UNIVERSITÉ PARIS-EST

Numéro
2009–005

THÈSE

En vue d'obtenir le grade de
DOCTEUR DE L'UNIVERSITÉ PARIS-EST

Intégration des événements non périodiques dans les systèmes temps réel – Application à la gestion des événements dans la spécification temps réel pour Java

sous la direction de Gille Roussel et Serge Midonnet

Spécialité **Informatique**
École doctorale **I**nformation, **C**ommunication, **M**odélisation et **S**imulation
Soutenue publiquement par **Damien Masson**
le **8 décembre 2008**

JURY :

Maryline Chetto	Université de Nantes	(rapporteur)
Laurent George	Université Paris-Est	
Serge Midonnet	Université Paris-Est	
Pascale Minet	INRIA Rocquencourt	(rapporteur)
Nicolas Navet	INRIA Nancy – Grand Est	
Marc Richard-Foy	AONIX	
Gilles Roussel	Université Paris-Est	

Ce document a été édité avec LaTeX.

Aux lecteurs...

Merci

Je tiens ici à remercier en premier lieu Serge Midonnet, qui m'a ouvert les portes du temps réel. C'est grâce à la confiance qu'il a su me témoigner dès mon stage de DEA que j'ai pu découvrir le monde de la recherche. Sans son enthousiasme, sa disponibilité et sa patience, ce travail n'aurait pas été possible.

Mais les portes du temps réel se seraient bien vite refermées sans Gilles Roussel qui a accepté de diriger cette thèse ; qu'il en soit donc ici remercié.

Je souhaite remercier chaleureusement Maryline Chetto et Pascale Minet d'avoir rapporté mon manuscrit. Le mois de décembre ne voit pas seulement proliférer les pères noël, c'est aussi la saison des soutenances de thèse, et donc des sollicitations de cette nature pour les chercheurs. Je les remercie par conséquent de l'intérêt particulier qu'elles ont porté à mes travaux de recherche en acceptant d'effectuer ce rapport et de participer au jury de ma thèse.

Que Laurent George, Nicolas Navet et Marc Richard-Foy se trouvent également remerciés de l'intérêt qu'ils montrent pour ces travaux en acceptant également de participer à ce jury.

Je souhaite remercier également l'ensemble des membres permanents et non permanents du laboratoire d'informatique Gaspard-Monge que j'ai eu la chance de côtoyer ces dernières années, pour savoir chaque jour transformer le quatrième étage du bâtiment Copernic en un endroit si particulier.

Je remercie particulièrement Pierre Peterlongo, Laurent Braud, Pierre Guillon, Frédéric Fauberteau et Florian *padawan* Sikora car ils ont successivement eu à me supporter dans leur bureau ; Nicolas Bedon car il m'a offert ma première expérience dans la recherche et déjà sur Java ; Marc Zipstein mon tuteur enseignant sans qui l'IGM connaîtrait une pénurie de café dont il ne se relèverait pas ; et puisqu'il est question de café Guillaume Blin avec qui ma balance commerciale dans ce domaine est certainement déficitaire. Avant de remercier également tous ceux que j'oublie, j'adresse un grand merci à Line Fonfrède pour l'ensemble des services que cette éminence grise du laboratoire rend chaque jour à tous les chercheurs qui le composent, et toujours avec le sourire.

Enfin je réserve ce dernier paragraphe à ma famille qui a forcément sa part de mérite dans l'accomplissement des dix neuf années d'étude que représente ce manuscrit.

Résumé

Les systèmes temps réel sont des systèmes informatiques composés de tâches auxquelles sont associées des contraintes temporelles, appelées échéances. Dans notre étude, nous distinguons deux familles de tâches : les tâches temps réel dur et les tâches temps réel souple. Les premières possèdent une échéance stricte, qu'elles doivent impérativement respecter. Elles sont de nature périodique, ou sporadique, et l'étude analytique de leur comportement fait l'objet d'un état de l'art conséquent. Les secondes sont de nature apériodique. Aucune hypothèse sur leur modèle d'arrivée ni sur leur nombre n'est possible. Aucune garantie ne saurait être donnée sur leur comportement dès lors que l'on ne peut écarter les situations de surcharge, où la demande de calcul peut dépasser les capacités du système.

La problématique devient alors l'étude des solutions d'ordonnancement mixte de tâches périodiques et apériodiques qui minimisent les temps de réponse des tâches apériodiques tout en garantissant les échéances des tâches périodiques. De nombreuses solutions ont été proposées ces vingt dernières années. On distingue les solutions basées sur la réservation de ressources, les serveurs de tâches, des solutions exploitant les instants d'inactivité du système, comme les algorithmes de vol de temps creux.

La spécification Java pour le temps réel (RTSJ) voit le jour dans les années 2000. Si cette norme répond à de nombreux problèmes liés à la gestion de la mémoire ou à l'ordonnancement des tâches périodiques, celui de l'ordonnancement mixte de tâches périodiques et apériodiques n'est pas abordé.

Nous proposons dans cette thèse d'apporter les modifications nécessaires aux algorithmes principaux d'ordonnancement mixte, le Polling Server (PS), le Deferrable Server (DS) et le Dynamic Approximate Slack Stealer (DASS) en vue de leur implantation avec RTSJ. Ces algorithmes ne peuvent en effet être implantés directement tels qu'ils sont décrits, car ils sont trop liés à l'ordonnanceur du système. Nous proposons des extensions aux APIs RTSJ existantes pour faciliter l'implantation de ces mécanismes modifiés, et nous fournissons les interfaces utiles à l'ajout d'autres solutions algorithmiques. Nous proposons également des modifications sur les APIs existantes de RTSJ afin de répondre aux problèmes d'intégration et d'implantation d'algorithmes d'analyse de faisabilité. Nous proposons enfin un algorithme d'estimation des temps creux, le Minimal Approximate Slack Stealer (MASS), dont l'implantation au niveau utilisateur, permet son intégration dans RTSJ.

Abstract

In computer science, real-time systems are composed of tasks. To each task is associated a timing constraint called a deadline. We distinguish two kinds of tasks : the hard ones and the soft ones. Hard tasks have hard deadlines, which must be respected to ensure the correctness of the system. So hard tasks are in essence periodic, or sporadic. Their behavior has been extensively studied. Soft tasks have soft deadlines that the system has to try to respect. When a task arrival model is unknown, i.e. when task is aperiodic, burst arrivals situation can happens, which makes the tasks timing behavior unpredictable. So aperiodic tasks can only have soft deadlines.

The studied problem in this thesis is then the joint scheduling of hard periodic tasks with soft aperiodic events, where the response times of soft tasks have to be as low as possible while the guarantee to meet their deadlines has to be given to hard tasks. A lot of solutions have been proposed these past two decades. We distinguish solutions based on resource reservation, like task servers, and solutions which take benefit from system idle times, like the slack stealer techniques.

The first version of the Real-Time Specification for Java (RTSJ) was proposed in early 2000. This specification addresses a lot of problems related to the memory management or the scheduling of periodic tasks. But if it proposes a model to write aperiodic events, advanced mechanisms for the integration of such events to handle the above-mentioned problem are not discussed.

We propose modifications to the main advanced mixed scheduling mechanisms like the *Polling Server* (PS), the *Deferrable Server* (DS) or the *Dynamic Approximate Slack Stealer* (DASS) in order to make their implementation possible with the RTSJ. Indeed, these algorithms are deeply connected to the system scheduler, and have to be adapted in order to be implemented in a user-land level. We propose extensions to current RTSJ APIs in order to integrate the modified algorithms and to allow the addition of other algorithms in a unified framework. We also propose some modifications to the RTSJ APIs in order to solve some problems we encountered during the integration of modified algorithms, especially in the field of the feasibility analysis algorithms integration in the specification. Finally, we propose the *Minimal Approximate Slack Stealer* algorithm (MASS), which is independent of the scheduler implementation and has a lower overhead than DASS.

Notes de lecture

Glossaires Des glossaires se trouvent à la fin de ce manuscrit. Ils contiennent les acronymes utilisés ainsi que les définitions jugées utiles. Les mots présents dans un glossaire sont présentés en *italique*. Un *astérisque** indique leur première occurrence.

Illustrations Toutes les illustrations présentées dans ce manuscrit sont les œuvres de son auteur. Les figures représentant des séquences d'ordonnancement utilisent toutes le même formalisme : les flèches verticales orientées vers le haut représentent les instants d'activation périodiques, celles qui pointent vers le bas marquent les échéances temporelles et une section hachurée correspond à l'occupation du processeur. Dans le cas de systèmes à échéances sur requêtes, une seule flèche verticale pointant vers le haut marque l'échéance et l'activation périodique.

Convention Pour simplifier l'écriture de certaines équations, plus une priorité est forte, plus sa valeur numérique est petite. En revanche, dans les phrases en français, nous parlons d'une priorité supérieure pour désigner une priorité plus forte.

Annexes Nous présentons dans les annexes des exemples d'utilisation de paradigmes de programmation, le code Java implantant les algorithmes que nous proposons ainsi qu'un logiciel de simulation que nous avons développé.

Table des matières

Remerciements v

Résumé vii

Abstract ix

Notes de lecture xi

Table des matières xiii

Table des figures xix

Introduction 21

I Temps réel 27

1 Modèles de tâche 31

1.1 Nature des tâches 31

1.2 Période et coût 33

1.3 Échéance et première activation 33

1.4 Gigue et temps de blocage 34

2 Ordonnancement 37

2.1 Ordonnancement hors ligne ou en ligne 38

2.2 Ordonnancement préemptif ou non préemptif 38

2.3 Ordonnancement à priorités statiques 39

2.3.1 Priorités basées sur la fréquence 39

2.3.2 Priorités basées sur l'échéance 40

2.4 Ordonnancement à priorités dynamiques 40

2.4.1 Priorités basées sur l'échéance 41

2.4.2 Priorités basées sur la laxité 41

3 Analyse de faisabilité 43
3.1 Étude de la charge . . . 43
3.2 Étude de la demande processeur . . . 44
3.3 Calcul du pire temps de réponse en priorité fixe . . . 45
3.3.1 Calcul de la période occupée de niveau i . . . 46
3.3.2 Calcul du temps de réponse d'une instance . . . 47
3.3.3 Test de faisabilité basé sur les pires temps de réponse . . . 47
3.4 Partage de ressources . . . 48

4 Gestion des tâches non périodiques 49
4.1 Approche sporadique . . . 50
4.2 Ordonnancement en tâche de fond . . . 50
4.3 Serveur à scrutation . . . 51
4.3.1 Contrôle d'admission . . . 51
4.3.2 Gestion de ressources partagées . . . 53
4.4 Serveur ajournable . . . 54
4.4.1 Contrôle d'admission . . . 55
4.4.2 Gestion de ressources partagées . . . 56
4.5 Vol de temps creux statique . . . 56
4.5.1 Calculs hors ligne . . . 58
4.5.2 Partie dynamique . . . 58
4.5.3 Implantabilité et optimalité . . . 59
4.6 Vol de temps creux dynamique . . . 61
4.6.1 Notations . . . 61
4.6.2 Calcul des temps creux à chaque niveau de priorité . . . 62
4.6.3 Algorithme de vol de temps creux dynamique . . . 63
4.6.4 Influence des tâches sporadiques . . . 64
4.6.5 Temps d'exécutions stochastiques . . . 64
4.6.6 Gigues d'activation . . . 65
4.6.7 Gestion de ressources partagées . . . 65
4.7 Vol de temps creux approché . . . 66
4.7.1 Approximation statique des temps creux . . . 66
4.7.2 Approximation dynamique des temps creux . . . 67

II Java pour le temps réel 69

5 Java n'est pas un langage pour le temps réel 73
5.1 La gestion de la mémoire . . . 73
5.2 Non respect des priorités . . . 75
5.3 Inversion de priorités . . . 75

5.4 Pas de contrôle de la charge . . . 76
5.5 Les optimisations dynamiques dans Java . . . 77

6 La spécification Java pour le temps réel (RTSJ) 79
6.1 Les principes directeurs . . . 80
6.2 Gestion des Priorités et entités ordonnançables . . . 81
6.3 Gestion de la mémoire . . . 82
6.4 Contrôle de la charge . . . 84
6.4.1 Écrire une tâche périodique . . . 84
6.4.2 Écrire une tâche sporadique . . . 86
6.4.3 Assurer les contraintes temporelles . . . 87
6.4.4 Analyse de faisabilité . . . 88
6.5 Interruption et transfert de contrôle asynchrone . . . 88

III Gestionnaire d'événements pour RTSJ 91

7 Serveurs de tâches 95
7.1 Contraintes et limitations . . . 95
7.2 Gestion de la capacité du serveur . . . 96
7.3 Ordonnancement des tâches au niveau utilisateur . . . 97
7.4 Politiques de gestion de la file . . . 97
7.5 Implantation . . . 99
7.5.1 Événements et file d'attente . . . 99
7.5.2 Serveur à scrutation . . . 100
7.5.3 Serveur ajournable . . . 101
7.6 Synchronisations . . . 101
7.7 Surcoût et intégration à l'analyse de faisabilité . . . 102

8 Voleurs de temps creux 103
8.1 Utilisation de DASS au niveau utilisateur . . . 103
8.2 Un algorithme de vol de temps creux minimal . . . 105
8.3 Données à collecter et à calculer . . . 105
8.4 Initialisation des données . . . 106
8.5 Mise à jour des données . . . 106
8.5.1 Maintien d'une borne minimale pour les $\bar{\mathbf{w}}_\mathbf{i}$ . . . 107
8.5.2 Maintien d'une borne maximale pour les $\bar{\mathbf{c}}_\mathbf{i}$ . . . 108
8.5.3 Bilan des opérations à effectuer . . . 108
8.6 Approximation de l'interférence . . . 109
8.6.1 Valeurs possibles de $\mathbf{Nba_i^j(t)}$ . . . 109
8.6.2 Choix de la valeur de $\mathbf{Nba_i^j(t)}$ . . . 110
8.7 Exemple comparatif de *DASS* et *MASS* . . . 112

8.8 Implantation 114

9 Propositions d'ajouts à la spécification RTSJ 117
9.1 Obtenir le temps CPU au niveau utilisateur 117
9.2 Ajout de code en début et fin d'instance 120
9.3 Analyse de faisabilité 120

10 Évaluation des performances 123
10.1 Méthodes de génération et de simulation 123
10.2 Service en tâche de fond 124
10.2.1 BS implicite 124
10.2.2 BS explicite 127
10.2.3 Comparaison de BS et MBS 128
10.3 Serveur à scrutation 129
10.3.1 PS non modifié 130
10.3.2 PS modifié 132
10.4 Serveur ajournable 132
10.4.1 DS non modifié 133
10.4.2 DS modifié 134
10.5 Vol de temps creux 136
10.5.1 Valeurs exactes des temps creux 138
10.5.2 Temps creux calculés avec DASS 140
10.5.3 Temps creux calculés avec MASS 142
10.6 Surcoûts des politiques DASS et MASS 144
10.7 Conclusions 145

Conclusions 149

Bibliographie 153

Acronymes 159

Vocabulaire 165

Annexes clxix

A Exemples clxix
A.1 Interruptions Asynchrones clxix
A.1.1 Exemple en utilisant `Timed` clxix
A.1.2 Exemple en utilisant `interrupt()` clxx
A.1.3 Exemple en utilisant `doInterruptible() / fire()` clxxi

B RTSS **clxxiii**

B.1 Simulation événementielle clxxiii

B.2 Architecture du programme clxxiv

C Code de notre gestionnaire d'événements pour RTSJ **clxxv**

C.1 AbstractSlackCompatibleSchedulable clxxvi

C.2 AbstractUserLandEventManager clxxvii

C.3 AbstractUserLandSlackStealer clxxx

C.4 AbstractUserLandTaskServer clxxxiii

C.5 DASSUserLandSlackStealer clxxxiii

C.6 ManageableAsyncEventHandler clxxxvii

C.7 ManageableAsyncEvent clxxxix

C.8 MASSUserLandSlackStealer cxc

C.9 UserLandDeferrableTaskServer cxcv

C.10 UserLandPollingTaskServer cxcvii

Table des figures

1.1 Utilité d'une tâche en fonction de sa date de terminaison 32
1.2 Modèle de tâche périodique 1 33
1.3 Modèle de tâche périodique 2 34
1.4 Modèle de tâche périodique 3 35

2.1 Ordonnancement *RM* du système de tâches du tableau 2.1 40
2.2 Ordonnancement *DM* du système de tâches du tableau 2.1 40
2.3 Ordonnancement *EDF* du système de tâches du tableau 2.1 41
2.4 Ordonnancement *LLF* du système de tâches du tableau 2.1 42

4.1 Système faisable avec un *PS*, non faisable avec un *DS* 54
4.2 Comparaisons BS, PS et DS 57
4.3 Non optimalité du vol de temps creux 60
4.4 Approximation statique de $S_i^C(\delta)$ 67

6.1 Écrire une tâche périodique avec un `RealtimeThread` 84
6.2 Écrire une tâche périodique avec un `AsyncEventHandler` 85

7.1 Intérêt de la duplication BS 98

8.1 Nombre d'activations - notations 110
8.2 Nombre d'activations - exemple 111
8.3 Calcul de la laxité de niveau 3 avec DASS et avec MASS 112
8.4 Méthode run() d'un Schedulable compatible avec un voleur de temps creux 114
8.5 Méthodes computeBeforePeriodic() et computeAfterPeriodic() 115

9.1 Obtention du temps CPU au niveau utilisateur 118

10.1 Influence du nombre de tâches composant le trafic périodique sur un *BS* . 125
10.2 Influence de la politique de gestion de la file sur un *BS* 126
10.3 Comparaison de *BS* et *MBS* 127
10.4 *BS-FIFO* et *MBS-LCF* : charge périodique de 70% 128
10.5 Période et capacité du *PS* en fonction du nombre de tâches périodiques . . 130
10.6 *PS* avec duplication : charges périodiques de 30 et 70% 131

10.7 *PS* : charges périodiques de 30 et 90% 131
10.8 Comparaison de *PS* et *MPS*, charge périodique de 70% 132
10.9 Période et capacité du *DS* en fonction du nombre de tâches périodiques . 133
10.10 *DS* avec duplication : Résultats pour des charges périodiques de 30 et 70% 133
10.11 *DS* : Résultats des charges périodiques de 30 et 90% 134
10.12 Comparaison de *PS* et *DS* 135
10.13 Comparaison de *DS* et *MDS* 136
10.14 Comparaison de *MPS* et *MDS* 137
10.15 Temps de réponse moyens pour *ESS* 138
10.16 Comparaison *ESS* et *OESS* 139
10.17 Temps de réponse moyens pour *DASS* 140
10.18 Comparaison *DASS* et *ODASS* 141
10.19 Temps de réponse moyens pour *MASS* 142
10.20 Comparaison *MASS* et *OMASS* 143
10.21 Surcoûts des ajouts avant et après chaque instance pour *DASS* et *MASS* . 144
10.22 Temps de réponse moyens pour chaque algorithme associé à *LCF&BS* . . 146
10.23 Charge périodique de 70% composée de 4 ou 40 tâches 147

Introduction

Systèmes temps réel étudiés

Un système informatique peut être défini comme un ensemble de tâches effectuant chacune un calcul. Le système est correct lorsque tous les résultats fournis par les tâches sont corrects.

