2) \in E$, si $v_1 \notin P$ et $v_2 \in (P - \{S\})$ remplacer $e(v_1, v_2)$ par $e'(v_1, v'_2)$, ou v'_2 est le prédécesseur immédiat de v_2 dans P. Ceci remplace chaque lien dont le nœud de départ n'est pas dans P et que le nœud d'arriver est dans P par un autre lien dont le nœud de départ est toujours le même, mais le nœud d'arrivée est le prédécesseur du nœud de départ d'origine.

Fin

---

② Phase 2 : Calcul du plus court chemin dans $G^*_{net}(V, E')$: Pour trouver le plus court chemin entre S et D dans le nouveau graphe G^*_{net}, on applique l'un des algorithmes qui permettent le calcul du chemin le plus court (par exemple l'algorithme de Dijkstra) à partir de la source jusqu'au destination, le nouveau chemin obtenu est désigné par la séquence de sommets $B = \{S, i'_1, i'_2, \cdots, i'_m, D\}$.

③ Phase 3 : Génération de la sauvegarde fragmentée du P dans G_{net}, supposons que la sauvegarde fragmentée est constituée des fragments de sauvegarde BF_1, BF_2, ...,BF_b ; ordonnés par ordre croissant de la position de leurs nœuds de départ dans la séquence P. Comme nous traversons la séquence B trouver dans la phase 2 de S vers D, nous générons des séquences de sommets pour ces segments BF_1, BF_2, BF_3, ... l'un après l'autre.

L'expression 'ouvrir un fragment' indique le début de la génération de ce fragment de sauvegarde, et 'fermer un fragment' indique la fin de la génération de ce fragment de sauvegarde. Ainsi, dans notre algorithme le fragment BS_1 est le premier ouvert, généré, et fermé, puis BS_2 et ainsi de suite, jusqu'à ce que toutes les sauvegardes segmentées soient générées.

Nous consommons par initialiser tous les fragments de la sauvegarde BF_1, BF_2, ...,BF_b à l'ensemble vide, A n'importe quel moment du parcours de B, nous utilisons BF_c pour noter le fragment en cours de création, et i'_c pour noter le nœud actuel, initialement $i'_c = S$. Pour chaque i'_c effectuer la première action applicable indiqué dans les étapes (a) à (d) de l'algorithme de création

des fragments de sauvegarde de l'Algorithme 6, dans cet ordre , Ce processus se termine en arrivant au nœud destination D. Nous utilisons les termes $Prec(i'_c)$ et $Succ(i'_c)$ pour désigner le nœud prédécesseur immédiat et le nœud successeur immédiat de i'_c dans P.

Algorithme 6 Algorithme de Création des fragments de sauvegarde.

Entrées : Le graphe $G^*_{net}(V, E')$.
Sorties : BF_1, BF_2, ...,BF_b.
Début

(a) **Si** $i'_c = S$ **Alors** BF_c=BF_0 et $i'_c = Succ(i'_c)$.
(b) **Si** $i'_c = D$ **Alors**
  i. **Si** $Prec(i'_c) \notin P$ **Alors** ajouter $\{i'_c\}$ au BF_c, **STOP**.
  ii. **Si** $Prec(i'_c) \in P$ **Alors** $BS_c = Succ(BF_c)$, ajouter $\{Prec(i'_c), i'_c\}$ au BF_c, **STOP**.
(c) **Si** $i'_c \neq i_k$ pour chaque $k \leq p$ $(i'_c \notin P)$ **Alors**
  i. **Si** $Prec(i'_c) \notin P$ **Alors** ajouter $\{i'_c\}$ au BF_c et $i'_c = Succ(i'_c)$.
  ii. **Si** $Prec(i'_c) \in P$ **Alors** $BF_c = Succ(BF_c)$, ajouter $\{Prec(i'_c), i'_c\}$ au BF_c et $i'_c = Succ(i'_c)$.
(d) **Si** $i'_c = i_k$ pour chaque $k \leq p$ $(i'_c \in P)$ **Alors**
  i. **Si** $Prec(i'_c) \notin P$ **Alors** ajouter $\{i_{k+1}\}$ au BF_c et $i'_c = Succ(i'_c)$.
  ii. **Si** $Prec(i'_c) \in P$ et $Pred_p(Prec(i'_c), i'_c)$ **Alors** $BF_c = Succ(BF_c)$, ajouter $\{Prec(i'_c), i'_c\}$ au BF_c et $i'_c = Succ(i'_c)$.
  iii. **Si** $Prec(i'_c) \in P$ et $Succ_p(Prec(i'_c), i'_c)$ **Alors** $i'_c = Succ(i'_c)$.

Fin

Les séquences de sommets résultantes définissent les fragments de sauvegarde qui forment la sauvegarde fragmentée la plus courte pour le chemin principal P dans G_{net}. L'Algorithme 7 résume tout le processus de la sauvegarde fragmentée.

Algorithme 7 Algorithme de La Sauvegarde fragmentée.

Entrées : Le graphe $G_{net}(V, E)$.
Sorties : BF_1, BF_2, ...,BF_b.
Début

1. Génération du Graphe modifié $G^*_{net}(V, E')$ à partir de $G_{net}(V, E)$ par l'application de l'algorithme de la transformation du Graphe (Algorithme 5)
2. Génération du chemin $B = \{S, i'_1, i'_2, \cdots, i'_m, D\}$, le plus court chemin entre S et D dans le nouveau graphe $G^*_{net}(V, E')$ (Générer à l'étape précédente par l'application de l'un des algorithmes des chemins les plus courts, comme Dijkstra).
3. Génération de la sauvegarde fragmentée du P dans G_{net} par l'application de l'algorithme de la création des fragments de sauvegarde (Algorithme 6).

Fin

7.4.4 Fonction de sélection des segments de sauvegarde avec pondération de l'apport temps réel

La technique de la sauvegarde fragmentée peut générer plusieurs fragments de sauvegarde indépendants pour le même fragment primaire, pour cela et dans le cadre de notre travail nous proposons une nouvelle technique qui permet de sélectionner un chemin de sauvegarde parmi l'ensemble des chemins de sauvegarde d'un chemin primaire donnée.

La fonction de sélection que nous proposons prend en considération l'importance de chacun des liens qui composent les fragments de sauvegarde et la longueur de ces chemins. l'importance d'un lien de communication est définie par les paramètres de la qualité de service de ses liens, à savoir dans notre cas le temps réel et la charge.

Soit $E_{frag} = \left\{\ldots, Frag_{(s)i}, \ldots\right\}$ l'ensemble des fragments de sauvegarde d'un fragment primaire donnée $Frag_{(p)i}$, on associe à chacun des chemins de l'ensemble E_{frag} une longueur $Long(Frag_{(s)i})$. Pour chaque lien d'un fragment de sauvegarde donné on calcul une mesure du temps réel qui permet le classement de ce lien dans une catégorie temps réel, le classement pondéré de ce lien permet la pondération de tout le fragment.

Une fonction F est calculée pour chaque fragment de sauvegarde $Frag_{(s)i}$ comme suite :

$$F(Frag_{(s)i}) = Long(Frag_{(s)i}) - \frac{M^{T-Rl}(Frag_{(s)i})}{M^{T-Rl}_{Moy}} \tag{7.2}$$

Où :

$$M^{T-Rl}_{Moy} = \frac{\sum_{l_{P_i,P_j} \in E_{frag}} M^{T-Rl}(l_{P_i,P_j})}{\sum_{k=1}^{k=nb_classes}(\frac{1}{1+k}Card(G_{net}(l)^{k}_{T-Rl})} \tag{7.3}$$

Pour le choix du fragment de sauvegarde parmi l'ensemble des chemins de sauvegarde d'un chemin primaire donné, on applique la fonction F, le fragment dont la valeur de F est la plus proche par rapport à au plus petit apport en temps réel.

