République Algérienne Démocratique Et Populaire
Ministère de l'enseignement supérieur et de la recherche scientifique

Université colonel HADJ LAKHDAR –BATNA-
Faculté des sciences de l'ingénieur
Département d'informatique

Mémoire

présenté

pour obtenir le grade de Magistère en informatique

par

Malika Bachir

Spécialité : Système d'Information et de Communication (SIC)

Titre

Tolérance aux fautes des systèmes temps-réel embarqués basée sur la redondance

---

Encadreur : Dr Zidani Abdelmadjid

Co-Encadreur : Dr Kalla Hamoudi

Composition du jury :

Dr	Mohammed Benmohammed *(prof.)*	Président	*Université de Constantine*
Dr	Belami Azzedine *(MCA)*	Examinateur	*Université de Batna*
Dr	Belattar Brahim *(MCA)*	Examinateur	*Université de Batna*
Dr	Zidani Abdelmadjid *(MCA)*	Rapporteur	*Université de Batna*
Dr	Kalla Hamoudi *(MCB)*	Co-Rapporteur	*Université de Batna*

2010-2011

Table des matières

Chapitre 3 Tolérance aux fautes dans les systèmes distribués temps réel embarqués

Chapitre 4 Description du problème

Chapitre 5 Etat de l'Art

Chapitre 6 Méthodologies proposées pour des architectures à liaison bus

Chapitre 7 Evaluation de la méthodologie AAA-FAULT$^{t.dep}$

Liste des figures

Liste des tableaux

Chapitre 1

Introduction générale

1.1. Problématique générale

Les systèmes *temps réel embarqués* ont depuis toujours acquis une importante place dans le développement des systèmes informatiques, ils sont de plus en plus utilisés dans divers domaines applicatifs importants comme les véhicules (automobiles, robots, avions, satellites, bateaux) et les systèmes de contrôle de processus industriels. Pour répondre aux nombreuses demandes de traitement, ils sont généralement composés de plusieurs processeurs spécialisés et *distribués* (interconnectés par un réseau). Une des caractéristiques importantes de ces systèmes est d'être réactif, un système réactif est un système qui réagit continûment à des stimuli externes d'un environnement à la vitesse imposée par cet environnement (la régulation de niveau d'eau dans un réservoir est un exemple). Il reçoit ces stimuli par l'intermédiaire de capteurs, il les traite en effectuant un certain nombre d'opérations et produit, grâce à des actionneurs, des sorties utilisables par l'environnement, appelées réactions.

Une application temps-réel effectue des fonctions de contrôle et de pilotage. Chaque fonctionnalité est assumée par une *tâche temps-réel*. Certaines fonctions sont critiques et ne peuvent pas être retardées, c.-à-d. que les tâches correspondantes doivent impérativement avoir terminé leur traitement sous un délai donné, à partir de leur date de réveil. Dans le cas contraire, les résultats produits sont faux et la tâche est considérée être invalide. De telles tâches sont dites *critiques*. Les systèmes temps réel embarqués sont des systèmes de traitement dont la validité est conditionnée non seulement par la correction des résultats mais surtout par les dates auxquelles ceux-ci sont délivrés. En effet, ces systèmes

sont essentiellement caractérisés par des *contraintes de temps* sur les actions à entreprendre, qu'il faut respecter de manière plus ou moins critique.

Grand nombre de systèmes distribués temps réel et embarqués, tels que l'aéronautique et le secteur militaire, réalisent des tâches complexes et critiques ce qui les rend plus sensibles aux *fautes,* ils sont soumis à de fortes contraintes en terme de temps et de fiabilité, si elles ne sont pas respectées elles peuvent conduire à une défaillance grave qui peut engendrer des conséquences catastrophiques (perte d'argent, de temps ou pire de vies humaines). La solution de *distribution et d'ordonnancement temps réel* s'impose afin d'assurer le respect de ces contraintes temporelles. L'ordonnancement consiste à organiser dans le temps la réalisation des tâches, compte tenu des contraintes temporelles (délais, contraintes d'enchaînement) et des contraintes portant sur la disponibilité des ressources requises. Puisque la présence de certaines fautes qu'elles soient matérielles ou logicielles, accidentelles ou intentionnelles est inévitable, ce type de systèmes doit être *sûr de fonctionnement*. La sûreté de fonctionnement d'un système informatique est définie comme la propriété qui permet à ses utilisateurs de placer une confiance justifiée dans le service qu'il délivre [21]. Pour assurer la sûreté de fonctionnement des systèmes temps réel critiques plusieurs méthodes sont utilisées dans la littérature [2, 21], parmi plusieurs autres on trouve la technique de *la tolérance aux fautes* qui est l'objet de ce mémoire. La tolérance aux fautes est définie par la capacité du système de se conformer à ses spécifications en dépit de présence des fautes dans n'importe lequel de ses composants (logiciels ou matériels). La *redondance* des composants matériels ou logiciels est la technique la plus efficace utilisée pour assurer la tolérance aux fautes.

La croissance continue de la complexité des processeurs et l'arrivée de nouvelles technologies indiquent la persistance du problème des fautes des processeurs, qui peuvent devenir pire dans l'avenir [1], donc ce mémoire est consacré à l'étude du problème de la tolérance aux fautes des processeurs dans les systèmes temps réel embarqués dans un environnement distribué. L'architecture matérielle de ces systèmes est composée de plusieurs calculateurs, capteurs et actionneurs. Ces composants sont reliés par un réseau de communication (dans notre cas, un bus). Dans les systèmes embarqués, le nombre de ces éléments est réduit au strict minimum pour satisfaire les contraintes de coût, de consommation électrique, de taille, …etc.

Pour réaliser des systèmes temps réel embarqués tolérants aux fautes dans un environnement réparti, nous cherchons plus particulièrement à résoudre un problème NP-difficile appelé problème de la distribution/ordonnancement temps réel tolérant aux fautes et *prédictif*. Nous tenons à proposer des techniques qui permettent de placer et d'ordonnancer les composants logiciels (tâches) sur les composants matériels tout en satisfaisant les contraintes temporelles, les contraintes de distribution et les contraintes de tolérance aux fautes.

- Les contraintes temporelles consistent à définir une limite ou une borne de l'exécution de l'algorithme sur le matériel.

- Les contraintes de distribution consistent à définir une relation d'exclusion entre certains composants matériels et certains composants logiciels (par exemple, une telle tâche ne peut pas s'exécuter sur tel processeur).
- Les contraintes de tolérance aux fautes définissent des hypothèses sur le nombre maximal de fautes des processeurs que le système doit tolérer.

En fin, la distribution/ordonnancement doit être prédictive ce qui signifie qu'il est possible de déterminer, avant la mise en exploitation du système, si les contraintes temps réel sont respectées ou non en présence et en absence de fautes.

1.2. Apport du mémoire

Dans ce mémoire, nous proposons deux heuristiques, basées sur la méthodologie AAA implantée dans SynDEx, qui permettent la réalisation des systèmes répartis temps réel et embarqués sûrs de fonctionnement en optant à la technique de la tolérance aux fautes des composants matériels et plus particulièrement les processeurs. Vu que les systèmes embarqués exigent certaines caractéristiques comme : la taille, le coût, le poids, la consommation électrique qui doivent être réduits au minimum, nous nous sommes intéressés uniquement aux *solutions logicielles* basées sur la redondance des *composants logiciels* pour assurer la tolérance aux fautes. Nous avons considéré le cas le plus simple qui est les fautes permanentes d'un seul processeur dans un système silence sur défaillance (ou bien fonctionne, et donne un résultat correct délivré à temps ou bien il est en panne, et ne fait rien).

1.2.1 Méthodologie *AAA-FAULT*$^{t.ind}$

Cette première méthodologie proposée consiste à générer automatiquement une distribution et un ordonnancement d'une architecture logicielle composée de tâches indépendantes (pas de communication entre elles) sur une architecture matérielle distribuée et hétérogène munie d'un bus de communication (liaison multipoint). La tolérance aux fautes est assurée par le principe de la redondance *active* des composants logiciels. Elle permet de tolérer hors-ligne une ou plusieurs fautes permanentes d'un seul processeur en deux phases. La première phase consiste à transformer le graphe d'algorithme non redondant en un nouveau graphe redondant avec des tâches dépendantes, alors que la deuxième consiste à générer une distribution/ordonnancement de ce nouveau graphe sur le graphe d'architecture matérielle, tout en garantissant que les contraintes citées précédemment soient respectées.

1.2.2 Méthodologie *AAA-FAULT*$^{t.dep}$

La méthodologie *AAA-FAULT*$^{t.dep}$ consiste à générer automatiquement une distribution et un ordonnancement temps réel d'une architecture logicielle composée de tâches dépendantes sur une architecture matérielle distribuée et hétérogène munie d'un bus de communication (liaison multipoint). La tolérance aux fautes est assurée par le principe de la redondance *passive* des composants logiciels. Elle permet de tolérer une ou plusieurs fautes permanentes d'un seul processeur en deux phases. La première phase consiste à

transformer le graphe d'algorithme non redondant en un nouveau graphe redondant. La deuxième phase, consiste à allouer spatialement et temporellement les composants logiciels de ce nouveau graphe d'algorithme avec redondance sur les composants matériels tout en garantissant le respect des mêmes contraintes. Dans cette heuristique l'implantation d'un mécanisme de détection d'erreurs est obligatoire.

1.3. Plan du mémoire

Le mémoire est divisé en sept chapitres :

- ***Le chapitre 1*** est une introduction générale sur le problème de la tolérance aux fautes dans les systèmes distribués temps réel embarqués.
- ***Le chapitre 2*** définit des notions sur ces systèmes et présente leurs caractéristiques et leurs spécifications. Par la suite, le problème de la conception de ces systèmes est exposé en présentant uniquement les principes des méthodologies basées sur la théorie d'ordonnancement. Le chapitre se termine par une présentation de l'algorithme de distribution et d'ordonnancement de SynDEx qui est la base de notre travail.
- ***Le chapitre 3*** présente une technique qui assure la sûreté de fonctionnement et qui est l'objet de ce mémoire, c'est la tolérance aux fautes. Tout d'abord une introduction aux notions de base de la sûreté de fonctionnement est présentée, ensuite les terminologies liées à la tolérance aux fautes et ses phases de réalisation sont exposées. A la fin de ce chapitre, le problème de la génération de distribution/ordonnancement temps réel et tolérant aux fautes est présenté.
- ***Le chapitre 4*** présente une modélisation du problème de la distribution et d'ordonnancement temps réel en décrivant le modèle d'algorithme, le modèle d'architecture et le modèle d'exécution. Le modèle d'algorithme décrit les spécifications des composants logiciels, le modèle d'architecture décrit les spécifications des composants matériels et le modèle d'exécution décrit le mode d'exécution des composants logiciels sur les composants matériels.
- ***Le chapitre 5*** présente un état de l'art des méthodes existantes qui réalisent une génération d'une distribution et d'un ordonnancement temps réel tolérante aux fautes. Plusieurs travaux ont effectivement traité ce genre de problème, mais nous cherchons à réduire au mieux la longueur de la distribution/ordonnancement temps réel tolérante aux fautes.
- ***Dans le chapitre 6***, nous proposons deux heuristiques pour traiter ce problème dans un environnement distribué et hétérogène muni d'un réseau de communication composé uniquement de liaison multipoint (bus).
- ***Le chapitre 7*** présente quelques résultats donnés par la méthodologie AAA-FAULT$^{t.dep}$.

Nous concluons enfin par un résumé des problèmes posés et traités dans ce mémoire et nous présentons les perspectives.

Chapitre 2

Introduction à l'ordonnancement dans les systèmes distribués temps réel embarqués

Ce chapitre a pour objectif de présenter le cadre de départ de notre travail. Nous introduisons dans un premier temps les principales caractéristiques des systèmes temps réel embarqués distribués. Nous présentons par la suite, le problème de distribution et d'ordonnancement dans ces systèmes en donnant tout d'abord les terminologies liées à ce problème, puis une classification des algorithmes d'ordonnancement pouvant le résoudre. Et enfin nous nous concentrons sur un algorithme de distribution et d'ordonnancement que nous exploitons dans ce travail.

2.1. Définitions

2.1.1. Système réactif et système temps réel

Selon la classification des systèmes informatiques introduits par D-Harel et A-Pnueli [6], un système est transformationnel, interactif, ou réactif.

Définition 1 (système transformationnel) : Le système transformationnel s'exécute avec des données en entrée et après un certain temps il termine en fournissant un résultat en sortie.

Définition 2 (système interactif) : Le système interactif réagit constamment avec son environnement mais à sa vitesse propre comme les systèmes d'exploitation.

Définition 3 (système réactif) : Un système réactif est tout système de traitement de l'information qui répond continuellement à des stimuli externes d'un environnement à la

vitesse imposée par ce dernier. En d'autre terme, *un système réactif réagit continuellement et instantanément à des événements ou stimuli qu'ils soient internes ou externes* [5].

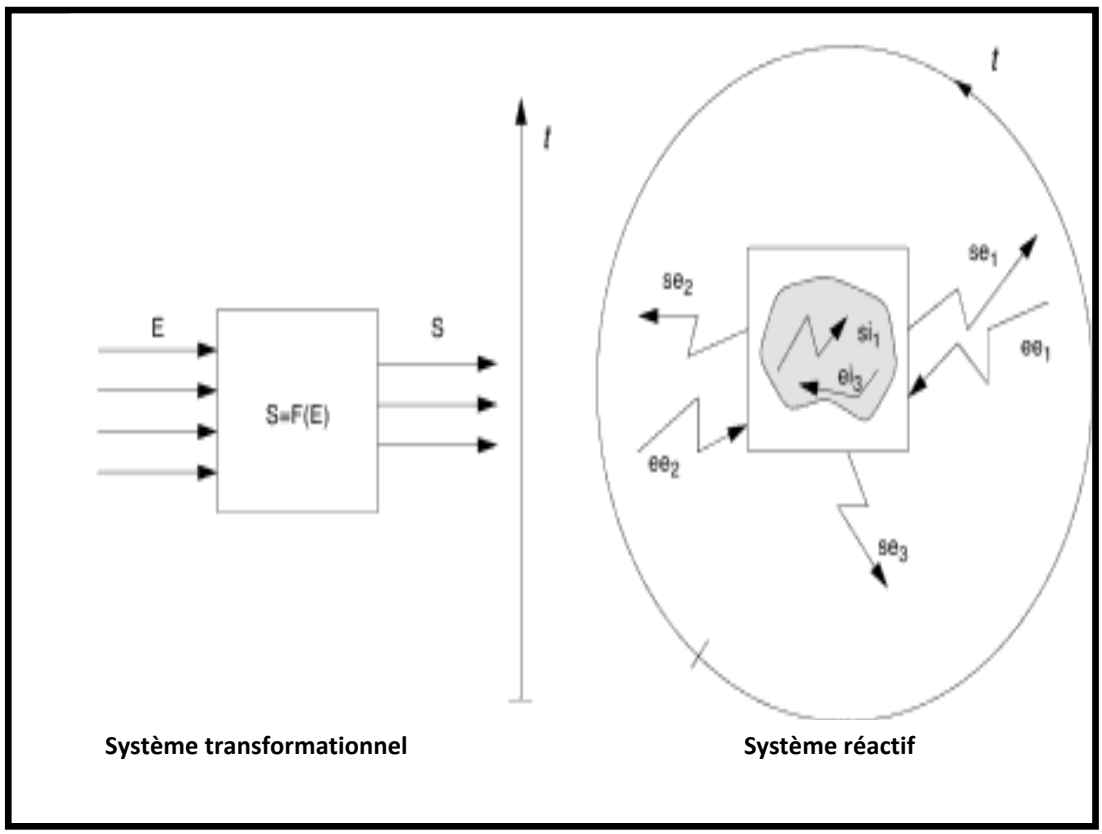


Figure2.1 : système transformationnel versus système

Nous pouvons représenter un système réactif comme étant composé de :

- L'environnement à contrôler.
- Un système de contrôle qui représente le système d'exploitation au dessus duquel l'application sera exécutée.
- l'application.

L'introduction de contraintes de temps inhérentes à l'application en spécifiant des délais avec lesquels seront fournies les réponses aux stimuli, permet d'introduire la notion de *système temps réel.*

Définition 4 (système temps réel) : Est un système qui permet de réaliser certaines tâches ou fonctions en réagissant avec son environnement qui lui-même évolue avec le temps, il exploite des ressources limitées. La correction d'un système temps réel ne dépend pas seulement des valeurs des résultats produits, mais également des délais dans lesquels ces résultats sont réalisés.

En d'autre terme, *un système est temps réel s'il est capable de respecter des échéances temporelles* [5].

La figure 2.2 schématise le comportement du système temps réel.

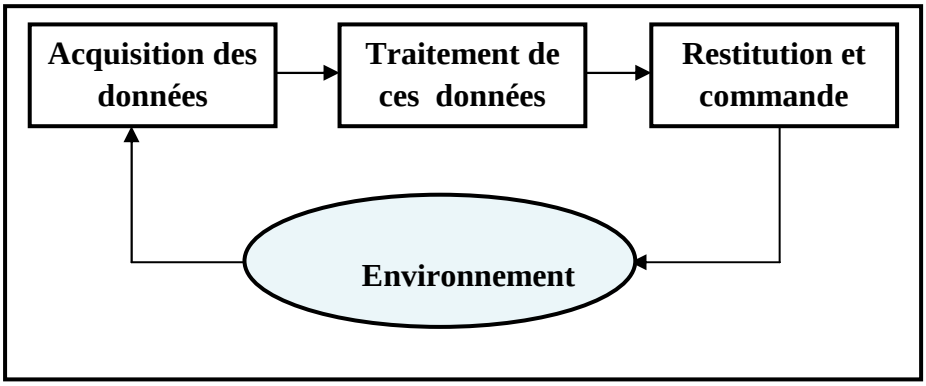


Figure2.2 : Schéma d'un système temps

En fonction du niveau de sévérité des contraintes temporelles qu'un système temps réel doit satisfaire, on considère usuellement deux grandes catégories :

- Système temps réel souple (ou système à contraintes relatives)
- Système temps réel dur (ou système à contraintes strictes)

a) Système temps réel souple

Dans ces systèmes, un retard dans l'obtention du résultat réduit progressivement son intérêt (dégradation des performances), mais les pénalités ne sont pas dramatiques. L'application de distributeur de billets est un exemple sur ce type de système.

b) Système temps réel dur

Dans ces systèmes, la réponse dans les délais est vitale. L'absence ou le retard d'une réponse est catastrophique, ainsi que celle erronée. Le système de contrôle d'avion, de freinage ABS et de détection de missile sont de bons exemples.

Remarque :

Le plus souvent, les systèmes temps réel sont embarqués, c'est-à-dire, physiquement liés au processus contrôlé.

Exemples de systèmes réactifs temps réel

- Contrôle des fonctions d'un véhicule automobile.
- Guidage d'une fusée.
- Simulation de vol.
- Contrôle d'une unité de production industrielle (usine chimique, traitement des centrales nucléaires).
- Contrôle des fonctions d'un satellite de télécommunication.

Dans un système temps réel, un processus est appelé tâche. Les états classiques d'une tâche et les transitions possibles entre eux sont donnés par la figure suivante :

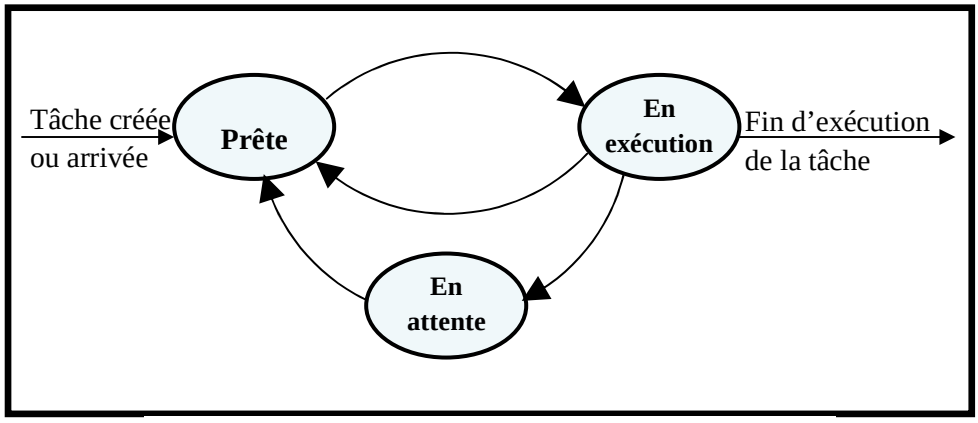


Figure2.3 : Etats d'une tâche

Une tâche temps réel est associée à des contraintes de temps et de ressources. Dans un système temps réel, les contraintes temporelles portent essentiellement sur les dates de début et de fin d'exécution des tâches. Les contraintes temporelles sont modélisées par les paramètres suivants [34] :

Date de création : c'est la date d'arrivée de la tâche au processeur (date de sa création ou éventuellement date à laquelle une tâche transférée par un autre processeur est reçue).

Temps d'exécution : c'est la durée d'exécution de la tâche (le temps entre la création d'une tâche et sa fin d'exécution). Il est déterminé par des simulations ou par une étude poussée du code source avant l'exécution.

Echéance : la date au plus tard à laquelle une tâche peut terminer son exécution.

Période : est le temps entre deux créations consécutives d'une tâche

Date de début d'exécution : la date à laquelle une tâche peut commencer son exécution. Certaines tâches sont dépendantes et ne peuvent commencer leurs exécutions que si les tâches qui les précèdent ont terminé.

En ce qui concerne la période, les tâches peuvent être classifiées comme tâches périodiques et tâches apériodiques. Les tâches périodiques se produisent à plusieurs reprises après une durée fixe de temps, tandis que les tâches apériodiques sont déclenchées par certains événements comprenant ceux externes au système. Les temps d'arrivée de ces événements ne sont pas connus à l'avance.

Les tâches sporadiques sont un cas particulier des tâches apériodiques où deux exemples successifs quelconques d'une tâche ont lieu séparément par un temps minimum.

En ce qui concerne les temps d'arrivée des tâches, les systèmes temps réel peuvent être classifiés dans les deux classes suivantes :

- **Systèmes statiques :** Sont ceux où les dates de début d'exécution de toutes les tâches sont connues avant l'exécution du système. Ils se composent typiquement des tâches périodiques.
- **Systèmes dynamiques :** Sont ceux où les tâches peuvent arriver au système pendant son fonctionnement (exécution) et les dates de début d'exécution ne sont pas connues à l'avance. Plus souvent, les systèmes dynamiques doivent considérer des tâches périodiques ou apériodiques produites par des événements externes.

Exactement comme dans un système d'exploitation, dans un système temps réel plusieurs tâches peuvent arriver en même temps. Le système doit décider quelle tâche doit s'exécuter d'abord. Cette décision dépend d'un algorithme appelé algorithme d'ordonnancement que nous allons voir en détails dans la suite de ce chapitre.

2.1.2. Système distribué et système embarqué

Définition 5 (système distribué) : Un système distribué est un ensemble d'ordinateurs indépendants, liés par un ensemble de moyens de communication comme la mémoire partagée, le bus et la liaison point-à-point qui apparait à l'utilisateur comme un système unique et cohérent. WWW, banques et système de fichier distribué sont des exemples parmi plusieurs autres de ce type de système.

Les systèmes distribués sont populaires pour plusieurs raisons :

- **Accès distant :** un même service peut être utilisé par plusieurs acteurs situés à des endroits différents.
- **Redondance :** des systèmes redondants permettent de pallier une faute matérielle ou de choisir le service équivalent avec le temps de réponse le plus court.
- **Performance :** la mise en commun de plusieurs unités de calcul permet d'effectuer des calculs parallélisables en des temps plus courts.
- **Confidentialité :** les données brutes ne sont pas disponibles partout, au même moment, seules certaines vues sont exportées.

Définition 6 (système embarqué) : Un système embarqué est une combinaison autonome de matériel et de logiciel (électronique plus informatique) dédié à réaliser généralement

une *tâche précise* en interaction avec son environnement et en respectant des contraintes souvent sévères tel que : consommation, poids, fiabilité, temps de réponse, coût, …etc.

Remarque :

Un système embarqué est souvent utilisé dans un environnement réactif soumit à des contraintes temps réel.

La conception des systèmes temps réel embarqués et distribués nécessite la description et l'expression des contraintes temps réel, la prédiction et l'estimation des temps de réponses et la sélection de l'architecture du matériel et du logiciel. Dans ce qui suit nous allons présenter ces spécifications.

2.2. Spécifications des systèmes distribués temps réel embarqués

La spécification des systèmes distribués temps réel et embarqués est réalisée en trois phases [1] : La spécification fonctionnelle, la spécification architecturale et la spécification des contraintes. La spécification fonctionnelle consiste à définir l'algorithme représentant les fonctionnalités du système, la spécification architecturale décrit le matériel sur lequel va s'exécuter l'algorithme et la spécification des contraintes consiste à attribuer des contraintes temporelles et matérielles à l'exécution de l'algorithme sur l'architecture.

2.2.1. Spécification algorithmique

Les fonctionnalités des systèmes temps réel embarqués combinent contrôle et traitement de données. Les fonctionnalités de traitement de données consistent à effectuer des calculs sur des données, alors que celles de contrôle consistent à mettre en séquence ces calculs [8]. L'algorithme représentant ces fonctionnalités est constitué de plusieurs composants logiciels où chaque composant peut être lui-même composé de plusieurs sous-composants et assure une fonctionnalité spécifique du système.

Dans ce mémoire, nous avons utilisé la spécification flot de données pour modéliser l'algorithme temps réel embarqué et distribué, où les nœuds sont les tâches de calcul de l'algorithme et les arcs sont les dépendances de données entre tâches. Une dépendance de données présentée par $A \rightarrow B$, désigne que la tâche B ne peut s'exécuter que lorsque la tâche A termine son exécution ; C'est-à-dire B s'exécute lorsque ses données d'entrée sont présentées.

Une tâche sans prédécesseur (resp. sans successeur) représente une interface d'entrée, c'est-à-dire un capteur, (resp. une interface de sortie, c'est-à-dire un actionneur) avec l'environnement.

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

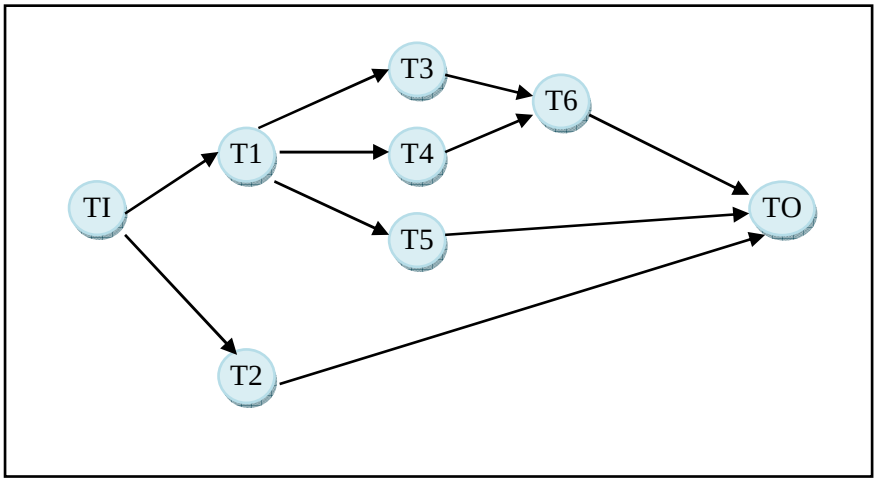


Figure2.4 : Exemple d'architecture logicielle

2.2.2. Spécification matérielle

L'architecture matérielle consiste à spécifier le matériel qui doit être utilisé pour satisfaire la réalisation d'un système distribué temps réel embarqué. Dans les systèmes distribués, l'architecture matérielle est composée de plusieurs machines mono ou multiprocesseurs, de moyens de communication et de plusieurs capteurs et actionneurs.