En temps réel, les tâches possèdent en plus une échéance, une date qui peut être fixe (échéance absolue) ou relative à leur date d'activation. Le résultat de la tâche doit être fourni au plus tard avant cette date.

On distingue plusieurs niveaux de contraintes temps réel. Le respect d'une échéance stricte, par exemple, sera aussi important que la correction du calcul. L'échéance peut également être souple, dans ce cas l'importance associée au résultat diminuera une fois l'échéance dépassée.

Les systèmes temps réel durs sont composés uniquement de tâches possédant des échéances strictes. La contrainte temps réel est alors très forte, et pour la garantir, il est nécessaire de maîtriser a priori un grand nombre de caractéristiques des tâches. Il est nécessaire de connaître par exemple leur temps d'exécution pire cas lorsqu'elles sont seules à occuper le processeur ainsi que leur modèle d'activation.

Ce besoin de déterminisme a longtemps limité l'étude des systèmes temps réel aux systèmes de tâches périodiques indépendantes les unes des autres et ne partageant pas d'autres ressources que le processeur. Ce modèle a été intensivement étudié et de nombreuses solutions d'ordonnancement proposées, depuis la méthode manuelle par tables temporelles spécifiant l'ordre d'exécution des tâches sur le processeur, aux méthodes automatisées par l'attribution de priorités aux tâches. Ces priorités peuvent être assignées de façon arbitraire ou en suivant une règle. Cette règle peut elle-même dépendre de paramètres statiques ou dynamiques. L'analyse de ces systèmes a ensuite été étendue en relâchant progressivement certaines contraintes, comme l'indépendance des tâches.

Cependant, les systèmes informatiques ne peuvent se limiter à la modélisation d'événements périodiques, le monde réel qu'ils cherchent à modéliser étant lui-même composé essentiellement d'événements apériodiques. Une contrainte temps réel dur ne peut pas être donnée à une tâche apériodique. En effet, si l'on ne peut pas borner le nombre d'occurrences de cette tâche ni la fréquence de son arrivée, la demande processeur peut dépasser la capacité de traitement du système, car un pic d'activations simultanées d'un grand

nombre de tâche peut survenir. On ne peut alors pas garantir le respect des échéances.

Si la tâche a réellement besoin de se voir attacher une contrainte temps réel dur, il faudra alors au moins borner sa fréquence d'activation. On peut ensuite analyser le système en considérant le pire cas : la tâche survient à sa fréquence maximale. La tâche n'est plus *apériodique*, elle est *sporadique**.

Les tâches purement apériodiques ne pourront avoir que des contraintes temps réel souples. L'utilité des résultats de ces tâches pour le système va dépendre de la durée qui aura été nécessaire pour leur obtention. Plus le *temps de réponse** de la tâche, c'est-à-dire le temps entre son activation et la fin de son exécution est grand, moins le résultat fourni sera utile. Il peut également exister une échéance qui, une fois atteinte, rend l'utilité de la tâche nulle.

L'ordonnancement d'un système mixte de tâches périodiques temps réel dur et apériodiques temps réel souple soulève par conséquent la problématique de la minimisation des *temps de réponse* pour les tâches apériodiques, tout en garantissant le respect des échéances pour les tâches périodiques.

Plusieurs approches sont proposées pour garantir que la prise en charge des événements apériodiques ne va pas remettre en question les garanties données aux tâches périodiques. La plus simple consiste à attribuer aux tâches apériodiques une plage de priorités inférieures à celles utilisées pour les autres tâches. Si cette approche est facile à implanter, elle ne répond qu'à la moitié du problème. La minimisation des *temps de réponse* n'est pas abordée. Une famille de techniques repose sur la réservation de ressources pour les tâches apériodiques : les serveurs de tâches. Un serveur est une tâche que l'on va pouvoir intégrer dans l'étude du système comme une tâche périodique possédant une contrainte temps réel dur. De nombreux serveurs ont été proposés. Certains nécessitent de modifier l'analyse du système, d'autre s'y intègrent exactement comme une tâche périodique ordinaire. Enfin, il existe une dernière approche, basée sur l'étude dynamique ou statique de la quantité d'unités de temps inutilisées par le système entre un instant donné et la prochaine échéance stricte à respecter. Ces temps peuvent alors être agrégés pour être affectés instantanément au traitement des tâches apériodiques. On appelle cette technique le vol de temps creux.

Un langage de haut niveau pour le temps réel

La programmation de systèmes temps réel est souvent assimilée à de la programmation bas niveau. Beaucoup pensent tout de suite à la programmation de pilotes devant interagir fortement avec du matériel, et à de la programmation assembleur, ou en langage C dans le meilleur des cas. Notons que les problématiques temps réel sont présentes dans un très grand nombre de domaines. Cela concerne effectivement des applications embarquées, mais également toutes sortes d'applications, du contrôle des procédés sur une chaine de montage industrielle, aux applications multimédia en passant par l'informatique de finance

et l'avionique. En réalité, tous les domaines de l'informatique peuvent être concernés par une contrainte temporelle, une échéance plus ou moins importante à respecter.

Aussi est-ce naturellement que la communauté temps réel s'est rapidement intéressée au langage Java, qui a su convaincre un grand nombre de développeurs. Ses atouts sont une grande portabilité, des paradigmes de programmation qui permettent une bonne réutilisabilité du code (et donc un gain de temps conséquent), une double vérification du code lors de la compilation et lors du chargement des classes et une gestion automatique de la mémoire par une machine virtuelle qui permet de dégager le programmeur de problèmes trop bas niveau, et ainsi d'éviter de nombreux bogues liés à la mémoire qui peuvent s'avérer très difficiles à détecter.

Mais ces avantages sont aussi autant d'obstacles à l'utilisation de Java pour écrire une application temps réel. La grande portabilité rend difficile le calcul des pires coûts d'exécution, le nettoyage automatique de la mémoire enlève du déterminisme à l'exécution des programmes, le modèle d'ordonnancement des processus légers de Java n'est pas spécifié, et dépend donc de la plateforme cible.

C'est pour cela qu'un groupe d'experts a été constitué pour établir une norme, la spécification Java pour le temps réel –*Real-Time Specification for Java (RTSJ)**–, qui spécifie d'une part les caractéristiques minimales pour une machine virtuelle temps réel, et fournit d'autre part un ensemble de classes et d'interfaces permettant le développement de programmes temps réel.

L'ordonnancement mixte avec Java

Les experts à l'origine de *RTSJ* se sont principalement attachés à résoudre les problèmes liés à la gestion de la mémoire, en augmentant le modèle de mémoire de Java avec de nouvelles zones particulières non concernées par le collecteur automatique. Ceci permet d'assurer qu'une tâche qui n'utilise que de telles zones de mémoire ne peut subir d'interférences de la part du collecteur. L'autre apport essentiel concerne le modèle d'ordonnancement. La spécification définit une nouvelle plage de priorités, destinée aux tâches temps réel. L'ordonnancement des processus légers de Java n'est pas modifié, mais les tâches temps réel peuvent être programmées à l'aide d'un nouveau type d'objet ordonnançable, le processus léger temps réel, dont l'accès au processeur est déterminé par un algorithme d'ordonnancement temps réel.

Cette spécification, encore jeune, ne propose aucune des solutions d'ordonnancement mixte que nous avons brièvement présentées ci-dessus. Le travail que nous présentons dans ce manuscrit résulte de l'étude de l'implantabilité de ces mécanismes en Java, dans le contexte de cette spécification.

Une contrainte forte a orienté cette étude : nous souhaitons que l'implantation des mécanismes avancés d'ordonnancement mixte reste portable, indépendante du système sous-jacent, dans la mesure où les spécifications minimales de *RTSJ* pour une machine

virtuelle temps réel sont respectées. Ceci nous a conduit à proposer des modifications aux algorithmes de la littérature, afin qu'ils ne fonctionnent non plus comme des ordonnanceurs, fortement intégrés dans le *système d'exploitation (OS)**, mais comme des gestionnaires s'exécutant au niveau utilisateur, au dessus d'un algorithme d'ordonnancement à priorités fixes préemptif (le seul ordonnanceur imposé par la spécification).

Les contributions de ce travail

Nous proposons dans cette thèse :

- une modification des algorithmes de services *Polling Server (PS)** et *Deferrable Server (DS)** ;
- une modification de l'algorithme d'utilisation des temps creux ;
- un algorithme d'évaluation dynamique d'une borne sur la quantité de temps creux, caractérisé par une faible complexité algorithmique et d'implantation ;
- une validation de ces politiques modifiées et de l'algorithme d'évaluation des temps creux par le biais de nombreuses simulations ;
- un ensemble de classes et d'interfaces pour implanter ces algorithmes et faciliter l'implantation d'autres politiques avec *RTSJ* ;
- des propositions de modifications pour *RTSJ*, résultant des problèmes rencontrés durant ce travail, afin d'améliorer le modèle d'ordonnancement et d'analyse de faisabilité de *RTSJ*.

Ce travail a également donné lieu au développement d'un simulateur événementiel de systèmes temps réel que nous distribuons sous la licence *GNU Public Licence (GPL)**.

Plan du manuscrit

Ce manuscrit est structuré en trois parties. La première présente un état de l'art général sur le problème de l'ordonnancement mixte de tâches apériodiques et périodiques dans les systèmes temps réel. La deuxième partie présente les problèmes liés à l'utilisation de Java pour écrire une application temps réel, les approches existantes pour résoudre ces problèmes, et plus particulièrement *RTSJ*. Enfin, la troisième partie présente les solutions que nous proposons pour l'adaptation et l'implantation des mécanismes décrit dans la première partie dans le contexte de *RTSJ*.

Première partie : Temps réel Un premier chapitre identifie et formalise les différents modèles de tâches que nous serons amenés à considérer. Le deuxième chapitre est consacré à la problématique de l'ordonnancement dans les systèmes temps réel. Le troisième chapitre présente les enjeux de l'analyse de faisabilité dans les systèmes temps réel. Enfin le quatrième chapitre présente les solutions existantes pour répondre au problème de l'ordonnancement mixte (tâches périodiques temps réel dur et tâches apériodiques temps

réel souple) : l'approche sporadique, le traitement en tâche de fond, les serveurs de tâches et les voleurs de temps creux.

Deuxième partie : Java pour le temps réel Cette partie présente les difficultés posées par l'utilisation de Java pour programmer des applications temps réel et les différentes approches pour les résoudre. Un chapitre est consacré particulièrement à l'approche adoptée par *RTSJ*.

Troisième partie : Gestionnaire d'événements pour *RTSJ* Les contributions de cette thèses sont développées dans cette partie. Un chapitre est consacré à l'implantation des serveurs de tâches, un autre à celle du vol de temps creux. Le lecteur y trouvera également une présentation de résultats de simulation évaluant les différents algorithmes.

Première partie

Temps réel

Définition 1 (Temps réel - [Sta88])**.** *In real-time computing the correctness of the system depends not only on the logical result of the computation but also on the time at which the results are produced.*

En programmation temps réel, la correction du système ne dépend pas seulement des résultats logiques des traitements, mais dépend en plus de la date à laquelle ces résultats sont produits.

Tout le monde a déjà entendu une fois le terme *temps réel*, mais sans forcément savoir de quoi il retourne. Contrairement à une idée répandue, cela n'est nullement un critère de rapidité. Il s'agit plutôt d'avoir des garanties sur la durée maximale de l'exécution des tâches composant un programme.

Le champ d'application des systèmes temps réel est très vaste, et couvre de nombreux secteurs d'activité. On les retrouve dans l'industrie, dans les salles de marché, dans le transport, ou encore dans le secteur des loisirs.

Ces systèmes temps réel se différencient des autres systèmes informatiques par la prise en compte de contraintes temporelles dont le respect est aussi important que l'exactitude du résultat.

On distingue le temps réel dur et le temps réel souple suivant l'importance accordée aux contraintes temporelles. Le temps réel strict ne tolère aucun dépassement de ces contraintes, ce qui est souvent le cas lorsque de tels dépassements peuvent conduire à des situations critiques, voire catastrophiques : le pilote automatique d'un avion, le système de surveillance d'une centrale nucléaire, etc. A l'inverse le temps réel souple s'accommode de dépassements des contraintes temporelles dans certaines limites au-delà desquelles le système devient inutilisable : vidéoconférence, jeux en réseau, etc. Dans la suite, si le contexte ne précise explicitement le contraire, l'expression *temps réel* désignera le temps réel dur.

Un système peut donc être qualifié de temps réel dès lors que toutes les tâches qui le composent sont assurées de terminer avant une échéance qui leur est assignée. Si ces échéances peuvent être exprimées en nanosecondes, elles peuvent également l'être en heures ou même en années. Ce qui est important, ce n'est pas la durée de l'exécution mais sa *prédictibilité*.

Chapitre 1

Modèles de tâche

Ce chapitre a pour but d'identifier des modèles de tâche, des plus simples aux plus complets. Ces modèles nous serviront dans la suite à illustrer les différentes politiques d'ordonnancement et les différents algorithmes d'analyse de faisabilité du système.

La terminologie, les notations et les modèles décrits dans ce chapitre sont inspirés de [LL73] qui est la publication de référence en matière d'ordonnancement de systèmes temps réel.

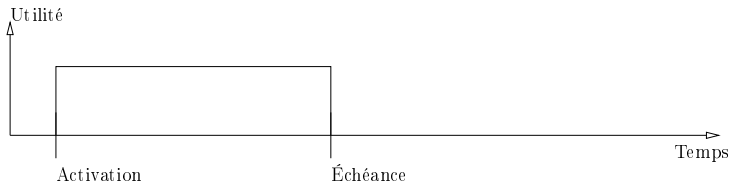
Ce chapitre présente une façon de voir la modélisation d'un système temps réel, le lecteur intéressé pourra se référer aux documents [Mok83, Bur91] pour approfondir le sujet.

1.1 Nature des tâches

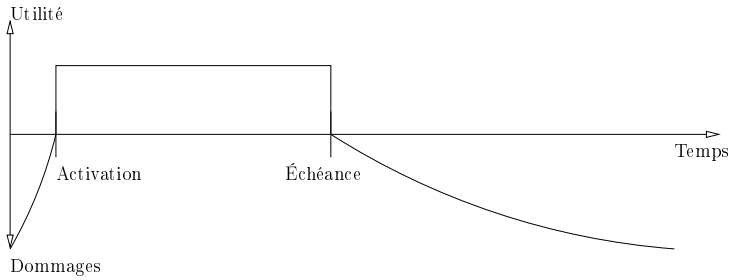
Un système informatique est composé de tâches à exécuter. En informatique temps réel, chaque tâche se voit associer une échéance temporelle. Il existe plusieurs types d'échéances. On distingue notamment les échéances strictes, les échéances critiques et les échéances souples.

La différence entre ces contraintes se mesure par l'utilité de la tâche en fonction du temps de terminaison. La figure 1.1 page suivante présente les fonctions d'utilité associées respectivement aux tâches temps réel souple, critique et dur. Pour une tâche à échéance stricte, le respect de l'échéance est aussi important pour sa correction que la valeur qu'elle calcule. Si l'échéance est ratée, le système commet une erreur : il est incorrect, ou défaillant. Une échéance critique peut être dépassée, mais plus le retard de livraison du résultat est important, plus les performances du système sont dégradées. En revanche, ne pas respecter une échéance souple ne dégrade pas le système, mais plus le retard de livraison du résultat est important, moins il aura d'utilité pour le système.

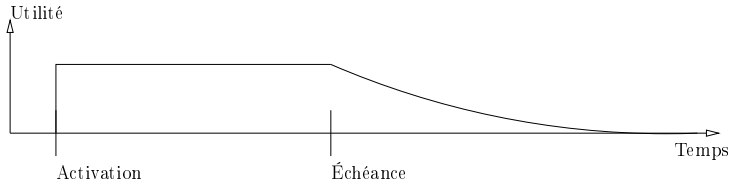
Les systèmes peuvent être composés uniquement de tâches de même type d'échéances, ou encore être mixtes.



(a) Échéance stricte



(b) Échéance critique



(c) Échéance souple

FIG. 1.1 – Utilité d'une tâche en fonction de sa date de terminaison

Les tâches sont par ailleurs caractérisées par leur modèle d'arrivée. Étant donné l'importance revêtue par le respect d'une échéance stricte, il est capital de pouvoir le garantir. Pour cela, il faut connaître le modèle d'arrivée de telles tâches. Le modèle étudié est un modèle d'arrivée périodique. Il correspond historiquement aux tâches de contrôle, qui vont périodiquement aller vérifier la valeur d'une donnée.