Exemple : Les trois fragments du sauvegarde de la Figure 7.8 sont les fragments de sauvegarde d'un même fragment primaire, l'application des formules de la fonction de sélection des fragments de sauvegarde avec pondération de l'apport temps réel permet d'avoir les résultats suivantes :

$$M^{T-Rl}_{Moy} = \frac{25}{\frac{1}{1} * (6) + \frac{1}{2} * (2) + \frac{1}{3} * (3)} = \frac{25}{8} = 3.125 \tag{7.4}$$

$$F(Frag_{(s)1}) = 2 - \frac{2}{M^{T-Rel}_{Moy}} = 2 - \frac{2}{3.125} = 1.36 \tag{7.5}$$

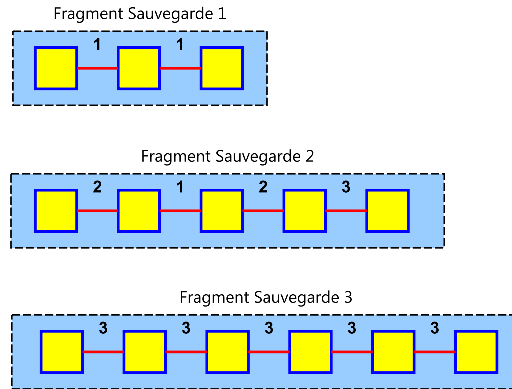


FIGURE 7.8 – Sélection d'un fragment de sauvegarde

$$F(Frag_{(s)2}) = 4 - \frac{8}{M_{Moy}^{T-Rel}} == 4 - \frac{8}{3.125} = 1.44 \quad (7.6)$$

$$F(Frag_{(s)3}) = 5 - \frac{15}{M_{Moy}^{T-Rel}} = 5 - \frac{15}{3.125} = 0.20 \quad (7.7)$$

7.4.5 Un exemple de la sauvegarde fragmentée

Nous illustrons maintenant les trois étapes ci-dessus (de l'algorithme de la sauvegarde fragmentée - Algorithme 7) à travers l'exemple de la Figure 7.9.

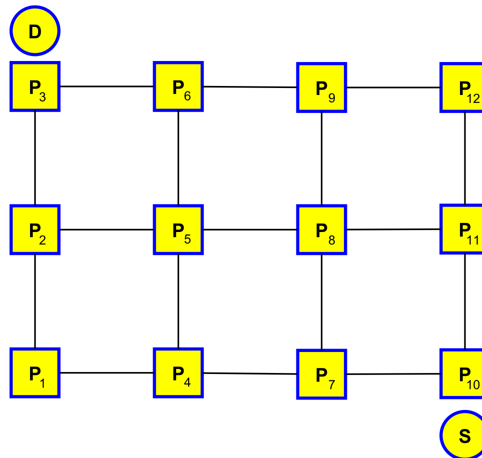


FIGURE 7.9 – La topologie du réseau de communications.

Le graphe orienté et pondéré G_{net} obtenue à partir de la topologie maillée $3 * 4$ est représenté sur la Figure 7.10, tandis que le graphe obtenu en modifiant le graphe

G_{net} en utilisant l'étape (1) est représenté sur par la Figure 7.11. Les bords le long de la voie primaire en direction de S à D (par exemple, le bord de P_5 à P_2) sont retirés, tandis qu'un poids nul (zéro) est attribué à ceux dans le sens inverse (par exemple, le bord de P_2 à P_5).

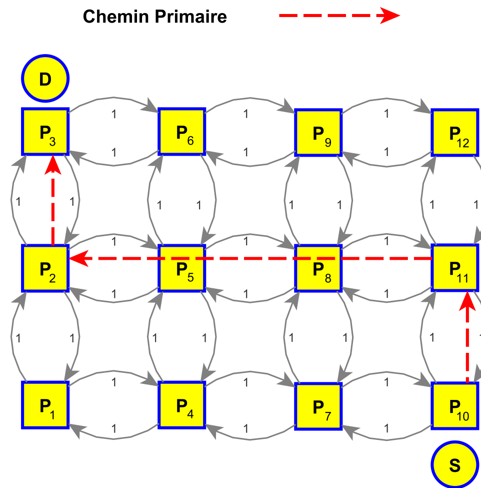


FIGURE 7.10 – Graphe G_{net}.

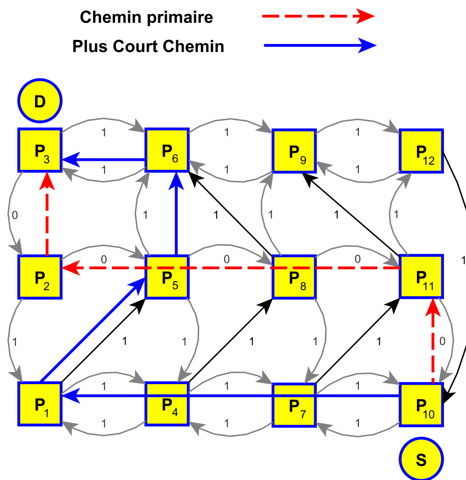


FIGURE 7.11 – Graphe G^*_{net}.

En outre, les bords pointant vers les nœuds sur le chemin primaire (par exemple, l'arrête de P_4 à P_5) sont redirigés pour pointer vers les sommets sur le chemin principal (l'arrête de P_4 à P_5 est redirigé vers P_N) précédent. Le plus court chemin dans G^*_{net} trouvé à l'étape (2) de l'algorithme est aussi montré dans cette même

figure avec une ligne pointillée.

Enfin, une sauvegarde fragmentée composée de deux fragments de sauvegarde : $BF_1 = \{l_{S,P_7}, l_{P_7,P_4}, l_{P_4,P_3}, l_{P_3,P_2}\}$ et $BF_2 = \{l_{P_5,P_6}, l_{P_6,D}\}$ obtenus à partir du plus court chemin à l'aide de l'étape (3) de l'algorithme est représentée sur la Figure 7.12.

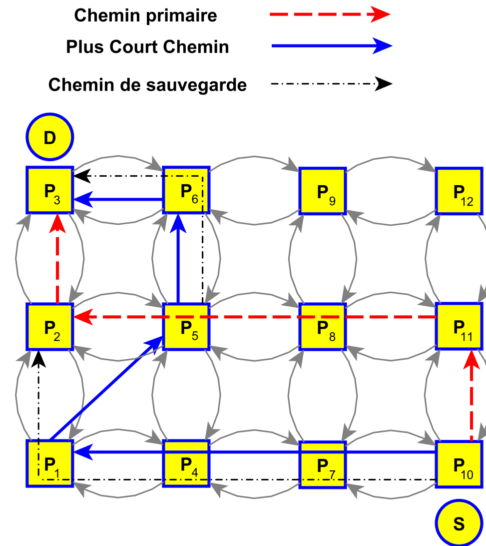


FIGURE 7.12 – La Sauvegarde Segmentée

On notera que dans l'étape 1(b) (étape (b) de l'Algorithme 6) , on redirige les arêtes de sorte que les fragments de sauvegarde successifs se chevauchent au moins sur une liaison de la voie primaire. Ce chevauchement est nécessaire pour tolérer des défaillances de nœuds.

7.5 Algorithme d'ordonnancement

L'heuristique d'ordonnancement est un algorithme de construction progressive de type glouton [98], il permet d'ordonnancer, à chaque étape (n), plusieurs opérations d'un algorithme G_{Task} donné sur plusieurs processeurs d'une architecture G_{Arch} donnée et assurer leurs dépendances de données par des chemins primaires qui se composent de plusieurs liens, dans le but de minimiser l'exécution du système, et tolère jusqu'à Nf_{chm} fautes de liens de communications.

L'heuristique d'ordonnancement *RBF-FB* (présentée par l'Algorithme 8), est composée des cinq phases suivantes :

— Phase d'initialisation : Cette phase consiste à initialiser la liste des opérations

candidates T_{cand} avec des tâches sans prédécesseurs c'est à dire les tâches d'entrée. La liste des tâches ordonnancées T_{ord} est initialement vide.

— Phase de vérification des routes disjointes : Cette phase consiste à vérifier que les routes sont complètement disjointes, et ne comprend aucun cycle.

— Phase de sélection : Pendent cette étape de sélection, un processeur est choisi parmi l'ensemble des processeurs de G_{Arch} pour ordonnancer une tâche et ses données de communications. La tâchel choisie est la candidate la plus urgente t_{urgent}. La règle de sélection choisie repose sur la pression d'ordonnancement. (voir chapitre 4)

— Phase d'ordonnancement : Après avoir sélectionner la meilleure candidate t_{best}, cette phase consiste en un premier temps à répliquer passivement cette tâche candidate en $Nf_{Proc}+1$ répliques, et en un deuxième temps, à ordonnancer chaque réplique t^k_{best} de t_{best} sur le k^{ieme} processeur p^k_{best} de $E_{Procbest}$.
Avant d'ordonnancer ces répliques, chaque dépendance de données ($t_j \rightarrow t_{best}$) est envoyée sur le chemin primaire qui connecte les nœuds source et destination
.