La spécification de l'algorithme est très liée à la spécification du matériel à titre d'exemple, les contraintes temporelles(échéance, durée d'exécution, …) des composants logiciels sont issues du type de matériel de calcul (ou de traitement) et de communications utilisées ; ainsi par exemple, pour gérer les communications inter-processeur entre des composants logiciels dépendants placés sur des processeurs distincts, il est nécessaire de connaitre le type du réseau de communication (bus, liaison point à point, mémoire partagée, …). Pour cela, la spécification de l'architecture matérielle consiste à caractériser tous les composants de l'architecture et à choisir la topologie du réseau de communication.

La Figure2.5 [1], représente un exemple d'une architecture distribuée. L'architecture est composée de cinq processeurs (P1 à P5), de trois mémoires locales (M1 à M3), d'un capteur et d'un actionneur. Les processeurs communiquent entre eux via un réseau de communication composé d'une mémoire partagée et d'une liaison physique de communication.

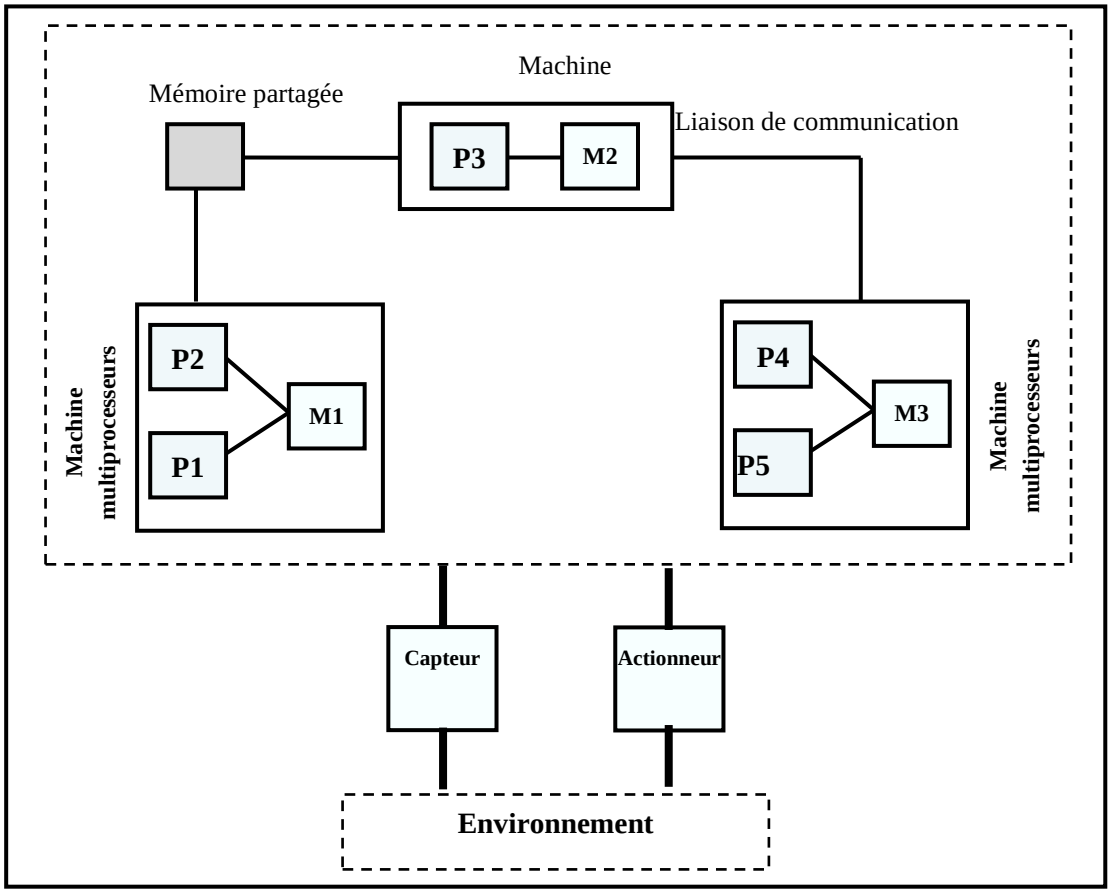


Figure2.5 : Exemple d'une architecture distribuée

2.2.3. Contraintes temporelles et d'embarquabilités

Le système temps réel embarqué permet, d'une part, de réaliser une spécification fonctionnelle conduisant à l'algorithme (ou l'ensemble d'algorithmes) qu'il faudra implanter sur une spécification matérielle et d'autre part de réaliser une spécification de contraintes. En ce qui concerne les contraintes temps réel, pour chaque composant logiciel de l'algorithme, on détermine une date de début d'exécution, et une date de fin d'exécution avant une échéance. Si ces dates ne sont pas respectées l'environnement risque de ne plus être contrôlé, ceci pouvant conduire à des conséquences catastrophiques. *Les contraintes temporelles attribuées aux composants de l'algorithme peuvent être des mesures exactes, moyennes, ou majoritaires ; ceci dépend des moyens utilisés pour obtenir ces mesures*[1]. En plus de ces contraintes temps réel, le système est soumis à des contraintes technologiques d'embarquabilité (matérielles) et de coût, qui incitent à minimiser les ressources matérielles nécessaires à sa réalisation.

2.3. Problème de distribution et d'ordonnancement temps réel

Etant donné, que notre objectif dans ce travail est de proposer une nouvelle heuristique pour la résolution du problème de la tolérance aux fautes dans les systèmes répartis temps réel embarqué, l'algorithme proposé est un ensemble de tâches qui s'exécutent sur plusieurs processeurs et satisfaisant certaines contraintes temporelles d'où l'ordonnancement et la distribution de ces tâches sur les différents processeurs s'imposent.

Pour bien présenter notre objectif, nous commençons d'abord par l'introduction de certaines définitions et terminologies concernant le problème de la distribution et d'ordonnancement temps réel.

2.3.1. Terminologies

Définition 7 (Ordonnancement temps réel) : Le problème d'ordonnancement met en relation des tâches à exécuter, des machines pour les exécuter et le temps [9]. Il détermine l'ordre d'exécution des tâches selon les critères (contraintes) spécifiés (échéance, temps de réponse, durée d'exécution, contraintes d'enchainement,…). L'ordonnancement est dit faisable s'il respecte toutes les contraintes temporelles.

Définition 8 (Ordonnancement statique et dynamique) : dans l'ordonnancement statique, la séquence d'exécution des tâches est déterminée avant le début d'exécution de l'algorithme et ne change pas durant cette exécution, tandis que dans l'ordonnancement dynamique la séquence déterminée au préalable est mise à jour et réordonnancée en fonction des nouvelles tâches créées.

Définition 9 (Allocation statique et dynamique) : l'allocation statique consiste à placer un ensemble de tâches sur un réseau de processeurs avant leur exécution, en respectant et optimisant certains critères. Par contre l'allocation dynamique désigne le placement des tâches créées durant l'exécution.

Définition 10 (Fonction de coût) : *Le rôle d'une fonction de coût est d'associer un poids à chaque composant logiciel, et ceci afin de calculer un ordre d'exécution entre ces composants.*

Définition 11 (Algorithme de distribution/ordonnancement temps réel) : c'est l'algorithme qui permet d'ordonnancer et de placer les tâches sur les processeurs en utilisant la fonction de coût dans un environnement distribué pour optimiser les performances du système temps réel.

Remarque :

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

2.3.2. Présentation du problème de distribution/ordonnancement

Le problème de distribution/ordonnancement consiste à trouver une allocation spatiale (distribution) et temporelle (ordonnancement) des composants logiciels sur l'architecture matérielle. Cette allocation doit être valide, c'est-à-dire elle satisfait les contraintes temporelles et matérielles. Ce problème devient d'ordre d'optimisation, lorsqu'il s'agit de rechercher une allocation valide qui doit optimiser en plus certains critères comme la réduction des coûts de communication. Si une telle solution existe, elle est dite optimale. Un problème d'optimisation peut être soit NP-difficile soit polynomiale [10].

On peut formaliser le problème de distribution et d'ordonnancement temps réel comme suit :

Problème 1[1] : *Etant donné :*

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

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

$A : ALG \rightarrow AMH$

$T_i \rightarrow A(T_i) = (P_j , O_k)$

Qui doit satisfaire C_{mt} *et* C_{tr} *, et optimiser les critères spécifiés.*

2.4. Classes d'algorithmes de distribution et d'ordonnancement temps réel

La résolution d'un problème d'ordonnancement consiste à choisir puis à placer dans le temps des tâches sur les processeurs compte tenu de contraintes temporelles et matérielles (contraintes portant sur l'utilisation et la disponibilité des ressources requises par les tâches). Pour atteindre ce but plusieurs classes d'algorithmes d'ordonnancement se présentent dans la littérature, nous ne nous intéressons qu'aux deux grandes classes à savoir les algorithmes enligne/hors-ligne et les algorithmes exacts/approchés.

[1] Problème de distribution et d'ordonnancement temps réel

2.4.1. Algorithmes hors-ligne et enligne

Selon les caractéristiques du système à réaliser et les contraintes temporelles qui sont connues avant l'exécution du système ou fixées au moment de son exécution, on trouve les algorithmes hors-ligne et les algorithmes enligne.

Dans les algorithmes hors-ligne, la séquence d'ordonnancement est pré-calculée avant l'exécution effective du système (ordonnancement statique) ainsi que l'allocation des tâches sur les processeurs est déterminée (allocation statique). Alors que dans les algorithmes enligne, Les décisions d'ordonnancement et le placement des tâches sur les processeurs sont prises au cours de l'exécution du système.

2.4.2. Algorithmes exacts et approchés

Lorsqu'on aborde la résolution d'un problème d'ordonnancement, on peut choisir entre deux grands types de stratégies, visant respectivement à l'optimalité des solutions par rapport à un ou plusieurs critères, ou plus simplement à leur admissibilité vis-à-vis des contraintes [10]. *Une méthode utilisant un critère d'optimisation est exacte si elle garantit l'optimalité des solutions trouvées ; sinon elle est dite approchée, ou heuristique, lorsqu'on observe empiriquement qu'elle fournit de « bonnes » solutions.*

Les algorithmes hors-ligne et enligne qui trouvent une solution optimale, si celle-ci existe, appartiennent à la classe des algorithmes exacts puisqu'ils renvoient toujours une solution optimale. Cependant, dans le cas général, ce problème est NP-difficile et la complexité est exponentielle. Trouver une solution optimale pour le problème de distribution/ordonnancement n'est pas usuellement une contrainte d'un système temps réel embarqué à contraintes strictes. C'est pourquoi, pour résoudre ce problème dans un temps polynomial, plusieurs heuristiques ont été développées dans la littérature pour chercher une solution valide et si possible proche de la solution optimale. Ces algorithmes de distribution et d'ordonnancement appartiennent à la classe des algorithmes approchés.

Parmi tous les algorithmes proposés dans la littérature, nous ne nous intéressons dans ce travail qu'aux algorithmes hors/ligne approchés basés sur celles de SynDEx.

2.5. Algorithme de distribution et d'ordonnancement de SynDEx

L'heuristique de distribution/ordonnancement implanté dans SynDEx[2] est un algorithme de type hors/ligne approché proposé dans sa première version en 1991 [11], ensuite il a été amélioré afin de pouvoir traiter des problèmes réalistes de types industriels où l'on cherche à distribuer des algorithmes de plusieurs centaines de composants logiciels sur quelques dizaines de processeurs et médias de communication.

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

2.5.1. Spécification de l'Algorithme de distribution/ordonnancement de SynDEx

Une application SynDEx est composée d'un graphe d'algorithme (définitions des tâches que l'application peut exécuter) et d'un graphe d'architecture (définitions d'un ensemble de processeurs reliés ensemble et circuits intégrés spécifiques). Exécuter une adéquation signifie l'exécution de l'heuristique cherchant l'implantation optimisée d'un algorithme donné sur une architecture donnée.

Dans le graphe d'algorithme, graphe flot de données, les sommets sont les tâches de l'algorithme et les arcs sont les dépendances de données entre ces tâches. Une tâche ne peut s'exécuter que lorsqu' elle reçoit ses données d'entrée. Elle consomme ses données et produit des données en sortie, qui sont ensuite utilisées par ses successeurs. Une tâche sans prédécesseur (resp. sans successeur) représente une interface d'entrée, c'est-à-dire un capteur, (resp. une interface de sortie, c'est-à dire un actionneur) avec l'environnement.

Le graphe d'architecture est spécifié par un graphe non orienté dans lequel les sommets désignent les processeurs et les mémoires, les arrêtes désignent les liaisons physiques entre mémoires et processeurs. Chaque processeur est composé d'une unité de calcul, d'une mémoire locale, et d'une ou plusieurs unités de communication pour communiquer avec les périphériques ou d'autres processeurs.

A chaque tâche T_i du graphe d'algorithme est associé un coût d'exécution $C_{exe}(T_i, P_k)$ sur chaque processeur P_k du graphe d'architecture. Ce temps d'exécution désigne une borne supérieure (WCET) du temps d'exécution de la tâche sur chaque processeur, il est déterminé par le concepteur du système en utilisant des méthodes statiques ou dynamiques [12]. Puisque l'architecture est hétérogène, alors ces coûts d'exécutions peuvent être distincts pour un même composant logiciel du graphe d'algorithme. De plus, à chaque dépendance de données (T_i, T_j) de l'architecture logicielle est associé un coût de transfert de données $C_{exe}(T_i, T_j, L_k)$ sur le lien de communication L_k de l'architecture matérielle.

Les contraintes matérielles sont spécifiées par l'association de la valeur 1 à $C_{exe}(T_i, P_k)$, ce qui signifie que la tâche T_i ne peut pas être implantée sur le processeur P_k . Enfin, une seule contrainte temps réel C_{tr} est prise en compte dans l'algorithme de distribution/ordonnancement, qui est la contrainte de la latence, c'est-à-dire que la longueur de la distribution/ordonnancement du graphe d'algorithme sur le graphe d'architecture doit être inférieure à un seuil défini par la latence.

2.5.2. Présentation de l'algorithme de distribution et d'ordonnancement de SynDEx

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

Tout d'abord, avant de présenter l'heuristique nous donnons les notations suivantes qui seront utilisées par cette heuristique et aussi dans le reste de ce travail :

pred(T_i) : l'ensemble des tâches prédécesseurs de la tâche T_i

succ(T_i) : l'ensemble des tâches successeurs de la tâche T_i

$C_{exe}(T_i, P_j)$ *: Le coût d'exécution de* T_i *sur le processeur* P_j

$X^{(n)}$ *: n désigne l''etape de l'heuristique, c'est-à-dire après avoir alloué la* $n^{ième}$ *tâche ; donc, X(n) désigne l'ensemble X à l'étape n de l'heuristique*

$T_{card}^{(n)}$ *: représente la liste des tâches candidates ; une tâche de ALG est dite candidate si elle est implantable, c'est-à-dire que tous ses prédécesseurs sont déjà alloués*

$T_{fin}^{(n)}$ *: représente la liste des tâches déjà allouées*

$St_{Ti,Pj}^{(n)}$ *: représente la date de début au-plus-tôt de* T_i *sur* P_j *, depuis le début [13]*

$st_{Ti,Pj}^{(n)}$ *: représente la date de début au-plus-tard de* T_i *sur* P_j *, depuis le début [13]*

$\overline{st}_{ti}^{(n)}$ *: représente la date de début au-plus-tard de* T_i *, depuis la fin [13]*

L'heuristique de distribution/ordonnancement est un algorithme glouton [14] de type ordonnancement de liste, basé sur une fonction de coût appelée la pression d'ordonnancement dont l'objectif est de minimiser la longueur de la distribution/ordonnancement.

Définition 12 (longueur d'une distribution/ordonnancement) : *La longueur de la distribution/ordonnancement d'un graphe d'algorithme sur un graphe d'architecture, noté* $R^{(n)}$*, est la durée d'exécution de ce graphe d'algorithme sur ce graphe d'architecture, c'est-à-dire la date de terminaison du dernier processeur en exécution.*

Définition 13 (Pression d'ordonnancement) : *La pression d'ordonnancement, notée* $\sigma_{Ti,Pj}^{(n)}$ *, est une fonction de coût qui induit des priorités d'ordonnancement entre les opérations d'ALG. Elle mesure à la fois la marge d'ordonnancement F(n) et l'allongement P(n) de la longueur de la distribution/ordonnancement. Elle est calculée pour chaque tâche candidate* T_i *sur chaque processeur* P_j *par :*

$$\sigma_{Ti,Pj}^{(n)} = P_{Ti,Pj}^{(n)} - F_{Ti,Pj}^{(n)} \quad (2\text{-}1)$$

Définition 14 (Pénalité d'ordonnancement) : *La pénalité d'ordonnancement, notée* $P_{Ti,Pj}^{(n)}$ *, est une fonction qui mesure l'allongement de la longueur de la distribution/ordonnancement* $R_{Ti,Pj}^{(n)}$ *après avoir allouer* T_i *sur* P_j *à la* $n^{ième}$ *étape de l'heuristique, en tenant compte des coûts de communication engendrés par l'allocation. Elle est définie par :*

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

Définition 15 (Flexibilité d'ordonnancement) : *La flexibilité d'ordonnancement, notée* $F_{Ti,Pj}^{(n)}$ *, est une fonction qui mesure la marge d'ordonnancement de* T_i *sur* P_j *à la* $n^{ième}$ *étape de l'heuristique. Elle est définie par :*

$$F_{Ti,Pj}^{(n)} = st_{Ti,Pj}^{(n)} - St_{Ti,Pj}^{(n)} \quad (2\text{-}3)$$

En utilisant l'´equation suivante [13] :

$$st_{Ti,Pj}^{(n)} = R_{Ti,Pj}^{(n)} - \bar{st}_{Ti}^{(n)} \quad (2\text{-}4)$$

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

$$\sigma_{Ti,Pj}^{(n)} = St_{Ti,Pj}^{(n)} + \bar{st}_{Ti}^{(n)} - R^{(n-1)} \quad (2\text{-}5)$$

Avant de présenter l'algorithme de distribution et d'ordonnancement de SynDEx nous décrivons en premier ces grandes lignes, alors :

- L'exécution de l'algorithme est effectuée par plusieurs itérations ; à chaque itération une liste des tâches candidates $T_{cand}^{(n)}$ est établie.
- Pour chaque tâche T_i de $T_{cand}^{(n)}$, une pression d'ordonnancement $\sigma_{Ti,Pj}^{(n)}$ sur chaque processeur P_j de AMH est calculée, puis le processeur (P_{best}) qui minimise cette pression d'ordonnancement est sélectionné.
- Parmi les couples (T_i, P_{best}), celui qui maximise la pression d'ordonnancement est sélectionné (t_{best} , P_{best}). C'est-à-dire la tâche T_{best} sera placée et ordonnancée sur le processeur P_{best}.
- Ce processus d'allocation est répété pour toutes les tâches restantes, jusqu'à ce qu'il n'en reste plus.

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

----------ALGORITHME----------

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

----------INITIALISATION

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

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

$T_{fin}^{(1)} := \emptyset$ *faire*

----------BOUCLE DE DISTRIBUTION ET D'ORDONNANCEMENT

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

----------SELECTION

- *Calculer pour chaque candidate* $T_{cand} \in T_{cand}^{(n)}$ *et chaque processeur* $P_j \in AHM$ *la pression d'ordonnancement (équation (2.5)) ;*
- *Sélectionner pour chaque candidate* T_{cand} *le processeur Pbest qui minimise la pression d'ordonnancement ;*
- *Sélectionner le meilleur couple (Tbest;Pbest) qui maximise la pression d'ordonnancement ;*

----------DISTRIBUTION ET ORDONNANCEMENT

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

----------VERIFICATION DES CONTRAINTES TEMPORELLES

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

----------MISE A JOUR

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

$$T_{fin}^{(n+1)} := T_{fin}^{(n)} \cup \{Tbest\};$$

$$T_{cand}^{(n+1)} := T_{cand}^{(n)} - \{Tbest\} \cup \left\{T \in succ(Tbest) \middle| pred(T) \subseteq T_{fin}^{(n+1)}\right\};$$

Fin tant que ----------FIN DE L'ALGORITHME

2.6. Conclusion

Nous avons présenté dans ce chapitre les terminologies liées à la distribution et à l'ordonnancement temps réel, ainsi que les algorithmes utilisés pour résoudre le problème d'ordonnancement dans les systèmes distribués temps réel et embarqués. Parmi la multitude des solutions proposées dans la littérature concernant le problème de distribution et d'ordonnancement temps réel hors-ligne, nous avons choisi de se baser dans le reste du mémoire sur l'algorithme de SynDEx pour des raisons de simplicité et d'efficacité. Des terminologies de base de la tolérance aux fautes vont être exposer dans le chapitre suivant.

Chapitre 3

Tolérance aux fautes dans les systèmes distribués temps réel embarqués

3.1. Introduction

Dans les systèmes temps réel embarqués et distribués le respect des contraintes temporelles tout au long de la vie du système est crucial, il constitue une condition nécessaire mais pas suffisante pour le bon fonctionnement de ces systèmes puisque quelles que soient les précautions prises, l'occurrence de fautes est inévitable (erreur humaine, malveillance, vieillissement du matériel, catastrophe naturelle, etc.). Au vu des conséquences catastrophiques (perte d'argent, de temps ou pire de vies humaines) que pourrait entraîner une faute dans un système temps réel critique, la présence des techniques qui assurent la sureté de fonctionnement [2, 21] est vitale dans la conception de ces systèmes. La tolérance aux fautes est l'une des méthodes utilisées dans la littérature pour assurer la sureté de fonctionnement des systèmes temps réel embarqués et distribués ; elle permet la conception des systèmes qu'ils continuent à rendre le service attendu (éventuellement un service gradé) même en présence de fautes. Les systèmes temps réel critiques doivent ainsi couvrir une propriété importante des systèmes sûrs de fonctionnement qui est la *fiabilité*. La fiabilité est la probabilité pour qu'un système soit continûment en fonctionnement sur une période donnée (entre 0 et t).

Dans ce mémoire, nous nous intéressons uniquement au concept de la tolérance aux fautes, nous commençons tout d'abord par les notions de base de la sûreté de fonctionnement qui nous permettront de cerner plus précisément la tolérance aux fautes. Ensuite Nous présentons les terminologies liées à la tolérance aux fautes et ses phases de

réalisation. Nous reprenons pour cela la taxonomie développée, entre autres, par Jean-Claude Laprie, Brian Randell et Algirdas Avizienis, qui fait largement référence.

3.2. Terminologies

Définition 16 (sûreté de fonctionnement) : *La sûreté de fonctionnement d'un système est définie comme la propriété qui permet à ses utilisateurs de placer une confiance justifiée dans la qualité du service qu'il leur délivre.*

La figure 3.1 montre les notions de base reliées aux systèmes sûrs de fonctionnement

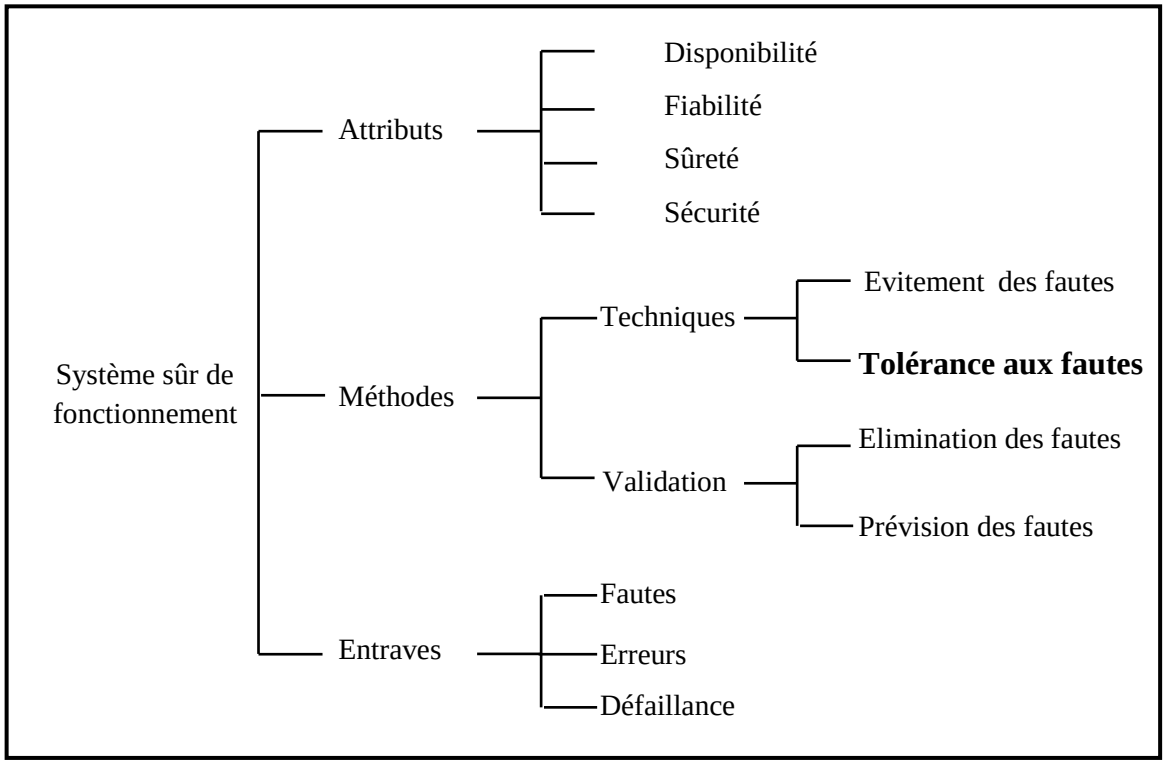


Figure3.1 : Arbre des systèmes sûrs de fonctionnement

Dans ce qui suit nous présentons quelques définitions classiques utilisées dans le domaine de la tolérance aux fautes d'où faute, erreur et défaillance sont les trois termes importants.

Définition 17 (Faute) : Est un défaut qui se produit dans certain composant matériel ou logiciel, c'est toute cause (événement, action, circonstance) pouvant provoquer une erreur. La faute ne produit pas immédiatement des erreurs, elle reste dormante durant l'exécution du système, et elle sera activée par un évènement intentionnel ou accidentel.

La faute peut être classifiée selon plusieurs critères qui sont récapitulés dans la figure 3.2 :

- **par leur nature :** accidentelle ou intentionnelle ;
- **par leur origine :** physique, humaine, interne, externe, conception, opérationnelle ;
- **par leur persistance :** permanente, transitoire ou intermittente.

Exemples : faute de programmation, malveillance, catastrophe naturelle, etc.