Une tâche de nature apériodique peut arriver de manière complètement aléatoire. Si l'on modélise ses activations par une loi de Poisson il est impossible de garantir les échéances de toutes les tâches impliquées, notamment lors du pic d'activation.

Si cette tâche est très importante pour le système, il faudra trouver un moyen de borner le temps minimal entre deux de ses activations, et considérer le scénario le moins favorable où la tâche survient systématiquement lors de sa date d'activation au plus tôt. On parle alors de tâche sporadique.

Dans la suite, nous considérerons donc toujours qu'une tâche sporadique ou périodique possède une échéance stricte, et qu'une tâche apériodique possède une échéance souple ou critique. Nous étudions alors les solutions permettant de minimiser les *temps de réponse*, et donc maximiser l'utilité pour le système des tâches apériodiques, sans compromettre les garanties données sur le respect des échéances des tâches sporadiques et périodiques.

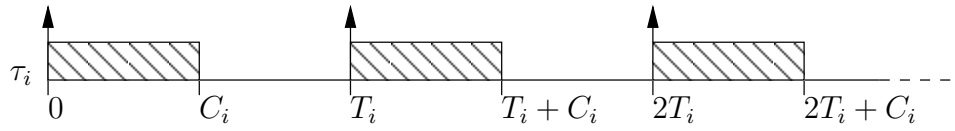


FIG. 1.2 – Modèle de tâche périodique 1

1.2 Période et coût

Un système informatique temps réel, Φ, peut être modélisé par un ensemble de n entités appelées *tâches*, $\Pi = \{\tau_1, ..., \tau_n\}$. Une tâche est caractérisée par plusieurs paramètres, qui peuvent être connus a priori ou non. Dans le modèle le plus simple, toutes les tâches sont périodiques de période T_i, c'est à dire composées d'une infinité de *requêtes*, ou *instances*. La k^e requête de la i^e tâche est notée $j_{i,k}$. La première est activée, c'est à dire prête pour l'exécution, au temps $t = 0$. On parle de modèle d'activation synchrone car toutes les tâches sont activées pour la première fois à la même date. La requête $j_{i,k}$ est activée au temps $t = (k-1)T_i$. Toutes les requêtes d'une même tâche ont le même coût, C_i. Le coût d'une requête est défini par le temps de calcul qui serait nécessaire au processeur pour la compléter si elle était seule à s'exécuter. Un ordonnancement correct doit garantir que chaque requête $j_{i,k}$ sera terminée au plus tard lors de l'activation de la requête $j_{i,k+1}$. Lorsque *l'échéance* des requêtes est ainsi fixée à la période de la tâche, on parle d'échéance sur requête.

Modèle 1. *Nous appelons* Modèle de tâche périodique 1 *le modèle dans lequel une tâche est caractérisée par le couple $\tau_i = (T_i, C_i)$, où T_i est la période d'activation des requêtes de τ_i et C_i leur coût. L'échéance est dite sur requête, c'est-à-dire que T_i est aussi le temps imparti pour compléter chaque instance de τ_i.*

La figure 1.2 illustre les notations utilisées pour ce modèle.

1.3 Échéance et première activation

Pour généraliser ce modèle à un plus grand nombre de systèmes, deux paramètres doivent être ajoutés : la date d'activation de la première requête, r_i, qui peut très bien être différente pour chaque tâche, et l'échéance, D_i, qui peut être différente de la période. Nous nous limitons toutefois à l'étude des tâches dont l'échéance est inférieure ou égale à la période. Cela implique que l'instance $j_{i,k}$ est activée à l'instant $t_1 = r_i + (k-1)T_i$ et doit se terminer au plus tard à l'instant $t_2 = t_1 + D_i$. On dit que D_i est l'échéance *relative* de la tâche, car elle indique une date de terminaison relative à la date d'activation. L'instant t_2 est appelé échéance *absolue* de la requête $j_{i,k}$ et est noté $d_{i,k}$.

De plus, nous relâchons l'hypothèse du modèle 1 selon laquelle toutes les requêtes d'une même tâche ont le même coût. La même séquence d'instructions peut en effet

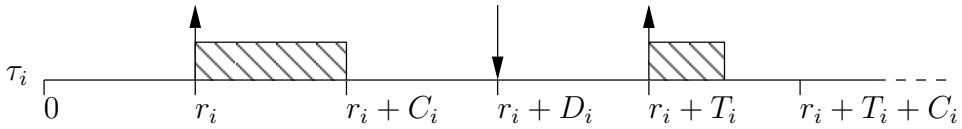


FIG. 1.3 – Modèle de tâche périodique 2

prendre un temps variable pour s'exécuter en raison de l'état de la mémoire, du temps d'accès aux disques, ou même sur les architectures modernes de la fréquence variable des processeurs, ou d'optimisations diverses (caches d'instructions, de données...). Aussi C_i désigne-t-il dans ce modèle le *pire temps d'exécution* ($WCET$) de la tâche, c'est-à-dire le temps d'exécution de l'une de ses requêtes si elle est seule à s'exécuter, dans le scénario le moins favorable.

Modèle 2. *Nous appelons* Modèle de tâche périodique 2 *le modèle dans lequel une tâche est caractérisée par le quadruplet* $\tau_i = (r_i, T_i, C_i, D_i)$*, où* r_i *est la date d'activation de la première requête de* τ_i*,* T_i *la période d'activation des suivantes,* C_i *leur pire temps d'exécution et* D_i *leur échéance.*

La figure 1.3 illustre les notations utilisées pour ce modèle.

1.4 Gigue et temps de blocage

Pour s'approcher davantage d'un système informatique réel, il faut encore considérer que les tâches peuvent subir des gigues d'activation et partager des ressources.

Un événement de nature purement périodique est assez rare dans le monde réel. Aussi les tâches périodiques qui répondent à des événements extérieurs périodiques peuvent elles subir de légères variations sur leur dated'activation. Ainsi, l'événement pourra survenir un peu plus tôt que prévu, ou un peu plus tard. Le décalage induit entre l'arrivée (périodique) de la requête et son activation (la tâche est prête) est appelé *gigue d'activation*. Il est nécessaire de borner cette gigue pouvant être subie par la tâche, et sa valeur maximale est notée J_i [ABR+93].

Si les tâches accèdent à des ressources communes, chaque requête peut alors subir un blocage correspondant à l'attente de l'accès à une ressource déjà occupée par une autre tâche. Des algorithmes d'héritage de priorité permettent de borner les temps de blocage. Le pire temps de blocage pouvant être subi par la tâche τ_i (même si elle-même n'accède à aucune ressource partagée) est noté B_i.

La requête $j_{i,k}$ est activée dans ces conditions à un instant $t_1 \in [r_i + (k-1)T_i - J_i, r_i + (k-1)T_i + J_i]$ et doit terminer son exécution avant l'instant $t_2 = t_1 + D_i$.

Modèle 3. *Nous appelons* Modèle de tâche périodique 3 *le modèle dans lequel une tâche est caractérisée par le sextuplet* $\tau_i = (r_i, J_i, T_i, C_i, B_i, D_i)$*, où* r_i *est la date d'arrivée de*

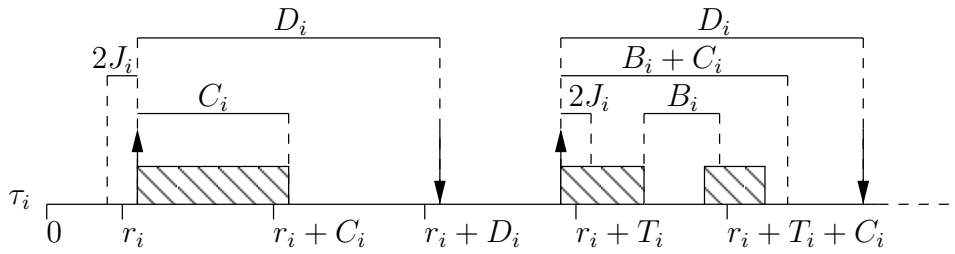


FIG. 1.4 – Modèle de tâche périodique 3

la première requête de τ_i, J_i la gigue d'activation maximale d'une de ses requêtes, T_i leur période d'activation, C_i leur pire temps d'exécution, B_i leur temps de blocage maximal par l'accès à des ressources partagées et D_i leur échéance.

La figure 1.4 illustre les notations utilisées pour ce modèle.

Notes Nous avons ici choisi de formaliser trois modèles différents en partant du modèle 1 très restreint pour relâcher progressivement une partie des contraintes et aboutir au modèle 3. Cette séparation, qui nous a permis de définir les différentes caractéristiques d'une tâche temps réel périodique, est arbitraire et nous servira par la suite à préciser le domaine d'application des résultats que nous présenterons.

Chapitre 2

Ordonnancement

Définition 1 (Ordonnancement)**.** *Organisation, agencement méthodique des différents éléments d'un ensemble, des diverses phases d'une fabrication.*

Dans un système informatique classique, dit à « temps partagé », l'ordonnanceur sert à répartir les ressources processeur entre les différents processus. L'objectif est double : rendre transparent pour l'utilisateur la gestion du ou des processeurs afin de donner l'illusion d'un système capable d'effectuer plusieurs tâches simultanément et empêcher les situations de *famine*, où une seule tâche occupe le processeur, empêchant les autres de s'exécuter.

Les tâches occupent le processeur à tour de rôle pour un temps borné : l'algorithme d'ordonnancement est appelé *Round Robin (RR)**. L'ordonnancement est dit *préemptif* car une tâche peut être suspendue avant d'avoir terminé son exécution. Dans un tel contexte, il n'est pas nécessaire de connaître a priori une borne sur le temps d'exécution des tâches. De même ces dernières peuvent avoir besoin d'accéder à d'autres ressources, mais ces besoins n'interviennent pas lors des décisions d'ordonnancement. L'ordonnanceur n'a aucune raison d'anticiper le comportement des tâches, dès lors que l'on prend garde à ne pas allouer toutes les ressources à un seul processus.

Il en va autrement dans un système temps réel : les tâches et les ressources dont elles ont besoin doivent être connues à l'avance. L'ordonnancement doit être *faisable*, c'est-à-dire que toutes les contraintes temporelles doivent être respectées, en tenant compte des besoins en ressources. L'objectif est d'obtenir un ordonnancement prévisible, afin de pouvoir déterminer sa faisabilité.

Plusieurs caractéristiques permettent de classer les algorithmes d'ordonnancement en familles :

- l'ordonnancement peut être oisif ou non oisif, c'est-à-dire que l'on autorise ou non le processeur à rester inactif alors que des tâches sont en attente ;
- il peut être établi hors ligne ou dirigé en ligne par des priorités,
- ces priorités peuvent être assignées de manière statique ou dynamique ;
- enfin l'ordonnancement est dit préemptif si une tâche peut être suspendue au profit d'une autre pour être reprise par la suite, et non préemptif sinon.

2.1 Ordonnancement hors ligne ou en ligne

Un ordonnancement hors ligne consiste à écrire de façon statique un plan d'ordonnancement déterminant une succession périodique de décisions d'ordonnancement. Le travail de l'ordonnanceur se limite ensuite à l'exécution de ce plan.

Le principal avantage réside dans la très faible complexité en ligne de la décision d'ordonnancement [KFG+92], puisqu'il suffit de lire la table. La table peut être créée de façon *ad hoc* [Foh94], optimale [JD90], ou par optimisation multi-critères [EJ00]. L'objectif d'obtenir un ordonnancement prévisible est atteint, l'ordonnancement est même déterministe, c'est-à-dire qu'il peut être établi sans exécuter le système.

Les inconvénients sont pourtant nombreux. D'une part, plus le système se complexifie, plus la taille de la table augmente, engendrant une forte complexité en mémoire. Ensuite, il n'existe pas toujours d'algorithme pour générer la table de façon optimale. Cette table est alors construite de façon *ad hoc*. Pour valider et perfectionner la table, seule l'exécution, coûteuse, d'une série de tests est possible. Changer un paramètre d'une tâche, ou ajouter une tâche peut radicalement changer la table, nécessitant de relancer tous les scénarios de test.

L'ordonnancement dirigé par priorités, *priority driven scheduling*, s'effectue en ligne. Il consiste à assigner une priorité à chacune des tâches, et à toujours choisir la tâche avec la plus forte priorité lorsqu'une décision d'ordonnancement doit être prise. Il existe de nombreuses politiques pour assigner les priorités aux tâches, certaines pouvant être statiques (les priorités sont établies une fois pour toute), d'autres dynamiques (les priorités varient en fonction de paramètres dépendant de l'exécution).

L'ordonnancement produit est prévisible. La priorité d'une tâche à un instant t dépend toujours d'un paramètre qui peut être prévu. Une analyse a priori du système, connaissant la politique d'ordonnancement, ou un contrôle d'admission effectué dynamiquement doit permettre de garantir le respect des contraintes temps réel qui lui sont attachées.

2.2 Ordonnancement préemptif ou non préemptif

Il existe deux grandes sous familles d'ordonnanceurs : les algorithmes préemptifs et les algorithmes non préemptifs. Les premiers permettent qu'une tâche soit interrompue pour exécuter une autre tâche, et être reprise par la suite. Avec un algorithme non préemptif, une tâche qui commence son exécution va jusqu'au bout (ou peut être abandonnée dans certains cas). Nous considérons dans la suite de notre étude des systèmes disposant d'un ordonnanceur préemptif mais les mécanismes que nous proposons dans la troisième partie peuvent être adaptés aux systèmes non préemptifs.

Tâche	Période	Échéance	Coût
τ_1	6	6	2
τ_2	7	4	3
τ_3	15	15	3

TAB. 2.1 – Données d'un exemple de système de tâches temps réel

2.3 Ordonnancement à priorités statiques

Avec ce type de politique, chaque tâche possède une priorité fixée avant son activation. La priorité peut être fixée de façon arbitraire, ou selon un critère d'importance de la tâche. Pourtant, ce type d'assignation n'est pas le plus performant en terme d'utilisation des ressources processeur. Aussi plusieurs méthodes d'assignation automatique de priorités ont-elles été proposées et évaluées. L'objet de ce chapitre et du suivant est de dresser un rapide bilan des techniques d'ordonnancement et de l'étude analytique de la faisabilité. Un état de l'art complet sur les algorithmes à priorités statiques et leurs conditions d'ordonnançabilité est dressé dans [ABD+95].

2.3.1 Priorités basées sur la fréquence

La politique *Rate Monotonic (RM)** [Ser72] assigne la priorité la plus forte à la tâche la plus fréquente.

Elle est dite *optimale* pour la classe d'algorithmes d'ordonnancement préemptif à priorités fixes de systèmes à échéances sur requêtes (modèle de tâche périodique 1). L'optimalité d'une politique pour une classe d'algorithmes signifie que pour un système donné appartenant à cette classe, si une assignation quelconque des priorités aboutit à un système ordonnançable, alors l'attribution des priorités résultant de l'algorithme optimal est également ordonnançable. Réciproquement, si l'attribution des priorités selon l'algorithme optimal n'aboutit pas à un système ordonnançable, aucun autre algorithme de la même classe ne peut y parvenir.

La qualité d'une politique d'assignation de priorités peut également se mesurer par la borne sur l'utilisation du processeur par les systèmes faisables en résultant. Pour des systèmes générés avec les scénarios les moins favorables, cette borne pour *RM* est de 69% [LL73]. Pour des systèmes générés aléatoirement, elle est de 88% [LSD89].

La figure 2.1 page suivante montre l'ordonnancement du système présenté dans le tableau 2.1 obtenu avec *RM*. On remarque que les tâches τ_2 et τ_3 dépassent leur échéance puisqu'elles terminent leur première instance respectivement au bout de 5 et 18 *unités de temps (ut)**. On parle alors de défaillances temporelles.

Un état de l'art exhaustif sur la politique *RM* et son analyse peut être trouvé dans [KRP+94].

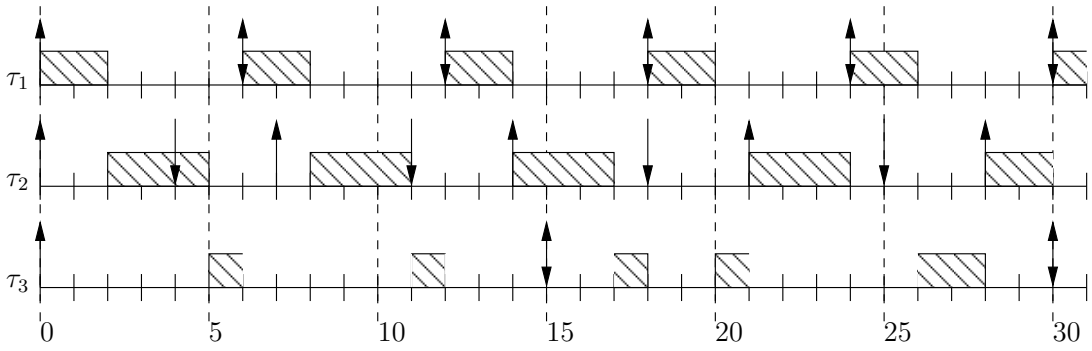


FIG. 2.1 – Ordonnancement *RM* du système de tâches du tableau 2.1

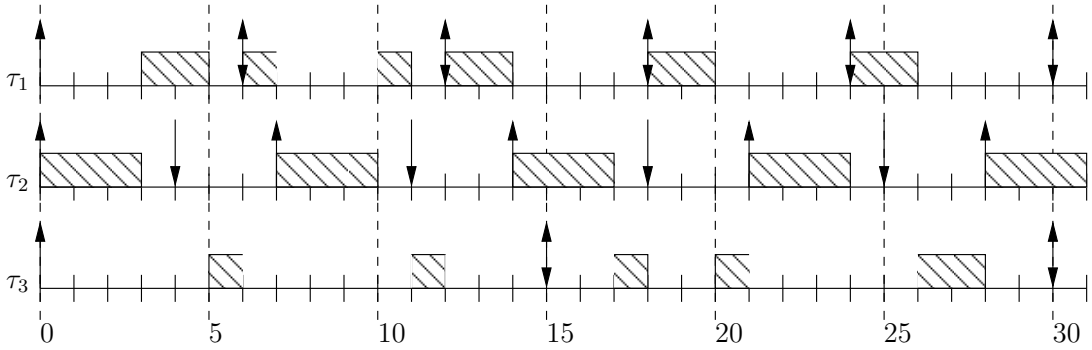


FIG. 2.2 – Ordonnancement *DM* du système de tâches du tableau 2.1

2.3.2 Priorités basées sur l'échéance

La politique *Deadline Monotonic (DM)** [LW82] assigne la priorité la plus forte à la tâche la plus urgente, c'est-à-dire avec l'échéance relative la plus petite. Dans le cas des systèmes avec échéances sur requêtes, *DM* est donc équivalent à *RM*.