— Phase de mise à jour : La tâche t_{best} est retirée de l'ensemble des tâches candidates pour ordonnancement T_{cand}, et les tâches de l'G_{Task} qui ont toutes leurs prédécesseurs dans le nouveau ensemble des tâches déjà ordonnancées sont ajoutées à cet ensemble.

7.6 Recouvrement des fautes(Reprise sur faute)

En cas de défaut d'un composant du réseau, toutes les connexions fiables qui le traversent doivent être déroutées par leurs chemins de sauvegarde. Ce processus est appelé reprise sur incident et il opère sur trois phases :

— Détection de la faute.

— Envoi du Rapport de faute.

— Activation de la sauvegarde.

Le temps de restauration, appelé délai de récupération après échec, est crucial pour de nombreuses applications temps réel, et doit être réduit au minimum. Dans notre modèle, nous supposons que quand un lien tombe en panne, ses nœuds d'extrémité peuvent détecter la faute comme expliquer dans [112], et le rapporter aux nœuds concernés pour la récupération après faute. C'est ce qu'on appelle des *rapports de défaillance*. Après que le rapport de défaillance atteint certains nœuds, la sauvegarde est activée par ces nœuds. Les rapports de défaillance et de l'activation de sauvegarde doivent utiliser des messages de contrôle ; pour cela on suppose qu'un canal de commande en temps réel (CCR) [106] est réservé pour l'envoi exclusive des messages de contrôle (ce canal doit garantir un débit et un délai minimum).

Algorithme 8 Algorithme d'ordonnancement RBF-FB.

Entrées : G_{Task}, G_{Arch}, Nf_{Proc}, Nf_{chm}.
Sorties : Ordonnacement de G_{Task} sur G_{Arch} tolérant au plus Nf_{Proc} fautes de processeurs et Nf_{chm} fautes de chemins de communication.
Début
Initialiser la liste des tâches candidates, et la liste des opérations déjà ordonnacées.
n := 0 ;
$T^{(0)}_{cand} := \{t \in T \mid pred(t) = \emptyset\}$; (tâche de G_{Task} sans prédécesseurs)
$T^{(0)}_{ord} := \emptyset$;
Tant que $T^{(n)}_{cand} \neq \emptyset$ **faire :**

1. Pour chaque tâche candidate t_{cand}, calculer $\sigma^{(n)}$ et $\Lambda(S_n)$ pour chaque processeur p_j.
   $\sigma^{(n)}(t_i, p_j) = S^{(n)}_{t_i,p_j} + \overline{S}^{(n)}_{t_i} - R^{(n-1)}$
   $\Lambda(S_n) = \frac{-\log R(S_n)}{U(S_n)}$
2. Pour chaque tâche candidate t_{cand}, sélectionner le meilleur processeur parmi $p^{t_{cand}}_{best}$ qui minimise $\sigma^{(n)}$ et le TGDS.
3. Sélectionner la tâche candidate la plus urgente t_{urgent}, parmi toutes les opérations t^i_{cand} de $T^{(n)}_{cand}$.
4. Ordonnancer t_{urgent} ;
5. Ordonnancer les données de t_{urgent} sur le chemin primaire ;
6. Mettre à jour la liste des : tâches candidates et des tâches déjà ordonnacées.
   $T^{(n)}_{ord} := T^{(n-1)}_{ord} \cup \{t_{urgent}\}$;
   $T^{(n+1)}_{cand} := T^{(n)}_{cand} - \{t_{urgent}\} \cup \{t' \in succ(t_{urgent}) \mid pred(t') \; subseteq T^{(n)}_{ord}\}$;
7. n := n + 1 ;

Fin Tant que
Fin

Les nœuds adjacents à un nœud défaillant du chemin primaire d'une connexion fiable permet de détecter la faute et d'envoyer des rapports de défaillance à la fois vers la source et vers la destination. Dans le modèle de sauvegarde du bout en bout, ces messages doivent parvenir à la source et la destination avant de pouvoir activer le chemin de sauvegarde, par contre avec notre approche, cela n'est pas nécessaire. Les fautes peuvent être manipulés plus localement. Les nœuds d'extrémité du fragment primaire contenant le composant défaillant peuvent initier le processus de récupération par sauvegarde fragmentée juste après la réception des rapports de défaillance.

Ces deux nœuds envoient le message d'activation sur le fragment de sauvegarde, et le service de connexion fiable est repris. Ce processus est illustré dans le cas de la sauvegarde fragmentée dans la Figure 7.13, et dans le cas de la sauvegarde de bout en bout dans la Figure 7.14.

Quand une faute se produit, non seulement nous faisons l'expérience d'une interruption de service pendant un certain temps, mais aussi les paquets transmis pendant la durée de l'envoi du rapports d'échec sont perdus. La plupart des ap-

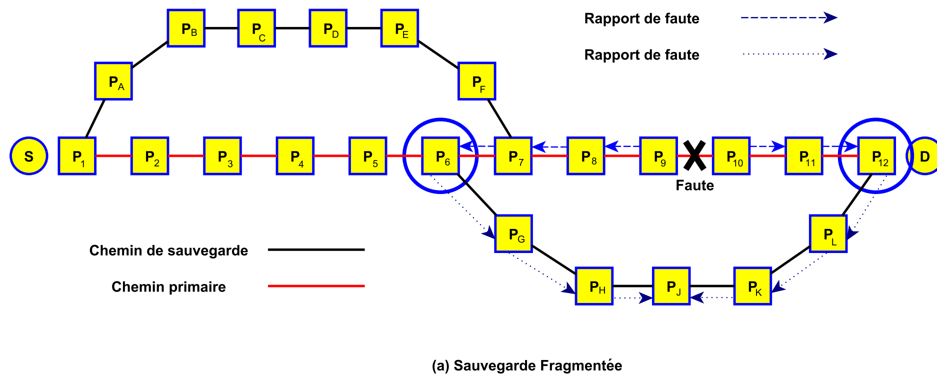


FIGURE 7.13 – Recouvrement des fautes : cas de la sauvegarde fragmentée

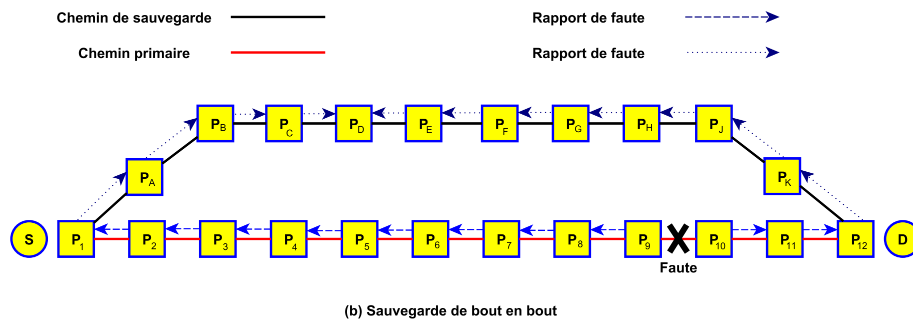


FIGURE 7.14 – Recouvrement des fautes : cas de la sauvegarde de bout en bout

plications temps réel ne peuvent pas tolérer beaucoup de perte de messages. Dans notre schéma la perte de message est réduite d'une façon considérable. Quand une faute se produit dans un fragment du chemin primaire, seuls les paquets qui sont entrés dans ce fragment à partir du moment de l'apparition de la faute jusqu'à ce que l'activation des fragments de sauvegarde sont perdus, les autres paquets dans les fragments avant et après que le fragment tombe en panne ne sont pas affectés et seront distribués normalement. Ceci est au contraire de la sauvegarde de bout en bout, où tous les paquets transmis dans le chemin primaire entre l'occurrence d'une faute et l'activation de sauvegarde, sont perdues.

7.7 Délai et Scalabilité

7.7.1 Délai

Avec les communications temps réel, il est essentiel d'avoir des délais les plus courts que possible, que ce soit pour le chemin primaire ou pour le chemin de sau-

vegarde. Par conséquent, nous pouvons définir des restrictions sur les valeurs par lesquelles le délai sur le chemin de sauvegarde peut dépasser celui du chemin primaire. Une simple spécification des paramètres de la QoS, peut définir que le délai sur le chemin de sauvegarde ne doit en aucun cas dépasser celui du chemin primaire.