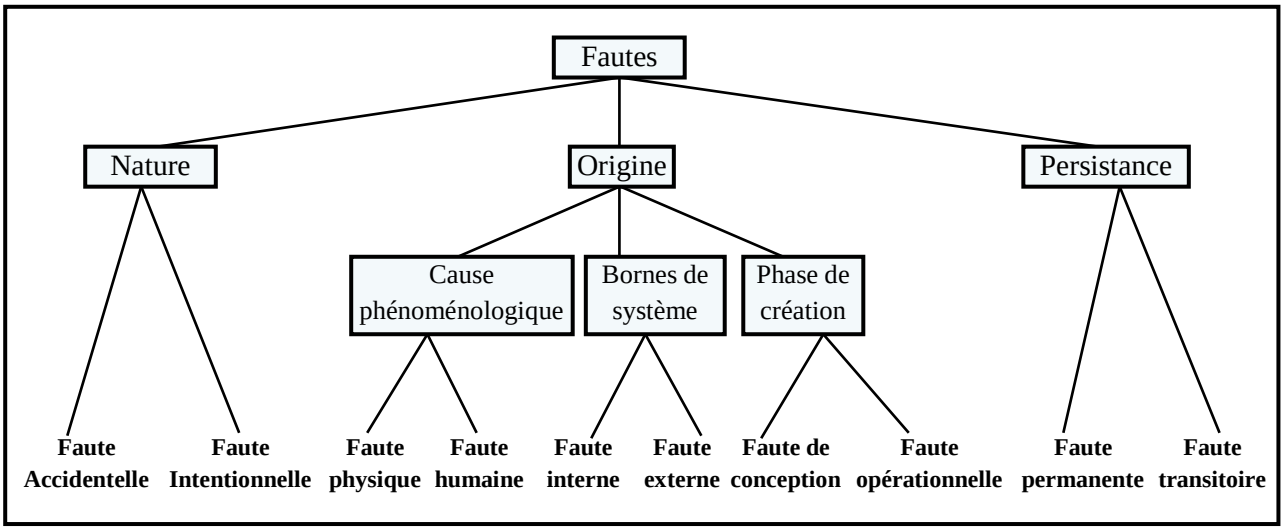


Figure3.2 : Classes de fautes

Définition 18 (Erreur) : une erreur est une manifestation d'une faute activée, c'est un état (ou partie de l'état) du système susceptible de provoquer une défaillance.

Exemple logiciel : la valeur d'une table ne vérifie pas un invariant spécifié.

Exemple matériel : une connexion est coupée entre deux points qui devraient être reliés entre eux.

Notes :

- Une erreur est latente tant qu'elle n'a pas provoqué de défaillance.
- Le temps entre l'apparition de l'état d'erreur et la défaillance est le délai de latence.
- plus le délai de latence est long, plus la recherche des causes d'une défaillance est difficile

Définition 19 (Défaillance) : Une défaillance d'un système est une déviation de celui-ci au service exigé, le système dans ce cas ne respect pas sa spécification. Les défaillances sont classifiées selon les critères suivants :

- **par leur domaine :**Défaillance de valeur et/ou défaillance temporelle ;
- **par leur perception :** par l'utilisateur ;
- **par leurs conséquences :** sur l'environnement.

Donc, la défaillance d'un système est la conséquence d'une erreur, et l'erreur est la conséquence d'une faute activée. En plus, étant donné qu'un système informatique est souvent composé de plusieurs sous-systèmes, la défaillance d'un sous-système peut créer et/ou activer une faute dans un autre sous-système ou dans le système lui même. La

relation entre ces trois termes faute, erreur et défaillance relativement aux sous systèmes du système peut être représentée par la figure suivante [1].

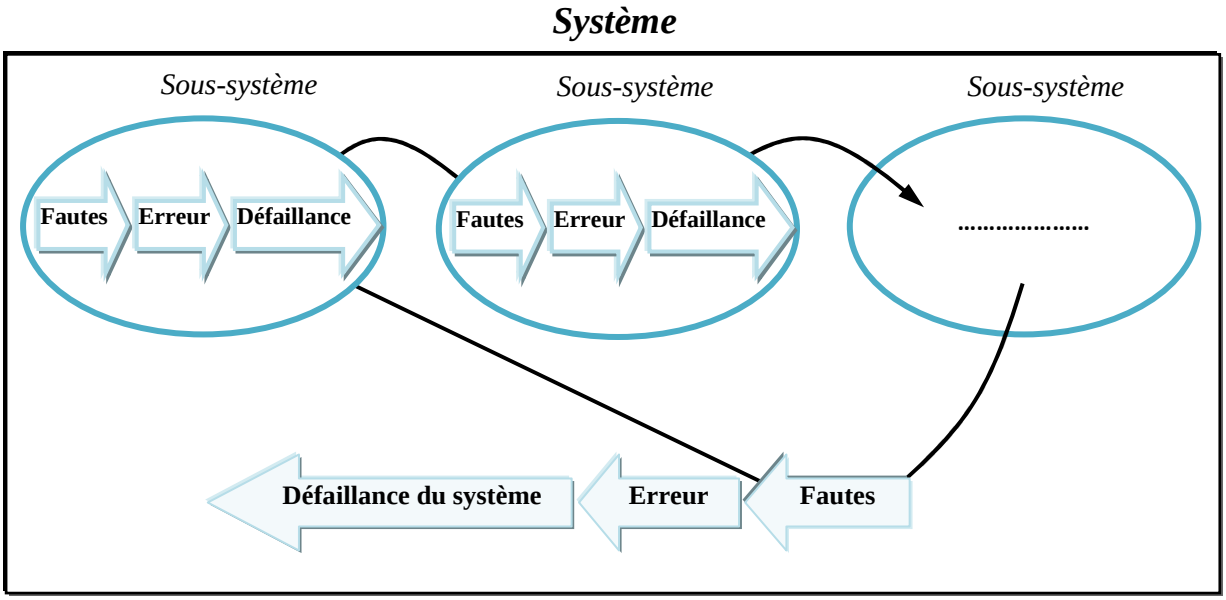


Figure3.3 : Relation entre faute, erreur et défaillance

Une défaillance dans un sous-système peut être vue comme une défaillance dans le système global.

Suivant les exigences fonctionnelles et temporelles d'un système temps réel embarqué et distribué, la défaillance de ce système peut être la conséquence de deux sources de fautes : fautes fonctionnelles (ou fautes de valeur) et fautes temporelles.

Définition 20 (Faute fonctionnelle) : *La valeur délivrée par le système est fausse, c'est-à-dire qu'elle n'est pas conforme à sa spécification ou elle est en dehors de l'intervalle des valeurs attendues* [1].

Définition 21 (Faute temporelle) : *L'instant auquel la valeur est délivrée est en dehors de l'intervalle de temps spécifié. Dans ce cas, la valeur est considérée soit temporellement délivrée trop tôt, soit trop tard, soit infiniment tard (jamais délivrée). La faute temporelle dans le cas où la valeur n'est jamais délivrée est appelée « faute par omission »* [1].

3.2.1. Degré de permanence des fautes

Les défaillances n'ont pas un comportement uniforme dans le temps, suivant la persistance temporelle d'une faute on distingue trois types de fautes: fautes permanentes, transitoires et intermittentes.

Définition 22 (Faute permanente) : Une faute permanente se caractérise par sa durée permanente, une fois activée, durant l'exploitation du système. Elle persiste donc indéfiniment (jusqu'à réparation) après son occurrence. Une faute de conception d'un composant matériel est un exemple typique de faute permanente.

Définition 23 (Faute transitoire) : Une faute transitoire se caractérise par sa durée limitée, une fois activée, durant l'exploitation du système. Les fautes transitoires sont souvent observées dans les systèmes de communication, où la présence des radiations électromagnétiques peut corrompre les données envoyées sur une liaison physique de communication. Ceci provoque une faute transitoire qui ne dure que la période de la présence de ces radiations.

Définition 24 (Faute intermittente) : Une faute intermittente est une faute transitoire qui se reproduit sporadiquement.

3.2.2. Hypothèses de défaillances (Classes de défaillances)

Il n'existe pas de méthodes de tolérance aux fautes valables dans l'absolu, seulement des méthodes adaptées à des hypothèses particulières d'occurrence de fautes. Ces hypothèses doivent donc être explicitement formulées après analyse soigneuse. Les hypothèses de défaillance des systèmes les plus utilisées dans la littérature [1] sont :

Définition 25 (Systèmes à défaillances en valeur) : Ces systèmes supposent que les valeurs sont délivrées à temps et qu'ils défaillent uniquement si les valeurs délivrées sont fausses.

Définition 26 (Systèmes à silence sur défaillances) : Un système à silence sur défaillances ou bien fonctionne et donne un résultat correct délivré à temps, ou bien il est en panne et ne fait rien (arrête de fonctionner).

Définition 27 (Systèmes à arrêt sur défaillances) : Ce sont des systèmes à silence sur défaillances mais, avant que le système arrête son fonctionnement, il délivre un message aux autres systèmes indiquant son arrêt. Un exemple de ces systèmes est le processeur à arrêt sur défaillance [3]. Ce processeur est un composant physique qui en présence de fautes spécifiées dans ses hypothèses de défaillances génère un message indiquant son arrêt.

Définition 28 (Systèmes à défaillances par omission) : Le système perd des messages entrants (omission en réception), sortants (omission en émission) ou les deux ; il n'y a pas d'autres déviations par rapport aux spécifications c.-à-d. dans ces systèmes les valeurs sont correctes, ils défaillent uniquement si la valeur n'est jamais délivrée.

Définition 29 (Systèmes à défaillances temporelles) : Les déviations par rapport aux spécifications concernent uniquement le temps (par exemple temps de réaction à un événement). Ces systèmes supposent que les valeurs délivrées sont correctes, et qu'ils défaillent uniquement si la valeur est temporellement délivrée trop tôt (en avance par rapport à sa plage de temps spécifiée) ou trop tard (en retard par rapport à sa plage de temps spécifiée).

Définition 30 (Systèmes à défaillances byzantines) : Ces systèmes défaillent d'une manière arbitraire. Le système peut faire n'importe quoi (y compris un comportement

malveillant). Ce mode de défaillance est parfois considéré par des systèmes à très haute fiabilité (nucléaire, spatial).

Notons qu'une hypothèse de défaillance peut couvrir une ou plusieurs autres hypothèses. La couverture entre les hypothèses que nous avons citées est illustrée sur la figure 3.2

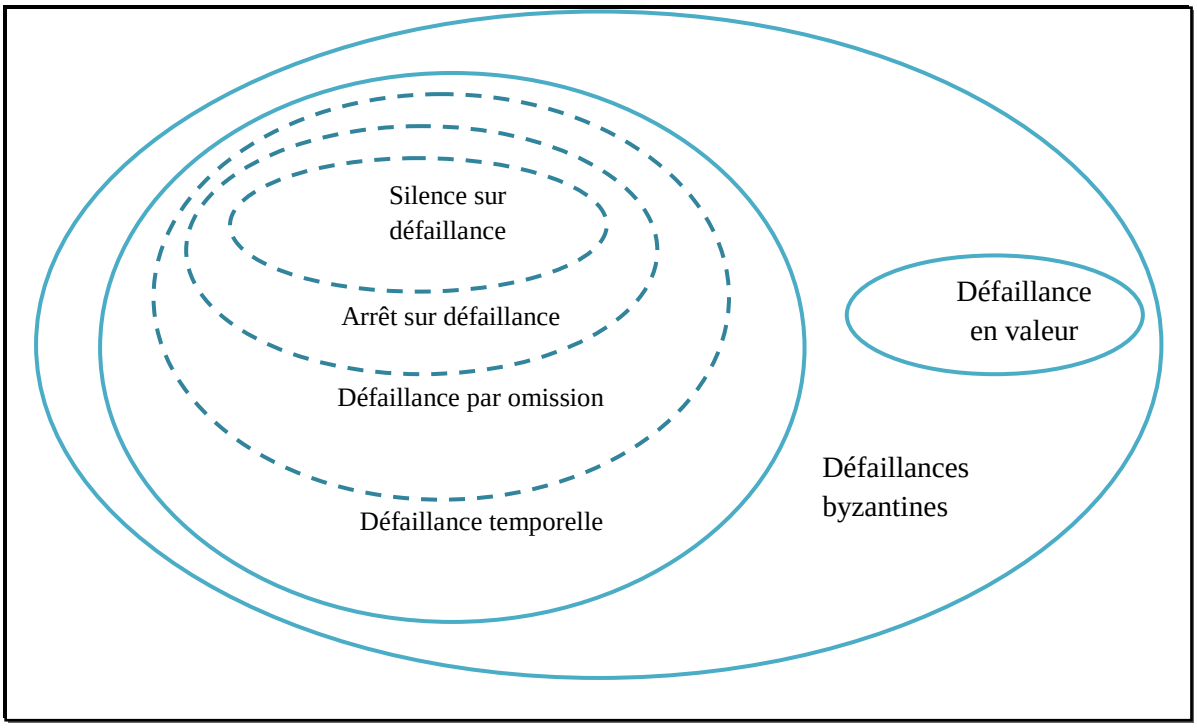


Figure3.2 : Couverture entre hypothèses de défaillances

Comme notre objectif est de réaliser des systèmes temps réel embarqués et distribués sûrs de fonctionnement, il ya plusieurs méthodes dans la littérature qui s'intéressent à donner les meilleurs résultats. Deux grandes classes de ces méthodes sont : l'évitement des fautes qui vise à empêcher l'occurrence de fautes par la prévention, l'évaluation ou la vérification des fautes, et la tolérance des fautes qui est l'objectif de notre mémoire. La tolérance aux fautes vise à préserver le service du système malgré l'occurrence de fautes.

Plusieurs travaux dans la littérature ont défini le concept de la tolérance aux fautes [4, 6, 7, 15], la définition suivante est la définition originale donnée par Algirdas Avizienis [7] en 1967:

Définition 31 (Tolérance aux fautes) : *On dit qu'un système informatique est tolérant aux fautes si ses programmes peuvent être exécutés correctement même en présence de fautes.*

3.2.3. Techniques de tolérance aux fautes

Les méthodes de tolérances aux fautes sont basées sur deux classes de techniques :

- Traiter les fautes,
- Traiter les erreurs

a. Traitement des fautes

Dans ce cas, l'algorithme de tolérance aux fautes vise à empêcher les fautes d'être activées. Elle concerne au moins deux étapes qui sont le diagnostique des fautes et l'inactivation des fautes.

- **Diagnostique des fautes :** détermine les causes de l'erreur en termes de location et de nature.
- **Inactivation des fautes :** prévient l'activation des fautes une autre fois (en les rendant passives).

b. Traitement des erreurs

Dans ce cas, l'algorithme de tolérance aux fautes consiste à détecter l'existence d'un état incorrect (erreur), puis remplacer l'état incorrect par un état correct conforme aux spécifications.

Dans tous les cas, la redondance est le principe unique utilisé afin de traiter les erreurs, il y a trois formes de redondance :

- **Redondance matérielle** : comporte les composants matériels ajoutés au système pour supporter la tolérance aux fautes (par exemple, utiliser un processeur disponible si l'un des processeurs en exécution est en panne),
- **Redondance logicielle (ou d'information)** : inclut tous les programmes et les instructions qui sont utilisés pour supporter la tolérance aux fautes (par exemple, en utilisant deux réalisations du même module).
- **Et redondance temporelle** : consiste à accorder un temps supplémentaire pour accomplir l'exécution des tâches pour supporter la tolérance aux fautes (par exemple, exécuter de nouveau un module plus tard).

Remarque :

Ce travail concerne les techniques de tolérance aux fautes basées sur le traitement d'erreurs.

3.2.4. Algorithmes de la tolérance aux fautes

Prenant la technique de tolérance aux fautes basées sur le traitement des erreurs, l'algorithme de tolérance aux fautes est réalisé en deux phases complémentaires : une phase de détection des erreurs et une phase de traitement des erreurs.

3.2.4.1. Phases d'un algorithme de tolérance aux fautes

a. Détection des erreurs

C'est la phase la plus importante, puisque la réussite de l'algorithme dépend d'elle, elle permet :

- D'identifier le type et l'origine des erreurs,
- De prévenir (si possible) l'occurrence d'une défaillance provoquée par l'erreur,
- D'éviter la propagation de l'erreur à d'autres composants

Cette détection peut être faite soit au niveau de l'environnement du système, soit au niveau de l'application du système [16]. Au niveau de l'environnement, c'est l'exécutif de l'application qui se charge de détecter certaines erreurs qui peuvent être par exemple de type : division par zéro, erreur d'entrée/sortie, accès interdit au périphérique. Au niveau de l'application, ce sont les composants redondants qui se chargent de réaliser cette tâche de détection.

Il ya plusieurs techniques de détection d'erreurs au niveau de l'application, parmi les techniques de base, on trouve la comparaison des résultats de composants répliqués et le contrôle de vraisemblance (vérification des temps de délivrance des résultats). La réussite d'une telle technique de détection des erreurs dépend de deux paramètres qui sont la latence (délai entre la production et la détection de l'erreur) et le taux de couverture (pourcentage d'erreurs détectées).

b. Traitement des erreurs

Cette phase consiste à traiter les états erronés (ou erreurs) détectés par la première phase, éliminer des erreurs de l'état du système à l'aide de l'une des deux techniques de base à savoir le recouvrement ou la compensation .

- **Recouvrement des erreurs**

Le recouvrement consiste à remplacer l'état d'erreur par un état correct, il nécessite pour cela la détection d'erreurs au moyen d'un test de vraisemblance explicite ou implicite. Il utilise soit la technique de la reprise soit la technique de la poursuit. La reprise consiste à mettre le système dans un de ses états précédents corrects (retour en arrière vers un état antérieur dont on sait qu'il est correct), et la poursuite permet soit la reconstitution d'un état courant correct (sans retour en arrière), soit la reconstitution partielle d'un état correct qui permet au système de fonctionner en mode dégradé.

- **Compensation des erreurs**

L'état possède une redondance interne d'informations suffisante pour détecter et corriger l'erreur (les tests externes de vraisemblance sont inutiles) pour masquer (rendre invisibles aux utilisateurs) ses conséquences.

Le choix entre ces deux techniques est un compromis entre plusieurs facteurs, tels que la complexité du système, les contraintes temporelles et matérielles et la criticité du système. Etant donné que nous visons des systèmes temps réel embarqués et distribués critiques, nous ne nous intéressons dans ce travail qu'aux techniques basées sur la redondance d'informations, donc aux techniques de compensation des erreurs.

3.2.4.2. Tolérance aux fautes matérielles et logicielles

La défaillance d'un système est due soit aux fautes matérielles, soit aux fautes logicielles.

a. Tolérance aux fautes logicielles

Les fautes logicielles sont dues aux erreurs de conception des composants logiciels de l'algorithme. L'explosion de la fusée Ariane 5 en juin 1996 à cause des erreurs de conception du logiciel est un exemple de ce type de fautes [17, 18]. Les techniques utilisées pour tolérer les fautes logicielles d'un système sont divisées en deux grands groups [19] : l'uni-version du logiciel et la multi-version du logiciel.

- **Uni-version du logiciel**

La technique uni-version se concentre sur l'amélioration de la tolérance aux fautes en utilisant une seule version du logiciel ou d'un de ses composants en ajoutant des mécanismes dans la conception visant la détection et le traitement des erreurs provoquées par l'activation des fautes de conception. Elle est basée sur l'utilisation de la redondance appliquée à une version unique d'un composant logiciel pour détecter et récupérer des fautes. Parmi les approches utilisant cette technique, on trouve : le traitement d'exception, la détection d'erreur et le point de reprise.

- **Multi-version du logiciel**

Cette technique emploie des versions multiples (ou des variantes) d'un composant logiciel dans une manière structurée d'une façon à s'assurer que les fautes de conception dans une version ne causent pas des défaillances du système. Elle est basée sur le principe de la redondance logicielle, où chaque composant logiciel est répliqué en plusieurs versions différentes. Parmi les approches utilisant cette technique, on trouve : N-vérification automatique de programmation et N-version de programmation qui est le cas des commandes de vol des avions Airbus et Boeing [20].

b. Tolérance aux fautes matérielles

Les fautes matérielles sont souvent dues soit aux fautes de conception et de fabrication des composants matériels du système, soit à l'interaction du système avec l'environnement qu'il contrôle. Elles peuvent être tolérées soit en utilisant des solutions logicielles, soit en utilisant des solutions matérielles. Les solutions logicielles sont basées sur la redondance des composants logiciels, et les solutions matérielles sont basées sur la redondance des composants matériels. Nous ne nous intéressons dans ce travail qu'aux solutions logicielles.

3.2.5. Problème de distribution et d'ordonnancement temps réel et tolérant aux fautes

Nous avons vu que le problème de distribution/ordonnancement consiste à la recherche d'une solution optimale (si elle existe) qui permet l'allocation spatiale et temporelle des composants logiciels sur l'architecture matérielle du système tout en respectant des contraintes temporelles ; tandis que, cette solution n'est pas garantie en cas de défaillance d'un composant matériel ou logiciel. Pour résoudre ce problème, un deuxième algorithme appelé algorithme de tolérance aux fautes est ajouté. Cet algorithme est une solution qui permet de garantir l'exécution complète du système[3] même en présence de fautes.

La coexistence de ces deux algorithmes nécessite la mise en œuvre des mécanismes de communication et de synchronisation entre eux. Puisque ces mécanismes sont coûteux et difficiles à réaliser, la plupart des systèmes combinent ces deux algorithmes en un seul algorithme appelé *algorithme de distribution/ordonnancement tolérant aux fautes*. Le but d'un tel algorithme est de résoudre le problème de la recherche d'une allocation valide des composants logiciels de l'algorithme sur les composants matériels de l'architecture, tout en tolérant des fautes matérielles. Ce problème peut être formalisé comme suit :

Problème 2[4]: Etant donné :

- *Une architecture matérielle hétérogène AMH composée de n composants matériels : AMH = {P1, ……. , Pn}*
- *Un algorithme ALG composé de m composants logiciels : ALG = {T_1 , ……. , T_m}*
- *Des coûts d'exécution C_{exe} des composants d'ALG sur les composants d'AMH*
- *Des contraintes temps réel C_{tr} et matérielles C_{mt}*
- *Un sous ensemble F d'AMH de composants matériels qui peuvent causer la défaillance du système.*
- *Un ensemble de critères à optimiser*

Il s'agit de trouver une application A qui réplique un ou plusieurs composants T_i d'ALG, place chaque composant répliqué T_i sur un composant P_j d'AMH, et qui lui assigne un ordre d'exécution O_k sur son processeur.

$A: \quad ALG \rightarrow AMH$

$T_i \quad \rightarrow A(T_i^j) = (P_j , O_k)$

Qui doit satisfaire C_{mt} et C_{tr} , tolérer les fautes des composants de F, et optimiser les critères spécifiés.

[3] Nous voulons dire par exécution complète du système que l'algorithme continue à délivrer un service spécifié en respectant les contraintes temporelles

[4] Problème de distribution/ordonnancement temps réel et tolérant aux fautes

3.3. Conclusion

Nous avons présenté dans ce chapitre la tolérance aux fautes qui est un moyen d'assurer la sûreté de fonctionnement d'un système informatique. Ce moyen consiste à faire en sorte que le système délivre un service correct malgré l'occurrence des fautes. Une faute est une anomalie affectant le matériel ou le logiciel, susceptible de provoquer la défaillance du système. Un système est défaillant lorsqu'il ne délivre plus un service conforme à sa spécification. Plusieurs techniques sont proposées pour résoudre le problème de la tolérance aux fautes, nous ne nous intéressons qu'aux algorithmes basés sur la redondance des composants logiciels pour tolérer les fautes matérielles. Dans ce qui suit, nous allons présenter un état de l'art sur les méthodologies utilisées pour résoudre le problème de la tolérance aux fautes dans les systèmes distribués temps réel embarqués en commençant d'abord par la description de notre problème dans le chapitre suivant.

Chapitre 4

Description du problème

4.1. Introduction

Ce travail a pour but d'intégrer les mécanismes de tolérances aux fautes dans la conception des systèmes temps réel distribués embarqués en se basant sur la méthodologie AAA [5] [24] implanté dans SynDEx. Afin de satisfaire toutes les contraintes (temps réel, d'embarquabilité et de la tolérance aux fautes) nous proposons une méthodologie globale prenant en compte toutes les phases du développement, de la spécification haut niveau des algorithmes et des architectures matérielles, à l'exécution du code. La spécification repose sur des modèles et l'exécution du code repose sur des techniques d'ordonnancement hors-ligne.

Notre méthodologie repose sur trois modèles : le modèle d'algorithme (ou architecture logicielle) qui est l'ensemble des tâches et des messages, le modèle matériel (ou architecture matérielle) qui est le support physique d'exécution et le modèle d'exécution ou d'implantation qui décrit le mode d'exécution de l'algorithme sur l'architecture matérielle.

Dans ce chapitre nous commencerons par une description des machines parallèles, préalable nécessaire à l'étude de notre modèle d'architecture de la méthodologie AAA.

4.2. Classification des machines

En 1946, Von Neumann a posé les bases d'un modèle d'architecture encore largement utilisé aujourd'hui. Ce modèle repose sur la coopération de cinq unités.

[5] Adéquation Algorithme architecture, consiste à mètre en correspondance de manière efficace l'algorithme sur l'architecture matérielle pour réaliser une implantation optimisée.

- *L'unité mémoire* contient des instructions et des données.
- *L'unité de traitements* transforme (effectue des calculs) les données stockées en mémoire.
- *Les unités d'entrée et de sortie* permettent de transférer des données entre la mémoire et l'environnement.
- *L'unité de commande* contrôle l'ensemble de ces unités, elle repose sur un *séquenceur* d'instructions qui lit ses instructions dans la mémoire et les applique à l'unité de traitement. L'ensemble unité de commande-unité de traitement est souvent appelé processeur ou CPU[6]

A partir d'un seul processeur il est possible de construire des machines relativement simples, de puissance égale à celle du processeur. Une autre technique que celle de l'utilisation de processeur unique, consiste à construire des machines parallèles par connexion de processeurs à l'aide de mémoires et de médias de communication. Dans ces machines, les unités de traitement de chaque processeur fonctionnent en parallèle de façon à résoudre un problème commun. Pour cela ils doivent coopérer, c'est à dire communiquer pour s'échanger des données mais aussi se synchroniser. Toutes les machines qu'il est possible de construire, sont basées sur trois notions génériques : le traitement, la mémoire et la communication, elles sont utilisées pour classer les machines [38]. La première identifie le type de parallélisme offert par la machine, il est basé sur la relation entre le nombre de séquenceurs et le nombre d'unités de traitements arithmétiques et logiques. Le second critère repose sur l'organisation de la mémoire dans la machine et le troisième identifie le type de communication qui peut avoir lieu dans la machine.

4.2.1. Traitement (séquenceur)

Dans la célèbre classification de Flynn, les machines sont organisées en quatre groupes correspondant au parallélisme d'instruction et de données.

- *Les machines SISD* sont basées sur un unique séquenceur d'instructions et une unique unité de traitement. Ce sont les machines les plus simples, elles n'offrent aucun parallélisme, leur architecture est comparable à celle de Von Neumann.
- *Les machines SIMD* n'ont toujours qu'un séquenceur d'instructions mais renferment plusieurs unités de traitement fonctionnant en parallèle de manière synchrone, multipliant d'autant leur capacité de calcul.
- *Les machines MISD* sont des machines possédant plusieurs séquenceurs d'instructions mais une seule unité de traitement (les machines pipeline sont parfois classées dans cette catégorie).
- *La dernière catégorie correspond aux machines MIMD,* elles possèdent plusieurs séquenceurs d'instructions indépendants et plusieurs unités de traitements et fonctionnent de ce fait de manière principalement asynchrone. Les MIMD sont homogènes si les séquenceurs et les unités de traitements qui les composent sont identiques, et hétérogènes s'ils sont différents.

[6] Central Processing Unit

4.2.2. Mémoire

L'organisation de la mémoire et ses méthodes d'accès par les processeurs sont la base d'une seconde classification [38] qui organise les machines MIMD en trois catégories : UMA, NUMA et NORMA.

Dans les machines de type UMA[7], tous les processeurs ont accès à toute la mémoire de la machine de façon uniforme. C'est le cas des machines à mémoire centralisée et partagée entre tous les processeurs au moyen d'un bus.

Dans les machines de type NUMA[8], tous les processeurs ont accès à tout l'espace mémoire de la machine (comme dans les machines UMA), mais le temps d'accès n'est plus uniforme, il dépend de la localisation géographique des données dans la machine. C'est le cas des machines à mémoire distribuée-partagée (DSM). Chaque processeur est connecté directement à une mémoire locale et à un bus ou à un réseau d'interconnexion qui lui permet d'accéder à chaque mémoire locale de chaque processeur de la machine (l'accès à sa mémoire locale étant plus rapide puisque ne passant pas par le réseau partagé, l'accès n'est plus uniforme). Bien que physiquement distribuée dans toute la machine, la mémoire est toujours vue, par tous les processeurs, comme un espace unique logiquement continu.