Elle est optimale pour la classe des algorithmes d'ordonnancement à priorités fixes préemptif de systèmes avec $D_i \leq T_i$. Cependant, cela n'est plus vrai lorsque les périodes et les échéances sont indépendantes, c'est-à-dire si l'on autorise $D_i > T_i$ [Leh90].

La figure 2.2 montre l'ordonnancement du système présenté dans le tableau 2.1 obtenu avec *DM*. Ici τ_2 devient plus prioritaire que τ_1, ce qui lui permet de ne pas rater sa première échéance, sans pour autant compromettre l'exécution de τ_1. En revanche, rien ne change pour τ_3 qui termine toujours sa première instance au bout de 18 *ut*.

Un état de l'art exhaustif sur la politique *DM* et son analyse peut être trouvé dans [ABRW92].

2.4 Ordonnancement à priorités dynamiques

Ce type d'algorithme permet à la priorité des tâches de changer d'une requête à l'autre et éventuellement pour une même requête. La règle d'assignation de la priorité ne se

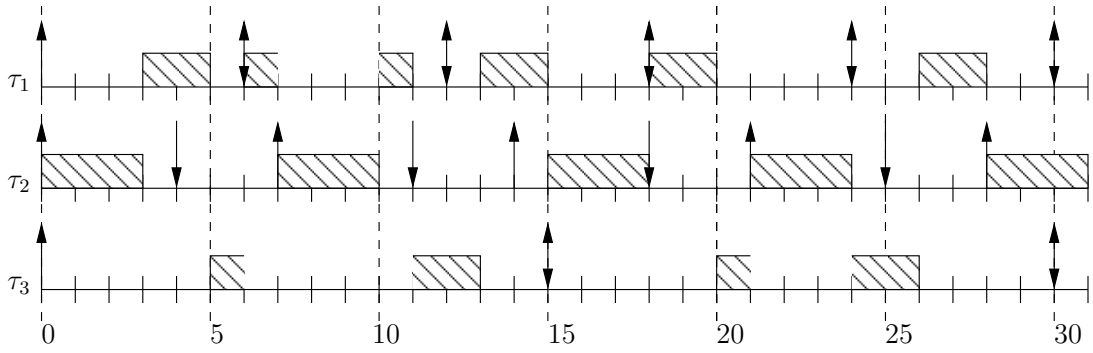


FIG. 2.3 – Ordonnancement *EDF* du système de tâches du tableau 2.1

repose plus sur une caractéristique constante de la tâche, mais sur des valeurs déterminées pendant l'exécution.

2.4.1 Priorités basées sur l'échéance

Lorsqu'une tâche se termine, l'ordonnanceur choisit la tâche possédant l'échéance absolue la plus proche afin de l'exécuter. Si une tâche dont l'échéance est plus proche est activée, elle la préempte. Cette politique est optimale pour la classe d'algorithmes d'ordonnancement préemptif non oisif [LL73]. *Earliest Deadline First (EDF)** permet également d'atteindre une utilisation du processeur maximale (100%). De plus, cette politique est optimale pour un autre problème, celui de la minimisation du retard maximal des tâches [But04].

La figure 2.3 montre l'ordonnancement du système présenté dans le tableau 2.1 obtenu avec *EDF*. Ici toutes les requêtes des tâches terminent leur exécution avant leur échéance. Notons que l'ordonnancement présenté est une possibilité, mais pas la seule. En effet à l'instant 14, alors que la tâche τ_1 est en cours d'exécution, la tâche τ_2 est activée. La prochaine échéance absolue étant la même pour les deux tâches (18), l'ordonnanceur a le choix : continuer l'exécution de τ_1 ou commencer celle de τ_2. Un scénario similaire peut être observé à l'instant 24 lorsque la tâche τ_1 est activée alors que τ_3 est en cours d'exécution : les deux tâches ont leur prochaine échéance à l'instant 30. Nous avons choisi de présenter la version de l'ordonnancement dans laquelle une tâche déjà commencée garde le contrôle du *processeur (CPU)**. Ceci permet de minimiser le nombre de préemptions.

2.4.2 Priorités basées sur la laxité

La laxité d'une tâche, $L_i(t)$, est le temps restant avant son échéance moins le temps de calcul qui lui reste à effectuer. Ceci ne doit pas être confondu avec la laxité du processeur au niveau de priorité i, qui prend en considération l'interférence des tâches plus prioritaires.

L'ordonnancement *Least Laxity First (LLF)** choisi d'exécuter la tâche possédant la laxité la plus faible. Cette politique est également optimale pour la classe d'algorithmes

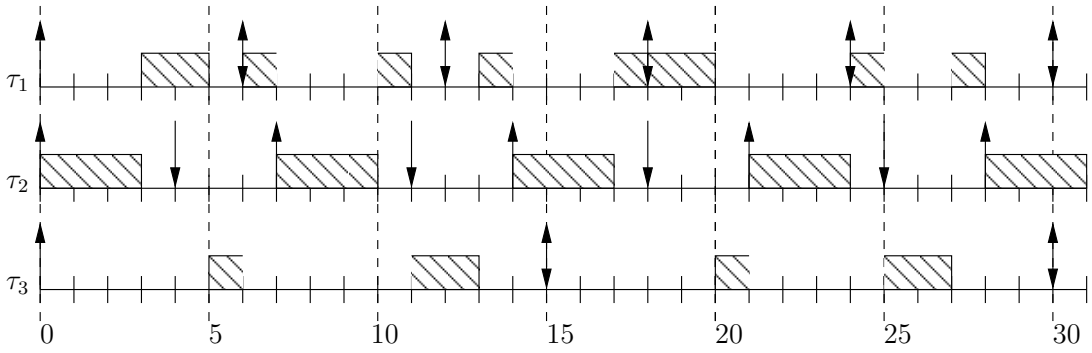


FIG. 2.4 – Ordonnancement *LLF* du système de tâches du tableau 2.1

d'ordonnancement préemptif non oisif et permet aussi une utilisation du processeur de 100% [LL73]. Toutefois, elle requiert plus de changements de contexte que *EDF*, car la laxité de la tâche qui s'exécute reste constante tandis que celle des autres tâches diminue.

La figure 2.4 montre l'ordonnancement du système présenté dans le tableau 2.1 obtenu avec *LLF*. Jusqu'à l'instant 14, l'ordonnancement est équivalent à *EDF*. Les laxités des tâches τ_1, τ_2 et τ_3 sont alors respectivement 3, 1 et 13. La tâche τ_2 préempte donc τ_1 bien que les deux tâches aient la même échéance. Notons que l'algorithme *EDF* aurait pu également accorder le processeur à τ_2 à l'instant 14, mais il a le choix, alors qu'ici *LLF* est obligé d'ordonnancer τ_2.

La situation à l'instant 24 illustre l'augmentation des changements de contextes. τ_1 et τ_3 ont alors la même laxité (4), et la même échéance (30). Il leur reste également toutes deux 2 unités de temps de calcul à effectuer. Dans le cas d'*EDF*, quelque soit la tâche choisie, ce choix n'a pas de raison de changer à l'instant 25. En revanche, avec *LLF*, la tâche choisie aura toujours une laxité de 4 à l'instant 25, alors que l'autre n'aura plus qu'une laxité de 3. Dans cet exemple, la tâche choisie termine son exécution à l'instant 28, mais si le coût des tâches était supérieur, il y aurait de nouveau un changement de contexte inutile.

Le but de notre travail consiste à implanter des mécanismes pour leur utilisation dans le cadre de *RTSJ*. Cette spécification requiert au minimum la présence d'un ordonnanceur à priorités fixes préemptif, non oisif. Nous n'étudions donc dans la suite que ce type d'ordonnanceur.

Chapitre 3

Analyse de faisabilité

L'étude des paramètres des tâches pour déterminer la faisabilité du système sous une politique d'ordonnancement donnée s'appelle l'*analyse de faisabilité**. Un système est faisable si chacune des tâches qui le composent est faisable, c'est-à-dire peut respecter toutes les contraintes temporelles qui lui sont attachées.

Dans ce chapitre, nous présentons les principes fondamentaux de l'analyse de faisabilité pour les algorithmes d'ordonnancement les plus utilisés. Nous commençons par étudier le cas des modèles de tâches les plus simples pour aller progressivement vers un modèle plus général.

Analyse de faisabilité Une analyse de faisabilité a pour but de garantir le respect des paramètres temporels attachés à l'ensemble des tâches temps réel constituant le système. Pour obtenir cette garantie, il faut déterminer un *test de faisabilité* ou *condition d'ordonnançabilité*. La condition recherchée peut être une condition nécessaire, une condition nécessaire et suffisante, ou même si l'on a besoin de déterminer la faisabilité de façon rapide mais certaine, simplement suffisante.

Trois principales approches sont possibles pour déterminer un *test de faisabilité* : l'étude de la charge du système, celle de la demande processeur, et l'analyse des *temps de réponse* des tâches.

3.1 Étude de la charge

Dans un système composé uniquement de tâches périodiques, l'équation 3.1 permet de calculer la charge.

$$U = \sum_{i=1}^{n} \frac{C_i}{T_i} \tag{3.1}$$

Une charge égale à 1 est maximale, car elle correspond à la situation où à tout instant de la vie du système du travail est en attente. Une condition nécessaire de faisabilité pour le système est donc simplement d'avoir une charge inférieure ou égale à 1.

Cette condition n'est pas suffisante pour un ordonnanceur à priorités fixes préemptif. On peut en effet facilement exhiber un système avec une charge inférieure à 1 mais non faisable, comme c'est le cas pour l'exemple du tableau 2.1 page 39. La charge est égale à $2/6 + 3/7 + 3/15 = 101/105 < 1$. Pourtant les ordonnancements résultant de *RM* et de *DM* présentent des dépassements d'échéances (respectivement aux temps 4 et 15).

Dans le cas $D_i = T_i$, ce test est nécessaire et suffisant pour *EDF*. Toujours avec l'hypothèse de l'échéance sur requête ($D_i = T_i$), une condition suffisante basée sur la charge est donnée pour *RM* par l'équation 3.2.

$$U \leq n(2^{1/2} - 1) \tag{3.2}$$

Cette condition n'est pas nécessaire, on a d'ailleurs vu que la borne d'utilisation processeur pour un système faisable avec *RM* était fixée expérimentalement à 88%, or $\lim_{n\to\infty} n(2^{1/2} - 1) = \ln 2 \simeq 0.69$.

3.2 Étude de la demande processeur

Alg. 1 Déterminer la faisabilité du niveau de priorité i

```
1: w_i ← Σ_{k≤i} C_k
2: loop
3:    if w_i > D_i then
4:       return false
5:    end if
6:    t ← w_i
7:    w_i ← Σ_{k≤i} ⌈t/T_k⌉ C_k
8:    if t = w_i then
9:       return true
10:   end if
11: end loop
```

Nous nous plaçons dans un premier temps dans l'étude du modèle de tâche 2, restreint aux conditions $D_i \leq T_i$ et $r_i = 0$. Cette section présente un test de faisabilité nécessaire et suffisant pour un système à priorités fixes préemptif, avec pour toutes les tâches $D_i \leq T_i$.

Ce test se base sur l'étude de la demande processeur. Nous allons, pour chaque niveau de priorité i, vérifier la condition nécessaire suivante :

il existe un instant t dans l'intervalle $]0, D_i]$ tel que $t = w_i(t)$;

où $w_i(t)$ est la quantité de travail demandée par les tâches de priorité supérieure ou égale à i dans l'intervalle $[0, t[$. Cette condition exprime le fait qu'il doit exister un instant inférieur à l'échéance de la tâche pour lequel toutes les demandes processeur des tâches de priorités supérieures et de la tâche étudiée sont satisfaites. Autrement dit, la somme

t	8	13	15
$w_3(t)$	13	15	18

TAB. 3.1 – Application de la procédure 1 à la tâche τ_3 de l'exemple 2.2

des retards subis par la tâche en raison des tâches plus prioritaires plus le coût de la tâche est inférieure à son échéance.

La quantité de travail demandé à un niveau de priorité i à l'instant t, $w_i(t)$ est donné par l'équation 3.3. Comme expliqué dans les notes de lecture au début de ce manuscrit, nous avons choisit pour convention d'attribuer la valeur numérique la plus petite à la priorité la plus forte.

$$w_i(t) = \sum_{k \leq i} \left\lceil \frac{t}{T_k} \right\rceil C_k \tag{3.3}$$

$$\exists ? t = \min_{t \leq D_i} \{t = w_i(t)\} \tag{3.4}$$

La procédure 1, équivalente à résoudre l'équation 3.4, est donc un test suffisant et nécessaire de faisabilité pour la tâche τ_i. En la réitérant sur toutes les tâches, on obtient un test de faisabilité pour le système. Cette procédure est convergente car la fonction $w_i(t)$ est croissante. Le tableau 3.1 donne la suite des couples $(t, w(t))$ dont le calcul est nécessaire pour montrer que la tâche τ_3 de la figure 2.2 n'est pas faisable en *DM*. Pour l'instant 15, la charge demandée est de 18, ce qui dépasse l'échéance de τ_3 (15).

Ce test n'est pas suffisant pour un système où les échéances peuvent être supérieures aux périodes, car une requête d'une tâche peut alors interférer sur les requêtes suivantes de la même tâche. Par conséquent, le respect de la première échéance, même dans le scénario d'activation le moins favorable (synchrone), n'implique pas le respect des suivantes.

Pour le modèle de tâche 3, il faudra considérer le scénario le moins favorable : remplacer, pour tout i, C_i par $C_i + B_i$ et considérer l'activation au plus tôt de toutes les tâches.

3.3 Calcul du pire temps de réponse en priorité fixe

Pour les systèmes avec des échéances indépendantes des périodes, la seule solution reste de calculer pour chaque tâche le pire *temps de réponse*. Le système est alors faisable à condition que le pire *temps de réponse* de chaque tâche soit inférieur à son échéance.

Le pire *temps de réponse* d'une tâche est le temps maximal qui peut s'écouler entre l'activation d'une instance de cette tâche et la fin de l'exécution de cette instance, ceci dans les pires scénarios d'activation et d'exécution.

Pour calculer ce pire *temps de réponse*, il faut commencer par identifier le pire scénario d'activation. Ce dernier dépend du modèle de tâche. Par exemple, pour le modèle 1, il est

t	18	21	23	26	28
$w_3(t)$	21	23	26	28	28

TAB. 3.2 – Application de la procédure 2 à la tâche τ_3 de l'exemple 2.2

prouvé que le scénario d'activation conduisant aux pires *temps de réponse* est le scénario synchrone, où toutes les tâches sont activées en même temps. En revanche, si l'on considère le modèle 3 pour lequel il existe une gigue d'activation, le pire scénario est toujours celui où les tâches arrivent au plus tôt.

Une fois le pire scénario d'activation identifié, il faut calculer la durée de la période à étudier. On appelle *période occupée de niveau i* (bp_i)* une période de temps pendant laquelle le processeur exécute des tâches de priorité supérieure ou égale à P_i. La plus longue bp_i résulte du pire scénario d'activation discuté dans le paragraphe précédent. Il est montré que le pire *temps de réponse* d'une tâche est obtenu durant la bp_i résultant du pire scénario d'activation.

Lorsque l'on connait le pire scénario d'activation et que l'on a calculé la bp_i qui en résulte, il reste à calculer les *temps de réponse* de chaque instance de τ_i qui est activée dans cette période. Le pire *temps de réponse* de la tâche est alors le maximum de ces valeurs.

3.3.1 Calcul de la période occupée de niveau i

Alg. 2 Déterminer la période occupée de niveau i

1: $w_i \leftarrow \sum_{k \leq i} C_k$
2: **loop**
3: $\quad t \leftarrow w_i$
4: $\quad w_i \leftarrow \sum_{k \leq i} \left\lceil \frac{t}{T_k} \right\rceil C_k$
5: $\quad$ **if** $t = w_i$ **then**
6: $\quad\quad$ **return** t
7: $\quad$ **end if**
8: **end loop**

$$bp_i = \min_{t>0}\{t = w_i(t)\} \tag{3.5}$$

La procédure 2, équivalente à résoudre l'équation 3.5, permet de calculer la bp_i. Cette équation est très similaire à la procédure 1, à la différence que l'on ne s'arrête pas lorsque l'échéance est dépassée. Pour résoudre l'équation il faut faire à l'instant t le bilan entre le travail demandé par les tâches de priorité i et de plus forte priorité d'une part, et le temps écoulé d'autre part. Lorsque le travail demandé depuis le commencement est égal au temps écoulé, la bp_i résultant du scénario d'activation le moins favorable est terminée.