Ainsi, la contrainte pour le choix d'une sauvegarde de bout en bout bout peut être définie comme :

$$delai_{chemin_{savegardedeboutenbout}} - delai_{chemin_{primaire}}$$

alors que pour le cas de la sauvegarde fragmentées cette contrainte est :

$$delai(Frag_{(s)i}) \leq delai(Frag_{(p)i})$$

Pour notre cas, nous devons minimiser l'augmentation des délais pour chaque fragment de façon indépendante. Ainsi le taux des communications fiable sans fautes est plus important du fait qu'il est plus facile de trouver de petits fragments de sauvegarde que de trouver des sauvegardes de bout en bout satisfaisants une contrainte de QoS bien déterminée.

7.7.2 Scalabilité

Le modèle de sauvegarde fragmentée est bien scalable car :

1. Il n'exige aucune connaissance globale du réseau ;
2. Il n'implique aucun type de diffusion ;
3. Il ne nécessite aucune gestion de trafic supplémentaire.

Par contre, avec la sauvegarde fragmentée chaque nœud doit faire le lien entre les chemins de sauvegarde qui le traversent et les chemins primaires. Ceci est facilement réalisable si l'information est mise à jour au moment de l'établissement du chemin de sauvegarde.

A la suite d'une faute, les messages de contrôle ne sont pas diffusés, mais envoyés uniquement à une partie limitée du réseau ; celle concernée par la faute. L'efficacité de la sauvegarde fragmentée augmente lorsque la longueur moyenne du chemin augmente.

7.8 RSVP

Le protocole RSVP [113] est la première proposition solide pour la réservation de ressources pour les réseaux multi-sauts ; il fournit l'initialisation et le contrôle de réservation des ressources. Toutefois, RSVP n'est pas un protocole de routage, pour notre cas, il ne permet pas la construction des chemins de sauvegarde mais

plutôt il travaille en collaboration avec ces protocoles pour permettre la réservation dynamique de ressources.

Son principe de fonctionnement est le suivant [114] :

- L'émetteur spécifie le trafic en terme de bande passante, délai d'acheminement dans un descripteur de trafic TSPEC, envoyer à l'aide d'un message RSVP-PATH, qui contient entre autre la spécification de trafic TSPEC et une spécification additionnelle ADSPEC ;
- Le message PATH est acheminé vers la destination à l'aide des protocole de routage ;
- Chaque nœud intermédiaire sur le chemin de sauvegarde enregistre les informations relatives au chemin constitué dans un PATH-STATE ;
- Grâce aux spécifications de TSPEC (décrites par l'émetteur) et ADSPEC (modifiées par le réseau pour rendre compte des possibilités réelles), et pour effectuer la réservation le récepteur envoi un message RSVP-RESV ;
- Le message RSVO-RESV revient au émetteur en utilisant le même chemin de sauvegarde indiqué dans le message PATH (technique de source routing) ;
- Chaque nœud intermédiaire reçoit le message RESV, et procède à l'allocation des ressources ;
- Lorsque l'émetteur ou le récepteur ferme la session RSVP, cela entraine la libération automatique de toutes les ressources réservées.

Par souci de simplification, nous avons supposé que les connexions sont de type unicast, c-à-d des connexions impliquant une source et une destination. Cependant la souplesse de RSVP permet de tenir facilement compte des connexions multicast, impliquant une sources vers plusieurs destinations. RSVP est orienté vers le récepteur, c-à-d que c'est le récepteur qui initie et maintient la réservation des ressources, et il est unidirectionnel.

7.9 Conclusion

Dans ce chapitre, nous avons présenté une nouvelle méthode, efficace, scalable et distribuée pour la tolérance aux fautes des communications. Cette technique est destinée aux réseaux multi-sauts et non fortement couplés. Le concept ainsi que le fonctionnement de la sauvegarde fragmentée, par opposition à la sauvegarde de bout en bout ont été présentés. Ensuite, une fonction de sélection des fragments de sauvegarde à été élaborée. Plus tard dans ce chapitre, il a été montré comment cette solution a été intégrée dans un algorithme d'ordonnancement, à la fin nous avons discuter les questions de la reprise après fautes, les délais, la scalabilité et comment notre algorithme peut être facilement mis en œuvre à l'aide des protocoles de réservation de ressources actuelles comme RSVP.

Chapitre 8

Conclusion Générale

Le travail présenté dans le cadre de cette thèse, se place dans le cadre général de la conception des systèmes sûrs de fonctionnement. Plus particulièrement, il s'intéresse aux systèmes réactifs, embarqués et temps réel. Il traite essentiellement l'aspect de la communication dont le rôle conditionne largement la capacité d'un système à respecter la ponctualité et les exigences de la sûreté de fonctionnement. Le problème consistait à chercher l'ordonnancement le plus fiable possible et tolérant aux fautes des processeurs et des médias de communication.

Le problème d'ordonnancement temps réel, fiable et tolérant aux fautes est un problème d'optimisation NP-difficile, puisque il s'agit ici de trouver une solution qui minimise le temps global de l'exécution d'un algorithme sur une architecture hétérogène et distribuée, en maximisant la fiabilité et en tolérant des fautes matérielles, tout en respectant un certain nombre de contraintes. La solution optimale et exacte à ce problème ne peut être trouvée que par des algorithmes exacts de complexité exponentielle ; c'est pourquoi nous avons proposé d'utiliser dans ce travail des heuristiques qui cherchent la solution la plus proche de la solution optimale, mais en étant de complexité polynômiale.

Comme nous ciblons des systèmes embarqués avec des ressources limitées (pour des raisons de poids, encombrement, consommation d'énergie, ou les contraintes de prix), nous ne nous intéressons qu'aux techniques basées sur des solutions logicielles où aucune ressource physique supplémentaire n'est ajoutée.

Les résultats des travaux de recherche effectués dans le cadre de cette thèse ont donnés lieu à la proposition de trois nouvelles approches de conception de systèmes distribués, embarqués, temps réel et tolérants aux fautes. La première et la deuxième solution sont adaptées aux architectures à liaisons bus de communications, et la troisième aux architectures à base de réseaux de communications multi-sauts.

La première approche, que nous avons proposée est dite **Approche d'ordonnancement fiable de communication basée sur la fragmentation variable des données**, elle permet un ordonnancement tolérant à n'importe quelle composition de plusieurs fautes de processeurs et de bus de communication. La tolérance aux fautes des processeurs est obtenue hors-ligne, par l'utilisation d'un mécanisme de préemption qui permet un ré-classement des tâches dans une nouvelle variété de

la redondance passive dite *Passive-Passive*, tandis que pour le cas des bus de communications, on utilise la redondance passive des communications combinée avec la fragmentation variable de données.

La deuxième approche, que nous avons appelée **Approche d'ordonnancement fiable de communication basée sur la désallocation des données**, est une approche hybride qui se base sur deux types de copies de sauvegarde, répliquées et désallouées. L'objectif principal est de minimiser la longueur des ordonnancements des données sur les bus de communications, par un mécanisme de chevauchement des copies de sauvegarde désallouées et réallouées , ce qui permet une utilisation plus optimale des bus de communications.

La troisième approche, que nous avons appelée **Approche d'ordonnancement fiable de communication temps réel basée sur les chemins de sauvegarde fragmentés** génère automatiquement des ordonnancements tolérantes à n'importe quelle combinaison de plusieurs fautes matérielles des liens du réseau de communication. Cette fois, la tolérance aux fautes est obtenue hors-ligne par l'utilisation des sauvegardes segmentées des chemins de communication.

Les ordonnancements temps réel fiables et tolérantes aux fautes générés par ces trois approches sont touts les trois prédictives. Donc les contraintes de temps réel et de fiabilité peuvent être vérifiées avant la mise en exploitation du système, à la fois en l'absence et en présence de défaillances des processeurs et des médias de communication.

Enfin, les travaux présentés dans cette thèse offrent un certain nombre de perspectives :

— Avant tout, les trois algorithmes d'ordonnancement proposés dans le cadre de ce travail doivent êtres testés plus extensivement sur des modèles de tâches et d'architectures générés aléatoirement afin de mieux comprendre l'influence des paramètres.

— Ensuite, ces trois approches doivent être testées sur des cas réalistes.

— Les trois approches peuvent être enrichies par des nouveaux moyens de sûreté de fonctionnement, tels que le *le fonctionnement en mode dégradé*. La question principale qui se pose ici est : comment prendre en compte le mode normal et le mode dégradé en même temps dans nos algorithmes d'ordonnancement, et surtôut comment gérer le passage entre ces deux modes ?

— 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.

Bibliographie

[1] P. Chevochot and I. Puaut, “An approach for fault-tolerance in hard real-time distributed systems,” in *Reliable Distributed Systems, 1999. Proceedings of the 18th IEEE Symposium on*, pp. 292–293, IEEE, 1999. (Cité en page 11.)