Dans les machines de type NORMA[9], les processeurs n'ont plus accès à toute la mémoire de la machine. Cela correspond aux machines à mémoire physiquement distribuée (comme les NUMA), mais non partagée. Chaque processeur est connecté à une mémoire locale (qui lui est privée), et il ne peut accéder à la mémoire locale d'un autre processeur. Les échanges de données se font de manière explicite par passage de message sur un réseau de communication qui connecte les processeurs.

4.2.3. Communication

Les communications au sein des machines UMA et NUMA, et des machines NORMA ne reposent pas sur les mêmes principes.

Dans le cas des premières machines, les communications sont dites par mémoire partagée (shared memory) car tous les processeurs ont accès à toute la mémoire, qu'elle soit physiquement centralisée ou distribuée. Cette mémoire est toujours appelée **RAM**[10] pour deux raisons :

- Quand la mémoire est de dimension N (c'est à dire composée de N registres correspondant à N emplacements spatiaux différents), il est possible d'accéder aléatoirement, en lecture ou écriture, à n'importe lequel d'entre eux car ils ont tous une adresse différente.

[7] Uniform Memory Access
[8] Non Uniform Memory Access
[9] No Remote Memory Access
[10] Random Access Memory

- L'ordre de lecture des données est totalement indépendant de leur ordre d'écriture, l'accès est aléatoire. Il est possible d'écrire plusieurs fois un même registre sans avoir lu sa valeur entre temps.

Dans les machines NORMA, où la mémoire est distribuée mais non partagée, les communications se font de manière explicite par passage de messages (message passing) sur le réseau d'intercommunication qui peut être basé sur la mémoire RAM partagée ou la mémoire **SAM**[11]. La mémoire SAM possède deux caractéristiques qui la distinguent des mémoires RAM [38] :

- Ce type de mémoire impose que chaque écriture dans la mémoire soit suivie d'une lecture, les données sont lues dans l'ordre de leur écriture.
- Quelque soit le nombre de registres de cette mémoire, elle ne possède qu'une adresse, les données sont donc toujours lues ou écrites à la même adresse. Les données stockées dans la mémoire SAM sont référencées temporellement par leur ordre d'écriture et non spatialement par une adresse comme dans le cas des RAM.

Il y a deus types de mémoire SAM, point-à-point et multipoint. Les mémoires SAM point à point ne peuvent être connectées qu'à deux processeurs. Si la mémoire est monodirectionnelle, un seul des processeurs est capable d'écrire dans la mémoire, l'autre étant uniquement capable d'y lire. Si elle est bidirectionnelle, les deux processeurs connectés sont capables d'y écrire (mais toute écriture de l'un doit être suivie d'une lecture par l'autre). Les mémoires SAM multipoint peuvent être connectées à plus de deux processeurs. Il existe deux types de mémoire SAM multipoint selon qu'elles supportent ou non la diffusion matérielle(Broadcast). Lorsque la mémoire supporte le Broadcast, les processeurs connectés peuvent faire une lecture de la même donnée simultanément. Si elles ne le supportent pas, un seul des processeurs est capable de lire une donnée. Si une donnée doit être utilisée par plusieurs processeurs, il faut effectuer autant d'écritures qu'il y a de destinataires.

4.2.3.1. DMA

Les mémoires partagées, soient-elles SAM ou RAM, ont généralement une bande passante inférieure à celle du processeur et des autres RAM. Lorsqu'un processeur effectue un transfert de données entre deux mémoires, il passe donc une partie de son temps à attendre que les données soient prêtes. Pour libérer en partie le processeur de la gestion fine des transferts de données, une unité DMA[12] est parfois ajoutée aux processeurs. Cette unité permet un réel parallélisme entre calculs et communications car elle est capable d'accéder directement au contenu de la mémoire (Direct Memory Access) pour transférer des données depuis et vers les unités d'entrée-sortie.

[11] Sequential Access Memory
[12] Direct Memory Access

4.2.3.2. Structure du réseau

La structure du réseau peut être statique ou dynamique (multiétage). Dans un réseau statique, chaque processeur peut communiquer directement avec un nombre fixe et déterminé de processeurs. Les performances des machines reposantes sur ces types de réseaux sont fortement dépendantes de la distribution des données et des calculs sur chaque processeur. Les réseaux dynamiques reposent sur des commutateurs, appelés aussi routeurs, souvent associées aux processeurs, ces unités sont dédiées aux routages des communications dans le réseau.

4.2.3.3. Topologie

La topologie d'un réseau d'interconnexion correspond à sa structure matérielle, à la façon dont sont connectés les processeurs (ou les routeurs) entre eux. Elle est souvent dictée par le type des algorithmes, ou imposée par l'environnement. En effet dans les systèmes embarqués, les capteurs et les actionneurs connectés aux unités d'entrée-sortie des processeurs sont souvent physiquement distribués. Pour minimiser les câblages, les processeurs sont souvent placés près de ces capteurs et actionneurs. Le classement des différents types de topologie repose sur trois critères, la distance, le diamètre et la connectivité [38]. Les topologies les plus répandues reposent sur les bus (simples, multiples, hiérarchisés), les réseaux à connexions directes (totalement connectés, anneau, étoile, grille, hypercube) ou hiérarchisés (arbres binaires, arbre anneau, pyramide).

Après ces trois types de classement, il est maintenant possible de décrire notre modèle d'architecture.

4.3. Modèle d'architecture

Nous distinguons deux types d'architectures matérielles, l'architecture monoprocesseur et l'architecture multiprocesseur. L'architecture monoprocesseur utilise un seul processeur pour l'exécution séquentielle des composants logiciels, donc certaines applications ne peuvent satisfaire les contraintes temps réel imposées par leur environnement. Tandis que l'architecture multiprocesseur utilise l'interconnexion de plusieurs processeurs pour l'exécution parallèle ou distribuée des composants logiciels, ce qui permet de satisfaire les contraintes temporelles dans la majorité des applications.

Les modèles des architectures parallèles et distribuées sont respectivement PRAM[13] et DRAM[14] [25]. Le premier modèle correspond à un ensemble de processeurs communiquant par mémoire partagée, alors que le deuxième correspond à un ensemble de processeurs à mémoire distribuée communiquant par passage de message.

Notre architecture est une architecture multiprocesseur hétérogène (les processeurs n'ont pas les mêmes caractéristiques, par exemple la vitesse de calcul) et plus particulièrement une architecture distribuée non partagée (machine NORMA). Pour des

[13] Parallel Random Access Machines
[14] Distributed Random Access Machines

raisons de tolérance aux fautes et de performance de calcul (satisfaire les contraintes temporelles) elle est présentée par un graphe non orienté dont chaque sommet est une machine à états finie (machine séquentielle) et chaque arc est une connexion physique entre deux machines à états finies. L'architecture est donc composée de :

- Processeurs et liens de communication dans le même processeur et inter-processeurs via les mémoires SAM qui fonctionne en FIFO[15].
- Chaque processeur est composé d'un opérateur pour séquencer des opérations de calcul (séquenceur d'instructions), et d'une mémoire RAM.
- Un opérateur est un regroupement d'une unité arithmétique et logique (UAL), d'une unité d'entrée/sortie (ES) et d'une unité de contrôle (UC)

La figure suivante montre un exemple des composants d'un processeur.

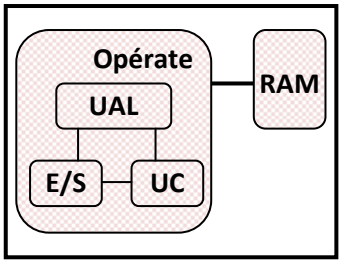


Figure4.1 : Exemple d'un processeur

Comme nous l'avons déjà vu, lors d'une opération de communication ou de transfert de données, il y aura un problème au niveau du processeur qui doit interrompre les opérations de calculs en cours et s'occuper de se communiquer ou de transférer les données aux autres processeurs, ce qui fait ralentir le temps d'exécution de la tâche actuelle et bien sûr le temps du système en général. Cette perte de temps peut affecter considérablement le respect des contraintes temporelles, pour résoudre ce problème le processeur est doté d'une unité de communication appelée DMA. Le DMA est souvent composé de plusieurs canaux de communication appelés *communicateurs*, son rôle est de libérer le processeur des opérations de transfert de données entre les mémoires RAM et SAM ainsi que les opérations de communication. Dans la figure suivante nous montrons un exemple d'un processeur doté d'un DMA composé de deux communicateurs.

[15] First in first out

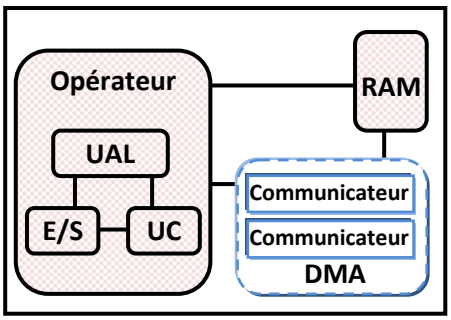


Figure4.2 : Exemple d'un processeur doté d'un DMA

Notre architecture est multiprocesseurs reliés par les mémoires SAM qui sont connectées entre les communicateurs de chaque processeur. La figure suivante présente une architecture composée de trois processeurs reliés par deux mémoires SAM.

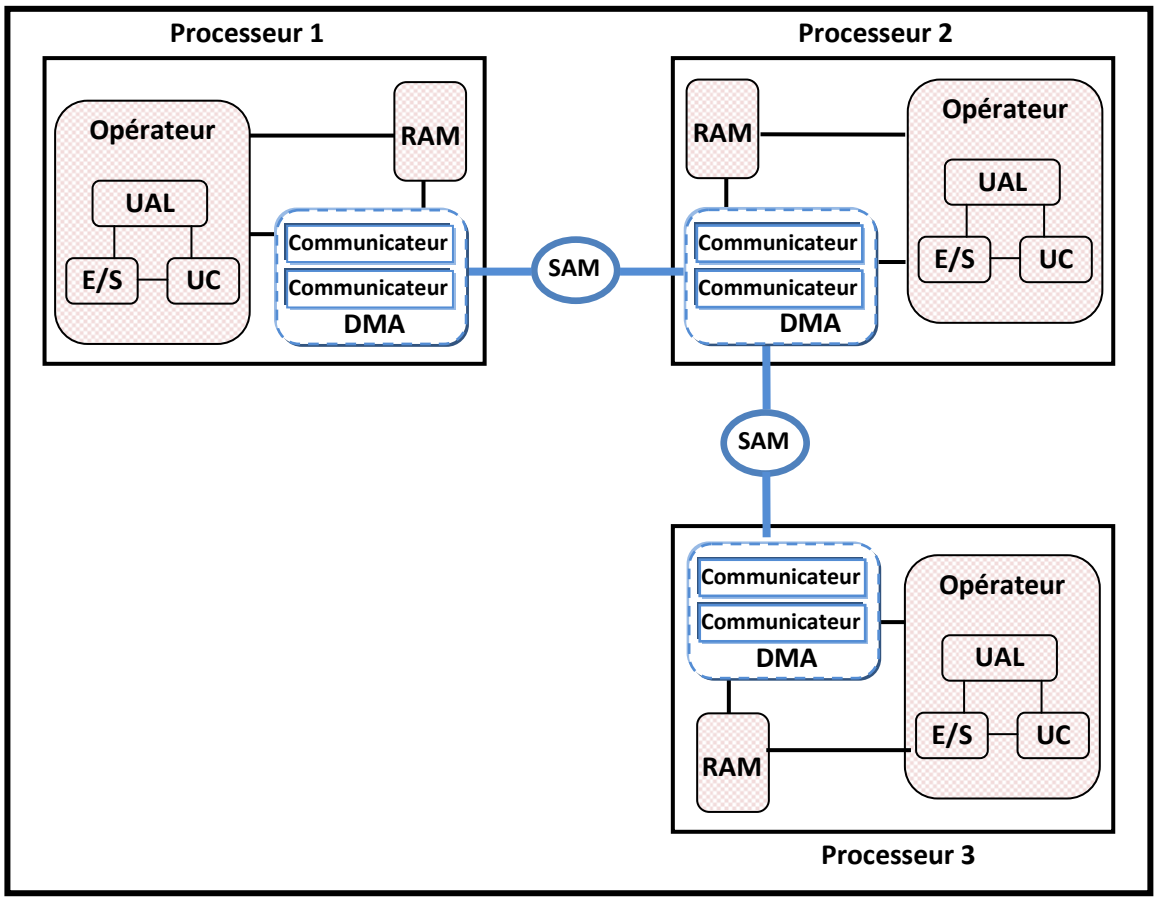


Figure4.3 : Exemple d'une architecture multiprocesseur distribuée

On appelle une liaison par mémoire SAM point-à-point un **lien**, et une liaison par mémoire SAM multipoint connectée à tous les processeurs un **bus.** Suivant le type de la mémoire SAM utilisée par le réseau de communication, on distingue deux types d'architecture : architecture à liaison point à point et architecture à liaison multipoint. Afin de modéliser ces deux types d'architectures plusieurs modèles ont été proposés, le

modèle classique est le modèle le plus utilisé dans le domaine d'ordonnancement. Dans ce modèle, l'architecture distribuée est représentée par un hypergraphe non orienté dont les sommets représentent les processeurs et les hyper-arrêtes représentent les liaisons physiques de communication point-à-point ou multipoint.

4.3.1. Architecture à liaisons point à point

Est une architecture représentée par un graphe non orienté dont les sommets représentent les opérateurs de calculs qui incluent les RAM, les communicateurs et les mémoires SAM ***point à point*** et les arcs représentent les communications dans le même processeur et inter processeurs. Dans le même processeur les arcs relient les opérateurs aux communicateurs, et entre les processeurs ils relient les communicateurs aux mémoires SAM point à point.

Dans l'exemple ci-dessous, nous modélisons l'architecture précédente (figure 4.3) par le graphe de la figure 4.4. Ce graphe contient trois processeurs : P_1, P_2 et P_3 et deux liens de communication. Chaque processeur P_i est composé d'un opérateur op_i et de deux communicateurs com_{i1} et com_{i2}. Les deux liens de communication sont respectivement : $\{com_{11}, SAM_{12}, com_{21}\}$ et $\{com_{22}, SAM_{23}, com_{31}\}$.

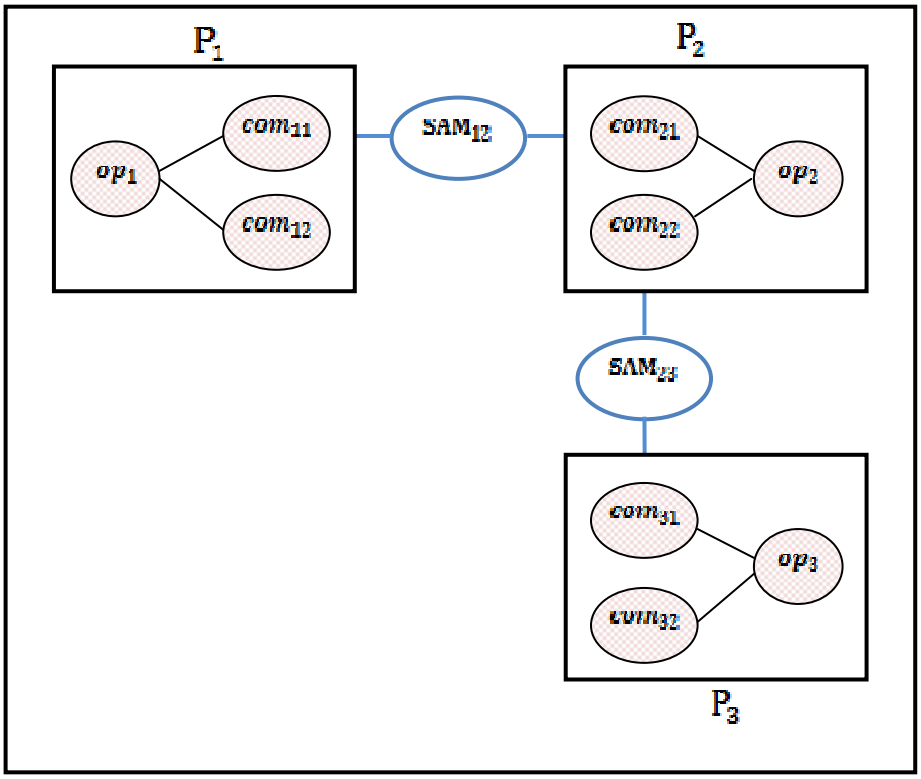


Figure4.4 : Exemple d'une architecture à liaisons point à point

4.3.2. Architecture à liaisons bus

Comme l'architecture précédente, l'architecture à liaison bus est représentée par un même graphe sauf que le type des mémoires SAM est multipoint. En effet, les sommets représentent les opérateurs incluant les RAM, les communicateurs les mémoires SAM ***multipoint*** et les arcs représentent les communications intra-processeur (relient l'opérateur à ses communicateurs) et inter-processeur (relient les communicateurs aux mémoires SAM multipoint).

La figure 4.5 représente un exemple d'une architecture à liaison bus composée de trois opérateurs op_1 , op_2 , op_3 et de deux bus de communication $\{com_{11}, com_{21}, com_{31}$ $SAM_1\}$ et $\{com_{12}, com_{22}, com_{32}, SAM_2\}$.

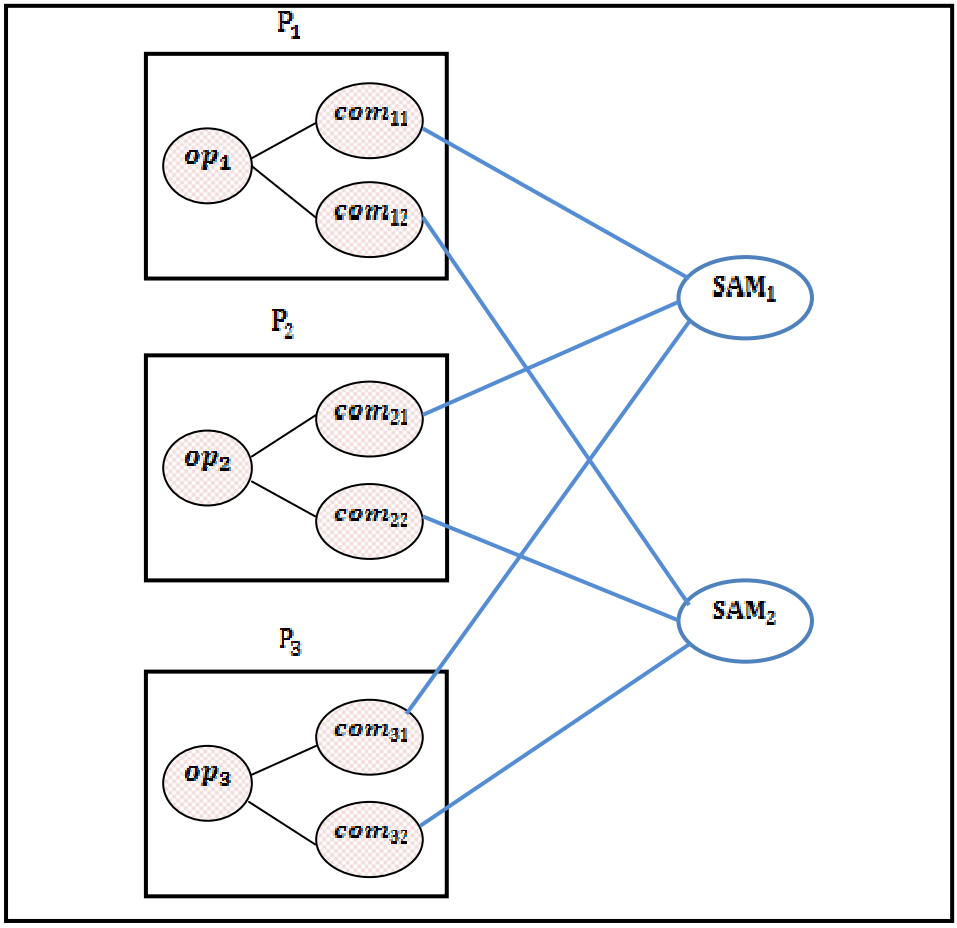


Figure4.5 : Exemple d'une architecture à liaisons bus

Il existe un troisième type plus général d'architecture constitué à la fois de liaisons point à point (lien) et de liaison multipoint (bus), cependant pour des raisons de simplicité nous ne considérons dans ce mémoire que les liaisons multipoints. Donc, notre spécification matérielle est une architecture multiprocesseur distribuée appelée graphe d'architecture, représentée par un graphe non orienté dans lequel les nœuds représentent les processeurs hétérogènes et les arcs représentent le bus de communication (réseau statique en bus).

4.4. Modèle d'algorithme

Pour modéliser l'architecture logicielle, nous considérons un graphe flot de données, c'est un hypergraphe orienté appelé graphe d'algorithme. L'ensemble des nœuds (sommets) T= $\{T_i, T_j, \dots T_n\}$ correspond aux tâches et l'ensemble des arcs $C \subseteq T \times T$ représente les dépendances de données entre ces tâches. Une tâche T_i est définie par ces caractéristiques temporelles (temps d'exécution sur chaque processeur). Les arcs sont évalués par la quantité de données échangées. Il y a deux types de communication entre tâches :

- **Communication locale sans délai** appelée communication intra-processeur (voir figure 4.6) : dans ce cas, les tâches sont sur un même processeur et utilisent sa mémoire pour communiquer.
- **Communication distante avec délai** appelée communication inter-processeurs (voir figure 4.7) : dans cette communication, les tâches sont placées sur des processeurs distincts et utilisent le réseau pour communiquer.

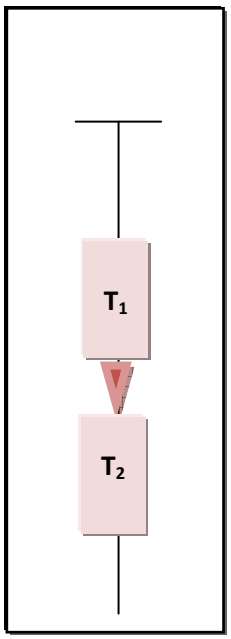


Figure4.6 : communication intra-processeurs

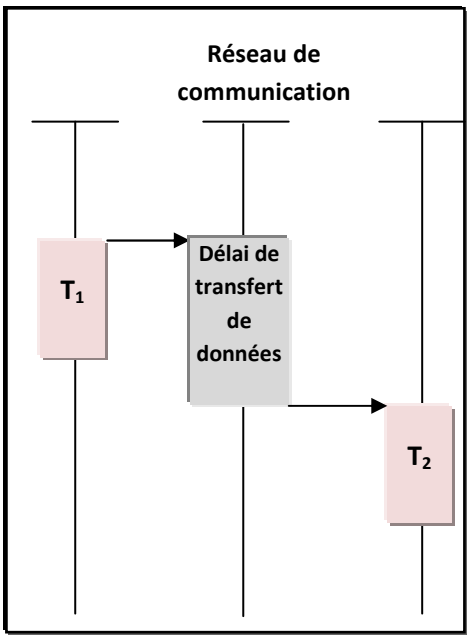


Figure4.7 : communication Inter-processeurs

Définition 32 (Tâche) : Une tâche est une entité localisée par une date de début et une date de fin, elle est désignée parfois dans notre mémoire par composant logiciel. On peut distinguer deux types de tâches dans le graphe d'algorithme :

- Une tâche de calcul : elle possède au moins un prédécesseur et un successeur, elle exploite et traite les données entrantes de son prédécesseur (données en entrée) et fait sortir des données à son successeur (données en sortie). La tâche de calcul ne peut s'exécuter que si ses données d'entrée sont présentes.

- Une tâche d'entrée/sortie : une tâche d'entrée (resp. de sortie) est une tâche qui n'a pas de prédécesseur (resp. de successeur), elle transforme les informations reçues des capteurs en données numériques (resp. transforme les données reçues des tâches de calcul en grandeurs physiques exploitées par les actionneurs).

La figure suivante présente un exemple d'un graphe d'algorithme contenant une tâche d'entrée A, cinq tâches de calcul (T_1, T_2, T_3, T_4, T_5), une tâche de sortie B et huit dépendances de données (arcs).

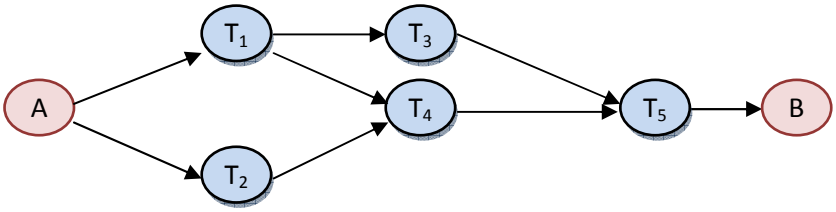


Figure4.8 : exemple d'un graphe d'algorithme

Définition 33 (Dépendance de données) : une dépendance de données $D_{i,j}$ représente une contrainte de précédence [9] entre deux tâches T_i et T_j, elle est donc créée si T_i doit précéder T_j.

Note :

Le graphe d'algorithme peut ne pas contenir des contraintes de précédence, c.-à-d. les dépendances de données entre les tâches, on l'appelle dans ce cas un graphe sans dépendances de données. L'exemple ci-dessous montre un graphe d'algorithme composé de quatre tâches indépendantes.

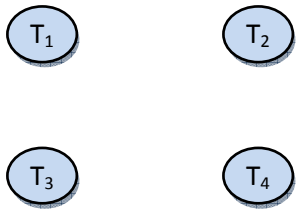


Figure4.9 : Exemple d'un graphe d'algorithme sans dépendances de données

Une exécution de toutes les tâches ainsi les dépendances de données (si elles existent) d'un graphe flot de données est appelée cycle. Dans les systèmes embarqués, l'exécution du cycle est répétée indéfiniment et cela ce qu'on appelle motif.

Définition 34 (motif) : *On appelle motif d'un graphe l'ensemble des opérations appartenant à la même répétition et l'ensemble des dépendances de données entre ces opérations.*

Lorsque l'exécution d'une tâche dans un motif N nécessite les données d'une autre tâche du motif $N-1$, une dépendance de données appelée dépendance inter-motif est créée entre ces deux tâches.

4.5. Modèle d'implantation

L'implantation d'un algorithme sur une architecture distribuée est un ordonnancement non seulement des opérations de l'algorithme sur les opérateurs de l'architecture mais aussi des opérations de communication, la manière d'exécuter cette implantation est définie par plusieurs modes [26], on distingue dans la littérature deux grandes classes : l'exécution périodique et l'exécution cyclique.

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

  À chaque top faire

  - lire les entrées
  - calculer
  - écrire les sorties

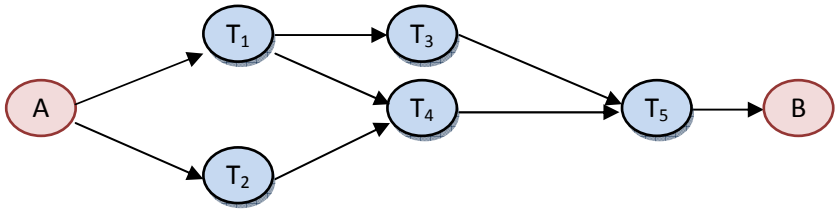
  Fin faire

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

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

Définition 35 (Problème d'ordonnancement cyclique) : consiste à ordonner dans le temps l'exécution répétitive d'un ensemble de tâches liées par des contraintes de précédence, en utilisant un nombre limité de ressources.