Si nous reprenons une nouvelle fois l'exemple de la tâche τ_3 de la figure 2.2, et que nous poursuivons le calcul des $w_3(t)$ présenté dans le tableau 3.1, on obtient alors le tableau 3.2 qui nous indique que la période occupée de niveau 3 qui commence au temps 0 est de 28 unités de temps. Si l'on étudie la tâche τ_3 dans l'intervalle $[0, 28]$, sont pire *temps de réponse* sera rencontré.

3.3.2 Calcul du temps de réponse d'une instance

On cherche à calculer le *temps de réponse* de l'instance j de la tâche τ_i, en numérotant les instances à partir de 1. Sa date de terminaison, notée F_i^j, est donnée par l'équation 3.6.

$$F_i^j = \min_{t>0}\{t = w_{i-1}(t) + (j-1)C_i\} \tag{3.6}$$

Son *temps de réponse*, R_i^j, est alors la différence entre sa date de terminaison et sa date d'activation. Il est donné par l'équation 3.7.

$$R_i^j = F_i^j - (r_i + (j-1)T_i) \tag{3.7}$$

3.3.3 Test de faisabilité basé sur les pires temps de réponse

Nous connaissons bp_i, la durée de l'intervalle d'étude. On obtient alors Q_i, le nombre d'instances de τ_i qu'il faut étudier, par l'équation 3.8.

$$Q_i = \left\lceil \frac{bp_i}{T_i} \right\rceil \tag{3.8}$$

Ces instances sont activées respectivement aux instants $\{r_i, r_i + T_i, ..., r_i + (Q_i - 1)T_i\}$. Il faut alors calculer les *temps de réponse* de toutes ces instances, donnés par l'équation 3.7. Le pire *temps de réponse* de τ_i, noté R_i, est la plus grande de ces valeurs (équation 3.9).

$$R_i = \max_{j\in\{1,...,Q_i\}} R_i^j \tag{3.9}$$

Un test suffisant et nécessaire de faisabilité est alors de vérifier l'équation 3.10.

$$\forall i, R_i \leq D_i \tag{3.10}$$

Une procédure pour calculer les pires *temps de réponse* obtenus avec un ordonnancement *EDF* est proposée dans [Spu96].

3.4 Partage de ressources

Si les tâches peuvent partager des ressources, l'accès concurrent à une même ressource par deux tâches peut provoquer une attente. Lorsqu'une tâche désire accéder à une ressource partagée, elle doit en premier lieu obtenir un verrou associé à cette ressource. Tant qu'elle ne relâche pas ce verrou, les autres tâches désirant accéder à la ressource sont *bloquées*. On dit que lorsqu'une tâche obtient un verrou, elle entre dans une *section critique*. Elle en sort lorsqu'elle relâche le verrou.

Dans un contexte temps réel, partager des ressources peut entrainer des inversions de priorité. En effet, lorsqu'une tâche de faible priorité obtient un verrou, elle peut bloquer l'exécution des tâches plus prioritaires qui tenteraient par la suite d'obtenir le même verrou.

Plusieurs protocoles dits de partage de ressources permettent de borner ces inversions de priorité. Il en ressort que l'on peut, comme nous l'évoquions dans la description du modèle de tâche 3, ajouter aux tâches un paramètre noté B_i, qui correspond au temps de blocage maximal qu'elle peut subir à cause d'un partage de ressource. Notons ici qu'une tâche peut très bien être bloquée à cause de l'accès à une ressource qu'elle même n'utilise jamais. Lors de l'analyse de faisabilité, il suffit d'ajouter ce paramètre supplémentaire au pire *temps de réponse* de chaque tâche.

Le temps de blocage maximal, B_i, dépend du protocole de partage de ressource utilisé. Les plus connus sont *Priority Inheritance Protocol (PIP)** et *Priority Ceiling Protocol (PCP)** [SRL90]. Avec *PIP*, si une tâche de priorité basse τ_b possède un verrou sur une ressource, lorsqu'une tâche de plus forte priorité τ_h demande ce verrou, τ_b hérite de la priorité de τ_h jusqu'à ce qu'elle relâche le verrou. Avec *PCP* (ou *Original Ceiling Priority Protocol (OCPP)**), si une tâche de priorité basse τ_b possède un verrou sur une ressource, lorsqu'une tâche de plus forte priorité τ_h demande un verrou sur cette ressource ou sur une autre, la tâche de basse priorité τ_b hérite de la priorité plafond de la ressource qu'elle possède jusqu'à ce qu'elle relâche le verrou. Par conséquent, si la priorité de τ_h est plus faible que celle de la ressource détenue par τ_b, elle sera bloquée. Le protocole *Immediate Ceiling Priority Protocol (ICPP)** repose sur la même idée, à la différence qu'une tâche qui obtient un verrou hérite immédiatement de la priorité plafond de la ressource concernée. Pour la norme *POSIX**, *ICPP* est appelé *Priority Protect Protocol (PPP)** alors que dans *RTSJ*, on le retrouve sous le nom de *Priority Ceiling Emulation (PCE)**.

Chapitre 4

Gestion des tâches non périodiques

Nous avons vu précédemment qu'un système temps-réel devait être composé de tâches possédant des paramètres d'activation prévisibles, afin de pouvoir étudier la faisabilité du système de façon statique. Pourtant, la majorité des événements du monde réel sont de nature apériodique, et le système doit être capable de les gérer tout en continuant à garantir le respect des contraintes temporelles des autres tâches (souvent périodiques).

Un événement apériodique peut être de deux natures : il peut s'agir d'un événement aussi important pour le système qu'une tâche temps réel dur, et posséder une échéance stricte, comme les traitements périodiques ; ou bien son utilité pour le système peut dépendre du temps dans lequel il est traité.

Dans le premier cas, il n'y a pas d'autre solution que de borner le temps minimal entre deux occurrences de cet événement. Ainsi son traitement peut être assimilé dans le pire scénario à une tâche périodique. On dit dans ce cas que l'événement est *sporadique*.

Dans le second cas de figure, la problématique devient la minimisation des *temps de réponse* des traitements des événements apériodiques, mais en continuant de garantir la faisabilité dans le pire scénario des tâches à échéance stricte (périodiques ou sporadiques).

Il existe principalement trois approches pour répondre à cette problématique d'ordonnancement mixte :

1. le *Background Scheduling (BS)**, qui consiste à exécuter les traitements des événements apériodiques dans une plage de priorités plus faibles que celle des tâches à échéance stricte ;
2. utiliser un serveur de tâche, c'est-à-dire déléguer le traitement des événements apériodiques à une tâche spéciale du système, que l'on sait intégrer dans l'analyse de faisabilité ;
3. voler des temps creux, c'est-à-dire retarder le plus possible le démarrage d'un traitement à échéance stricte afin de profiter de ce retard pour exécuter les traitements des événements apériodiques en attente.

Pour chaque approche, plusieurs politiques différentes sont possibles. Une description détaillée et une comparaison de ces politiques peut être trouvée dans [But04, chapitre

5]. Nous présentons dans ce chapitre les principales, dans le cadre d'un ordonnancement non oisif à priorités fixes préemptif. La plupart de ces politiques ont été adaptées pour fonctionner avec un ordonnanceur à priorités dynamiques. Le lecteur pourra consulter [CC89, SB94, SB96].

4.1 Approche sporadique

L'approche sporadique est utilisée lorsque le traitement d'un événement apériodique est aussi important pour le système que les autres tâche à échéance stricte. Elle consiste à assigner une pseudo période à une tâche de nature non périodique. Cette pseudo période correspond à borner la fréquence d'arrivée de deux requêtes successives de cette tâche.

Cette valeur peut être arbitraire, ou découler d'une caractéristique réelle de l'événement. Si la valeur est arbitraire, la tâche pourra donc en pratique arriver plus souvent. Il faudra alors veiller à traiter dynamiquement cette violation du temps minimal d'inter arrivée.

Plusieurs approches sont possibles : la requête qui arrive trop tôt peut être ignorée, elle peut être mise en file d'attente pour être réactivée à la prochaine pseudo période, elle peut encore remplacer la requête en attente la plus ancienne.

Ces approches sont déjà intégrées dans *RTSJ*, comme nous le verrons au chapitre 6. Aussi nous sommes nous davantage intéressés aux approches de traitements d'événements apériodiques à contrainte souple durant nos travaux.

4.2 Ordonnancement en tâche de fond

L'approche *BS* consiste à traiter les tâches apériodiques en arrière plan du trafic temps réel dur : les tâches apériodiques ne sont traitées que lorsque le processeur est libre, et immédiatement préemptées si une tâche à échéance stricte est activée.

Cette approche ne répond qu'en partie à la problématique de l'ordonnancement mixte. Elle permet en effet de s'assurer que le traitement des événements apériodiques ne peut pas perturber l'exécution des tâches à échéance stricte, et par conséquent ne remet pas en question les garanties fournies par l'analyse de faisabilité. En revanche, rien n'est fait pour tenter de minimiser les temps de réponse des événements apériodiques.

Cette solution est par ailleurs très simple à mettre en œuvre : il suffit de réserver la priorité temps réel la plus faible pour traiter les événements apériodiques, ou une plage de plusieurs priorités faibles si une hiérarchie existe entre ces événements. Cette stratégie fonctionne aussi bien pour un ordonnanceur à priorités fixes que pour un ordonnanceur à priorités dynamiques. Le seul surcoût d'exécution réside dans la gestion de la ou des files d'attente pour les traitements apériodiques. Il faudra également veiller à gérer d'éventuels dépassements de la taille de cette ou de ces files.

La figure 4.2(b) page 57 présente un exemple d'utilisation du *BS*.

4.3 Serveur à scrutation

Les serveurs de tâches ont été introduits pour la première fois en 1987 par Lehoczky, Sha et Strosnider dans [LSS87]. Cette publication propose trois algorithmes de service : le *PS*, le *DS* et le *Priority Exchange Server (PES)**.

Les propriétés de ces politiques et la mise au point de nouvelles ont depuis fait l'objet de nombreux travaux. En particulier, une analyse approfondie du *DS* pourra être trouvée dans [SLS95], le *Sporadic Server(SS)** est introduit dans [SSL89, Spr90] et le *Extended Priority Exchange Server (EPES)** dans [SLS88].

Nous présentons ici le *PS*, appelé en français « serveur à scrutation ». Ce serveur est une tâche périodique ordinaire, possédant par conséquent une période et un coût. Cette tâche est chargée de traiter les tâches apériodiques dans la limite de ce coût, que l'on appelle capacité maximale. Pour cela, le serveur peut accéder à une file d'attente dans laquelle sont ajoutés les traitements associés au événements apériodiques qui surviennent. Cette file peut être implantée avec une structure de type *premier arrivé, premier servi (FIFO)** ou *dernier arrivé, premier servi (LIFO)**, ou encore être triée si un ordre existe entre les événements apériodiques.

Le serveur se déclenche à chaque période avec sa capacité maximale. S'il y a des tâches apériodiques en attente dans la file, elles sont traitées. La capacité du serveur diminue lorsqu'il traite des tâches. S'il n'y a pas ou plus de tâche en attente, la capacité restante est immédiatement perdue. Lorsque la capacité est épuisée (soit par un traitement, soit parce qu'il n'y a plus de traitement en attente), le serveur s'endort jusqu'à sa prochaine activation.

Par conséquent, le temps d'exécution maximal d'une instance d'un *PS* est égal à sa capacité. De plus, il ne peut être activé que périodiquement (perte de la capacité en absence de tâche à traiter). Dans le pire des cas, où la somme de traitements apériodiques à effectuer dépasse sa capacité, le serveur provoque sur les tâches périodiques moins prioritaires la même interférence que s'il s'agissait d'une tâche périodique à échéance stricte ordinaire.

L'inconvénient de cette politique vient de la perte de sa capacité en l'absence de tâche apériodique à servir. En effet, si une tâche survient à un instant $\alpha T_s + \epsilon$ où α est un entier positif quelconque, et que le serveur n'avait pas d'autre tâche apériodique en attente, cette dernière ne pourra être traitée qu'à l'instant $(\alpha + 1)T_s$.

Une amélioration simple du *PS* consiste à traiter les tâches avec une politique de *BS* lorsque la capacité du serveur est épuisée.

La figure 4.2(c) page 57 illustre l'utilisation d'un *PS*.

4.3.1 Contrôle d'admission

Si les événements apériodiques possèdent une échéance au-delà de laquelle leur utilité pour le système devient nulle, il est possible d'analyser dynamiquement la faisabilité de la

tâche, c'est-à-dire d'effectuer un test permettant de savoir si elle a une chance de respecter son échéance. Si le test est négatif, la tâche peut être abandonnée tout de suite. On appelle ce procédé un *contrôle d'admission*.

Dans un premier temps, il est plus facile d'étudier le cas particulier d'une seule requête apériodique, J_a, qui survient à l'instant r_a, avec un coût C_a et une échéance relative D_a. Nous notons la période du serveur T_s et sa capacité C_s.

Dans le pire cas de figure, J_a survient juste après une période du serveur, c'est à dire $r_a = \alpha T_s + \epsilon$, et la tâche va donc devoir attendre la période suivante avant de commencer. Si on suppose que $C_a \leq C_s$, une condition suffisante de faisabilité est donnée par l'équation 4.1.

$$2T_s \leq D_a \tag{4.1}$$

Si on lève la contrainte sur C_a, il faut au plus $\lceil C_a/C_s \rceil$ instances du serveur pour exécuter J_a. La condition suffisante de faisabilité est alors donnée par l'équation 4.2.

$$T_s + \left\lceil \frac{C_a}{C_s} \right\rceil T_s \leq D_a \tag{4.2}$$

Ces conditions ne sont que suffisantes. On ne peut avoir un test nécessaire et suffisant que si l'on exécute le serveur à la plus forte priorité. En effet, dans ce cas particulier, on sait que le serveur exécute des tâches uniquement entre les instants αT_s et $\alpha T_s + C_s$.

Soit G_a l'instance du serveur qui débutera l'exécution de J_a, F_a le nombre de fois que le serveur va devoir dépenser toute sa capacité pour traiter J_a et R_a le temps nécessaire pour terminer le traitement de J_a dans la dernière instance du serveur. Si on note d_a l'échéance absolue de J_a, et que les instances du serveur sont numérotées en partant de 0, la condition nécessaire et suffisante est alors de vérifier l'équation 4.3.

$$(F_a + G_a)T_s + R_a \leq d_a$$

$$\text{avec} \begin{cases} F_a &= \left\lceil \dfrac{C_a}{C_s} \right\rceil - 1 \\ G_a &= \left\lceil \dfrac{r_a}{T_s} \right\rceil \\ R_a &= C_a - F_a C_s \end{cases} \tag{4.3}$$

Pour étendre ce résultat au traitement de plusieurs tâches apériodiques, on suppose que le serveur traite les requêtes selon leur priorité (statique ou dynamique). Pour calculer f_k, la date de terminaison de la tâche J_k, il faut alors considérer $C_{ape}(t, P_k)$, la somme des coûts des tâches apériodiques non traitées à l'instant t dont la priorité est supérieure à P_k. Autrement dit, $C_{ape}(t, P_k)$ est la charge de travail apériodique à traiter avant la fin de J_k.

Si $c_s(t)$ est la capacité restante du serveur à l'instant t, F_k le nombre d'instances du

serveur qui devront être entièrement consacrées à $C_{ape}(t, P_k)$, G_k l'instance du serveur qui débutera le traitement de $C_{ape}(t, P_k)$ et R_k le temps nécessaire dans une dernière instance du serveur pour finir de traiter $C_{ape}(t, P_k)$, la date de terminaison de la tâche J_k (f_k) est donnée par l'équation 4.4.

$$f_k = \begin{cases} t + C_{ape}(t, P_k) & \text{si } C_{ape}(t, P_k) \leq c_s(t) \\ (F_k + G_k)T_s + R_k & \text{sinon.} \end{cases}$$

$$\text{avec} \begin{cases} F_k &= \left\lceil \dfrac{C_{ape}(t, P_k) - c_s(t)}{C_s} \right\rceil - 1 \\ G_k &= \left\lceil \dfrac{t}{T_s} \right\rceil \\ R_k &= C_{ape}(t, P_k) - c_s(t) - F_k C_s \end{cases} \tag{4.4}$$

Notons qu'une portion de $C_{ape}(t, P_k)$ égale à $c_s(t)$ est immédiatement exécutée, c'est pourquoi on a les équations précédentes pour F_k, G_k et R_k.

Pour savoir si l'arrivée d'une nouvelle tâche apériodique J_i est faisable et ne modifie pas la faisabilité des autres tâches apériodiques, il faut donc vérifier l'équation 4.5, où n_a est le nombre de tâche apériodiques acceptées.

$$\forall k \in \{1, ..., n_a\}, f_k \leq d_k \tag{4.5}$$

Il est également possible de supposer que le serveur traite les tâches dans l'ordre de leur arrivée. Dans ce cas l'admission d'une nouvelle tâche ne perturbe pas la faisabilité des tâches déjà acceptées et il suffit de vérifier la faisabilité de J_a.

4.3.2 Gestion de ressources partagées

Si les tâches à échéance stricte sont autorisées à partager des ressources, un protocole pour borner les inversions de priorité doit être utilisé. Les tâches peuvent alors subir un temps de blocage maximal, qui doit être intégrer dans l'analyse de faisabilité. Cela n'interfère nullement avec le serveur, qui continue à se comporter comme une tâche périodique ordinaire puisque lui même n'accède à aucune ressource partagée et possède la priorité la plus élevée du système.

En revanche, si l'on considère un partage de ressource entre les tâches temps réel dur et les tâches temps réel souple traitées par le serveur, cela implique que le serveur puisse subir un blocage, ou faire subir un blocage.