[2] P. Chevochot and I. Puaut, “Tolérance aux fautes dans les systèmes répartis temps-réel strict,” *TSI. Technique et science informatiques*, vol. 18, no. 8, pp. 837–870, 1999. (Cité en page 11.)

[3] E. Grolleau, *Ordonnancement temps réel hors-ligne optimal à l'aide de réseaux de pétri en environnement monoprocesseur et multiprocesseur.* PhD thesis, LISI-ENSMA, 1999. (Cité en page 11.)

[4] M. Garey and D. Johnson, *Computers and Intractability, a Guide to the Theory of NP-Completeness.* San Francisco : W. H. Freeman Company, 1979. (Cité en pages 14 et 19.)

[5] T. Grandpierre, C. Lavarenne, and Y. Sorel, “Optimized rapid prototyping for real-time embedded heterogeneous multiprocessors,” in *Proceedings of the seventh international workshop on Hardware/software codesign*, pp. 74–78, ACM, 1999. (Cité en page 14.)

[6] L. Cucu, R. Kocik, and Y. Sorel, “Real-time scheduling for systems with precedence, periodicity and latency constraints,” in *Proceedings of 10th Real-Time Systems Conference, RTS'02*, 2002. (Cité en page 14.)

[7] T. Grandpierre and Y. Sorel, “From algorithm and architecture specifications to automatic generation of distributed real-time executives : a seamless flow of graphs transformations,” in *Formal Methods and Models for Co-Design, 2003. MEMOCODE'03. Proceedings. First ACM and IEEE International Conference on*, pp. 123–132, IEEE, 2003. (Cité en page 14.)

[8] M. Marouf, *Ordonnancement temps réel dur multiprocesseur tolérant aux fautes appliqué à la robotique mobile.* PhD thesis, école nationale supérieure des mines de Paris, 2012. (Cité en pages 15, 16 et 32.)

[9] A. Colin, I. Puaut, C. Rochange, and P. Sainrat, “Calcul de majorants de pire temps d'exécution : état de l'art,” *Techniques et Sciences Informatiques (TSI)*, vol. 22, no. 5, pp. 651–677, 2003. (Cité en page 16.)

[10] R. KOCIK, *Sur l'optimisation des systèmes distribués temps réel embarqués : application au prototypage rapide d'un véhicule électrique semi-autonome.* PhD thesis, Université de Rouen , Spécialité informatique industrielle, 2000. (Cité en page 18.)

[11] C.-H. Lin and C.-J. Liao, “Makespan minimization for multiple uniform machines,” *Computers & Industrial Engineering*, vol. 54, no. 4, pp. 983–992, 2008. (Cité en page 18.)

[12] N. Ventroux, *Contrôle en ligne des systèmes multiprocesseurs hétérogènes embarqués : élaboration et validation d'une architecture.* PhD thesis, Université de Rennes,, 2006. (Cité en page 18.)

[13] F. Parain, *Ordonnancement sous contraintes énergétiques d'applications multimédia sur une plate-forme multiprocesseur hétérogène.* PhD thesis, Rennes 1, 2002. (Cité en page 18.)

[14] A. B. A. Garcia, *Estimation et optimisation de la consommation dans les descriptions architecturales des systèmes intégrés complexes.* PhD thesis, Université de Paris 6,, 2005. (Cité en page 18.)

[15] R. A. Walker and S. Chaudhuri, "High-level synthesis : Introduction to the scheduling problem," in *IEEE Design & Test*, 1995. (Cité en page 19.)

[16] P. Chevochot and I. Puaut, "Scheduling fault-tolerant distributed hard real-time tasks independently of the replication strategies," in *Real-Time Computing Systems and Applications, 1999. RTCSA'99. Sixth International Conference on*, pp. 356–363, IEEE, 1999. (Cité en pages 23, 36, 40 et 53.)

[17] P. Pop, V. Izosimov, P. Eles, and Z. Peng, "Design optimization of time-and cost-constrained fault-tolerant embedded systems with checkpointing and replication," *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, vol. 17, no. 3, pp. 389–402, 2009. (Cité en pages 23 et 36.)

[18] K. Chaaban, M. Shawky, and P. Crubille, "A distributed framework for real-time in-vehicle applications," in *Intelligent Transportation Systems, 2005. Proceedings. 2005 IEEE*, pp. 925–929, IEEE, 2005. (Cité en pages 23 et 36.)

[19] V. Izosimov, P. Pop, P. Eles, and Z. Peng, "Design optimization of time-and cost-constrained fault-tolerant distributed embedded systems," in *Proceedings of the conference on Design, Automation and Test in Europe-Volume 2*, pp. 864–869, IEEE Computer Society, 2005. (Cité en pages 23 et 36.)

[20] J.-C. Laprie, *Guide de la sûreté de fonctionnement.* Cépadués, 1995. (Cité en page 23.)

[21] J. Arlat, Y. Crouzet, Y. Deswarte, J.-C. Laprie, D. Powell, P. David, J. Dega, C. Rabéjac, H. Schindler, and J.-F. Soucailles, "Fault tolerant computing," *Wiley Encyclopedia of Electrical and Electronics Engineering*, 1999. (Cité en page 23.)

[22] J. Arlat, A. Costes, J.-P. Blanquart, and J.-C. Laprie, *Guide de la sûrté de fonctionnement.* Cépaduès-éditions, 1996. (Cité en page 24.)

[23] A. Avizienis, J.-C. Laprie, B. Randell, and C. Landwehr, "Basic concepts and taxonomy of dependable and secure computing," *Dependable and Secure Computing, IEEE Transactions on*, vol. 1, no. 1, pp. 11–33, 2004. (Cité en page 25.)

[24] J. Arlat, Y. Crouzet, Y. Deswarte, J.-C. Fabre, J.-C. Laprie, and D. Powell, "Tolérance aux fautes," *Encyclopédie de l'informatique et des systèmes d'information. Vuibert, Paris, France*, p. 92, 2006. (Cité en pages 25, 27, 29, 32, 33, 35, 37, 39 et 42.)

[25] J.-C. Laprie, “Sûreté de fonctionnement des systèmes : concepts de base et terminologie,” *Revue de l’électricité et de l’électronique (REE),*, vol. 11, pp. 95–105, 2004. (Cité en pages 25 et 27.)

[26] J.-C. Laprie, *Dependability : Basic Concepts and Terminology.* Springer-Verlag, 1992. (Cité en pages 28, 30, 40 et 46.)

[27] A. Avižienis, “Design of fault-tolerant computers,” in *Proceedings of the November 14-16, 1967, Fall Joint Computer Conference*, AFIPS ’67 (Fall), (New York, NY, USA), pp. 733–743, ACM, 1967. (Cité en page 33.)

[28] A. Burns and A. J. Wellings, *Real-Time Systems and Programming Languages : ADA 95, Real-Time Java, and Real-Time POSIX.* Boston, MA, USA : Addison-Wesley Longman Publishing Co., Inc., 3rd ed., 2001. (Cité en pages 33 et 34.)

[29] J. Wakerly, *Error detecting codes, self-checking circuits and applications.* Elsevier, 1978. (Cité en page 34.)

[30] P. A. Lee and T. Anderson, *Fault tolerance.* Springer, 1990. (Cité en page 35.)

[31] G. Muller, M. Banâtre, N. Peyrouze, and B. Rochat, “Lessons from ftm : an experiment in design and implementation of a low-cost fault tolerant system,” *Reliability, IEEE Transactions on*, vol. 45, no. 2, pp. 332–340, 1996. (Cité en pages 35 et 42.)

[32] I. Koren and C. M. Krishna, *Fault-tolerant systems.* Morgan Kaufmann, 2010. (Cité en page 36.)

[33] E. N. Elnozahy, L. Alvisi, Y.-M. Wang, and D. B. Johnson, “A survey of rollback-recovery protocols in message-passing systems,” *ACM Computing Surveys (CSUR)*, vol. 34, no. 3, pp. 375–408, 2002. (Cité en page 37.)

[34] R. Guerraoui and A. Schiper, “Fault-tolerance by replication in distributed systems,” in *Reliable Software Technologies-Ada-Europe’96*, pp. 38–57, Springer, 1996. (Cité en pages 39 et 46.)

[35] R. Guerraoui and A. Schiper, “Software-based replication for fault tolerance,” *Computer*, vol. 30, no. 4, pp. 68–74, 1997. (Cité en page 39.)

[36] X. Qin, Z. Han, H. Jin, L. Pang, and S. 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)*, 2000. (Cité en pages 40 et 50.)