L'exécution de l'algorithme sur l'architecture de l'exemple 4.3 prend donc la forme suivante :



While système ok do

Exécuter A
Transmettre le résultat de A à t_1 et t_2
Exécuter t_1 et t_2
Transmettre le résultat de t_1 à t_3 et à t_4
…..
…..
Exécuter B

End while

Comme nous avons cité précédemment, dans l'exécution cyclique l'ordonnancement est statique, donc notre solution se base sur la non préemption des tâches ayant chacune un temps d'exécution différent sur chaque processeur. Cette différence dans la durée d'exécution est due à notre architecture qui est hétérogène.

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

Définition 36 (contraintes de distribution) : consistent à assigner à chaque paire (processeur, tâche) une valeur du temps d'exécution de cette tâche sur ce processeur. De même, elles assignent une durée de communication à chaque paire (dépendance de données, lien de communication).

Le tableau ci-dessous représente les durées d'exécution de chaque tâche du graphe d'algorithme de la figure 4.8 sur les opérateurs de calcul de chaque processeur du graphe

D'architecture de la figure 4.4. La valeur ∞ signifie que cette tâche ne peut pas être exécutée sur cet opérateur.

		Tâches						
		A	t_1	t_2	t_3	t_4	t_5	B
Opérateurs	OP_1	1.1	2	5	1	4	3	∞
	OP_2	2	3	6.5	∞	5	4.5	3
	OP_3	∞	1.5	7	0.0	6	5	4

Tableau 4.1: Durées d'exécution des tâches sur les processeurs

A chaque média (bus) de communication est associée une durée de communication ou de transfert de données pour chaque dépendance de données. Cette durée dépend de la quantité de données échangées entre deux tâches et des caractéristiques physiques du médium de communication (telles que la taille de la mémoire SAM et la vitesse de chaque communicateur).

Le tableau 4.2 représente les durées de transfert des données entre les tâches du graphe d'algorithme de la figure 4.8 sur les médias de communication du graphe d'architecture de la figure 4.4.

		Tâches							
		A→ t_1	A→ t_2	t_1→t_3	t_1→t_4	t_2→t_4	t_3→t_5	t_4→t_5	t_5→B
Liens	L_{12}	1.1	2.25	1.00	1.00	1.75	2.00	1.00	3.25
	L_{23}	1.2	1.00	2.75	1.50	1.25	1.75	2.25	2.00

Tableau 4.2 : Durées de transfert des données entre les tâches

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

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

4.6. Conclusion

La modélisation de notre problème est basée sur la méthodologie AAA. La méthodologie AAA vise le prototypage rapide et l'implantation optimisée d'applications distribuées temps réel embarquées, elle est fondée sur des modèles de graphes pour spécifier les algorithmes applicatifs et les architectures matérielles distribuées. Notre modélisation revient à résoudre un problème d'optimisation consistant à choisir une implantation de l'architecture logicielle sur l'architecture matérielle distribuée et hétérogène dont les performances déduites des caractéristiques des composants matériels respectent les contraintes temps réel et d'embarquabilité. Pour satisfaire le concept de la tolérance aux fautes, la technique de redondance est utilisée dans la plupart des travaux. Dans le chapitre suivant nous survolons sur les différentes techniques de base de la redondance ainsi qu'aux travaux réalisés dans la littérature pour donner des solutions qui conviennent à nos propres hypothèses au problème de la tolérance aux fautes dans les systèmes distribués temps réel embarqués.

Chapitre 5

Etat de l'Art

5.1. Introduction

Sachant que le système temps réel embarqué à réaliser est critique, et l'évitement des défaillances qui sont matérielles et/ou logicielles est impossible, alors le recours à une méthode qui assure la sûreté de fonctionnement est inévitable et obligatoire. La tolérance aux fautes est l'une des méthodes utilisée [37] pour garantir le bon fonctionnement du système même en présence de fautes. L'approche la plus utilisée pour assurer le concept de la tolérance aux fautes est celle de la redondance matérielle et/ou logicielle. Vu que les systèmes embarqués exigent certaines caractéristiques concernant par exemple la réduction du coût, de la taille et de l'énergie, nous ne considérons que les ***solutions logicielles***. Aussi bien, comme nous avons vu avec l'exemple d'Ariane 501, il est essentiel de bien cerner la nature des fautes qu'on cherche à tolérer, car de telles méthodes se diffèrent sur plusieurs critères tels que : la nature des fautes (matérielles ou logicielles), l'origine de la faute matérielle (processeur, média de communication, capteur, actionneur, …) et la persistance de la faute (transitoire, permanente, …).

Nous ne considérons dans ce mémoire que les fautes ***matérielles permanentes d'un seul processeur***. Et puisque nous visons des solutions logicielles, nous ne présentons que des méthodes basées sur la théorie de l'ordonnancement [1].

Dans ce chapitre nous verrons comment la redondance (appelée aussi réplication) est utilisée dans les systèmes répartis pour mettre en œuvre la tolérance aux fautes en rappelant quelques résultats dans la littérature.

5.2. Stratégies de réplication

La réplication consiste à la création des copies multiples des processus, appelée dans notre mémoire tâches ou composants logiciels, sur des processeurs différents. On trouve dans la littérature trois techniques de base de réplication : la réplication active, la réplication passive et la réplication hybride (semi-active)

Remarque :

Le nombre n de répliques de chaque composant dépend directement du nombre k de fautes à tolérer ainsi que de type de ces fautes. Par exemple, pour tolérer au plus *k* fautes permanentes de processeurs, chaque composant logiciel est répliqué en une copie primaire et en k copies secondaires, d'où ***n = k+1***.

5.2.1. Réplication active

Définition 37 (réplication active) : La réplication active (active replication ou state machine approach) se définie par la symétrie des comportements des copies d'un composant répliqué. Chaque copie joue un rôle identique à celui des autres.

a) Principe

La réplication active est définie ainsi :

- **Réception des requêtes :** toutes les copies reçoivent la même séquence de requêtes ;
- **Traitement des requêtes :** toutes les copies traitent les requêtes de manière déterministe ;
- **Emission des réponses :** toutes les copies émettent la même séquence de réponses.

Cette technique permet en particulier de mettre en œuvre un vote sur les sorties afin de se prémunir contre les défaillances byzantines.

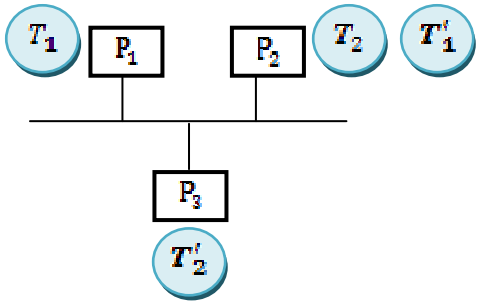
b) Tolérance aux fautes des processeurs

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

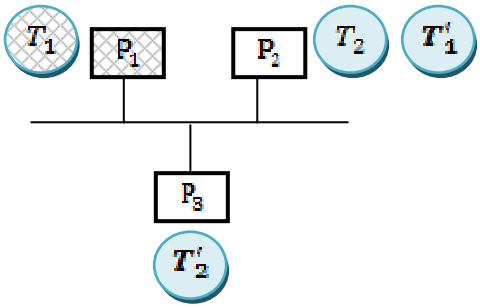
La figure 5.1 montre un exemple des tâches répliquées activement pour assurer la tolérance aux fautes d'un seul processeur sur une architecture matérielle composée de trois processeurs : P_1, P_2 et P_3 reliés par un bus de communication. L'architecture logicielle est

Composée de deux tâches : T_1 et T_2 , leurs répliques T_1' et T_2' sont placées activement sur deux processeurs distincts.

L'architecture logicielle



Chaque tâche est répliquée sur un processeur différent. Les deux copies d'une tâche reçoivent les mêmes données et s'exécutent au même temps (réplication active).



Par exemple en cas de défaillance de P_1, la tâche T_1 devient inactive (défaillante) alors sa copie T_1' masque cette défaillance et le système continue son exécution sans aucune défaillance.

Figure5.1 : Exemple de la réplication active

c) Présentation de quelques algorithmes basés sur la redondance active

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

Plusieurs approches utilisent la redondance active [1, 35, 36, 27, 29,…] comme solution au problème de tolérance aux fautes. Par exemple, Kalla Hamoudi et al dans [36] présentent une heuristique basée sur la réplication active pour tolérer un nombre arbitraire de défaillances des processeurs et des liens de communication, dont les défaillances sont

assumées d'être silence sur défaillance (voir section 3.2.2) dans les systèmes distribués temps réel et embarqués. La tolérance aux fautes est obtenue hors-ligne en deux phases, la première s'appuie sur un formalisme de transformation d'une spécification d'un graphe non redondant en une spécification avec redondance des composants logiciels pour tolérer les fautes des processeurs et des liens de communication. La deuxième phase, consiste à allouer spatialement et temporellement les composants logiciels de ce nouveau graphe redondant sur l'architecture matérielle par une heuristique de distribution et d'ordonnancement temps réel.

Un autre mécanisme de tolérance aux fautes est présenté par Koji Hashimoto et al dans [35], ce mécanisme permet de tolérer les fautes d'un seul processeur en utilisant la duplication active des composants logiciels. Leur algorithme, appelé HBP[16], se base sur des algorithmes présentés dans leurs travaux précédents (DSH , RSR_k, GRD , PHS^1). Dans un premier temps, l'algorithme HBP partitionne l'ensemble des tâches en groupes de tâches selon leur taille (ou hauteur) calculée par une formule donnée afin d'améliorer le temps d'exécution du système dans le cas de défaillance d'un processeur, et dans un deuxième temps, il appelle un algorithme de base pour chaque groupe. L'algorithme de base s'exécute en deux étapes, dans la première étape les tâches de chaque groupe sont ordonnancées suivant leurs priorités[17] en permettant à ses prédécesseurs de s'ordonnancer en premier sur le même processeur, et dans la deuxième étape toutes les tâches de chaque groupe sont dupliquées et ordonnancées de la même façon que l'étape1 à condition que chaque instance d'une tâche soit ordonnancée sur un processeur différent. Ce mécanisme permet la tolérance aux fautes et l'élimination, dans la plupart du temps, des coûts de communication.

P.Ramanathan et K.Shin ont proposé dans [27] une approche basée sur la réplication active pour résoudre le problème de délivrance des messages critiques dans leur date d'échéance (deadline) dans le cas de défaillance des processeurs ou des liens de communication en un moindre coût. Le modèle utilisé est basé sur une architecture distribuée à liaison point-à-point (maille hexagonale et une topologie hypercube), et les fautes considérées sont des fautes transitoires. L'heuristique consiste donc à dupliquer chaque message au moins en deux copies, et cela en fonction de son criticité et du nombre de processeurs et des liens de communication qu'il doit traverser, puis les diffuser sur des routes disjointes pour réduire le coût de la retransmission des messages. Le même principe est proposé dans [28] par Kandasamy et al et dans [29] par Molina et al.

5.2.2. Réplication passive

Définition 38 (réplication passive) : La réplication passive distingue deux comportements d'un composant répliqué : la copie primaire (primary copy) et les copies secondaires (backups). La copie primaire est la seule à effectuer tous les traitements. Les copies secondaires ne traitent pas de messages, elles surveillent uniquement la copie

[16] Height – Based Partitionning
[17] La tâche ayant la plus grande hauteur est considérée la plus prioritaire

primaire. En cas de défaillance de la copie primaire, la copie secondaire doit détecter cette défaillance et devient la nouvelle copie primaire. Cette technique nécessite un mécanisme de détection d'erreur.

a) Principe

La réplication passive est définie ainsi :

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

b) Tolérance aux fautes des processeurs

Dans ce cas, un composant logiciel est répliqué en n exemplaires (voir la remarque de la page 54), mais une seule des n répliques effectue le calcul. Les n-1 autres répliques sont passives et ne prennent la relève que si la réplique active est défaillante. Pour que cette stratégie fonctionne, il est nécessaire que la réplique active transmette aux répliques passives à intervalles réguliers son état d'exécution pour la mettre à jour. Si la réplique active est défaillante, une des répliques passives est activée et reprend l'exécution du calcul à partir du dernier point de reprise enregistré. On dit que le processus effectue un retour arrière.

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

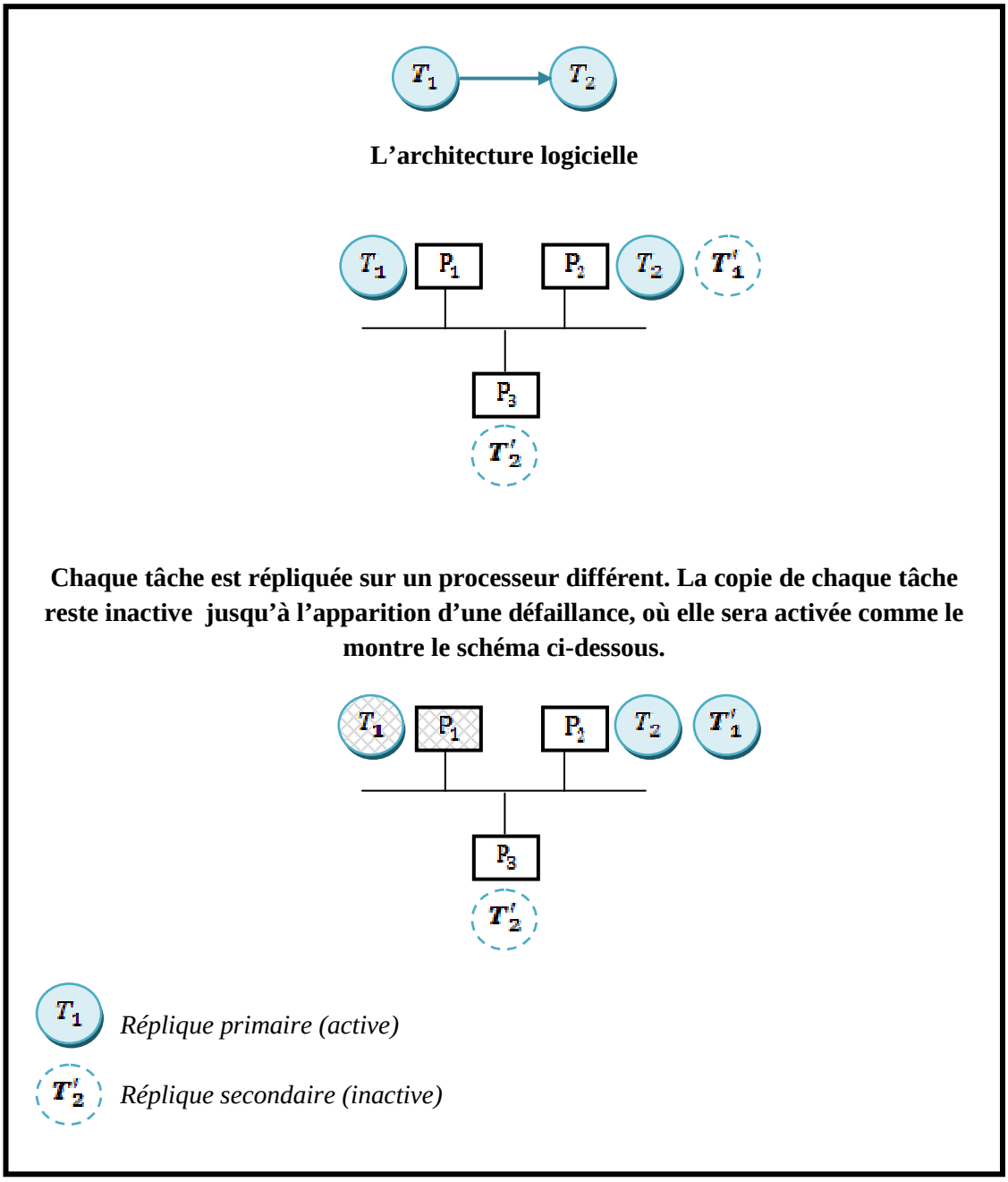


Figure5.2 : Exemple de la réplication passive

d) Présentation de quelques algorithmes basés sur la redondance passive

Plusieurs travaux dans la littérature s'intéressent à la duplication passive pour atteindre des systèmes sûrs de fonctionnement en moindre coût. Par exemple Xiao Qin et al ont présenté dans [30] une heuristique d'ordonnancement tolérante aux fautes dans les systèmes temps réel distribués et hétérogènes[18] en se basant sur la redondance passive. Les tâches sont supposées indépendantes (absence de dépendances de données) et non-préemptives, elles doivent satisfaire leur date d'échéance même en présence de

[18] Chaque tâche à un temps d'exécution différent sur chaque processeur

fautes d'un seul processeur. Pour arriver à ce type de système, ils ont choisi de répliquer chaque tâche en deux copies, l'une primaire et l'autre de sauvegarde qui ne s'exécute qu'à l'apparition d'une défaillance d'un processeur. Ces deux copies sont nécessairement placées sur deux processeurs distincts.

Même principe était suivi par Y. Oh et S.H.Son dans [31], ils ont proposé une heuristique d'ordonnancement tolérante aux fautes dans les systèmes temps réel distribués et hétérogènes, où les tâches sont indépendantes et non-préemptives. Leur but est de réaliser des systèmes dont les tâches (qui sont strictes) doivent finir leur exécution le plus tôt possible avant ou dans leur date d'échéance. Alors l'heuristique proposée est un ordonnancement hors ligne qui assure la satisfaction des dates limites de chaque tâche même à la présence de défaillances arbitraires d'un seul processeur, elle est appelée 1TFT[19]. La redondance des tâches spécifiées dans ce cas correspond à la redondance passive où chaque tâche est répliquée en une copie primaire et une copie de sauvegarde qui doivent être ordonnancées de façon séquentielle (pas de chevauchement) sur deux processeurs distincts.

La réplication passive est utilisée ainsi pour tolérer les fautes des processeurs utilisés pour le routage dans une route de communication afin d'assurer la qualité des services de communication (comme le délai d'arrivée des messages et le taux d'erreur). S.Han et K.G.Shin ont proposé dans [32] une heuristique pour reconstituer rapidement les routes temps réel (real-time channels)[20] dans le cas de défaillances des processeurs dans les réseaux de communication. Pour cela, des routes de sauvegardes sont installées à priori en plus de chaque route primaire. Les routes de sauvegarde ne portent aucun message dans le cas normal, et elles sont activées en cas de défaillance d'un ou plusieurs processeurs.

5.2.3. Réplication semi-active (hybride)

Définition 39 (semi-active) : La réplication semi-active (semi-active replication) est une technique hybride entre la réplication active et la réplication passive. Contrairement à la réplication passive, les copies secondaires ne sont pas oisives. La copie primaire est appelée leader et les copies secondaires sont appelées suiveuses.

a) Principe

La réplication semi-active est définie ainsi :

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

[19] 1-Timely-Fault-tolerance : est définie par l'ordonnancement dans lequel aucun deadline de tâches n'est manqué malgré les fautes arbitraires d'un seul processeur.

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

b) Tolérance aux fautes des processeurs

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

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

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

H.Kalla dans [1] présente une heuristique appelée AAA-TB, basée sur la redondance hybride pour tolérer les fautes arbitraires des processeurs et des bus de communication dans un système réactif. La redondance hybride consiste, dans ce cas, d'une part à répliquer les opérations de l'architecture logicielle en plusieurs copies placées activement sur plusieurs processeurs distincts, et d'autre part à répliquer les dépendances de données en plusieurs répliques dont une seule s'exécute et les autres restent inactives jusqu'à l'apparition de défaillances de bus ou processeur implantant la copie primaire. Cela nécessite un mécanisme de recouvrement d'erreurs qui peut augmenter la longueur de la distribution/ordonnancement. Pour accélérer le recouvrement d'erreurs, il a proposé de fragmenter les données de communication.

Une autre approche présentée par P.Chevochot et I.Puaut dans [33] consiste à tolérer les fautes transitoires et permanentes d'un composant physique dans un système distribué temps réel dur. Dans cette approche, un site qui est composé d'un processeur, d'une mémoire et d'une horloge est supposé de type silence sur défaillance, il a un accès à des capteurs et à des actionneurs ; les capteurs sont exposés à des erreurs temporelles et fonctionnelles tandis que les actionneurs sont supposés fiables. La tolérance aux fautes est achevée par le développement d'un outil de réplication appelé HYDRA, ce dernier permet d'intégrer la réplication active, passive ou hybride dans l'algorithme d'ordonnancement.

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

active n'est applicable qu'à des processus au comportement déterministe, dans le cas contraire, les copies risqueraient de diverger. En raison de sa propriété de recouvrement rapide, ce type de réplication est particulièrement adapté à un environnement exigeant des temps de réponses bornés.

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

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

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

5.3. Conclusion

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

Chapitre 6

Méthodologies proposées pour des architectures à liaison bus

6.1. Introduction

Notre objectif dans ce travail est de résoudre le problème de la génération automatique de distribution/ordonnancement temps réel tolérante aux fautes. Sachant que nous travaillons sur les systèmes distribués temps réel embarqués, nous allons présenter deux nouvelles méthodologies adaptées aux architectures matérielles munies d'un réseau de communication liaison à bus, elles permettent de tolérer les fautes permanentes d'un seul processeur en utilisant la redondance logicielle. La tolérance aux fautes est obtenue hors-ligne en trois phases :

La première consiste à une transformation du graphe d'algorithme sans redondance à un nouveau graphe redondant où chaque tâche est répliquée en deux copies : primaire et secondaire. Dans la première méthodologie, nous utilisons la réplication active dans laquelle les deux copies sont exécutées en parallèle, tandis que dans la deuxième méthodologie nous utilisons la réplication passive où seule la première copie s'exécute au démarrage de chaque cycle d'exécution et la deuxième ne s'exécute qu'à la détection d'une erreur d'où la nécessité d'un mécanisme de détection d'erreurs. Dans ce cas, la détection d'une panne est obtenue par l'ajout d'une nouvelle tâche nommée « watchdog » au graphe comme successeur de chaque tâche.

La deuxième phase consiste à une allocation spatiale et temporelle de ce nouveau graphe d'algorithme sur le graphe d'architecture en utilisant l'algorithme de distribution et d'ordonnancement de SynDEx.

Enfin, la dernière phase consiste à :

1. **Dans la première méthodologie (réplication active):** en cas de défaillance d'un processeur, les copies secondaires de ses tâches masquent cette défaillance et la méthodologie met à jour la distribution/ordonnancement en supprimant le processeur défaillant et les répliques des tâches non défaillantes, puis elle suit cette nouvelle distribution/ordonnancement à chaque cycle. Dans le cas contraire (le non défaillance), à chaque cycle d'exécution la méthodologie annule l'exécution des copies secondaires pour diminuer le surcoût.
2. **Dans la deuxième méthodologie (réplication passive):** en cas de défaillance d'un processeur, la méthodologie doit détecter cette défaillance (par les tâches watchdogs), mettre à jour la distribution/ordonnancement tout en ignorant le processeur défaillant, activant les répliques de ses tâches implantées sur d'autres processeurs et annulant les répliques des tâches non défaillantes et les tâches watchdogs, et enfin elle suit cette nouvelle distribution/ordonnancement.

Les deux méthodologies se distinguent par leur graphe d'architecture logicielle, nous distinguons deux types de graphes :

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

Nous commençons par la présentation du modèle de fautes puis de l'heuristique proposée pour résoudre le problème de distribution/ordonnancement tolérant aux fautes dans le cas le plus simple : un *système dont les tâches sont indépendantes.* Ensuite, nous abordons la solution au même problème dans le cas d'un *système avec tâches dépendantes*.

6.2. Présentation de l'algorithme de distribution/ordonnancement tolérant aux fautes des tâches indépendantes AAA-FAULT$^{t.ind}$

Pour bien présenter le problème de distribution/ordonnancement tolérant aux fautes des tâches indépendantes, nous introduisons tout d'abord notre modèle de fautes.

6.2.1. Modèle de fautes

Puisque toute solution est dépendante des hypothèses de défaillance définies par les modèles de fautes, nous introduisons dans ce qui suit les hypothèses supposées dans notre modèle.

Hypothèse 1 : les tâches sont indépendantes.

Hypothèse 2 : Les fautes sont des fautes de processeurs.

Hypothèse 3 : Le système accepte les fautes d'un seul processeur durant tous les cycles d'exécution du graphe d'algorithme sur le graphe d'architecture.

Hypothèse 4 : Les fautes sont permanentes.

Hypothèse 5 : Le bus de communication est supposé fiable.

Hypothèse 6 : Le logiciel est supposé sans fautes.

6.2.2. Données du problème

A fin d'aboutir à la génération automatique d'une distribution/ordonnancement tolérante aux fautes qui satisfait les contraintes temps réel (C_{tr}) et matérielles (C_{mc}) et qui minimise la longueur de distribution/ordonnancement en absence et en présence de défaillance, un ensemble de données doit être défini avant d'aborder le principe général de la méthodologie proposée.

Problème 3[21] : étant donné :

- Une architecture matérielle hétérogène AMH composée d'un ensemble P de processeurs et d'un ensemble L qui contient un bus B de communication (liaison à bus).

  P = {P_1, P_2, …….P_m} , L = {B}

Exemple :

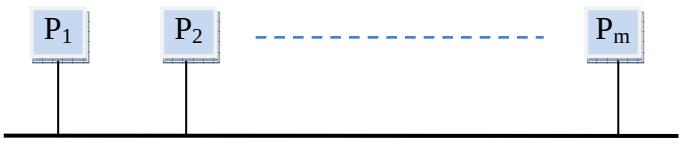


Figure6.1 : Architecture liaison à bus

[21] Problème de distribution/ordonnancement tolérant aux fautes des tâches indépendantes.

- Un algorithme ALG, composé d'un ensemble T de tâches indépendantes.

  T= $\{t_1, t_2, \ldots\ldots t_n\}$

Exemple :

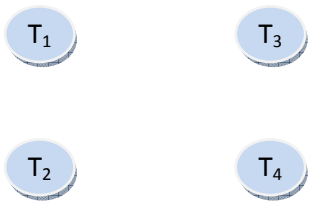


Figure6.2 : Exemple d'un graphe d'algorithme avec tâches indépendantes

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

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

$A :$ $ALG \rightarrow AMH$

$T_i \rightarrow A(T_i) = (P_j , O_k)$

6.2.3. Principe général de la méthodologie AAA-FAULT$^{t.ind}$

Le problème 3 peut être vu comme un problème à deux parties : Un problème de temps réel noté P_{tr} et un problème de tolérance aux fautes noté P_{tf}.

Le problème de temps consiste à minimiser la longueur de distribution/ordonnancement, tandis que le problème de tolérance aux fautes consiste à tolérer les fautes d'un seul processeur, c'est à dire même un processeur est en panne le système continue son exécution et délivre les résultats prévus.

Le problème P_{tr} est un problème d'optimisation car il s'agit de trouver une solution optimale. Il a été démontré comme étant NP-difficile [1].

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

Le problème de tolérance aux fautes P_{tf} est résolu dans la littérature par plusieurs techniques [1, 25, 26, 27, 28, 29]. Etant donné que nous visons des systèmes embarqués, la redondance logicielle est la technique utilisée dans ce travail pour réduire le coût physique et financier exigé par les systèmes embarqués.

Principe 1 : redondance logicielle

La redondance logicielle est une technique utilisée dans les systèmes répartis pour tolérer les fautes matérielles (processeur ou lien de communication), elle consiste à répliquer les processus sur différents processeurs.

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

Définition 40 (AAA-FAULT$^{t.ind}$) : Est une nouvelle méthodologie pour optimiser d'une part la génération automatique de distribution/ordonnancement des tâches indépendantes et d'autre part pour tolérer les fautes permanentes d'un seul processeur. Dans cette méthodologie, la tolérance aux fautes est basée en premier temps sur la redondance active des tâches indépendantes et en deuxième temps sur le blocage de l'exécution de la copie secondaire de chaque tâche non défaillante. Ce blocage sert à minimiser le surcoût qui est plus élevé dans la redondance active, donc notre méthodologie bénéficie des avantages des deux stratégies de redondance (active et passive). En effet, dans le cas de défaillance la méthodologie n'a pas besoin de la détecter, elle a un temps de réponse prévisible et une reprise immédiate après cette défaillance, et dans le cas où aucun processeur n'est en panne, c'est-à-dire l'exécution des répliques de chaque tâches sera bloquée, la méthodologie AAA-FAULT$^{t.ind}$ bénéficie de la réduction du surcoût.