Le serveur cesse alors de se comporter comme une tâche périodique ordinaire. En effet, si une tâche temps réel souple traitée par le serveur obtient un verrou, et que la capacité du serveur se trouve épuisée avant que la tâche n'ait relâché ce verrou, toutes les tâches temps réel dur qui devraient accéder à la même ressource se verraient bloquées au moins jusqu'à la prochaine instance du serveur. Comme il n'est pas possible de prévoir le modèle

	Coût / capacité	Période
DS	2	4
τ_1	2	5

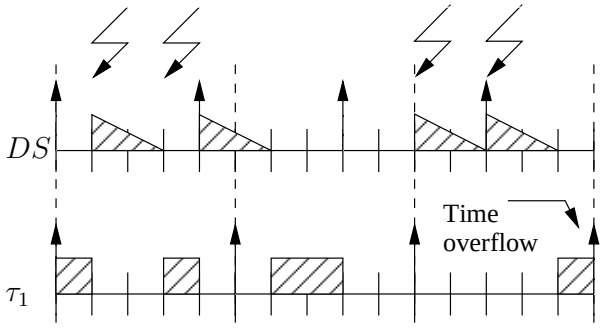


FIG. 4.1 – Système faisable avec un *PS*, non faisable avec un *DS*

d'arrivée des tâches apériodiques, il est impossible de borner le temps de blocage des tâches qui partagent des ressources avec des tâches apériodiques.

Dans le cas d'une approche par serveur de type *PS*, il semble donc impossible d'autoriser le partage de ressources entre les tâches périodiques et les tâches apériodiques. La solution consiste à s'assurer, lorsqu'une tâche apériodique demande un verrou, que la durée maximale de sa section critique, c est inférieur à la capacité restante du serveur. Si tel n'est pas le cas, l'obtention du verrou ne pourra être accordé et le serveur pourra soit se suspendre, soit ordonnancer la tâche apériodique suivante dans sa file.

Toute tâche périodique utilisant des ressources pouvant être partagées avec des tâches apériodiques pourra alors subir un blocage égal à la capacité du serveur, qu'il faudra prendre en compte lors de l'analyse de faisabilité du système.

4.4 Serveur ajournable

« Serveur ajournable » est l'appellation en français du *DS*. Il est ainsi nommé car, s'il possède comme le *PS* une capacité et une période, il peut suspendre son exécution sans perdre sa capacité s'il n'a pas de tâche à traiter.

Il est introduit en même temps que le *PS* dans [LSS87].

Sa période ne correspond plus aux instants où il peut être activé, mais à ceux où sa capacité est restaurée à sa valeur maximale.

L'avantage de cette politique dite à conservation de capacité est une amélioration des *temps de réponse* des tâches apériodiques. En effet, dans la mesure où le serveur n'a pas encore utilisé toute sa capacité, une requête apériodique peut être traitée instantanément.

Le gros défaut de cette politique est qu'elle rend caduque l'analyse de faisabilité classique pour le système. Pour s'en convaincre, il suffit de prendre l'exemple de la figure 4.1. À l'instant 8, aucune tâche n'est exécutée, pourtant le serveur conserve sa capacité, lui

permettant d'empêcher l'exécution de la tâche périodique à l'instant 10, puisque survient en même temps un événement apériodique. Le phénomène observé entre les instants 10 et 14 est souvent appelé *double hit*, le serveur provoque le retard maximal pour la tâche moins prioritaire, à savoir deux fois sa capacité.

Le scénario d'activation le moins favorable à analyser n'est plus l'activation synchrone du serveur avec l'ensemble des tâches, mais l'activation du serveur à un temps t_s, suivi d'une activation synchrone des tâches utilisateur au temps $t = t_s + T_s - C_s$. Le test de faisabilité nécessaire et suffisant basé sur l'utilisation du processeur pour les systèmes avec $D_i \leq T_i$, présenté dans le chapitre 3, est alors valable. Il en résulte que l'utilisation maximale du processeur en présence d'un *DS* est plus faible qu'avec un *PS* de même période et même capacité, puisque le test de faisabilité est plus pessimiste. C'est le prix à payer pour offrir de meilleurs *temps de réponse* aux tâches apériodiques.

La figure 4.2(d) page 57 illustre l'utilisation d'un *DS*.

4.4.1 Contrôle d'admission

Comme le *PS*, le *DS* peut servir à exécuter des tâches apériodiques possédant des échéances souples. Dans ce cas, il faut fournir un test d'ordonnançabilité en ligne pour les tâches apériodiques.

Nous reprenons les mêmes notations que pour le contrôle d'admission des tâches traitées par un *PS*. Dans un premier temps, nous considérons toujours le cas d'une seule tâche J_a. Lorsque J_a survient, trois cas de figures sont possibles :

1. $C_a \leq c_s(r_a)$, alors J_a termine au temps $f_a = r_a + C_a$;

2. $C_a > c_s(r_a)$ avec $c_s(r_a)$ inférieure au temps restant avant la prochaine activation du serveur, dans ce cas la portion de C_a exécutée par l'instance courante du serveur est $\Delta_a = c_s(r_a)$;

3. $C_a > c_s(r_a)$, mais le serveur va être réactivé, et donc sa capacité rechargée avant que $c_s(r_a)$ soit complètement consommée ; dans ce cas la portion de C_a exécutée par l'instance courante du serveur est $\Delta_a = G_a T_s - r_a$.

Dans les cas 2) et 3), la portion de C_a exécutée par l'instance courante du serveur est donnée par l'équation 4.6.

$$\Delta_a = \min(c_s(r_a), G_a T_s - r_a) \tag{4.6}$$

La date de terminaison de J_a, f_a, est alors donnée par l'équation 4.7.

$$f_a = \begin{cases} r_a + C_a & \text{si } C_a \leq c_s(r_a) \\ (F_a + G_a)T_s + R_a & \text{sinon.} \end{cases}$$

$$\text{avec} \begin{cases} F_a &= \left\lceil \dfrac{C_a - \Delta_a}{C_s} \right\rceil - 1 \\ G_a &= \left\lceil \dfrac{r_a}{T_s} \right\rceil \\ R_a &= C_a - \Delta_a - F_a C_s \end{cases} \tag{4.7}$$

On peut étendre ce résultat pour un serveur traitant une quantité non bornée de requêtes en remplaçant dans les équations 4.6 et 4.7 C_a par C_{ape} représentant la quantité de travail apériodique à fournir avant la fin de J_a. Comme pour le *PS*, si une priorité est affectée aux tâches apériodiques, il faudra revérifier l'ordonnançabilité de toutes les tâches moins prioritaires que J_a, alors que si le serveur traite les tâches dans leur ordre d'arrivée, une tâche admise l'est définitivement.

4.4.2 Gestion de ressources partagées

Les remarques concernant le partage de ressources entre tâches périodiques et tâches apériodiques que nous avons faites sur le *PS* sont également valables pour l'étude du *DS*. Toutefois, lorsqu'une tâche apériodique servie par un *DS* demande un verrou et que la capacité restante, c_r, du serveur est inférieure à la longueur de sa section critique, c, il convient de distinguer deux cas. Soit la prochaine période du serveur intervient avant l'épuisement de la capacité restante, soit elle intervient plus tard. Dans le premier cas, et si $c \leq c_r + C_s$, le verrou pourra être accordé à la tâche. Dans ce cas, dans le scénario le moins favorable, le temps de blocage maximal pouvant être infligé par le serveur sur une tâche partageant des ressources avec une tâche apériodique est égal à deux fois la capacité du serveur.

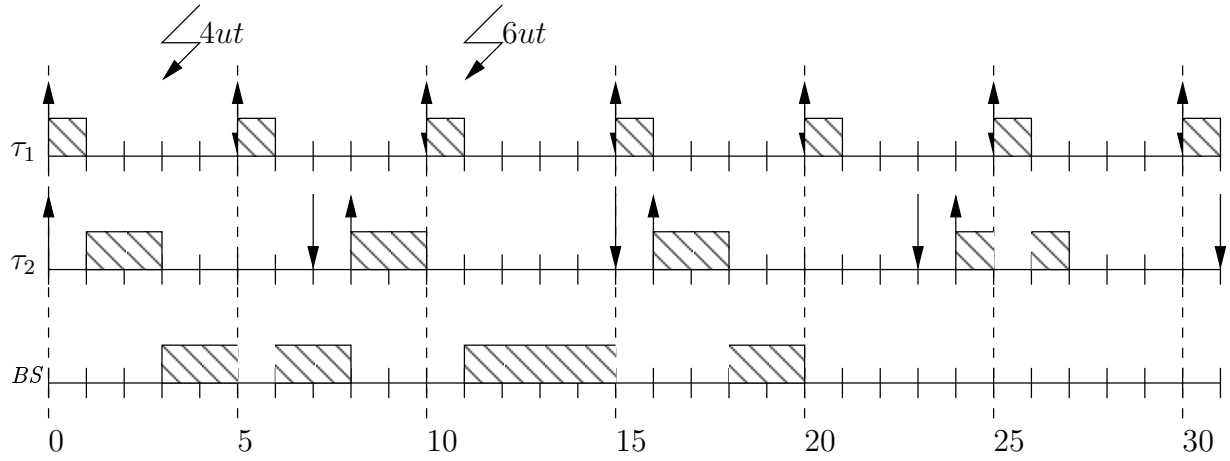
4.5 Vol de temps creux statique

Les algorithmes de vol de temps creux ont été introduits par Lehoczky et Ramos-Thuel dans [LRT92] et étendus aux tâches sporadiques temps réel dur dans [RTL93].

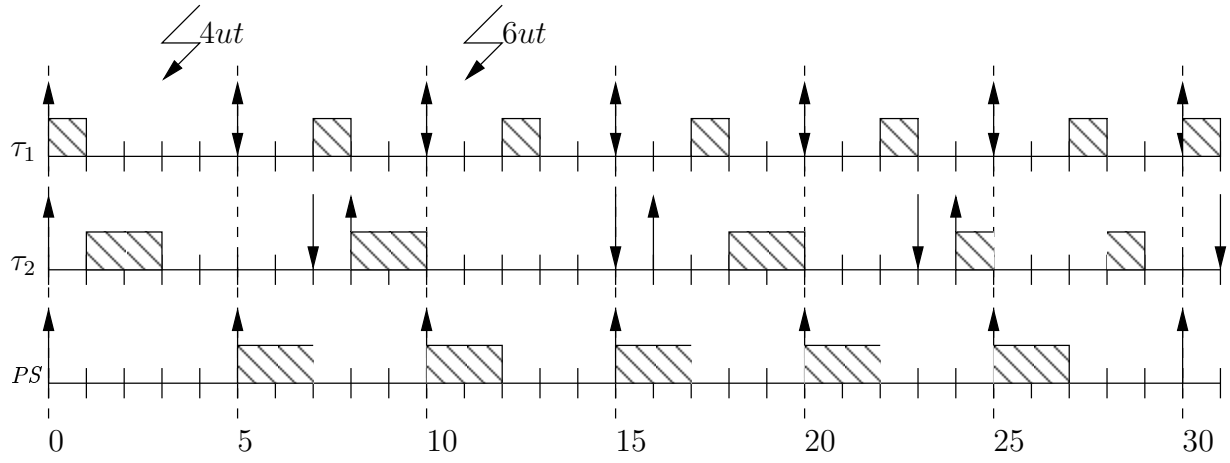
L'idée de départ est simple : puisque l'on veut seulement garantir qu'une tâche critique termine avant son échéance, mais que l'on souhaite qu'une tâche apériodique soit terminée le plus vite possible, alors si on peut calculer le retard maximal que chaque tâche périodique peut subir sans que cela n'entraine le dépassement de son échéance, ce retard peut être mis à profit pour l'exécution au plus tôt des tâches apériodiques. Le vol de temps creux se distingue de l'approche des serveurs par l'absence de réservation de ressource.

	T_i	D_i	C_i
τ_1	5	5	1
τ_2	8	7	2
τ_s	5	$\emptyset$	2

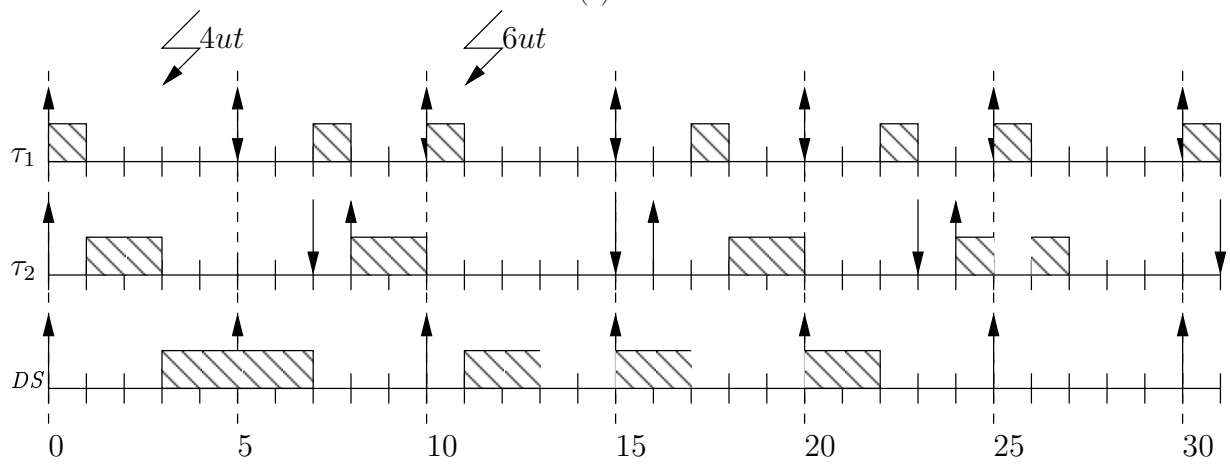
(a) Données



(b) BS



(c) PS



(d) DS

FIG. 4.2 – Comparaisons BS, PS et DS

On essaye d'utiliser le plus tôt possible les futurs temps d'innocupation du processeur.

Un voleur de temps creux est composé de deux éléments : un algorithme permettant de calculer le retard maximal que peut subir chaque tâche périodique à un instant t, et un algorithme qui utilise cette donnée pour ordonnancer les tâches apériodiques. Le mécanisme proposé dans [LRT92] repose sur un calcul statique de fonctions associant à un instant le retard maximal de chaque tâche.

4.5.1 Calculs hors ligne

La première étape consiste à calculer de façon statique la laxité du processeur en fonction du temps. La laxité du processeur au temps t est la durée pendant laquelle il peut suspendre tout traitement sans qu'aucune tâche ne rate son échéance. La fonction de laxité est une fonction croissante en escalier. On la note $S(t)$.

Pour obtenir cette fonction, il faut commencer par calculer les fonctions de laxité du processeur à chaque niveau de priorité, notées $S_i(t)$. La laxité du processeur au niveau de priorité i au temps t, encore appelée laxité de niveau i, est définie ici comme la somme de travail supplémentaire qui aurait pu être acceptée au niveau de priorité i dans l'intervalle $[0, t]$ sans qu'aucune tâches de niveau de priorité i ne rate d'échéance.

Pour que le processeur puisse se suspendre, il faut qu'il existe de la laxité pour tous les niveaux de priorité, on a donc la propriété 4.8.

$$S(t) = \min_{1 \leq k \leq n} S_k(t) \tag{4.8}$$

Les fonctions $S_i(t)$ sont des fonctions croissantes en escalier. Croissantes car la laxité du processeur au niveau i est ici définie de manière cumulative, depuis l'instant initial. Les marches correspondent à l'échéance de chaque requête de la tâche, car à ces dates il est possible d'accepter plus de travail, puisque la contrainte de l'échéance à respecter se déplace dans le temps vers la prochaine échéance.

Pour calculer les $S_i(t)$, on utilise l'équation 4.9, dans laquelle d_{ij} est l'échéance absolue de l'instance j de la tâche i, S_{ij} est la valeur de la marche de l'escalier de $S_i(t)$ correspondant à l'instance j de la tâche i et $P_i(t)$ est la somme du travail périodique de priorité supérieur ou égal à i demandé à l'instant t (non inclus). Cette formule est en fait dérivée du test nécessaire et suffisant de faisabilité présenté dans le chapitre 3.

$$\min_{0 \leq t \leq d_{ij}} \{S_{ij} + P_i(t) = t\} \tag{4.9}$$

4.5.2 Partie dynamique

Durant l'exécution du système, des accumulateurs sont mis en place, qui gardent en mémoire la somme de travail apériodique effectué ainsi que la durée d'inactivité de niveau i (le temps processeur utilisé à un niveau de priorité inférieur à i plus le temps d'inactivité

du processeur) pour chaque niveau de priorité. Ces accumulateurs sont réinitialisés au début de chaque *hyperpériode (plus petit commun multiple des périodes)**. On note $\mathcal{A}$ la somme de travail apériodique et $\mathcal{I}_i$ le temps d'inactivité de niveau i.

Pour mettre à jour ces accumulateurs, à chaque fois que le processeur commence une activité a au temps t_1 pour la finir ou la suspendre au temps t_2 :

- si a est une tâche apériodique, on ajoute $t_2 - t_1$ à $\mathcal{A}$;
- si a est la tâche périodique de plus haute priorité (τ_1), on ne fait rien ;
- si a est une tâche périodique de priorité i, on ajoute $t_2 - t_1$ à $\mathcal{I}_1, ..., \mathcal{I}_{i-1}$;
- si a correspond à une inactivité du processeur, on ajoute $t_2 - t_1$ à $\mathcal{I}_i$ pour tout i.

À l'instant t, la laxité du processeur est donnée par l'équation 4.10.

$$S(t) = \min_{1 \leq i \leq n} (S_i(t) - \mathcal{I}_i) - \mathcal{A} \tag{4.10}$$

Il faut en effet retrancher le temps d'inactivité au temps total disponible, ainsi que le temps déjà consommé. On remarque que seules les fonctions de laxité par niveau sont nécessaires. Le calcul devant être fait pour chaque tâche, il s'effectue en temps linéaire en nombre de tâches.