[37] C. Dima, A. Girault, C. Lavarenne, and Y. Sorel, “Off-line real-time fault-tolerant scheduling,” in *Parallel and Distributed Processing, 2001. Proceedings. Ninth Euromicro Workshop on*, pp. 410–417, IEEE, 2001. (Cité en pages 40 et 50.)

[38] Y. Oh and S. H. Son, “Scheduling real-time tasks for dependability,” *Journal of the Operational Research Society*, pp. 629–639, 1997. (Cité en pages 40, 50, 51 et 52.)

[39] K. Ahn, J. Kim, and S. Hong, "Fault-tolerant real-time scheduling using passive replicas," in *Fault-Tolerant Systems, 1997. Proceedings., Pacific Rim International Symposium on*, pp. 98–103, IEEE, 1997. (Cité en pages 40 et 50.)

[40] R. Al-Omari, A. K. Somani, and G. Manimaran, "A new fault-tolerant technique for improving schedulability in multiprocessor real-time systems," in *Parallel and Distributed Processing Symposium., Proceedings 15th International*, pp. 8–pp, IEEE, 2001. (Cité en pages 40 et 50.)

[41] 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. (Cité en pages 40, 46, 49 et 50.)

[42] A. Girault, M. Sighireanu, Y. Sorel, *et al.*, "An algorithm for automatically obtaining distributed and fault-tolerant static schedules," in *Proceeding of International Conference on Dependable Systems and Networks*, pp. 165–190, 2003. (Cité en pages 40, 46 et 49.)

[43] P. Felber and A. Schiper, "Optimistic active replication," in *Distributed Computing Systems, 2001. 21st International Conference on.*, pp. 333–341, IEEE, 2001. (Cité en pages 40 et 46.)

[44] P. Felber, X. Defago, P. Eugster, and A. Schiper, "Replicating corba objects : a marriage between active and passive replication," in *Distributed Applications and Interoperable Systems II*, pp. 375–387, Springer, 1999. (Cité en pages 40 et 53.)

[45] K. Hashimoto, T. Tsuchiya, and T. Kikuno, "Effective scheduling of duplicated tasks for fault tolerance in multiprocessor systems," *IEICE Transactions on Information and Systems*, vol. 85, no. 3, pp. 525–534, 2002. (Cité en pages 40, 53 et 54.)

[46] V. Bobin, S. Whitaker, and G. Maki, "Links between n-modular redundancy and the theory of error-correcting codes," in *Idaho Univ, The 1992 4 th NASA SERC Symposium on VLSI Design 11 p(SEE N 94-21694 05-33)*, 1992. (Cité en page 41.)

[47] T. Smith and J. N. Yelverton, "Processor architectures for fault tolerant avionic systems," in *Digital Avionics Systems Conference, 1991. Proceedings., IEEE/AIAA 10th*, pp. 213–219, IEEE, 1991. (Cité en page 41.)

[48] J.-x. Zou and H.-b. Xu, "Design and reliability analysis of emergency trip system with triple modular redundancy," in *Communications, Circuits and Systems, 2009. ICCCAS 2009. International Conference on*, pp. 1006–1009, IEEE, 2009. (Cité en page 42.)

[49] T. J. Dysart and P. M. Kogge, "System reliabilities when using triple modular redundancy in quantum-dot cellular automata," in *Defect and Fault Tolerance of VLSI Systems, 2008. DFTVS'08. IEEE International Symposium on*, pp. 72–80, IEEE, 2008. (Cité en page 42.)

[50] R. Guerraoui and A. Schiper, "Software-based replication for fault tolerance," *Computer*, vol. 30, pp. 68–74, Apr. 1997. (Cité en page 46.)

[51] C. Arar, M. S. Khireddine, A. Belazoui, and R. Megulati, "An energy-efficient fault-tolerant scheduling algorithm based on variable data fragmentation," in *5th IFIP TC 5 International Conference, CIIA 2015 Saida, Algeria, May 20-21*, IFIP Advances in Information and Communication Technology, 2015. (Cité en pages 46 et 59.)

[52] C. Arar and M. s. Khiereddine, "A reliable multi-bus fault-tolerant scheduling algorithm based on variable data fragmentation," *Asian Journal of Information Technology*, vol. 14, no. 02, pp. 67–73, 2015. (Cité en pages 46 et 59.)

[53] C. Arar, H. Kalla, S. Kalla, and R. Hocine, "Reliable fault-tolerant multi-bus scheduling algorithm," *International Journal of Computer Applications*, vol. 71, no. 12, pp. 11–15, 2013. (Cité en page 46.)

[54] C. Arar, H. Kalla, S. Kalla, and S. S. Bendib, "Energy-efficient fault-tolerant scheduling approach for embedded real time systems," *International Journal of Electrical Energy*, vol. 1, no. 4, pp. 274–278, 2013. (Cité en page 46.)

[55] C. Arar, H. Kalla, S. Kalla, and H. Riadh, "A reliable fault-tolerant scheduling algorithm for real time embedded systems," in *Proceedings of Workshop DECS (ERCIM/EWICS Workshop on Dependable Embedded and Cyber-physical Systems) of the 32nd International Conference on Computer Safety, Reliability and Security*, 2013. (Cité en page 46.)

[56] C. Arar, H. Kalla, S. Kalla, and S. S. Bendib, "Energy-efficient fault-tolerant scheduling approach for embedded real time systems," in *6th International Conference on Computer and Electrical Engineering (ICCEE 2013)*, IACSIT, 2013. (Cité en page 46.)

[57] C. Arar, H. Kalla, S. Kalla, and S. S. Bendib, "Fault-tolerant real-time scheduling algorithm for energy-aware embedded systems," *SAFECOMP 2013 FastAbstract*, 2013. (Cité en page 46.)

[58] H. Bettahar, C. Arar, and A. Boubdallah, "An efficient qos server selection protocol for duplicated layered multicast servers," in *Information and Communication Technologies : From Theory to Applications, 2004. Proceedings. 2004 International Conference on*, pp. 675–676, IEEE, 2004. (Cité en page 46.)

[59] K. Hashimoto, T. Tsuchiya, and T. Kikuno, "Effective scheduling of duplicated tasks for fault-tolerance in multiprocessor systems," *TIS*, vol. E85-D, pp. 525–534, Mar. 2002. (Cité en page 48.)

[60] S. Dulman, T. Nieberg, J. Wu, and P. Havinga, "Trade-off between traffic overhead and reliability in multipath routing for wireless sensor networks," in *Wireless Communications and Networking Conference*, 2003. (Cité en pages 49 et 50.)

[61] A. Banerjea, “Simulation study of the capacity effects of dispersity routing for fault-tolerant real-time channels,” in *SIGCOMM Symposium*, pp. 194–205, Aug. 1996. (Cité en pages 49 et 50.)

[62] B. Kao, H. Garcia-Molina, and D. Barbara, “Aggressive transmissions of short messages over redundant paths,” *TPDS*, vol. 5, pp. 102–109, Jan. 1994. (Cité en pages 49 et 50.)

[63] N. Kandasamy, J. Hayes, and B. Murray, “Dependable communication synthesis for distributed embedded systems,” in *International Conference on Computer Safety, Reliability and Security, SAFECOMP'03*, (Edinburgh, UK), Sept. 2003. (Cité en pages 49 et 50.)

[64] P. Fragopoulou and S. G. Akl, “Fault-tolerant communication algorithms on the star network using disjoint paths,” in *Proceedings of the 28th Hawaii International Conference on System Sciences (HICSS'95)*, (Kingston, Ontario, Canada), 1995. (Cité en pages 49 et 50.)

[65] C. D. A. G. C. Lavarenne ; and Y. Sorel, “Off-line real-time fault-tolerant scheduling,” in *9th Euromicro Workshop on Parallel and Distributed Processing*, pp. 410–417, 2001. (Cité en pages 49 et 50.)

[66] H. Kopetz, *Real-time systems : design principles for distributed embedded applications*. Springer Science & Business Media, 2011. (Cité en page 50.)

[67] X. Qin, Z. 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. (Cité en pages 52 et 55.)

[68] Q. Zheng and K. G. Shin, “Fault-tolerant real-time communication in distributed computing systems,” in *IEEE Transactions on Parallel and Distributed Systems*, pp. 470–480, 1998. (Cité en pages 52 et 106.)

[69] S. Han and K. Shin, “Fast restoration of real-time communication service from component failures in multi-hop networks,” in *SIGCOMM Symposium*, Sept. 1997. (Cité en page 52.)