Principe 2 : Redondance active

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

Principe 3 : Masquage d'erreurs

Si une défaillance d'un processeur quelconque apparaît, les copies secondaires des tâches implantées sur ce processeur masquent cette erreur. Alors cette technique n'exige pas un mécanisme de détection d'erreurs.

Principe 4 : Tolérance aux fautes permanentes

La méthodologie **AAA-FAULT**$^{t.ind}$ sert à générer automatiquement une distribution/ordonnancement tolérante aux fautes permanentes d'un seul processeur à n'importe qu'elle étape dans chaque cycle d'exécution de l'algorithme. Elle est composée de deux phases : la phase de transformation du graphe et la phase d'adéquation de l'architecture logicielle sur l'architecture matérielle.

- ***La phase de transformation*** consiste à transformer le graphe d'algorithme non redondant ALG à un nouveau graphe ALG^n redondant dont :

- Chaque tâche est répliquée en deux copies : primaire t_i et secondaire t_i' ;
- Une dépendance de données est créé entre les tâches : (t_i , t_i').

La figure suivante montre un exemple de transformation de chaque tâche t_i du graphe d'algorithme :

Donc, la tolérance aux fautes peut se présenter comme suit :

Si t_i' reçoit un message de t_i^{pt} alors (t_i^{pi} *désigne que* t_i *est placée sur le processeur* p_i)

- t_i' se bloque 

Si non (p_i *en panne*)

- t_i' masque l'erreur de t_i 

Fin Si

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

La figure 6.3 schématise une représentation de la méthodologie **AAA-FAULT**$^{t.ind}$.

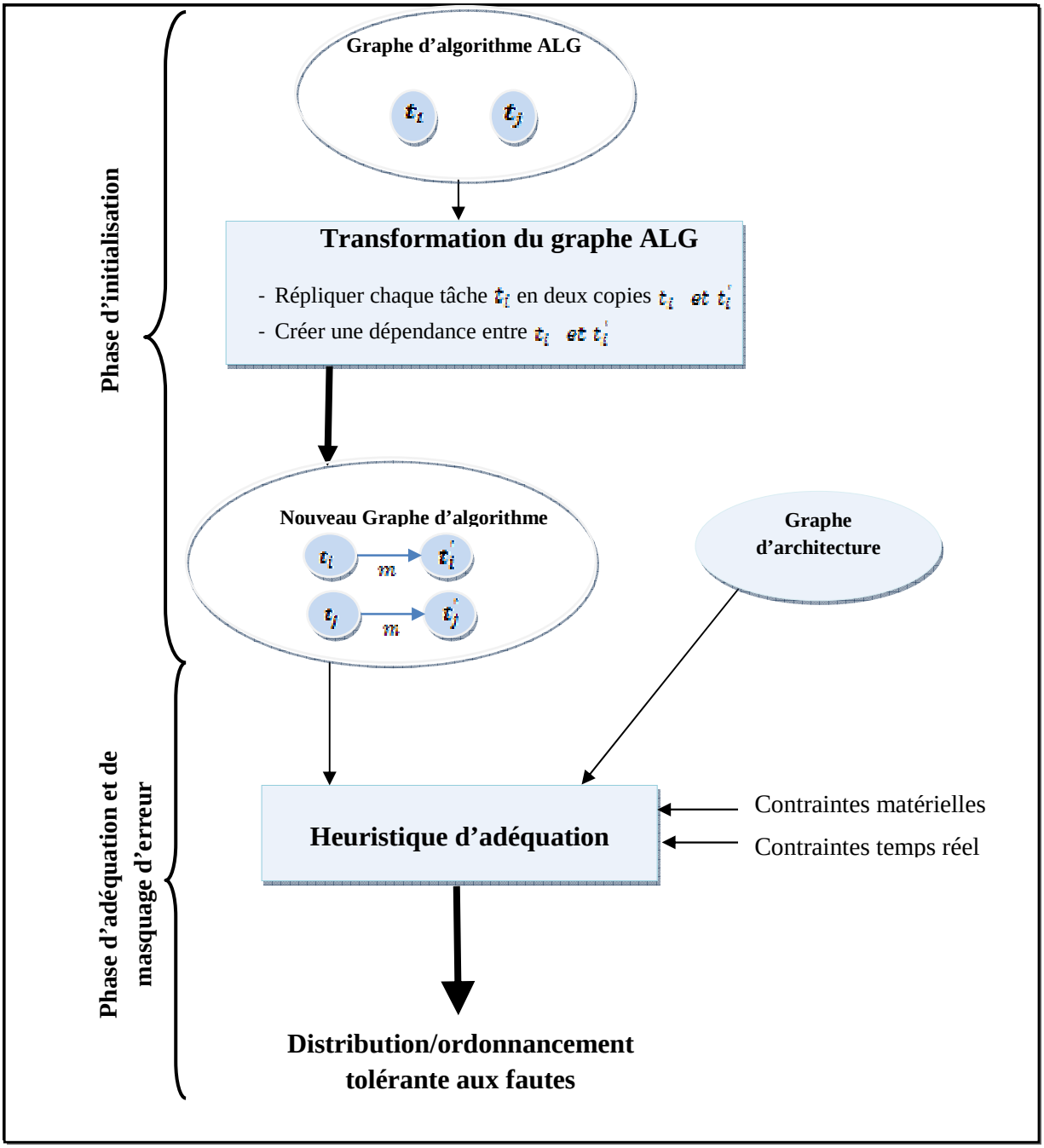


Figure6.3 : Méthodologie AAA-FAULT$^{t.ind}$

1) Phase de transformation du graphe

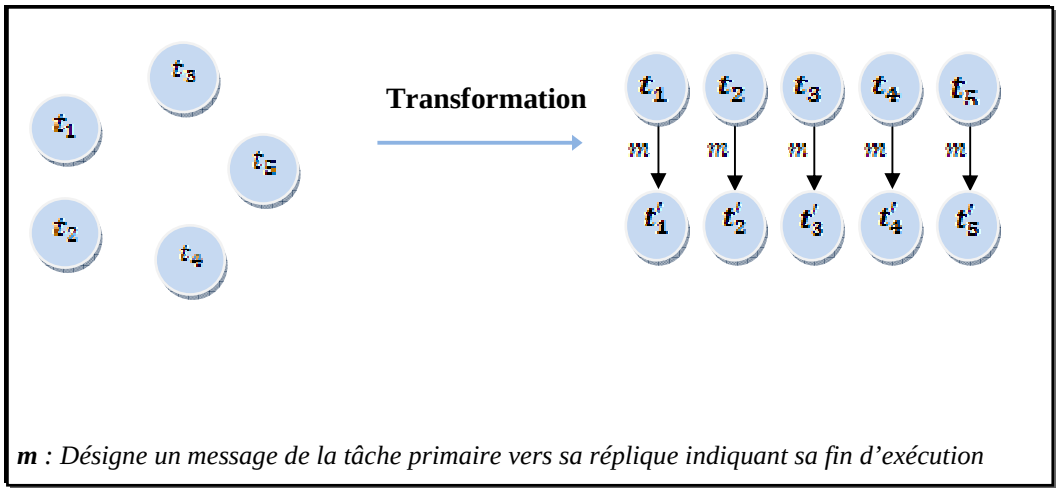
L'objectif de cette phase est la redondance logicielle d'une façon que chaque tâche du graphe d'algorithme soit répliquée et une dépendance de données soit créée entre ces deux répliques.

Définition 41 (Graphe ALG^n) : A la différence du premier graphe, le nouveau graphe est un graphe orienté. Dans ce nouveau graphe chaque tâche t_i est répliquée en deux copies

t_i et t_i' (tâche primaire et tâche secondaire) connectées entre elles par une dépendance de données ($t_i \rightarrow t_i'$).

Définition 42 (Tâche réplique) : *Une tâche réplique* t_i^k *de* ALG^n *représente la* $k^{ième}$ *réplique de la tâche ti de ALG. Puisque le système est sans faute logicielle, toutes les répliques d'une même opération sont identiques, c'est-à-dire qu'elles ont le même code informatique.*

Un exemple de transformation est donné par la figure 6.4 pour tolérer la faute d'un seul processeur.



***m** : Désigne un message de la tâche primaire vers sa réplique indiquant sa fin d'exécution*

Figure6.4 : Exemple de transformation d'un graphe d'algorithme

Remarque :

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

2) Phase d'adéquation (distribution/ordonnancement)

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

a) Tolérance aux fautes des processeurs

Nous avons supposé que le bus de communication est fiable, Donc notre solution est basée sur la redondance active des composants logiciels pour tolérer les fautes d'un seul processeur. Pour cela chaque tâche t_i doit être répliquée et les deux répliques doivent être

placées activement sur deux processeurs distincts (P_i , P_j). Si le processeur de la tâche primaire défaille alors sa réplique masque cette erreur et l'ordonnancement sera reconfiguré, si non, à sa terminaison, elle envoie un message de blocage à sa réplique.

b) Tâches principales de l'heuristique d'adéquation

- Calcule des dates de début de toutes les tâches du graphe d'algorithme en utilisant la pression d'ordonnancement [1]. Chaque tâche contient deux dates de début : T_{best} et T_{worst}.

  T_{best} **:** C'est la date de début des copies primaires.

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

Remarque :

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

c) Notation

Nous utilisons les mêmes notations que celles données dans la section 2.5.2.

d) Principes de l'heuristique

L'heuristique que nous proposons est basée sur une fonction de coût appelée la pression d'ordonnancement notée $\sigma^n_{(T_i,P_j)}$, donnée par l'équation 2.5 (page 24), dont l'objectif global est de minimiser la longueur de distribution/ordonnancement en absence et en présence de défaillance d'un seul processeur.

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

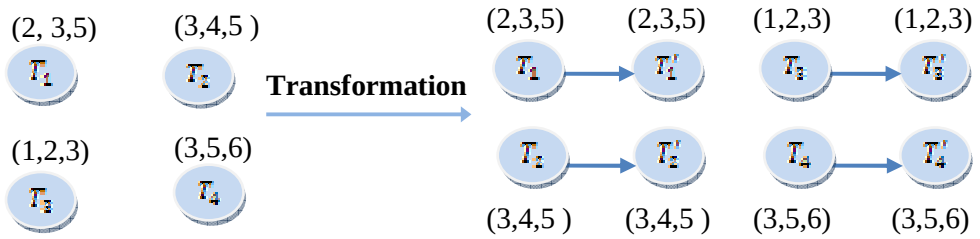
- Toutes les tâches s'exécutent au même temps, selon leur date de début d'exécution.
- Si aucun processeur n'est en panne, l'exécution des répliques de toutes les tâches sera ignorée (pour le cycle d'exécution en cours).
- A la défaillance d'un processeur, les répliques de ses tâches masquent cette défaillance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

L'exemple ci-dessous présente une distribution/ordonnancement d'un graphe d'algorithme sur un graphe d'architecture dans laquelle les tâches sont représentées par des boites dont la hauteur est proportionnelle à leur durée d'exécution.

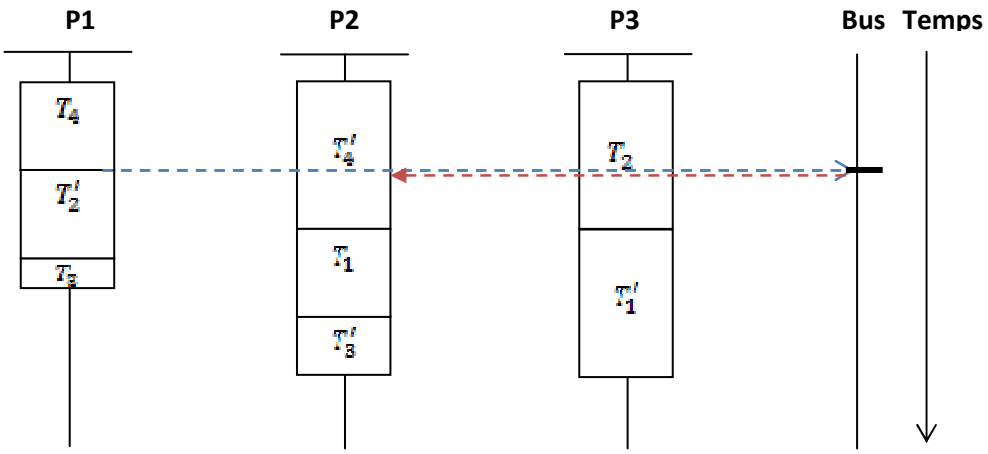
Soit l'architecture matérielle suivante :

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

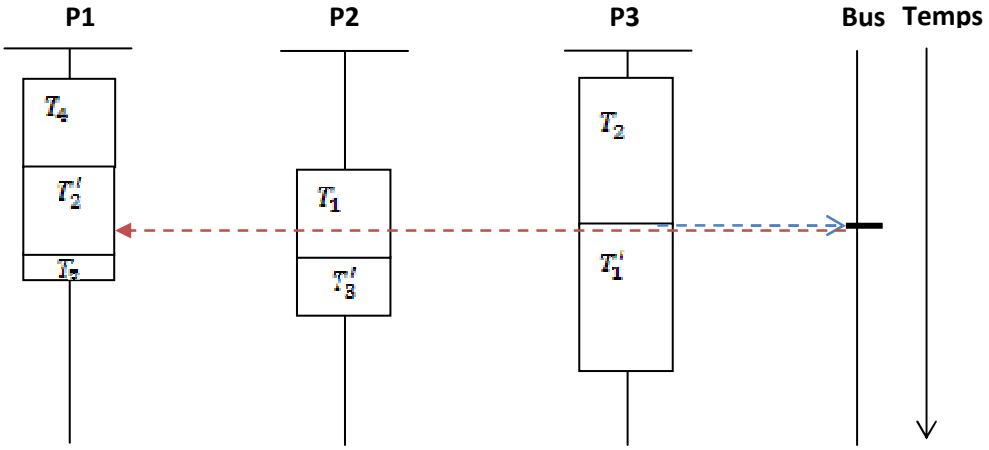
Et soit le graphe d'architecture logicielle ALG transformé en ALG^n :



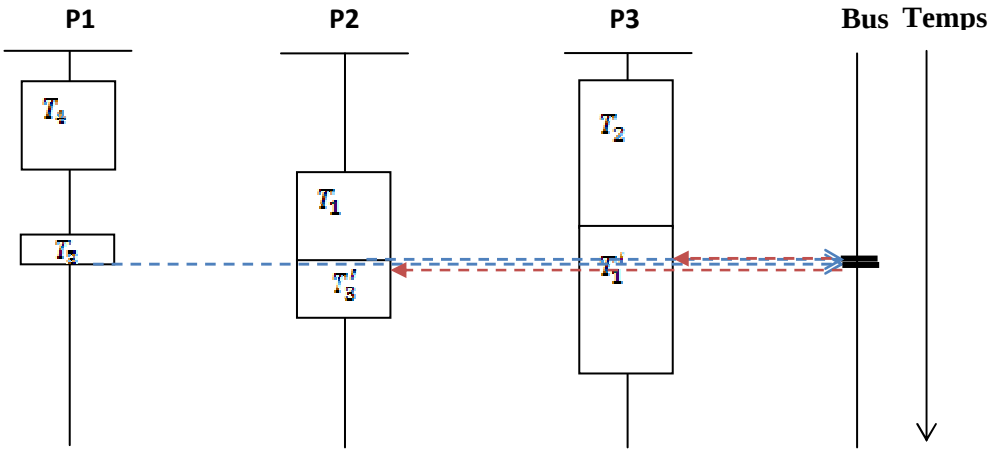
Alors la distribution/ordonnancement tolérante aux fautes est schématisée comme suit :



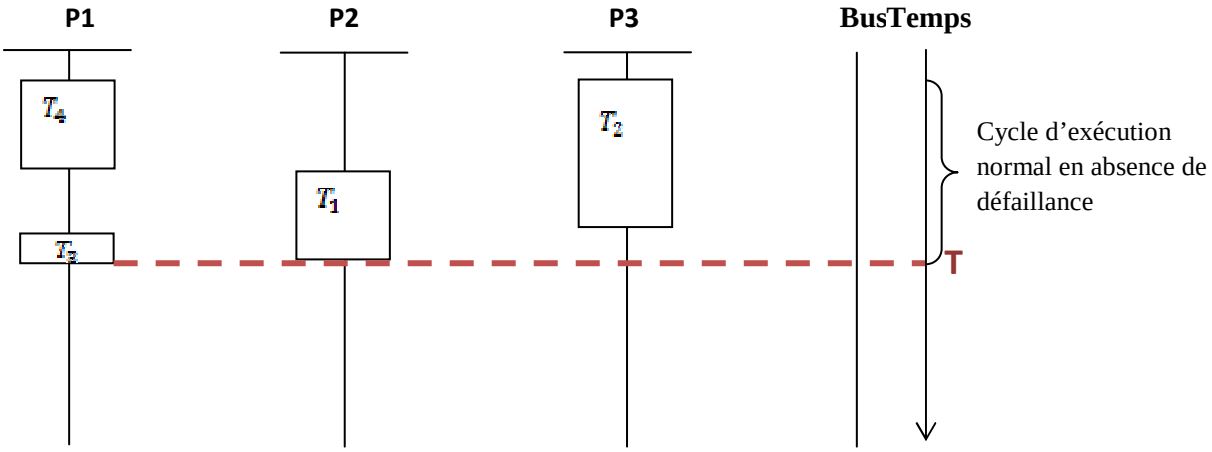
T_4 est exécutée normalement alors T_4' se bloque, l'ordonnancement devient donc :



T_2 est exécutée normalement alors T_2' se bloque, l'ordonnancement devient donc :



A la fin T_1 termine son exécution et bloque T_1', même chose pour T_3 .

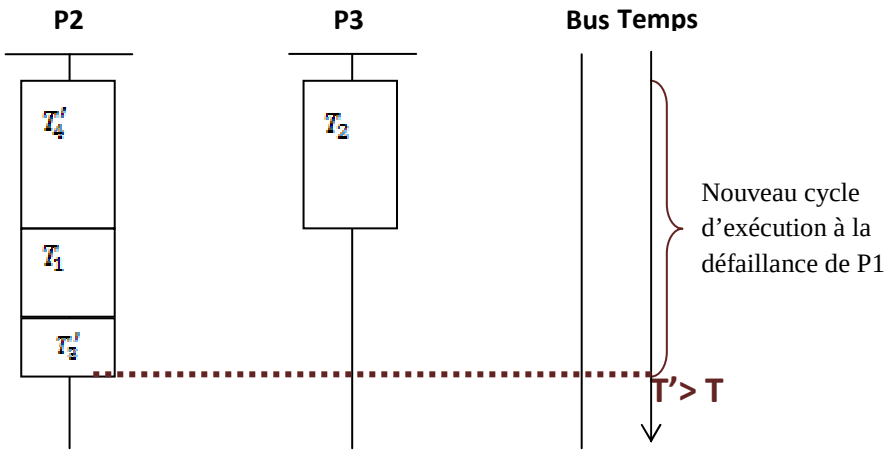


Distribution/ordonnancement sans défaillance

Note :

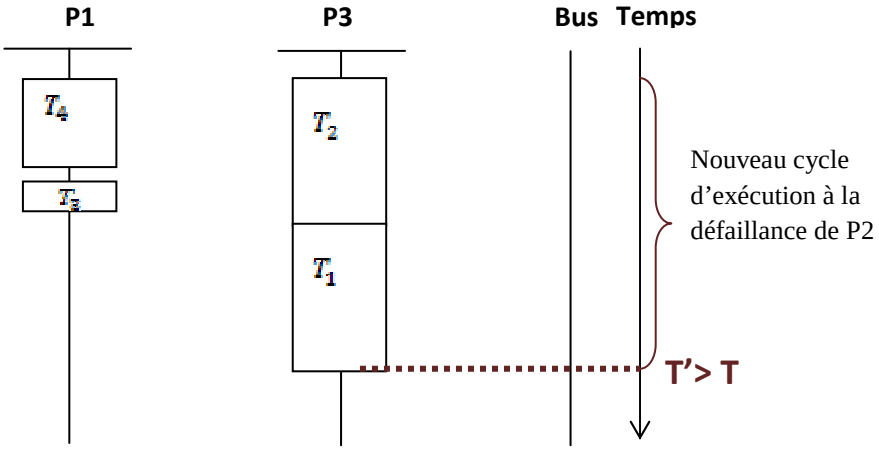
▬▬ : Présente le message de blocage (temps de communication est nul)

Dans le cas d'une panne par exemple de P1 au début du cycle, c.à.d. t_4 et t_3 ne s'exécutent pas, alors leurs répliques implantées dans P2 masquent cette défaillance et l'ordonnancement des tâches sera mis à jour en éliminant P1 et toutes les tâches secondaires des tâches non défaillantes. Le système suit donc le nouvel ordonnancement.



Distribution ordonnancement avec défaillance de P1

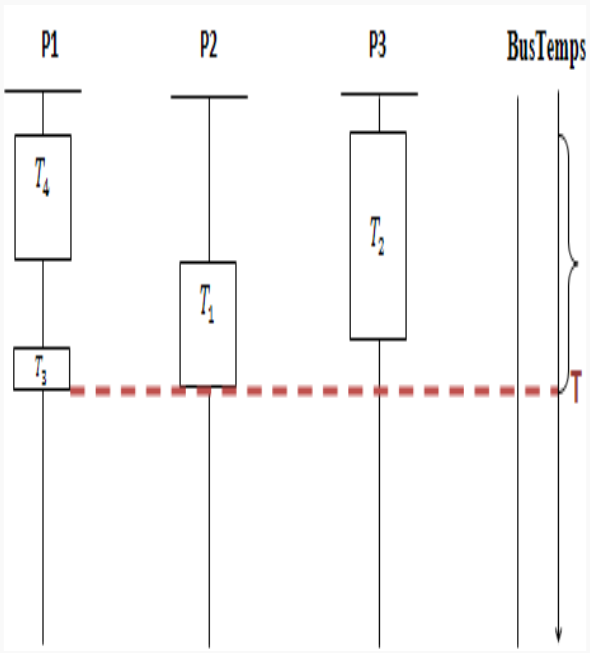
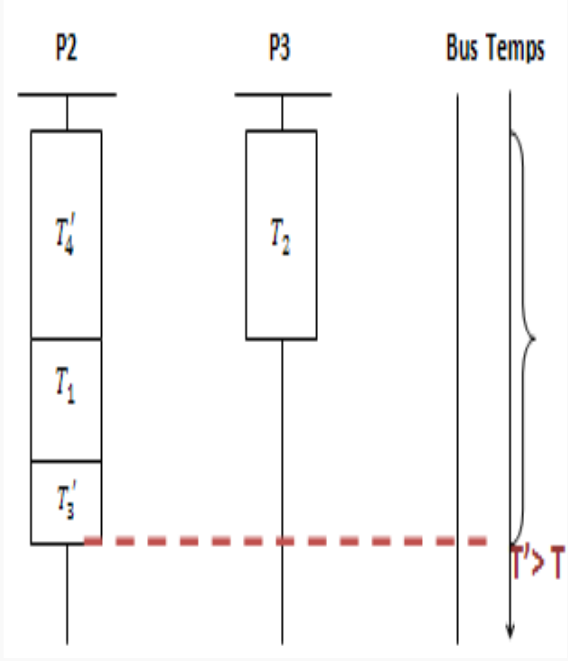
Dans le cas d'une panne de P2 au moment de l'exécution de T_1, t_1 ne peut s'exécuter. La réplique de ce dernier masque sa défaillance et l'ordonnancement des tâches sera mis à jour en éliminant P2 et toutes les tâches secondaires des tâches non défaillantes. Le système suit donc le nouvel ordonnancement.



Distribution ordonnancement avec défaillance de P2

Nous constatons, que le temps d'exécution du système dans chaque cycle est augmenté dans le cas de défaillance d'un processeur comme le montre le tableau ci-dessous.

Tableau 1 : Comparaison entre la distribution/ordonnancement avant et après la défaillance de P1

Distribution/ordonnancement avant l'apparition de la défaillance	Distribution/ordonnancement après la défaillance de P1
	

e) Présentation de l'heuristique

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

- **Phase d'initialisation :** Consiste à placer les tâches qui n'ont pas de prédécesseurs dans une liste appelée $\mathbf{T_{cand}}$, dans ce cas (tâches indépendantes) toutes les copies primaires du graphe sont concernées. Une autre liste appelée T_{fin} est initialisée à $\emptyset$, cette liste contient les tâches déjà placées.

- **Phase de sélection :** Le but de cette phase est de sélectionner la candidate la plus urgente pour être placée et ordonnancée. Elle est réalisée en trois étapes :

- Tout d'abord, la fonction de la pression d'ordonnancement est calculée pour chaque tâche candidate sur tous les processeurs.
- Deuxièmement, un processeur P_{best}, parmi tous les processeurs, qui minimise la pression d'ordonnancement est sélectionné pour chaque tâche candidate. Pour la copie secondaire de chaque tâche, le processeur P_{best} doit être distinct de celui de sa copie primaire.
- Puis, le couple le plus urgent (t_{best}, P_{best}) parmi tous les couples (t_i , P_{best}) qui maximise la pression d'ordonnancement est sélectionné pour l'ordonnancement.

- **Phase de distribution et d'ordonnancement :** Cette phase est chargée de placer/ordonnancer chaque tâche sélectionnée sur son meilleur processeur, ainsi que la dépendance de données entre cette tâche et sa réplique.

- **Phase de vérification des contraintes temporelles :** à chaque fois, l'algorithme doit vérifier le respect de la contrainte de temps (deadline) spécifiée par le système.

- **Phase de mise à jour :** l'objectif de cette phase est de mettre à jour la liste des tâches déjà allouées T_{fin}, dans laquelle les tâches nouvellement placées sont ajoutées ; et la liste des tâches candidates T_{cand} , dans laquelle les tâches déjà placées doivent être supprimées et les nouvelles tâches ajoutées à cette liste sont celles qui ont tous leurs prédécesseurs dans la liste des tâches déjà placées.

----------ALGORITHME----------

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

----------INITIALISATION

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

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

$T_{fin}^{(1)} := \emptyset$ *faire*

----------BOUCLE DE DISTRIBUTION ET D'ORDONNANCEMENT

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

----------SELECTION

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

----------DISTRIBUTION ET ORDONNANCEMENT

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

----------VERIFICATION DES CONTRAINTES TEMPORELLES

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

----------MISE A JOUR

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

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

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

Fin tant que ----------FIN DE L'ALGORITHME

6.3. Présentation de l'algorithme de distribution/ordonnancement tolérant aux fautes des tâches dépendantes AAA-FAULT$^{t.dep}$

6.3.1. Modèle de fautes

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

Hypothèse 1 : les tâches sont dépendantes.

Hypothèse 2 : Les fautes sont des fautes des processeurs.

Hypothèse 3 : Le système accepte les fautes d'un seul processeur durant tous les cycles d'exécution du graphe d'algorithme sur le graphe d'architecture.

Hypothèse 4 : Les fautes sont permanentes.

Hypothèse 5 : Le bus de communication est supposé fiable.

Hypothèse 6 : Le logiciel est supposé sans fautes.

6.3.2. Données du problème

Afin d'aboutir à la génération automatique d'une distribution/ordonnancement tolérante aux fautes qui satisfait les contraintes temps réel (C_{tr}) et matérielles (C_{mt}) et qui minimise la longueur de la distribution/ordonnancement en absence et en présence de défaillance, un ensemble de données doit être défini avant d'aborder le principe général de la méthodologie proposée.