Il reste à déterminer à quel instant calculer la laxité. Comme du temps utile supplémentaire ne peut être obtenu que lorsqu'une tâche périodique termine une instance (dans l'hypothèse où du trafic apériodique est toujours demandé, les paliers de la fonction $S(t)$ correspondent aux échéances des tâches périodiques), il convient de calculer la laxité lorsqu'une tâche apériodique arrive alors que la file des apériodiques en attente est vide et lorsqu'une tâche périodique se termine alors que cette même file n'est pas vide.

4.5.3 Implantabilité et optimalité

Optimalité Cet algorithme a d'abord été considéré comme optimal pour le problème général de minimisation des *temps de réponse* des tâches apériodiques (en ordonnancement préemptif non oisif à priorités fixes). En effet, il permet de démarrer instantanément la tâche apériodique lorsqu'elle survient, et pour le temps maximal au delà duquel un dépassement d'échéance serait provoqué. Pourtant, il a été montré par la suite [TLS96] que l'utilisation au plus tôt des temps creux n'avait pas forcément pour conséquence de minimiser le *temps de réponse* des tâches. La figure 4.3 page suivante présente un contre exemple. Les auteurs montrent également que seul un algorithme clairvoyant, connaissant le modèle d'arrivée des tâches apériodiques, peut résoudre de façon optimale le problème de la minimisation des *temps de réponse* de toutes les tâches apériodiques. Le vol de temps creux donne toutefois la valeur exacte de la laxité du système au plus tôt.

Implantabilité Le principal inconvénient de cette technique est qu'elle nécessite de garder en mémoire un grand nombre de données, nombre qui dépend de la taille de

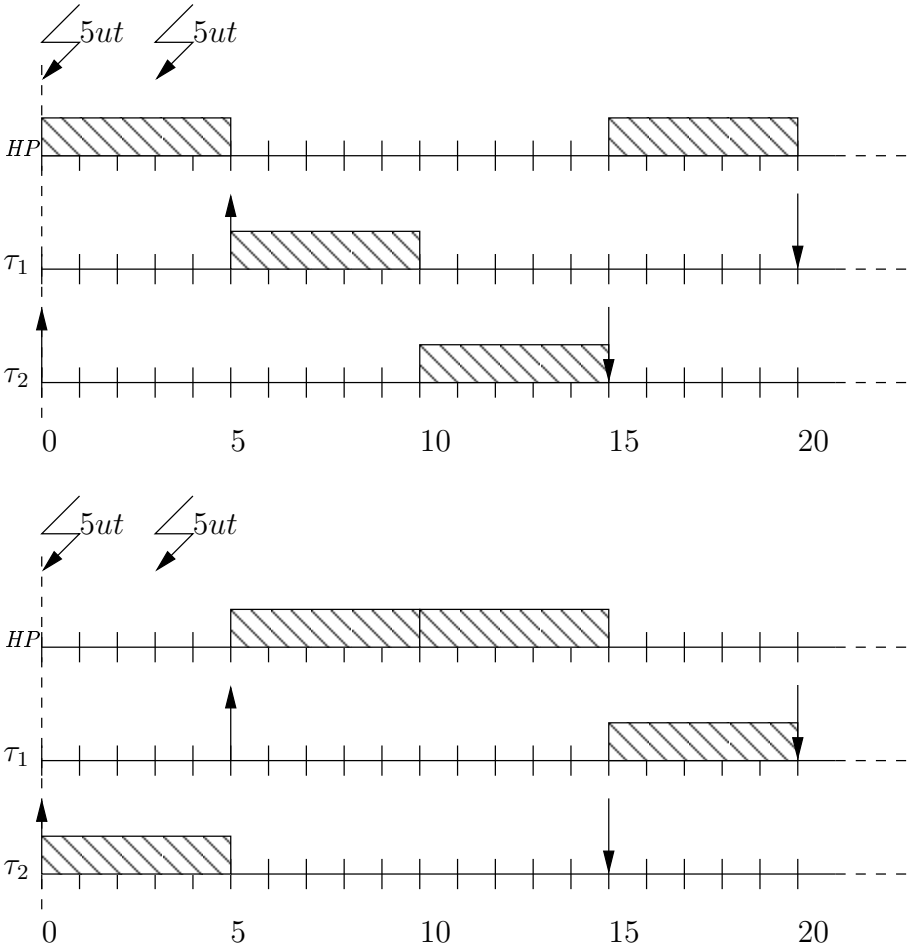


FIG. 4.3 – Non optimalité du vol de temps creux pour la minimisation des *temps de réponse* : en haut, la laxité est utilisée immédiatement, en bas, bien que de la laxité existe, on reporte l'exécution de la tâche apériodique ; pourtant, tout le trafic apériodique est servi au temps 15 en bas, alors qu'il faut attendre l'instant 20 en haut.

l'*hyperpériode*. Cette *hyperpériode* peut vite devenir très grande, dès que l'on considère plus d'une dizaine de tâches périodiques.

D'autre part, les auteurs ne proposent pas d'intégrer les temps de blocage dans le calculs des fonctions de laxité. Cet algorithme en l'état ne fonctionne donc pas avec des systèmes de tâches périodiques partageant des ressources.

Par ailleurs, si le système est composé pour partie de tâches sporadiques temps réel dur, il n'est pas possible de récupérer le temps non utilisé par une tâche sporadique qui arrive plus tard que sa date d'activation au plus tôt pour le traitement des tâches apériodiques. Ce problème est résolu dans [RTL93].

Comme le calcul des fonctions de laxité pour chaque niveau de priorité s'effectue hors ligne, il est également impossible d'y intégrer le temps non utilisé par une exécution plus rapide que prévu d'une tâche périodique. De façon générale, tout événement dynamique modifiant la laxité d'une tâche ne peut être pris en compte.

La complexité de la partie en ligne dépend du trafic apériodique, puisque la laxité doit être recalculée ($\mathcal{O}(n)$) à chaque fin de tâche périodique si la file des apériodiques n'est pas vide et à chaque arrivée de tâches apériodiques si la file est vide. On pourrait proposer une solution approchée qui recalcule systématiquement la laxité à chaque fin de tâche périodique, et uniquement à ce moment là, ce qui permet d'inclure le coût du calcul dans le coût des tâches périodiques. Ou bien, comme le proposent Davis, Tindell et Burns dans [DTB93], ajouter une tâche périodique chargée du calcul de la laxité.

4.6 Vol de temps creux dynamique

Le vol de temps creux présenté par Lehockzy et Ramos-Thuel se sert, comme on l'a vu, du calcul hors ligne des fonctions de laxité (cumulée) des tâches sur l'ensemble de l'*hyperpériode*. Cette solution, améliorée, permet de voler du temps creux à des tâches sporadiques temps réel dur, mais exclut de considérer une gigue d'activation pour les tâches périodiques ou encore des systèmes avec des ressources partagées entre tâches. De plus, le passage à l'échelle sur des systèmes comprenant un grand nombre de tâches est délicat en raison de la complexité en mémoire.

Davis, Tindell et Burns proposent dans [DTB93] un autre algorithme de vol de temps creux, qui permet de calculer la laxité dynamiquement, et par conséquent de prendre en compte les particularités des cas ci-dessus.

4.6.1 Notations

- X_0 : équivalent à $\max(0, X)$;
- $lp(i)$: priorités plus faibles ou équivalentes à i (i.e. $\{k \in \mathbb{N} \backslash k \geq i\}$) ;
- $hp(i)$: priorités plus fortes que i (i.e. $\{k \in \mathbb{N} \backslash k < i\}$) ;
- $l_i(t)$: la date de la dernière activation de la tâche τ_i ;

- $x_i(t)$: la date de la prochaine activation de la tâche τ_i ;
- $d_i(t)$: la date de la prochaine échéance de la tâche τ_i ;
- $c_i(t)$: le temps d'exécution restant pour l'instance courante de la tâche τ_i ;
- $S_i(t)$: la laxité du processeur au niveau de priorité i au temps t ou encore les temps creux disponibles à la priorité i dans l'intervalle $[t, d_i(t)[$, c'est-à-dire le temps disponible dans cet intervalle pour des tâches de priorités inférieures à i ;
- $S(t)$: la laxité du processeur au temps t, soit le temps disponible à l'instant t à la plus haute priorité pour que toutes les tâches respectent leur échéance.

Notons que dans le cas d'une tâche sporadique, si elle n'est pas active, $x_i(t)$ et $d_i(t)$ sont donnés dans l'hypothèse où la tâche arrive à sa fréquence maximale.

4.6.2 Calcul des temps creux à chaque niveau de priorité

L'algorithme de vol de temps creux dynamique repose sur le calcul au temps t à chaque niveau de priorité i de $S_i(t)$, la durée pendant laquelle la tâche de priorité i peut être suspendue avant qu'elle ne rate une échéance. Cette durée est en fait équivalente au nombre d'unités de temps non utilisées à une priorité supérieure ou égale à i entre t et la prochaine échéance de τ_i.

Pour servir une tâche à la plus haute priorité en garantissant les échéances de toutes les tâches temps réel dur, il faut que $S_i(t)$ soit non nul pour tous les niveaux de priorité. La quantité de temps creux « volable », c'est-à-dire disponible immédiatement pour un service à la plus haute priorité, est alors $S(t) = \min_{\forall i} S_i(t)$.

Alg. 3 Déterminer $S_i(t)$

$S_i(t) \leftarrow 0$
$w_i^{m+1}(t) \leftarrow 0$
repeat
  $w_i^m(t) \leftarrow w_i^{m+1}(t)$
  $w_i^{m+1}(t) \leftarrow S_i(t) + \sum_{\forall j \leq i} \left(c_j(t) + \left\lceil \frac{(w_i^m(t) - x_j(t))_0}{T_j} \right\rceil C_j \right)$
  if $w_i^m(t) = w_i^{m+1}(t)$ **then**
    $S_i(t) \leftarrow S_i(t) + \nu_i(t, w_i^m(t))$
    $w_i^{m+1}(t) \leftarrow w_i^{m+1}(t) + \nu_i(t, w_i^m(t)) + \varepsilon$
  end if
until $w_i(t)^{m+1} \leq d_i(t)$
return $S_i(t)$

À partir de ces données, l'algorithme 3 permet de calculer $S_i(t)$. Cet algorithme repose sur le constat que l'intervalle d'étude, $[t, d_i(t)[$, peut être vu comme une succession de périodes d'activité de niveau supérieur ou égal à i suivies de périodes d'activité de niveau inférieur à i. L'algorithme calcul alors $w_i^{m+1}(t)$ le prochain point d'inactivité (au niveau i) du processeur, puis $\nu_i(t, w_i^{m+1}(t))$ la durée de cette inactivité qui sera ajoutée au temps

creux de niveau i en cours de calcul, puis le prochain point d'inactivité et ainsi de suite. La durée de l'inactivité est donnée par l'équation 4.11.

$$\nu_i(t, w_i(t)) = \min\left((d_i(t) - w_i(t))_0, \min_{\forall j \geq i}\left(\left\lceil \frac{w_i(t) - x_j(t)}{T_j} \right\rceil_0 T_j + x_j(t) - w_i(t)\right)\right) \quad (4.11)$$

4.6.3 Algorithme de vol de temps creux dynamique

Dans la version statique du vol de temps creux, les tâches temps réel souple apériodiques sont exécutées à la priorité maximale lorsque du temps creux est disponible à tous les niveaux de priorité. Cette approche n'est plus possible ici car on veut prendre en compte les systèmes acceptant des tâches temps réel dur sporadiques. En effet s'il existe une tâche sporadique de très haute priorité avec une échéance égale à son coût et une très forte fréquence d'arrivée, il n'y aura jamais de temps creux disponible alors que cette tâche peut ne jamais survenir.

La version dynamique de l'algorithme consiste donc, lorsqu'une tâche temps réel souple J_a arrive, à calculer les temps creux disponibles à chaque niveau de priorité inférieur ou égal à k, où k est la priorité de la tâche temps réel dur exécutable la plus prioritaire. Une tâche est exécutable lorsqu'elle a été activée et placée dans la file d'attente de l'ordonnanceur. S'il existe des temps creux à tous ces niveaux de priorité, la tâche temps réel souple peut préempter la tâche de priorité k. La condition pour ordonnancer J_a à la priorité k est donnée par l'équation 4.12. La tâche devra être interrompue (ou la quantité de temps creux recalculée) après $\min_{\forall i \geq k} S_i(t)$ unités de temps (ut).

$$\min_{\forall i \geq k} S_i(t) > 0 \quad (4.12)$$

Cet algorithme nécessite potentiellement de recalculer les temps creux disponibles à tous les niveaux de priorité à chaque unité de temps. Ce n'est évidemment pas envisageable en pratique. Pour réduire les opérations nécessaires, on va maintenant chercher à exprimer les temps creux disponibles au temps t' en fonction des temps creux disponibles au temps t. Pour cela, il faut analyser les situations qui font varier la quantité de temps creux disponible.

Le premier cas de figure est celui où aucune tâche temps réel dur ne termine son exécution dans l'intervalle $[t, t'[$. Il existe trois possibilités : soit le processeur est en train d'exécuter des tâches temps réel souple, soit il est inactif, soit il exécute la tâche temps réel dur τ_i. S'il est inactif ou s'il exécute des tâches temps réel souple, alors les temps creux disponibles à tous les niveaux de priorité sont réduits de $(t' - t)$. En revanche, s'il exécute τ_i, il est par définition inactif aux priorités plus fortes, et par conséquent les temps creux disponibles à tous les niveaux de priorité supérieurs à i (tous les k tels que $k < i$) sont réduits de $(t - t')$.

Le deuxième cas de figure est celui où une tâche temps réel dur, τ_i, termine son exécution à l'instant t'' dans l'intervalle $[t, t'[$. Tous les temps d'inactivité de niveau i présents

dans $[t, d_i(t)[$, avec $d_i(t) \geq t''$, le seront également dans l'intervalle $[t, d_i(t)+T_i[= [t, d_i(t'')[$, c'est-à-dire le nouvel intervalle à considérer pour le calcul des temps creux de niveau i. Par conséquent, la terminaison de τ_i ne peut pas diminuer la quantité de temps creux disponible, au contraire, cette dernière peut augmenter. Il faut donc recalculer les temps creux de niveau i chaque fois que la tâche τ_i termine une exécution. Afin de diminuer le coût du calcul, on peut initialiser la récurrence avec $S_i(t'') = S_i(t)$.

La procédure est alors :

1. si aucune tâche temps réel dur ne termine dans $[t, t'[$:
   (a) si le processeur est inactif ou en train d'exécuter des tâches temps réel souple on applique la formule 4.13,
   $$\forall j > 1 : S_j(t') = S_j(t) - (t' - t) \tag{4.13}$$
   (b) si le processeur est occupé par la tâche temps réel dur τ_i on applique la formule 4.14,
   $$\forall j < i : S_j(t') = S_j(t) - (t' - t) \tag{4.14}$$
2. si la tâche temps réel dur τ_i termine à l'instant $t'' \in [t, t'[$, il faut calculer $S_i(t'')$ avec l'algorithme 3 en prenant $S_i(t'') = S_i(t)$ comme valeur initiale.

Autrement dit, $S_i(t)$ diminue chaque fois que le processeur est occupé par une priorité inférieure à i et doit être recalculé lorsque le traitement d'une requête de τ_i se termine.

4.6.4 Influence des tâches sporadiques

Lorsque l'on souhaite prendre en compte dans le système des tâches sporadiques temps réel dur, un nouveau facteur qui peut modifier la quantité de temps creux disponible est introduit : les tâches sporadiques qui n'arrivent pas à leur fréquence maximale. Si la tâche τ_i n'est pas arrivée à l'instant t_2 alors qu'elle aurait pu arriver dès l'instant t_1, on a $d_i(t_1) = t_1 + D_i$ et $d_i(t_2) = t_2 + D_i$. Il en résulte $S_i(t_2) > S_i(t_1)$. Les équations 4.13 et 4.14 donnent une borne minimale pour $S_i(t_2)$ mais la valeur exacte reste à calculer avec l'algorithme 3 (que l'on peut initialiser avec la valeur de $S_i(t_1)$).

Si l'on veut utiliser la valeur exacte de $S_i(t_i)$, et que τ_i n'arrive jamais, il faut donc potentiellement la recalculer à chaque unité de temps.

4.6.5 Temps d'exécutions stochastiques

Lorsqu'une tâche termine son exécution en moins de temps que son pire temps d'exécution, du temps libre apparait dans le système, car l'analyse de faisabilité garantit son

ordonnançabilité dans le cas où toutes les tâches consomment la totalité de leur coût. On appelle *gain* le temps libéré par la terminaison précoce d'une tâche.

Pour identifier ce gain le plus tôt possible, on peut jalonner les tâches de bornes et mesurer le pire temps d'exécution entre les bornes. Ainsi, du gain peut être identifié a chaque borne durant l'exécution de la tâche [DSZ90, Aud93].

Il est très facile de modifier le vol de temps creux pour pouvoir tirer partie du temps inutilisé lorsque les tâches temps réel dur s'exécutent avec un coût inférieur à leur pire temps d'exécution (C_i). En fait, le *gain*, c'est-à-dire la différence entre le pire coût et le coût réel, peut être ajouté directement à $S_j(t)$ pour tout j supérieur ou égal à i [LRT92].

4.6.6 Gigues d'activation

Si une tâche τ_i est sujette à une gigue d'activation J_i, il suffit de ne prendre en compte que sa date d'arrivée au plus tôt. On modifie alors le calcul de $x_i(t)$, ce qui donne $x_i(t) = (l_i(t) + T_i - J_i)_0$.

4.6.7 Gestion de ressources partagées

Jusqu'à présent, nous avons supposé que les tâches ne partageaient pas de ressources, et ne subissaient donc pas blocages. Nous relâchons cette hypothèse dans cette sous section.

Partage de ressources entre tâches temps réel dur Nous considérons dans un premier temps l'influence du partage de ressource entre deux tâches temps réel dur sur le calcul de la quantité de temps creux disponible.