[70] K. P. Gummadi, M. J. Pradeep, and C. S. R. Murthy, “An efficient primary-segmented backup scheme for dependable real-time communication in multihop networks,” *IEEE/ACM Transactions on Networking*, vol. 11, February 2003. (Cité en pages 52, 107 et 113.)

[71] R. Boppana and S. Chalasan, “Fault-tolerant multicast communication in multicomputers,” in *Proceedings of the 1995 International Conference on Parallel Processing*, vol. 1, pp. 118–125, 1995. (Cité en page 52.)

[72] C. Pope and J. Yantchev, “Fault-tolerant real-time communication with reduced resource overhead,” in *Australian Computer Science Communications*, vol. 17, pp. 441–448, February 1995. (Cité en page 52.)

[73] D. Ganesan, R. Govindan, S. Shenker, and D. Estrin, “Highly-resilient, energy-efficient multipath routing in wireless sensor networks,” in *Mobile Computing and Communications Review (MC2R)*, vol. 1, 2002. (Cité en page 52.)

[74] A. Banerjea, C. Parris, and D. Ferrari, “Recovering guaranteed performance service connections from single and multiple faults,” in *Proc. GLOBECOM‘94*, (San Francisco, CA), November 1994. (Cité en pages 53 et 106.)

[75] V. Izosimov, P. Pop, P. Eles, and Z. Peng, “Scheduling and optimization of fault-tolerant embedded systems with transparency/performance trade-offs,” *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 11, no. 3, p. 61, 2012. (Cité en page 54.)

[76] B. Chen, S. Kamat, and W. Zhao, “Fault-tolerant real-time communication in FDDI-based networks,” in *IEEE Real-Time Systems Symposium*, pp. 141–151, 1995. (Cité en pages 54 et 106.)

[77] C. Krishna, “Fault-tolerant scheduling in homogeneous real-time systems,” *ACM Computing Surveys (CSUR)*, vol. 46, no. 4, p. 48, 2014. (Cité en page 54.)

[78] D.-J. Chen, M.-S. Chang, M.-C. Sheng, and M.-S. Horng, “Time-constrained distributed program reliability analysis,” *J. Inf. Sci. Eng.*, vol. 14, no. 4, pp. 891–911, 1998. (Cité en page 54.)

[79] Y. He, Z. Shao, B. Xiao, Q. Zhuge, and E. Sha, “Reliability driven task scheduling for tightly coupled heterogeneous systems,” in *IASTED International Conference on Parallel and Distributed Computing and Systems*, (Marina Del Ray, USA), Nov. 2003. (Cité en pages 54 et 55.)

[80] S. Kartik and C. S. R. Murthy, “Improved task allocation algorithms to maximize reliability of redundant distributed computing systems,” *IEEE Transactions On Reliability*, vol. 44, Dec. 1995. (Cité en page 54.)

[81] S. Kartik and C. S. R. Murthy, “Task allocation algorithms for maximizing reliability of distributed computing systems,” *IEEE Transactions On Computers*, vol. 41, Sept. 1997. (Cité en page 54.)

[82] M. Lin, M. Chang, and D. Chen, “Efficient algorithms for reliability analysis of distributed computing systems,” *Information Sciences*, vol. 117, no. 1-2, pp. 89–106, 1999. (Cité en page 54.)

[83] M. Lin, D. Chen, and M. Horng, “The reliability analysis of distributed computing systems with imperfect nodes,” *The Computer Journal*, vol. VOL. 42, 1999. (Cité en page 54.)

[84] X. Qin and H. Jiang, “Dynamic, reliability-driven scheduling of parallel real-time jobs in heterogeneous systems,” in *Proceedings of the 30th International Conference on Parallel Processing (ICPP 2001*, (Valencia, Spain), pp. 113–122, Sept. 2001. (Cité en page 54.)

[85] S. Srinivasan and N. Jha, “Safety and reliability driven task allocation in distributed systems,” *IEEE Transactions on Parallel and Distributed Systems*, vol. 10, pp. 238–251, Mar. 1999. (Cité en page 54.)

[86] A. Dogan and F. Özgüner, "Optimal and suboptimal reliable scheduling of precedence-constrained tasks in heterogeneous distributed computing," in *Proceedings of the 2000 International Conference on Parallel Processeing (ICPP00-Workshops)*, (Toronto, Canada), August 2000. (Cité en page 55.)

[87] N. Auluck and D. P. Agrawal, "Reliability driven, non-preemptive real time scheduling of periodic tasks on heterogeneous systems," in *Fourteenth IASTED International Conference on Parallel and Distributed Computing and Systems (PDCS 2002)*, (MIT, Cambridge, USA), pp. pp. 803–809, November 4-6 2002. (Cité en pages 55 et 56.)

[88] X. Qin, H. Jiang, and D. R. Swanson, "An efficient fault-tolerant scheduling algorithm for real-time tasks with precedence constraints in heterogeneous systems," in *Proceedings of the 31th International Conference on Parallel Processing (ICPP 2002)*, (Vancouver, British Columbia, Canada), pp. 360–386, August 2002. (Cité en page 55.)

[89] S. Shatz, J. Wang, and M. Goto, "Task allocation for maximizing reliability of distributed computer systems," in *IEEE Trans. Computers*, vol. 41, pp. 156–168, September 1992. (Cité en pages 55 et 56.)

[90] T. Yang and A. Gerasoulis, "List scheduling with and without communication delays," *Parallel Computing*, vol. 19, no. 12, pp. 1321–1344, 1993. (Cité en page 55.)

[91] A. Girault, "A novel bicriteria scheduling heuristics providing a guaranteed global system failure rate," *Dependable and Secure Computing, IEEE Transactions on*, vol. 6, no. 4, pp. 241–254, 2009. (Cité en pages 60, 63 et 64.)

[92] P. Jalote, *Fault-Tolerance in Distributed Systems.* Englewood Cliffs, New Jersey : Prentice Hall, 1994. (Cité en page 60.)

[93] H. Kopetz and G. Bauer, "The time-triggered architecture," *Proceedings of the IEEE*, vol. 91, pp. 112–126, oct 2003. (Cité en page 60.)

[94] 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. (Cité en page 61.)

[95] W. Torres-Pomales, "Software fault tolerance : A tutorial," October 2000. National Aeronautics and Space Administration (NASA) Langley Research Center. (Cité en page 61.)

[96] M. Pizza, L. Strigini, A. Bondavalli, and F. Di Giandomenico, "Optimal discrimination between transient and permanent faults," in *High-Assurance Systems Engineering Symposium, 1998. Proceedings. Third IEEE International*, (Bethesda, MD, USA), pp. 214–223, IEEE, 1998. (Cité en page 62.)

[97] J. Reisinger and A. Steininger, "The design of a fail-silent processing node for the predictable hard real-time system mars," *Distributed Systems Engineering*, vol. 1, no. 2, p. 104, 1993. (Cité en page 62.)

[98] J.-P. Beauvais, *Etude d'algorithmes de placement de taches temps réel périodiques complexes dans un système réparti.* PhD thesis, École Centrale de Nantes, June 1996. (Cité en pages 77 et 120.)

[99] D. Zhu, R. Melhem, D. Mosse, and E. Elnozahy, "Analysis of an energy efficient optimistic tmr scheme," in *Parallel and Distributed Systems, 2004. ICPADS 2004. Proceedings. Tenth International Conference on*, pp. 559–568, IEEE, 2004. (Cité en page 80.)

[100] 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. (Cité en page 81.)

[101] Z. Jun, E. H. Sha, Q. Zhuge, J. Yi, and K. Wu, "Efficient fault-tolerant scheduling on multiprocessor systems via replication and deallocation," *International Journal of Embedded Systems*, vol. 6, no. 2-3, pp. 216–224, 2014. (Cité en page 88.)

[102] V. Zivojnovic, J. M. Velarde, C. Schlager, and H. Meyr, "Dspstone : A dsp-oriented benchmarking methodology," in *Proceedings of the International Conference on Signal Processing Applications and Technology*, pp. 715–720, 1994. (Cité en page 99.)

[103] R.-T. S. group, "Storm." `http://storm.rts-software.org`. (Cité en page 101.)

[104] H. Kopetz and G. Grunsteidl, "Ttp-a protocol for fault-tolerant real-time systems," *Computer*, vol. 27, no. 1, pp. 14–23, 1994. (Cité en page 106.)