Problème 3[22] **:** étant donné :

- Une architecture matérielle hétérogène AMH composée d'un ensemble P de processeurs et d'un ensemble L qui contient un bus B de communication (liaison à bus).

P = {P_1, P_2, …….P_m} , L = {B}

Exemple :

[22] Problème de distribution/ordonnancement tolérant aux fautes des tâches dépendantes

- Un algorithme ALG, composé d'un ensemble T de tâches et d'un ensemble D de dépendances de données.

T= {t_1 , t_2 , ……t_n} , D = {……,($t_i \rightarrow t_j$), ……}

Exemple :

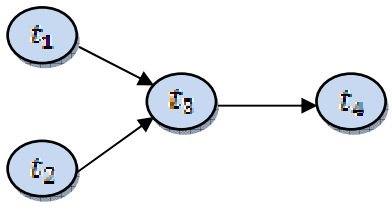


Figure6.5 : exemple d'un graphe d'algorithme

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

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

A : $ALG \rightarrow AMH$

$T_i \rightarrow A(T_i) = (P_j , O_k)$

6.3.3. Principe général de la méthodologie AAA-FAULT$^{t.dep}$

Semblablement au problème 2, le problème 3 peut être vu comme étant un problème à deux parties ; un problème de temps réel et un problème de tolérance aux fautes. C'est un problème d'optimisation NP-difficile. Pour le résoudre en temps polynomial, nous proposons une méthodologie appelée **AAA-FAULT$^{t.dep}$** qui essaie de trouver une solution proche de la solution optimale. A la différence de la méthodologie **AAA-FAULT$^{t.ind}$**, **AAA-FAULT$^{t.dep}$** est basée sur la redondance passive des tâches du système et en conséquence sur un mécanisme de détection d'erreurs.

Définition 43 (AAA-FAULT$^{t.dep}$) : Est une nouvelle méthodologie pour optimiser d'une part la génération automatique de distribution/ordonnancement et d'autre part pour tolérer les fautes permanentes d'un seul processeur durant un cycle d'exécution de l'architecture logicielle sur l'architecture matérielle. Dans cette méthodologie, la tolérance aux fautes est basée sur la redondance passive, donc nécessite un mécanisme de détection d'erreurs qui est dans notre cas une nouvelle tâche ajoutée au système appelée watchdog, notée dorénavant « W ».

En plus du *principe1* de la méthodologie précédente, nous présentons ainsi les principes suivants pour cette nouvelle méthodologie :

Principe 2 : Redondance passive

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

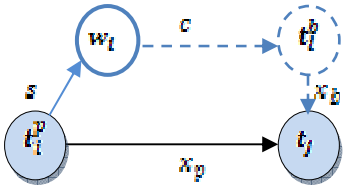
Principe 3 : Tolérance aux fautes permanentes

La méthodologie **AAA-FAULT$^{t.dep}$** sert à générer automatiquement une distribution/ordonnancement tolérante aux fautes permanentes d'un seul processeur à n'importe quelle étape dans chaque cycle d'exécution de l'algorithme. Elle est composée de trois phases : la phase de transformation, la phase d'adéquation et la phase de tolérance aux fautes détectées.

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

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

La figure suivante montre un exemple de transformation de chaque tâche t_i et chaque dépendance de données $t_i \rightarrow t_j$



La tâche qui n'a pas de successeur à la forme suivante dans le graphe :

- ***Dans la phase d'adéquation,*** l'heuristique proposée met en correspondance le nouveau graphe ALG^n et l'architecture matérielle AMH pour réaliser un ordonnancement optimal.
- ***La phase de tolérance aux fautes*** consiste à détecter la défaillance d'un processeur puis mettre à jour l'ordonnancement de ses tâches.

Principe 4 : Détection d'erreurs

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

Le rôle de la tâche W est de recevoir un signal indiquant l'exécution de son prédécesseur qui est la tâche t_i. Ainsi, après un certain temps Δ[23] si elle ne reçoit aucun signal elle détecte alors que le processeur P_i qui exécute la tache t_i est en panne, par suite elle réveille (active) la copie secondaire de la tâche (t_i) implantée sur un autre processeur, et cela par l'envoi d'un message de réveil à travers le bus ou via une communication intra-processeur.

Le corps de la tâche W a la forme suivante :

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

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

Fin Si

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

La figure 6.6 schématise une représentation de la méthodologie **AAA-FAULT**$^{t.dep}$.

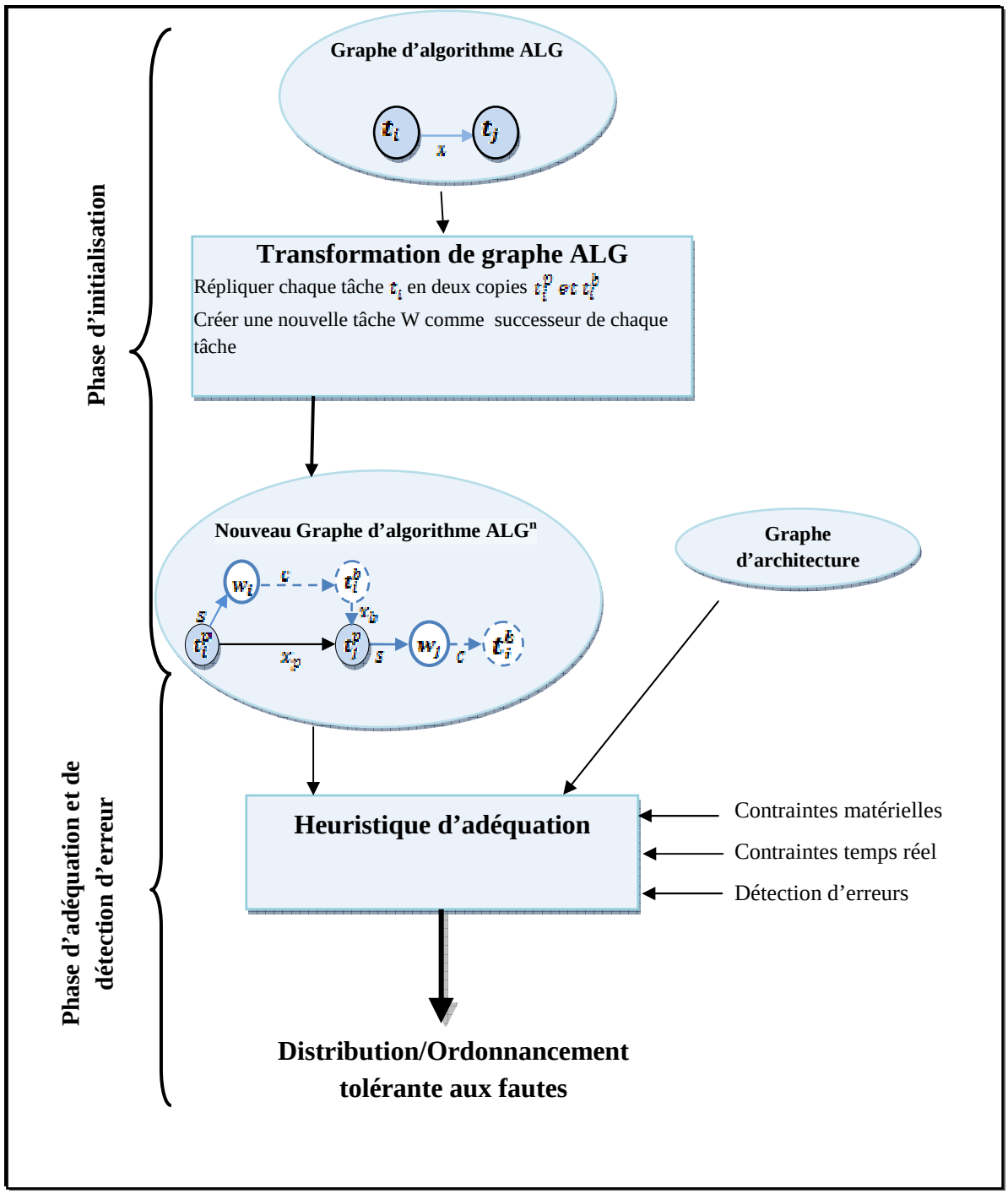


Figure6.6 : Méthodologie AAA-FAULT$^{t.dep}$

1) Phase de transformation du graphe

L'objectif de cette phase est la redondance logicielle d'une façon où chaque tâche du graphe d'algorithme est répliquée et pour chaque tâche une autre tâche W est créée pour la surveiller.

Définition 44 (Graphe ALG^n) : Le nouveau graphe est aussi bien un graphe orienté. Dans ce nouveau graphe, deux nouvelles tâches sont ajoutées à chaque tâche t_i. La première c'est sa copie secondaire, et la deuxième appelée W surveille le bon fonctionnement de son processeur. Trois nouvelles dépendances sont en effet créées comme l'explique le principe 3.

Définition 45 (Tâche watchdog) : Une tache W est une tâche dont le rôle est de détecter la panne d'un processeur puis d'activer les répliques de ses tâches, son temps d'exécution est presque nul.

Définition 46 (Dépendance conditionnée) : Une dépendance est dite conditionnée si son exécution dépend de la panne d'un processeur, c.-à-d. elle s'exécute si un processeur donné est défaillant si non elle ne s'exécute jamais.

Un exemple de transformation est donné par la figure 6.7 pour tolérer les fautes d'un seul processeur.

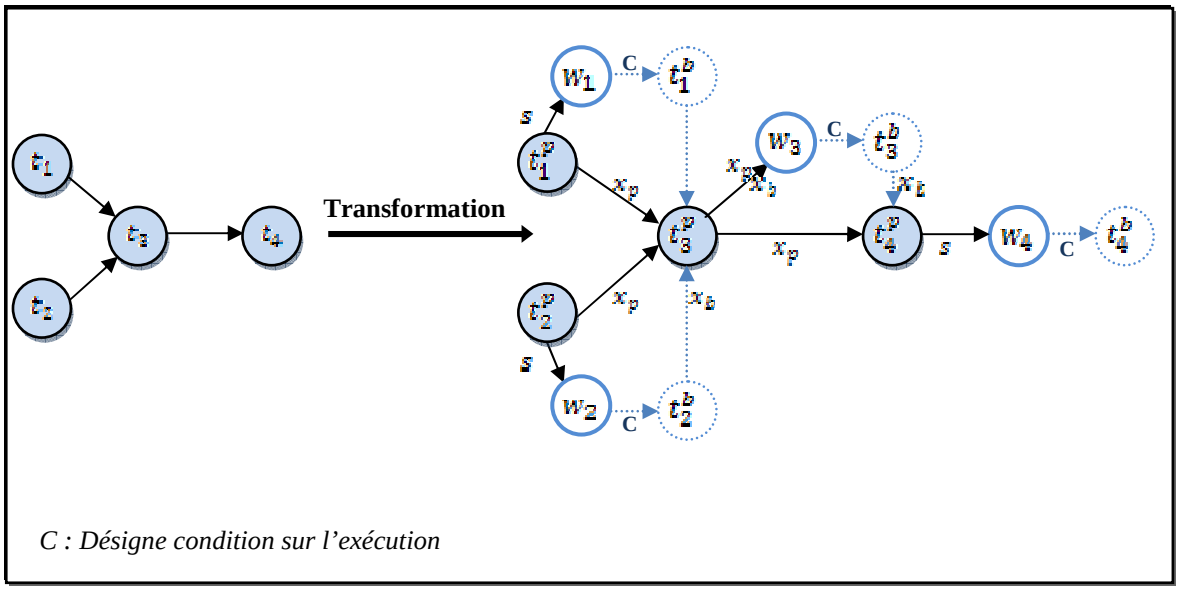


Figure6.7 : Exemple de transformation d'un graphe flot de données

2) Phase d'adéquation (distribution/ordonnancement) et de tolérance aux fautes

Elle est basée sur une heuristique de distribution/ordonnancement hors-ligne qui consiste à mettre en correspondance de manière efficace le nouveau graphe d'algorithme ALG^n sur le graphe d'architecture AMH.

a) Tolérance aux fautes des processeurs

Nous avons supposé que le bus de communication est fiable, donc notre solution est basée sur la redondance passive des composants logiciels pour tolérer les fautes d'un seul processeur. Pour cela chaque tâche t_i doit être répliquée et placée sur deux processeurs distincts (P_i , P_j).

Comme nous avons déjà cité, cette technique nécessite un mécanisme de détection d'erreurs qui est la tâche W. Son rôle est dans un premier temps de recevoir un signale indiquent l'exécution de son prédécesseur (la tâche t_i^p), et dans un deuxième temps d'envoyer un message qui réveille la copie backup (t_i^b) de la tâche t_i^p placée dans un autre processeur si elle ne reçoit pas ce signale après un certain temps Δ .

Rappels et Notes :

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

b) Tâches principales de l'heuristique d'adéquation

- Calcul des dates de début de toutes les tâches du nouveau graphe d'algorithme en utilisant la fonction de la pression d'ordonnancement [1]. Chaque tâche contient deux dates de début : T_{best} et T_{worst} .

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

  T_{worst} **:** C'est la date de début des copies backups (à la présence d'une défaillance).
- Placement actif des copies primaires : au démarrage du système, seules les copies primaires de chaque tâche sont à exécuter.
- Placement passif des copies secondaires : pour ne pas surcharger les processeurs, les copies backups ne s'exécutent qu'à la détection d'une défaillance.
- Prise en compte de toutes les contraintes : le coût d'exécution, la contrainte temps réel et les contraintes matérielles.
- Prévision du comportement temps réel (prédictibilité)

Remarque :

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

c) Notation

Nous utilisons les mêmes notations que celles données dans la section 2.5.2.

d) Principe de l'heuristique

L'heuristique que nous proposons est basée sur une fonction de coût appelée la pression d'ordonnancement notée $\sigma^{n}_{(Ti,Pj)}$, donnée par l'équation 2.5 (page 24), dont l'objectif global est de minimiser la longueur de la distribution/ordonnancement en absence et en présence de défaillance d'un seul processeur. Les principes de cette heuristique sont :

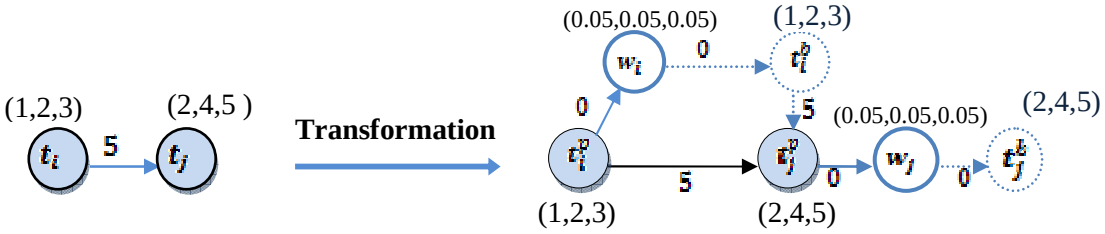
- Chaque tâche envoie ces données de dépendances dans le bus, seules les tâches concernées reçoivent le message.
- En cas du non réception d'un signal qui signifie la bonne exécution de chaque tâche primaire par la tâche W après un temps « timeout », W réveille les copies secondaires par une communication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

L'exemple ci-dessous présente une distribution/ordonnancement d'un graphe d'algorithme sur un graphe d'architecture dans laquelle les tâches (resp.les communications) sont représentées par des boites dont la hauteur est proportionnelle à leur durée d'exécution (resp.de communications).

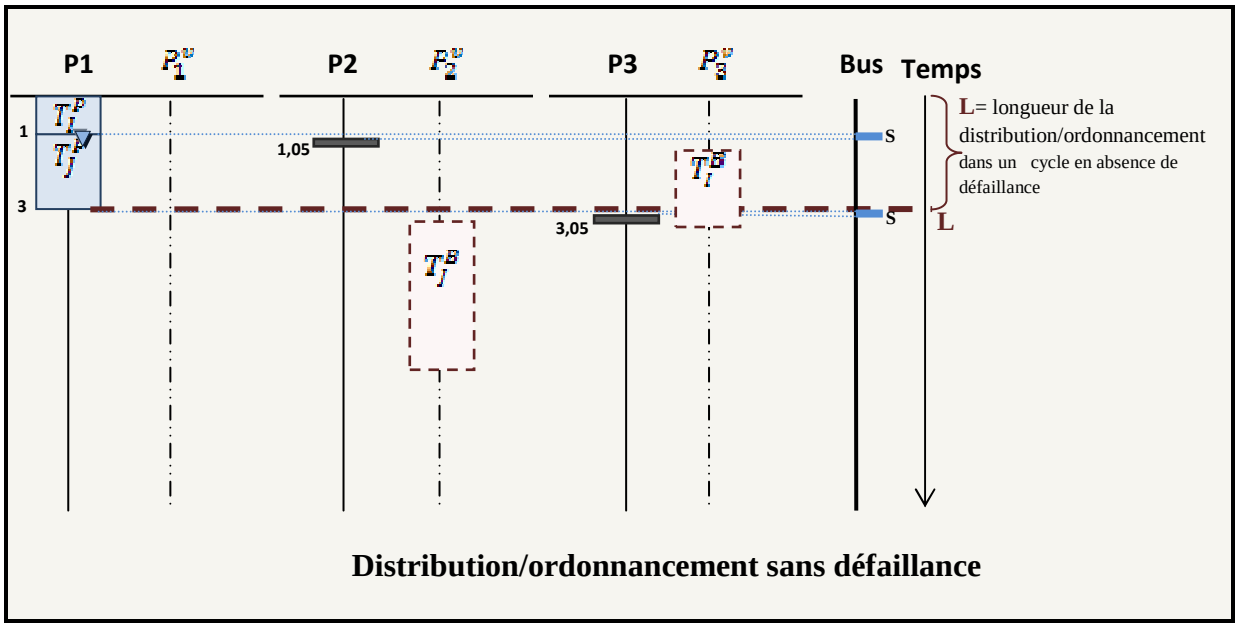
Soit l'architecture matérielle hétérogène suivante :

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

Et soit le graphe d'architecture logicielle ALG transformé en ALG^n dont chaque nœud (tâche) est associé par ses durées d'exécution sur les différents processeurs, et chaque arc (dépendance) est associé par la durée de communication entre les nœuds qu'il relie :



Alors la distribution/ordonnancement tolérante aux fautes est schématisée comme suit :



Distribution/ordonnancement sans défaillance

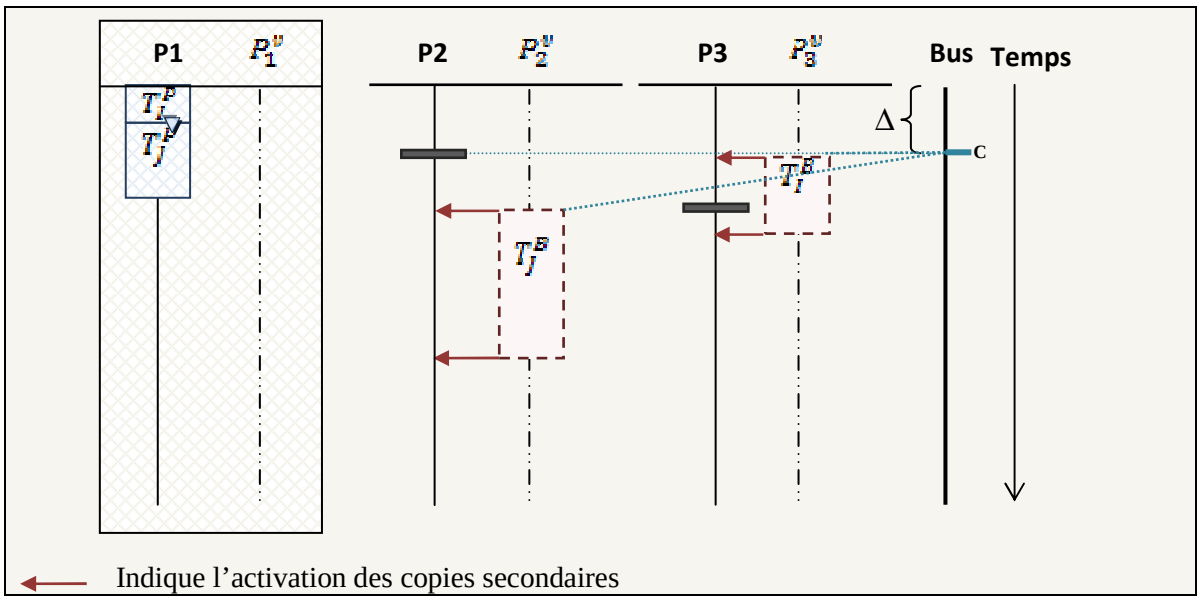
Notes :

▬ : Présente la tâche watchdog

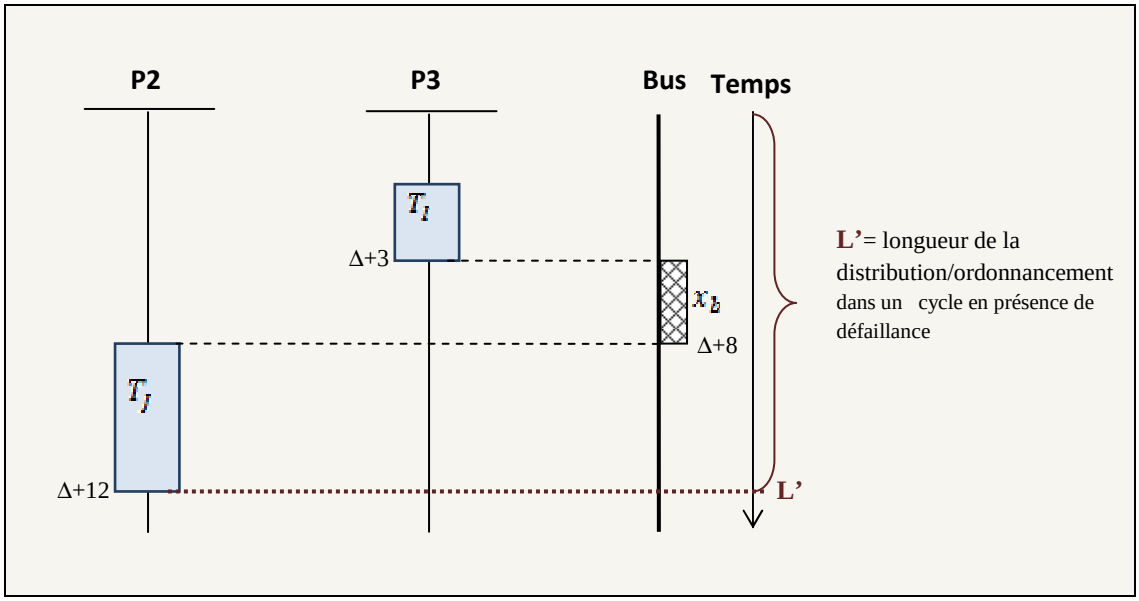
▼ : Présente une communication intra-processeur

P_i^v (Processeur Pi virtuel) : désigne le placement passif des répliques des tâches sur le processeur Pi en attendant leur activation.

Dans le cas d'une panne par exemple de P1 au début du cycle, W_i ne reçoit pas un signal de T_i^P , donc après le timeout ΔW_i réveille la réplique backup T_i^B implantée sur P3 en exécutant la dépendance conditionnée C, de même ce message de réveil est reçu par la tâche T_j^B. Donc la longueur de la distribution/ordonnancement devient :

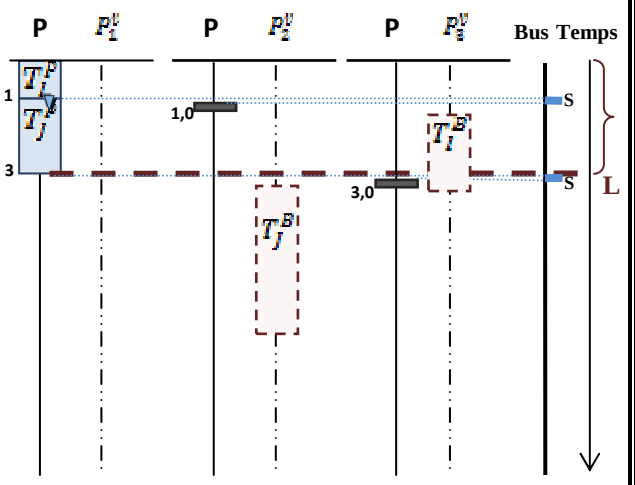
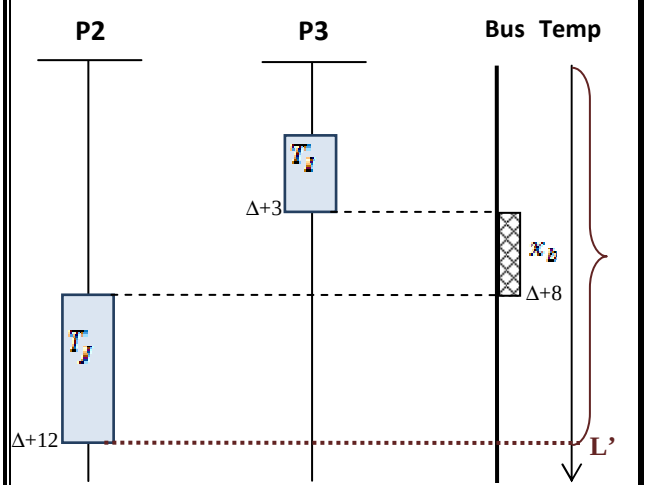


Alors T_I^B sera activée et exécutée sur P3, elle envoie le résultat x_b à T_J^B par une communication inter-processeur pour être exécutée et fournir les résultats. Le processeur P1 (resp. P_1^W) sera exclu de l'architecture matérielle, ainsi que toutes les répliques et les tâches W .



En fait l'exécution du système dans les cycles suivants suit cette nouvelle distribution/ordonnancement.

Tableau 1 : Comparaison entre la distribution/ordonnancement avant et après la défaillance de P1

Distribution/ordonnancement avant l'apparition de la défaillance	Distribution/ordonnancement après l'apparition de la défaillance
	

Remarque :

Le temps d'exécution en présence de défaillance est plus grand par apport au cas normal, c'est-à-dire en absence de défaillance. Mais, ce qui nous intéresse c'est le respect des contraintes temps réel (deadline) et la continuité du système à délivrer les services prévus même en présence des fautes qui conduisent à des défaillances.

f) Présentation de l'heuristique

L'heuristique de distribution/ordonnancement consiste à placer et à ordonnancer, à chaque exécution répétitive de l'algorithme « étape (n) », plusieurs tâches (resp. dépendances de données) sur plusieurs processeurs (resp. liens). Elle est composée de cinq phases.

----------ALGORITHME----------

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

----------INITIALISATION

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

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

$T_{fin}^{(1)} := \emptyset$ *faire*

----------BOUCLE DE DISTRIBUTION ET D'ORDONNANCEMENT

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

----------SELECTION

- *Calculer pour chaque candidate* t_i *de* $T_{cand}^{(n)}$ *et chaque processeur* P_j *∈ AHM la pression d'ordonnancement ;*
- *Sélectionner pour chaque candidate* t_i *le processeur* p_{best} *qui minimise la pression d'ordonnancement à condition que le processeur de la réplique d'une tâche doit être différent de celui de cette dernière ;*
- *Sélectionner parmi les couples* (t_i , p_{best}) *le meilleur couple* (t_{best}, p_{best}) *qui maximise la pression d'ordonnancement ;*

----------DISTRIBUTION ET ORDONNANCEMENT

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

----------VERIFICATION DES CONTRAINTES TEMPORELLES

- ***si*** $\left(R_{Ti.Pj}^{(n)} > C_{tr}\right)$ ***alors*** *terminer et répondre « échec » ;*

----------MISE A JOUR

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

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

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

Fin tant que ----------FIN DE L'ALGORITHME

L'algorithme est divisé en :

Phase d'initialisation : dans laquelle :

- Les tâches qui n'ont pas de prédécesseurs sont placées dans une liste appelée T_{cand} ;
- Une autre liste appelée T_{fin} est initialisée à $\emptyset$, cette liste contient les tâches déjà placées.