[105] R. Kawamura, K.-i. Sato, and I. Tokizawa, "Self-healing atm networks based on virtual path concept," *Selected Areas in Communications, IEEE Journal on*, vol. 12, no. 1, pp. 120–127, 1994. (Cité en page 106.)

[106] S. Han and K. G. Shin, "A primary-backup channel approach to dependable real-time communication in multihop networks," *Computers, IEEE Transactions on*, vol. 47, no. 1, pp. 46–61, 1998. (Cité en pages 106, 114 et 121.)

[107] Q. Zheng and K. G. Shin, "Fault-tolerant real-time communication in distributed computing systems," in *Fault-Tolerant Computing, 1992. FTCS-22. Digest of Papers., Twenty-Second International Symposium on*, pp. 86–93, IEEE, 1992. (Cité en page 106.)

[108] C. H. Yang and S. Hasegawa, "Fitness-failure immunization technology for network services survivability," in *Global Telecommunications Conference, 1988, and Exhibition.'Communications for the Information Age.'Conference Record, GLOBECOM'88., IEEE*, pp. 1549–1554, IEEE, 1988. (Cité en page 106.)

[109] J. E. Baker, "A distributed link restoration algorithm with robust preplanning," in *Global Telecommunications Conference, 1991. GLOBECOM'91.'Countdown to the New Millennium. Featuring a Mini-Theme on : Personal Communications Services*, pp. 306–311, IEEE, 1991. (Cité en page 106.)

[110] J. Anderson, B. T. Doshi, S. Dravida, and P. Harshavardhana, “Fast restoration of atm networks,” *Selected Areas in Communications, IEEE Journal on*, vol. 12, no. 1, pp. 128–138, 1994. (Cité en page 106.)

[111] G. Ranjith, C. S. R. Murthy, and G. Krishna, “A distributed primary-segmented backup scheme for dependable real-time communication in multihop networks,” in *Parallel and Distributed Processing Symposium, International*, vol. 2, pp. 0139–0139, IEEE Computer Society, 2002. (Cité en page 107.)

[112] S. Han and K. G. Shin, “Experimental evaluation of failure-detection schemes in real-time communication networks,” in *Fault-Tolerant Computing, 1997. FTCS-27. Digest of Papers., Twenty-Seventh Annual International Symposium on*, pp. 122–131, IEEE, 1997. (Cité en page 121.)

[113] L. Zhang, S. Deering, D. Estrin, S. Shenker, and D.Zappala, “Rsvp : A new resource reservation protocol,” *IEEE Network*, vol. 7, pp. 8–18, 1993. (Cité en page 124.)

[114] J. L. Melin, *Qualité de service sur IP*. Eyrolles, 2001. (Cité en page 125.)

Liste des publications et communications

[1] Sara Khernane, Abdelaziz Khernane and Chafik Arar, "Efficient Fault-Tolerant Scheduling On Computer Network by using Particle Swarm Optimization," in *The 6th International Conference on Metaheuristics and Nature Inspired Computing, META'16 Morocco, Oct 27-31 2016.*

[2] Chafik Arar, and Mohamed Salah Khireddine. "Hybrid Software Redundancy Approach for Building Reliable Communication in Multi-BUS Heterogeneous Systems." *International Journal of Reliability, Quality and Safety Engineering*, vol. 23, no. 04,50013, 2016.

[3] C. Arar, M. S. Khireddine, "An Efficient Fault-Tolerant Multi-Bus Data Scheduling Algorithm Based on Replication and Deallocation,"*Cybernetics and Information Technologies*, vol. 16, no. 02, pp. 69–84, 2016.

[4] C. Arar, M. S. Khireddine, A. Belazoui, and R. Megulati, "An energy-efficient fault-tolerant scheduling algorithm based on variable data fragmentation," in *5th IFIP TC 5 International Conference, CIIA 2015 Saida, Algeria, May 20-21*, IFIP Advances in Information and Communication Technology, 2015.

[5] C. Arar and M. s. Khiereddine, "A reliable multi-bus fault-tolerant scheduling algorithm based on variable data fragmentation," *Asian Journal of Information Technology*, vol. 14, no. 02, pp. 67–73, 2015.

[6] C. Arar, et al. , "Reliable fault-tolerant multi-bus scheduling algorithm," *International Journal of Computer Applications*, vol. 71, no. 12, pp. 11–15, 2013.

[7] C. Arar, et al, "Energy-efficient fault-tolerant scheduling approach for embedded real time systems," *International Journal of Electrical Energy*, vol. 1, no. 4, pp. 274–278, 2013.

[8] C. Arar, et al, "A reliable fault-tolerant scheduling algorithm for real time embedded systems," in *Proceedings of Workshop DECS (ERCIM/EWICS Workshop on Dependable Embedded and Cyber-physical Systems) of the 32nd International Conference on Computer Safety, Reliability and Security*, 2013.

[9] C. Arar, et al, "Energy-efficient fault-tolerant scheduling approach for embedded real time systems," in *6th International Conference on Computer and Electrical Engineering (ICCEE 2013)*, IACSIT, 2013.

[10] C. Arar, et al, "Fault-tolerant real-time scheduling algorithm for energy-aware embedded systems," *SAFECOMP 2013 FastAbstract*, 2013.

[11] H. Bettahar, C. Arar, and A. Boubdallah, "An efficient qos server selection protocol for duplicated layered multicast servers," in *Information and Communication Technologies : From Theory to Applications, 2004. Proceedings. 2004 International Conference on*, pp. 675–676, IEEE, 2004.

« Redondance logicielle pour la tolérance aux fautes des communications »

Résumé : Le travail présenté dans cette thèse, se place dans le cadre général de la conception des systèmes sûrs de fonctionnement. Plus particulièrement, il s'intéresse aux systèmes, embarqués et temps réels, et il traite essentiellement l'aspect de la communication dont le rôle conditionne largement la capacité d'un système à respecter la ponctualité et les exigences de la sûreté de fonctionnement.
La proposition, la définition et la caractérisation d'algorithmes d'ordonnancement temps réel tolérants aux fautes adaptés aux architectures hétérogènes, sont les principaux objectifs de notre travail. Notre première proposition dite *Approche d'ordonnancement fiable de communication basée sur la fragmentation variable des données*, est adaptée aux architectures basées sur les communications par bus. La deuxième approche dite *Approche d'ordonnancement fiable de communication basée sur la désallocation des données*, cible les mêmes architectures, et utilise un mécanisme de chevauchement de données, pour assurer des communications optimales et sans fautes. Alors que la troisième proposition, dite *Approche d'ordonnancement fiable de communication temps réel basée sur les chemins de sauvegarde fragmentés*, est destinée pour les architectures à base de réseaux de communications multi sauts.
Dans ce travail, on a essayé de contourner la difficulté de proposer des algorithmes d'ordonnancement tolérants aux fautes des processeurs et des communications temps réel ; offrant à la fois une optimalité théorique du point de vue de l'utilisation des ressources de calcul et de communication, mais également une efficacité pratique traduite par une sûreté de fonctionnement maximisée. **Mots-clés** : redondance logicielle, systèmes temps réel embarqués, architectures hétérogènes, tolérance aux fautes, heuristiques d'ordonnancement, fragmentation variable de données, désallocation, sauvegarde fragmentée, fiabilité.

« Software redundancy for communications fault-tolerance »

Abstract : The work presented in this thesis, is located in the general context of the design of fault tolerant systems. In particular, it is dedicated to real time embedded systems, and it mainly deals with communication's aspect, whose role largely determines the ability of a system to maintain the guaranteed timeliness and the requirements of dependability.
The proposal, the definition and characterization of fault-tolerant real-time scheduling algorithms, dedicated to heterogeneous architectures, are the main objectives of our work. Our first proposition is called *A reliable communication fault-tolerant scheduling approach based on variable data fragmentation*, and is adapted to multi-bus communication architectures. The second approach, called *An efficient fault-tolerant multi-bus data scheduling algorithm based replication and deallocation* target same architectures and uses a data overlap mechanism, to ensure optimum communications and without faults. While the third proposal called *A reliable real-time communication fault-tolerant scheduling approach based on fragmented backup paths*, is for architectures based on multi-hop communications networks.
In this work, we try to around the difficulty of proposing fault-tolerant scheduling algorithms for processor and real-time communications, offering both a theoretical optimality from the perspective of the use of computing and communications resources, but also a practical efficiency resulted in an maximized dependability.
Keywords : embedded real-time systems, heterogeneous distributed architectures, software redundancy, scheduling heuristics, fault-tolerance, variable data fragmentation, deallocation, fragmented backups, reliability.