Phase de sélection : dans cette phase, un choix de la tâche la plus urgente qui va s'implanter sur le meilleur processeur est effectué par le calcul de la pression d'ordonnancement pour chaque tâche candidate sur tous les processeurs. Dans le cas où la meilleure tâche sélectionnée est une copie secondaire, alors l'euristique doit lui choisir un deuxième processeur distinct de celui de sa copie primaire.

Phase de distribution et d'ordonnancement : Après avoir sélectionné la meilleure candidate t_{best} et son meilleur processeur p_{best} pendant la phase de sélection, il ne reste plus qu'à la placer/ordonnancer, la première copie de chaque tâche sélectionnée est allouée activement sur son processeur (à exécuter), par contre sa deuxième copie est allouée passivement sur le deuxième processeur choisi en attendant son activation.

Phase de vérification des contraintes temporelles : à chaque fois, l'algorithme doit vérifier le respect de la contrainte de temps, deadline, spécifiée par le système.

Phase de mise à jour : l'objectif de cette phase est de mettre à jour :

- La liste des tâches déjà allouées T_{fin} : dans laquelle les tâches nouvellement placées sont ajoutées ;
- La liste des tâches candidates T_{cand} : les tâches déjà placées doivent être supprimées de cette liste et ses nouvelles tâches ajoutées sont celles qui ont leurs prédécesseurs dans la liste des tâches déjà placées.

6.4. Prédiction du comportement temps réel

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

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

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

6.5. Conclusion

Dans ce chapitre, nous avons proposé deux nouvelles heuristiques de la tolérance aux fautes permanentes d'un seul processeur dans une architecture distribuée hétérogène à liaison bus pour assurer la sureté de fonctionnement des systèmes temps réel embarqués. Les deux heuristiques sont basées sur la redondance des composants logiciels afin de satisfaire les contraintes temps réel et d'embarquabilité. La première appelée AAA-FAULT$^{t.ind}$ se base sur la redondance active, tandis que la deuxième appelée AAA-FAULT$^{t.dep}$ se base sur la redondance passive.

Dans la première méthodologie, chaque tâche et son réplique s'exécutent à la fois selon la date de début d'exécution de chacune, mais au moment où la tâche primaire termine son exécution elle envoie un message à sa réplique, qui est allouée sur un autre processeur, pour bloquer son exécution et éviter de perdre le temps d'exécution des autres tâches implantées sur ce même processeur. De cette façon nous pouvons masquer les tâches défaillantes et de plus gagner le temps dans le cas contraire.

Dans la deuxième méthodologie et dans le cas du non défaillance d'aucun processeur, l'heuristique profite de la réduction du temps et de la surcharge des processeurs; tandis que en cas de défaillance d'un processeur, l'heuristique a besoin d'un mécanisme de détection d'erreur qui est une tâche surveillante dont le temps d'exécution est presque nul et les durées de communication qui la concernent sont aussi nulles.

Chapitre 7

Evaluation de la méthodologie *AAA-FAULT*$^{t.dep}$

7.1. Introduction

Il existe deux manières d'évaluer une méthodologie, relativement à d'autres méthodologies du même type ou bien de façon absolue.

Par ailleurs, il faut définir les critères utilisés pour cette évaluation. Il nous semble ici intéressant d'évaluer la longueur de la distribution/ordonnancement introduit par la méthodologie AAA-FAULT$^{t.dep}$ en présence et en absence de défaillances d'un seul processeur. Nous avons implémenté cette heuristique par le langage visuel c++.

7.2. Paramètres d'évaluation

Nous avons appliqué AAA-FAULT$^{t.dep}$ à un ensemble de graphes d'algorithmes générés aléatoirement et à deux graphes d'architecture, une architecture composée de six processeurs et une autre composée de cinq processeurs. Les processeurs de chaque architecture sont reliés par un bus de communication. Afin de générer les graphes d'algorithmes aléatoires, nous avons fait varier deux paramètres dans notre générateur aléatoire qui va être présenté plus loin dans ce chapitre :

- Le nombre d'opérations : N=10, 20, 30, ….,80 ;
- Le rapport entre le temps moyen de communication et le temps moyen d'exécution : CCR = 0.1, 0.5, 0.8, 1, 2, 2.5, 5, 10, 25.

7.3. Les résultats

7.3.1. Effet du rapport entre le temps moyen de communication et le temps moyen d'exécution sur AAA-FAULT$^{t.dep}$

Nous avons tracé dans la figure 7.1 la moyenne de la longueur de la distribution/ordonnancement de 50 graphes aléatoires pour N=50 taches, P=6 processeurs et CCR = 0.1, 0.5, 0.8, 1, 2, 2.5, 5 10 et 25.

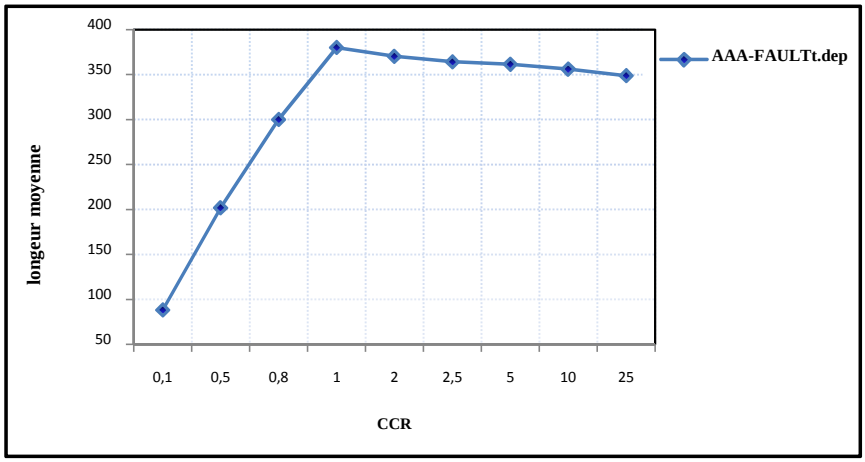


Figure7.1 : Effet du CCR pour P = 6 et N =50

Nous constatons dans ce graphe que les performances de la méthodologie AAA-FAULT$^{t.dep}$ s'augmentent lorsque le temps moyen d'exécution diminue par rapport au temps moyen de communication. Ceci est expliqué par l'augmentation considérable des dates de début d'exécution des copies secondaires lorsque le temps moyen d'exécution de leurs copies primaires est élevé.

7.3.2. Effet du nombre de tâches sur AAA-FAULT$^{t.dep}$

Afin d'évaluer l'effet du nombre de tâches N sur notre heuristique, nous avons tracé dans la figure 7.2 la longueur de la distribution/ordonnancement de 80 graphes aléatoires pour N = 10, 20, …, 80 , CCR=2 et p = 5, et ceci en absence et en présence de défaillance. Dans le graphe, L désigne la longueur de la distribution/ordonnancement temps réel. Cette figure montre que les performances de AAA-FAULT$^{t.dep}$ décroissent lorsque le nombre de tâches croît. Ceci s'explique par le fait que chaque tâche est répliquée en deux exemplaires, ainsi la topologie utilisée (bus) pour relier les différents processeurs (qui ne permettent que la transmission des données d'une seule tâche à la fois) augmente d'une manière considérable cette longueur.

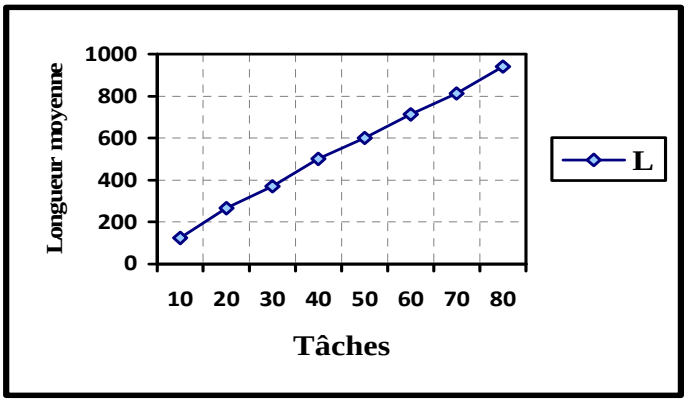


Figure7.2 : Effet de N sur AAA-FAULT$^{t.dep}$ pour p=5 et CCR=2

7.3.3. Effet du nombre de processeurs sur AAA-FAULT$^{t.dep}$

Pour évaluer l'effet du nombre de processeurs sur notre heuristique, nous avons fait varier le nombre de processeurs : P = 2, 3, 4, 5, 6, 7, 8, 9 sur un graphe d'algorithme composé de 40 tâches et un CCR égal à 1 pour tracer la moyenne de la longueur de la distribution/ordonnancement temps réel. Donc le graphe de la figure 7.3 montre que les performances de la méthodologie croissent lorsque le nombre de processeurs croît, mais à un moment donné elles commencent à diminuer.

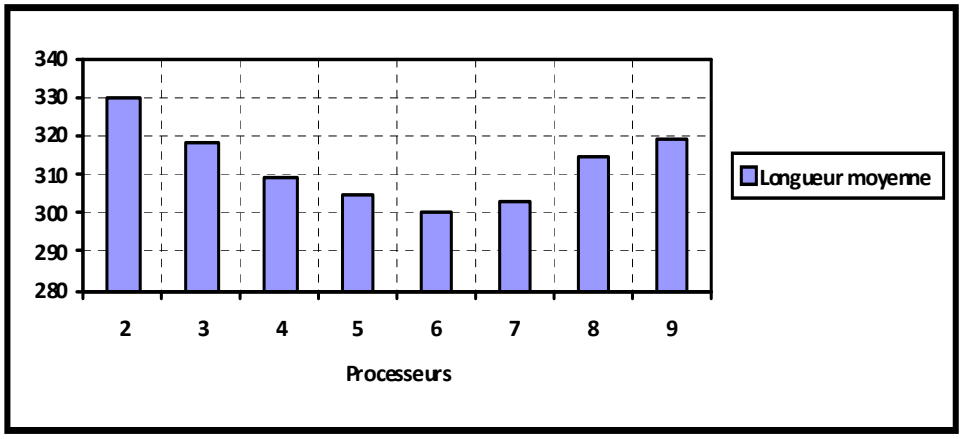


Figure7.3 : Effet du nombre de processeurs sur AAA-FAULT$^{t.dep}$ pour N= 40 CCR=1

Afin d'aboutir à ces études comparatives, il a été nécessaire de disposer d'un ensemble de graphes d'algorithmes que nous avons généré aléatoirement par un générateur de graphes aléatoires. Dans ce qui suit nous allons présenter notre générateur de graphes d'algorithmes.

7.4. Générateur de graphe d'algorithme

Notre générateur s'inspire des travaux précédents de Kalla [1]. Nous utilisons une technique de génération par niveaux, c'est-à-dire que le graphe d'algorithme est représenté par plusieurs niveaux (≥ 2). Chaque niveau i est composé de plusieurs nœuds (tâches), et chaque nœud t_i de niveau i possède au moins un prédécesseur t_j de niveau inférieur j tel que i > j.

Ce générateur est basé sur trois paramètres :

1. ***La taille du graphe n* :** c'est le nombre de nœuds dans le graphe ;
2. ***La hauteur maximale du graphe H* :** c'est le nombre maximum de niveaux dans le graphe ;
3. ***La largeur maximale du graphe L* :** c'est le nombre maximum de nœuds indépendants dans un niveau du graphe.

La figure suivante présente un exemple sur le processus de génération aléatoire de graphes.

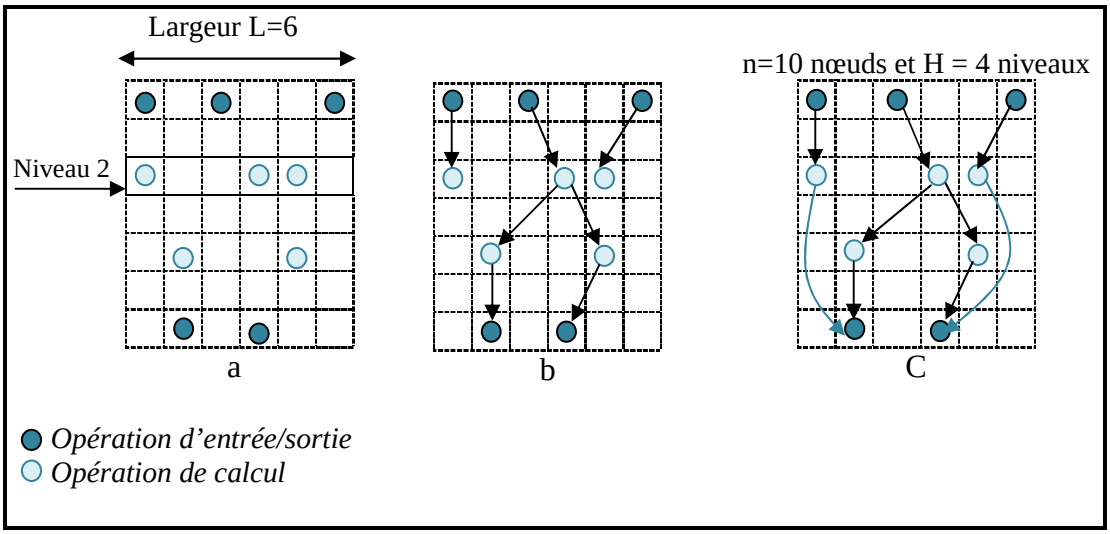


Figure7.4 Etapes de génération aléatoire d'un graphe d'algorithme

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

La phase de génération de nœuds est elle-même réalisée en deux étapes comme la présente la figure ci-dessus :

- Premièrement, nous tirons aléatoirement n nœuds dans une matrice de dimensions (L*H) ; L et H agissent sur la forme du graphe.
- Ensuite, nous construisons les N niveaux du graphe ; chaque niveau k est composé des nœuds situés sur une même ligne horizontale dans la matrice.

Dans la phase de génération d'arcs, nous avons générer les arcs en deux étapes :

- En premier lieu, nous choisissons aléatoirement pour chaque nœud t_i de niveau i un ou plusieurs nœuds comme prédécesseurs de niveau j, avec j = i-1.
- Après, nous choisissons aléatoirement pour chaque nœud t_i de niveau i zéro ou plusieurs nœuds comme prédécesseurs de niveau j, avec j < i-1.

7.5. Conclusion

Dans ce chapitre nous avons présenté quelques résultats donnés par la méthodologie AAA-FAULT$^{t.dep}$, nous avons évalué la longueur de la distribution/ordonnancement en faisant varier quelques paramètres (CCR, nombre de tâches, nombre de processeurs) pour y savoir leur effet. A la fin nous avons présenté notre générateur de graphes aléatoires qui permet de générer plusieurs graphes d'algorithmes.

Conclusion et perspectives

Dans ce mémoire, nous avons considéré une propriété cruciale qui doit être prise en compte à la conception des systèmes temps réel embarqués critiques, qui est la sûreté de fonctionnement. La tolérance aux fautes est la technique adoptée dans ce travail pour réaliser des systèmes sûrs de fonctionnement. Ces systèmes doivent en plus respecter des contraintes temps réel en absence et en présence de fautes (prédictifs). Pour atteindre ces deux objectifs, nous avons présenter deux heuristiques pour résoudre le problème de distribution et d'ordonnancement temps réel statique et tolérant aux fautes dans un système multiprocesseurs hétérogènes non préemptifs connectés par une liaison à bus. La première vise à présenter une solution dans une architecture logicielle composée d'un ensemble de tâches indépendantes, tandis que la deuxième heuristique présente une solution dans une architecture avec des tâches dépendantes.

Dans un système distribué temps réel et embarqué, le respect des contraintes temporelles et d'embarquabilités est réalisé par l'ordonnancement et la distribution de l'ensemble des tâches constituant le système sur l'ensemble des processeurs et des médias de communication. La solution d'un tel problème qui doit être tolérant aux fautes était prouvé NP-difficile. La solution optimale à ce type de problème ne peut être atteinte que par des algorithmes exactes dont la complexité est exponentielle ; vu qu'une telle solution est difficile voir même impossible à réaliser nous avons proposé des heuristiques qui permettent d'approcher le maximum possible leur solution à la solution optimale tout en étant de complexité polynomiale.

La solution que nous avons adopté pour tolérer les fautes permanentes d'un seul processeur est la redondance des composants logiciels. Ce choix est du au fait que nous visons des systèmes embarqués, donc l'ajout d'un ou plusieurs processeurs à l'architecture pour assurer la continuité de l'exécution du système si un processeur est en panne engendre par exemple des coûts, de la taille et de la consommation électrique supplémentaires dont la conception du système ne peut les accepter.

Par rapport aux solutions proposées dans la littérature, nos solutions permettent de mieux minimiser la longueur de distribution et d'ordonnancement temps réel en présence de fautes puisque nous considérons que les fautes des processeurs d'une part, et

d'autre part le principe des deux heuristiques proposées permet de réduire le temps entre l'apparition de la défaillance d'un processeur et la reprise ou la continuité de l'exécution de ses tâches sur d'autres processeurs.

Dans la première méthodologie appelée ***AAA-FAULT$^{t.ind}$*** , la tolérance aux fautes permanentes d'un seul processeur est assurée hors-ligne par la duplication active de chaque tâche en deux copies primaire et secondaire, dans ce cas la défaillance de la copie primaire est masquée par la copie secondaire implantée dans un processeur distinct et il n'y aura aucun temps perdu. Le mieux ici, est que l'exécution de la copie secondaire sera bloquée si aucune défaillance n'a pu apparaitre, cela influe considérablement sur la longueur globale de la distribution/ordonnancement.

La deuxième méthodologie appelée ***AAA-FAULT$^{t.dep}$*** assure la tolérance aux fautes du même problème par la duplication passive des tâches en deux copies implantées sur deux processeurs distincts. Dans ce cas la défaillance d'un processeur est détectée par une tâche appelée wotchdog qui a comme rôle la surveillance de l'exécution des tâches.

Comme perspectives a ce travail, nous proposons :

- Premièrement, les deux heuristiques doivent être testées toutes les deux sur un outil de simulation plus puissant à savoir l'outil SynDEx afin de mieux comprendre l'influence des paramètres.

- Deuxièmement, nous avons utilisé la liaison à bus pour relier les processeurs ce qui permet d'augmenter la longueur de la distribution/ordonnancement, utiliser la liaison point à point permet de réduire cette longueur.

- Enfin, dans ce mémoire nous nous sommes restreints à l'ordonnancement hors ligne non préemptif des calculs et des communications avec une seule contrainte temporelle de latence. Dans le but d'élargir le champ d'application de notre méthodologie, il faudrait introduire la possibilité de spécifier des applications devant respecter plusieurs contraintes temporelles. Pour cela il sera nécessaire d'introduire de la préemption dans les ordonnancements hors-ligne, en essayant de minimiser son effet, car son coût est loin d'être négligeable.

Bibliographie

[1] Hamoudi KALLA. *Génération automatique de distributions/ordonnancements temps réel fiables et tolérants aux fautes*, 2004. 179p. Thèse Docteur, Mathématiques, Sciences et technologies de l'information, L'INPG, 2004.

[2] A. Avizienis, J.-C. Laprie, and B. Randell. Dependability and its threats: taxonomy. In *Building the information society, 18th IFIP World Computer Congress*, pages 91–120, Toulouse, France, August 2004. Kluwer Academic Publishers.

[3] R.D. Schilichting and F.B. Schneider. Fail stop processors: An approach to designing fault tolerant computing systems. *ACM Transactions on Computer Systems*, 3(1):145–154, August 1983.

[4] J. C. Laprie, editor. *Dependability*: Basic *Concepts and Terminology*. Springer-Verlag Wien New York, 1991.

[5] Luigi Zaffalon , Pierre Breguet. Programmation concurrente et temps réel avec ADA55

[6] P. Jalote. *Fault-Tolerance in Distributed Systems*. Prentice Hall, Englewood Cliffs, New Jersey, 1994.

[7] A. Avizienis. Design of fault-tolerant computers. *In Fall Joint Computer*, pages 733–743, 1967.

[8] Nicolas Permet , Yves Sorel. Transformation de spécifications incluant du contrôle en spécification flot de données pour implantation distribuée

[9] Groupe Gotha, sous la direction de Philippe Baptiste, Emmanuel Néron, Francis Sourd. *Modèle et algorithme en ordonnancement, 2002*

[10] Pierre Lopez. Approche par contraintes des problèmes d'ordonnancement et d'affectation : *structures temporelles et mécanismes de propagation*

[11] 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.

[12] 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), 22(5) : 651–677, 2003.

[13] A. Vicard. *Formalisation et optimisation des systèmes informatiques distribués temps-réel embarqués*. PhD thesis, Université Paris XIII, July 1999.

[14] T. Yang and A. Gerasoulis. List scheduling with and without communication delays. *Parallel Computing*, 19(12) :1321–1344, 1993.

[15] A. Burns and A.Wellings. *Real-time systems and programming languages*. Addison-Wesley, 1997.

[16] A. Burns and A.Wellings. *Real-time systems and programming languages*. Addison-Wesley, 1997.

[17] Sacha Krakowiak. *Tolérance aux fautes – 1 Introduction, techniques de base.* Université Joseph Fourier Projet Sardes (INRIA et IMAG-LSR).

[18] http ://www.cnes.fr/espace pro/communiques/cp96/rapport 501/rapport 501 2.html.

[19] Wilfredo Torres-Pomales. *Software Fault Tolerance: A Tutorial*, Langley Research Center, Hampton, Virginia 23681-2199. October 2000.

[20] P. Traverse, I. Lacaze, and J. Souyris. Airbus fly-by-wire : a total approach to dependability. *In Building the information society, 18th IFIP World Computer Congress*, pages 191–212, Toulouse, France, August 2004. Kluwer Academic Publishers.

[21] Algirdas Avizienis, Jean-Claude Laprie, Brian Randell, and Carl Landwehr. *Basic concepts and Taxonomy of Dependable and Secure Computing*, IEEE transactions on dependable and secure computing, 2004

[22] J. C. Laprie, editor. Dependability : *Basic Concepts and Terminology*. Springer-Verlag Wien New York, 1991.

[23] O. Lundkvist. *Implementation of Fault-Tolerance Techniques for Real-Time Multiprocessor Scheduling*. PhD thesis, Departement of Computer Engineering, Chalmers University of Technoloy, Goteborg, December 1997.

[24] Institut national de recherche en informatique et en automatique, équipe Ostre. *Optimisation des Systèmes Temps Réel Embarqués*, rapport d'activité 2002

[25] A. ZOMAYA. *Parallel and distributed computing handbook.* McGraw-Hill, 1996.

[26] Hadrien Cambazard, Pierre-Emmanuel Hladik, Anne-Marie, Déplanche Narendra Jussien. *Une contrainte globale pour l'ordonnancabilité des tâches temps réel dur*, Cork Constraint Computation Centre Department of Computer Science, University College Cork, Ireland, 2007

[27] Parameswaran Ramanathan, Kang G. Shin. *Delivery of Time-Critical Messages Using a Multiple Copy Approach*, ACM Transactions on Computer Systems, Vol 10, No 2, May 1992, Pages 144-166

[28] Nagarajan Kandasamy, John P. Hayes et Brian T. Murray. *Dependable Communication Synthesis for Distribute Embedded Systems*, a research supported by a contract from The Delphi Corporation.

[29] Hector Garcia-Molina, Ben Kao et Daniel Barbara. *Agressive Transmissions over Redundant Paths for Time Critical Messages*.

[30] X. Qin, Z.F. Han, H. Jin, L. P. Pang, and S. L. Li. *Real-time fault-tolerant scheduling in heterogeneous distributed systems*. In Proceeding of the International Workshop on Cluster Computing-Technologies, Environments, and Applications (CC-TEA'2000), Las Vegas, USA, June 2000.

[31] Y. Oh and S. H. Son. *Scheduling real-time tasks for dependability*. Journal of Operational Research Society, 48(6) :629–639, June 1997.

[32] S. Han and K.G. Shin. *Fast restoration of real-time communication service from component failures in multi-hop networks*. In SIGCOMM Symposium, September 1997.

[33] Pascal. Chevochot and Isabelle. Puaut. *Scheduling fault-tolerant distributed hard real-time tasks independently of the replication strategie*. In The 6th International Conference on Real-Time Computing Systems and Applications (RTCSA'99), pages 356–363, HongKong, China, December 1999.

[34] Leïla Baccouche. *Un Mécanisme d'Ordonnancement Distribué de Tâches Temps Réel*, 1995. 166p. Thèse Docteur , L'INPG,Paris sud, 1995.

[35] koji Hashimoto, Tatsuhiro Tsuchiya et Tohru Kikuno. *Effective scheduling of duplicated tasks for fault tolerance in multiprocessor systems*. IEICE TRANS.INF.& SYST, Vol.E85-D, No.3 march 2002

[36] Alain Girault, Hamoudi Kalla et Yves Sorel. *An active replication scheme that tolerate failures in distributed embedded real-time systems, Processors and communication links failures.*

[37]Gérard Le Lann, Pascale Minet, David Powell. *Tolérance aux fautes et systèmes répartis concepts et mécanismes*, Rapport LAAS n° 93449, Décembre 1993.

[38] Thierry GRANDPIERRE, *Modélisation d'architectures parallèles hétérogènes pour la génération automatique d'exécutifs distribués temps réel optimises*. 2000. 249p. Thèse Docteur en science, Paris sud, 2000.

Résumé :

Les systèmes distribués temps réel embarqués sont aujourd'hui au cœur de nombreuses applications industrielles, ils sont essentiellement caractérisés par des contraintes de temps sur les actions à entreprendre qu'il faut respecter de manière plus ou moins critique. Au vu des conséquences catastrophiques (perte d'argent, de temps, ou pire de vies humaines) que pourrait entraîner une défaillance, ces systèmes doivent être sûrs de fonctionnement. La tolérance aux fautes est l'une des méthodes les plus utilisées pour assurer la sûreté de fonctionnement. Dans ce mémoire, nous avons présenté deux nouvelles heuristiques pour résoudre le problème de la génération automatique d'une distribution et d'ordonnancement temps réel qui en plus tolérante aux pannes ; Ce problème étant NP-difficile. Les deux heuristiques proposées sont basées sur la redondance des composants logiciels pour tolérer les fautes permanentes d'un seul processeur pour une architecture à liaison bus, la première se base sur la redondance active, et la deuxième sur la redondance passive. Elles offrent de bonnes performances sur des graphes d'architecture et des graphes d'algorithme générés aléatoirement.

Mots-clés : systèmes distribués temps réel embarqués, systèmes critiques, heuristiques de distribution/ordonnancement, sûreté de fonctionnement, tolérance aux fautes, fautes permanentes, redondance logicielle, architectures réparties hétérogènes.

Abstract :

The distributed embedded real time systems are today in the heart of many industrial applications, they are primarily characterized by time constraints on the actions to undertake which should be respected in a more or less critical way. Within catastrophic consequences (loss of money, time, or worse of human lives) that could involve a failure, these systems must be dependable. The fault-tolerance is one of the most methods used to ensure the dependability. In this memory, we presented two news heuristics to solve the problem of the automatic generation of a distribution and scheduling real time which in more fault-tolerant; This Problem is Np-hard. The two heuristics proposed are based on the redundancy of the software components to tolerate the permanent faults of only one processor for an architecture with multipoint communication links, the first is based on the active redundancy, and the second on the passive redundancy. They offer good performances on architecture graphs and algorithm graphs randomly generated.

Keywords: distributed embedded real time systems, critical systems, distribution and scheduling heuristic, dependability, fault-tolerance, permanent faults, software redundancy, heterogeneous distributed architectures.