Chaque tâche τ_i peut donc pour chacune de ses invocations subir un temps de blocage borné par B_i, comme spécifié dans le modèle de tâche 3 où les tâches sont caractérisées par le sextuplet $(r_i, J_i, T_i, C_i, B_i, D_i)$.

La laxité de la tâche τ_i, donnée par l'algorithme 3 doit alors être décrémentée de B_i. La condition pour ordonnancer J_a à la priorité k donnée par l'équation 4.12 devient :

$$\min_{\forall i \geq k} (S_i(t) - B_i) > 0 \tag{4.15}$$

Notons alors que l'algorithme ne donne plus la valeur exacte de la quantité de temps creux disponible, mais une borne, puisque B_i est le temps de blocage maximal. Davis propose une méthode dynamique pour obtenir une borne plus précise, notée $b_i(t)$, qui se base sur l'hypothèse de l'utilisation du protocole *PCP*.

Partage de ressources entre tâches temps réel dur et tâches temps réel souple Comme l'exécution des tâches temps réel souple ne doit pas modifier la faisabilité du système de tâche, si on autorise une tâche apériodique à prendre un verrou, il faut s'assurer qu'elle le relâche avant d'être préemptée. Autrement dit, il faut s'assurer que la laxité est

supérieure au temps de blocage maximal d'une tâche apériodique avant d'autoriser son exécution.

La condition pour ordonnancer J_a à la priorité k donnée par l'équation 4.12 devient :

$$\min_{\forall i \geq k} (S_i(t) - B_i) > B_a \tag{4.16}$$

où B_a désigne le temps de blocage maximale de J_a donné par l'algorithme utilisé pour borner les inversions de priorité.

4.7 Vol de temps creux approché

Dans sa thèse de doctorat [Dav95], Davis propose deux algorithmes pour calculer une borne minimale sur la quantité de temps creux disponible, et en tire trois procédures, dont l'une est paramétrable, pour voler du temps creux avec une complexité réaliste.

4.7.1 Approximation statique des temps creux

Cette approximation repose sur le temps entre la terminaison et l'échéance de la prochaine activation d'une tâche τ_i, intervalle de temps que l'on note δ.

Si on dispose d'une fonction $S_i^C(\delta)$, qui à un intervalle associe la quantité de temps libre au niveau de priorité i, alors on peut facilement donner une borne minimale sur la quantité de temps creux dans le système à l'instant t.

En effet, à la fin d'une instance de τ_i, la quantité de temps creux disponible au niveau i est précisément $S_i^C(\delta)$.

On peut alors faire une estimation pire cas de la fonction $S_i^C(\delta)$. Lorsque la tâche τ_i se termine, aucune tâche plus prioritaire ne peut être active par définition. Par conséquent le pire scénario est celui où toutes les tâches plus prioritaires sont activées au moment où τ_i termine.

De plus, δ est compris dans l'intervalle $[T_i, D_i + T_i]$, correspondant à la date de terminaison au plus tôt et au plus tard d'une instance de τ_i.

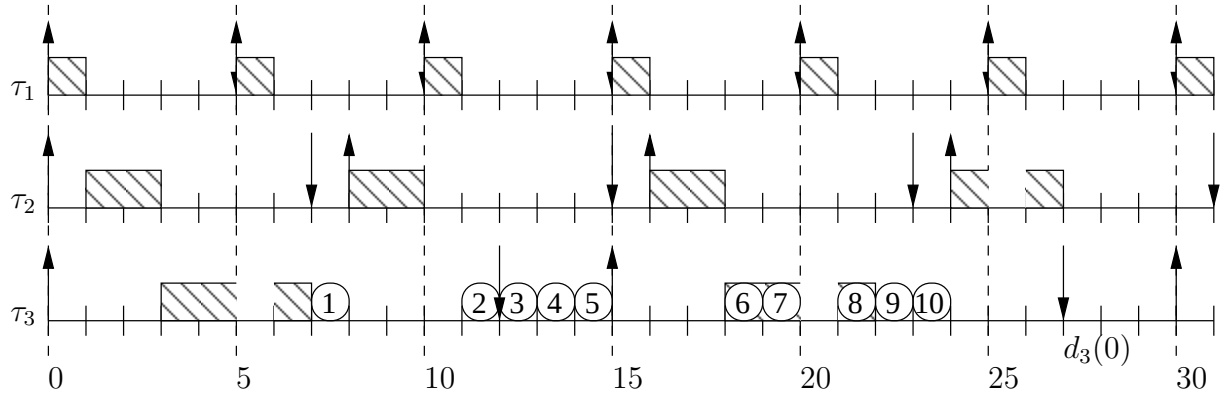
$S_i^C(\delta)$ est alors équivalent à la quantité de temps libre au niveau i dans l'intervalle $[0, \delta[$ avec le scénario pire cas d'activation au temps 0, et une seule activation de τ_i.

La figure 4.4 présente un exemple issu de [Dav95]. Les paramètres des tâches périodiques sont donnés par le tableau 4.4(a). On veut estimer $S_3^C(\delta)$ où δ peut prendre les valeurs entre T_3 et $T_3 + D_3$, c'est-à-dire dans l'intervalle $[15, 27]$. On construit alors la table 4.4(c). Les valeurs numériques entourées d'un cercle sur le diagramme indiquent le nombre d'unités de temps non utilisées aux priorités plus fortes que 3 entre la fin de l'exécution de la première instance de τ_3 (7) et sa prochaine échéance (27). Elles permettent de comprendre comment la table est construite.

L'algorithme qui utilise cette évaluation des temps creux s'appelle *Static Approximate Slack Stealing (SASS)**.

	Priorité	Période	Échéance	Coût
τ_1	forte	5	5	1
τ_2	moyenne	8	7	2
τ_3	basse	15	12	3

(a) Données



(b) Diagramme d'exécution

δ	15	16	17	18	19	20	21	22	23	24	25	26	27
$S_3^C(\delta)$	5	5	5	5	6	7	7	8	9	10	10	10	10

(c) Table des $S_3^C(\delta)$

FIG. 4.4 – Approximation statique de $S_i^C(\delta)$

4.7.2 Approximation dynamique des temps creux

Rappelons que la quantité de temps creux disponible au niveau i au temps t, $S_i(t)$, est égale au nombre d'unités de temps inutilisé au niveau i dans l'intervalle $[t, d_i(t)[$. La quantité de temps creux utilisable à la plus haute priorité est alors le minimum des $S_i(t)$ pour tout i.

Pour estimer cette quantité, on va calculer $I_i^j(a,b)$ l'interférence subie par τ_j causée par τ_i, une tâche plus prioritaire, dans l'intervalle $[a,b]$. Cette interférence est donnée par l'équation 4.17.

$$I_i^j(a,b) = c_i(t) + f_i(a,b)C_i + \min(C_i, (b - x_i(a) - f_i(a,b)T_i)_0) \tag{4.17}$$

La fonction $f_i(a,b)$ retourne le nombre de requêtes de τ_i qui peuvent commencer et s'exécuter complètement dans l'intervalle $[a,b]$. L'équation 4.18 permet de la calculer.

$$f_i(a,b) = \left\lfloor \frac{b - x_i(a)}{T_i} \right\rfloor_0 \tag{4.18}$$

L'interférence de τ_i dans l'intervalle $[a,b]$ est alors constituée de la quantité de calcul restant à traiter pour terminer l'éventuelle requête en cours, de $f_i(a,b)$ requêtes complètes et d'une dernière requête éventuellement incomplète.

Une borne minimale sur la quantité de temps creux au niveau i est alors donnée par

la longueur de l'intervalle d'étude, $d_i(t) - t$, moins la somme des interférences venant des tâches de priorité supérieure ou égale à i sur cet intervalle. Elle est donnée par l'équation 4.19.

$$S_i(t) = \left(d_i(t) - t - \sum_{\forall j \leq i} I_j^i(t, d_i(t)) \right)_0 \tag{4.19}$$

L'interférence d'une tâche se calcule en temps constant. L'interférence totale, et par conséquent $S_i(t)$, peut donc être estimée en temps linéaire. Par conséquent, une borne sur la quantité de temps creux disponible à la plus haute priorité s'obtient en temps quadratique.

Comme pour l'algorithme exact, $S_i(t)$ n'a besoin d'être recalculé que lorsqu'une requête de τ_i termine son exécution. Aux autres instants, $S_i(t)$ diminue de dt si le processeur a été occupé pendant une durée dt avec une tâche de priorité inférieure, et reste constante sinon. Ceci permet d'obtenir la borne sur la quantité de temps creux en temps linéaire à condition d'ajouter des opérations également en temps linéaire à chaque changement de contexte.

L'approximation vient du fait que l'on considère l'interférence maximale qu'une tâche plus prioritaire va pouvoir causer. Or si cette tâche n'est elle même pas la plus prioritaire, elle va subir une interférence qui sera déjà prise en compte lorsque l'on somme les interférences venant de toutes les tâches plus prioritaires.

Alg. 4 Échéance effective de τ_i

```
e_i ← d_i(t)
for all j < i do
  if x_j(t) + f_j(t, e_i)T_j + C_j ≥ e_i then
    e_i ← x_j(t) + f_j(t, e_i)T_j
  end if
end for
return e_i
```

L'effet de bord peut être diminué en calculant e_i, *l'échéance effective* de τ_i c'est-à-dire la date à partir de laquelle le processeur ne sera plus occupé que par des tâches de priorité supérieure à i jusqu'à la prochaine échéance de τ_i. La procédure 4 permet de calculer e_i en temps linéaire. Cette procédure examine tour à tour toutes les tâches τ_j de plus forte priorité que τ_i. Si la date de terminaison au plus tôt de la dernière instance de τ_j activée dans l'intervalle d'étude est plus grande que e_i, alors on sait qu'entre la dernière activation de τ_j et e_i, le processeur sera occupé à traiter des tâche plus prioritaires que τ_i. On peut donc assigner cette valeur à e_i. L'intervalle d'étude pour calculer $S_i(t)$ devient alors $[t, e_i[$.

L'algorithme qui utilise cette évaluation des temps creux s'appelle *Dynamic Approximate Slack Stealing (DASS)**, avec ou sans l'amélioration apportée par le calcul de l'échéance effective.

Deuxième partie

Java pour le temps réel

Java est un langage de programmation fortement orienté objet développé par Sun Microsystems dont la première version est parue en 1995. Ce langage a la spécificité d'être semi compilé dans un langage assembleur simplifié appelé *bytecode**. Ce code n'est pas directement exécutable par un processeur traditionnel, mais peut être interprété par une *machine virtuelle Java (JVM)**. Ceci permet à un programme écrit en Java et compilé en *bytecode* d'être exécuté sur n'importe quelle architecture, pourvu qu'une *JVM* y soit présente. La philosophie qui a fait naitre Java est souvent résumée par le slogan *Write Once, Run Anywhere (WORA)** : écrivez une seule fois, exécutez partout.

Cette *JVM* gère également automatiquement la mémoire au travers d'un mécanisme appelé ramasse miette ou *Garbage Collector (GC)**. Cette gestion automatique simplifie énormément le travail du programmeur et permet d'éviter un grand nombre d'erreurs. En s'associant à la très grande portabilité du langage elle a assuré à Java une très forte popularité.

Depuis sa version 2, Java est décliné sous plusieurs variantes selon le type d'architecture visé : *Java Micro Edition (ME)* pour les environnements disposant de peu de ressources, *Java Standard Edition (SE)* pour les machines de bureau et Java *Enterprise Edition (EE)* ciblant les applications distribuées et la programmation web.

Depuis mai 2007, Java est entièrement disponible sous la licence *GPL*, et est donc un logiciel libre.

Toutes ces caractéristiques ainsi que l'existence de nombreux *environnements de développement intégrés (IDE)** performants comme la plateforme *Eclipse* ont permis de réduire les coûts de développement des logiciels. C'est donc naturellement qu'il intéresse aujourd'hui les communautés de développeurs d'applications embarquées et temps réel.

Chapitre 5

Java n'est pas un langage pour le temps réel

Les nombreux avantages qui ont permis à Java de s'imposer dans de nombreuses branches de l'informatique lui ont valu un intérêt particulier de la part des concepteurs de systèmes temps réel. Pourtant Java n'est pas adapté à la programmation temps réel, principalement pour les raisons suivantes :

- la mémoire est gérée automatiquement par un *GC* ;
- l'ordonnancement ne respecte pas les priorités ;
- les inversions de priorité dues aux synchronisations ne sont pas bornées ;
- il n'y a pas de mécanisme de contrôle de charge ;
- des optimisations dynamiques sont mises en œuvre.

5.1 La gestion de la mémoire

Un programme informatique a besoin de se voir allouer une part de la mémoire du système pour stocker les structures de donnée qu'il utilise. Pour cela, un exécutable peut demander au système de lui réserver une plage d'adresses mémoire qu'il sera seul à utiliser. C'est *l'allocation*.

Si le programme est amené à s'exécuter longtemps, et s'il peut redemander de la mémoire durant son cycle de vie (allocation *dynamique*), il devient nécessaire de libérer les zones mémoires réservées qui ne seront plus utilisées.

En langage C, le programmeur peut obtenir de la mémoire de façon statique en utilisant des structures de donnée de tailles fixées avant la compilation, ou faire appel explicitement à une primitive d'allocation de mémoire, qui lui retourne un pointeur. Dans ce cas, c'est à lui de veiller à libérer la mémoire pointée lorsqu'elle n'est plus utile. S'il oublie cette étape, et si le pointeur vers la zone mémoire allouée disparait du contexte d'exécution, il n'y a plus de moyen de libérer la mémoire avant la terminaison du programme : c'est une fuite de mémoire. Les fuites de mémoire sont la source de nombreux bogues célèbres

y compris dans des systèmes dont la mise en œuvre a profité de moyens importants. Les bogues liés à la gestion de la mémoire sont en effet difficiles à détecter et il est rarement aisé d'en retrouver l'origine.

En Java, de la mémoire est allouée pour la *JVM* qui va la répartir entre les applications Java qu'elle héberge. Les objets sont alloués dans une structure de donnée de type *tas*. Les champs d'une classe qui ne sont pas des types primitifs du langage (`int`, `boolean`, `float` ...) sont des références vers une autre partie du tas où se trouve l'objet modélisant le champ.

En plus du tas, une pile contenant des références vers les objets manipulés est créée lorsqu'une méthode est invoquée. Cette pile est détruite, ainsi que les références qu'elle contient, lorsque la méthode se termine, et une référence sur l'objet représentant le retour de la méthode est placée sur la pile de la méthode appelante. Il peut ainsi arriver qu'il n'existe plus aucune référence vers un objet, qui occupe pourtant de la mémoire.

Les objets sont construits explicitement par un appel à leur constructeur (mot clé `new`), mais il n'existe pas de destructeurs, comme en C++, ni de primitive permettant de libérer la mémoire associée à un objet. C'est la machine virtuelle qui se charge de rechercher les objets du tas qui ne sont plus accessibles et de les détruire. Un objet est accessible si une référence sur celui-ci est présente sur une pile ou possédée par un autre objet accessible.

Un processus léger particulier, le *GC*, est responsable de rechercher ces objets. Durant cette collecte automatique, le *GC* peut préempter toutes les autres tâches pour une durée non bornée, car la complexité de l'algorithme dépend entre autre de l'état de la mémoire, de la taille du tas et du nombre de références cycliques. Ceci n'est pas acceptable pour un système temps réel dur, puisque l'exécution du *GC* n'est pas intégrable dans l'analyse de faisabilité.

Le programmeur Java n'a pas à se soucier du nettoyage de la mémoire, ce qui permet d'éviter de nombreuses erreurs de développement difficiles et coûteuses à détecter. Si c'est une caractéristique qui a grandement contribué à rendre ce langage populaire, elle parait incompatible avec une approche temps réel dur.

Les solutions possibles Trois approches, qui peuvent d'ailleurs être combinées, sont possibles pour résoudre ce problème :

1. Ne pas nettoyer la mémoire, c'est-à-dire désactiver le *GC*. La spécification d'une machine virtuelle Java n'impose en effet pas que la mémoire allouée par le programme soit libérée avant la fin de son exécution. Si cette solution est retenue, il faudra veiller à créer le minimum d'objets temporaires.
2. Changer l'algorithme du *GC* pour permettre qu'il soit préempté immédiatement sans laisser la mémoire dans un état incohérent. On parle alors de *GC temps réel (RT-GC)**. Le lecteur intéressé par les algorithmes de *GC* temps réel pourra consulter [Sie02]

3. Modifier la spécification de la machine virtuelle pour ajouter de nouvelles zones de mémoire non gérées par un *GC*.

5.2 Non respect des priorités

Java propose en théorie jusqu'à dix niveaux de priorité. Pourtant, rien dans la spécification de la *JVM* ne prévoit que ces priorités soient utilisées pour l'ordonnancement. En pratique les *processus légers (threads)** sont même souvent des *threads* de l'*OS* sous-jacent, ordonnancés par celui-ci. La priorité affectée au *thread* par Java pourra éventuellement être utilisée lors de la construction du *thread* système correspondant pour paramétrer l'algorithme de *Round Robin* (*RR*) utilisé par la plupart des *OSs*, mais sans aucune obligation ni aucune précision sur la manière d'utiliser cette valeur de priorité. Ceci n'est pas compatible avec une approche temps réel.

Les solutions possibles Utiliser une machine virtuelle qui ordonnance les *threads* selon une politique temps réel est l'unique solution. Cela suppose également que l'*OS* sur lequel tourne la machine virtuelle implante cette politique. Cela pose toutefois un vrai problème de compatibilité ascendante des programmes et des *API** Java. En effet, la majorité des applications concurrentes développées sont testées dans un contexte où l'ordonnancement empêche les situations de famine. Même si cela n'est pas spécifié par le langage, les développeurs Java présument bien souvent que l'ordonnanceur est basé sur un *RR*. Si l'on veut pouvoir exécuter des programmes Java normaux sur la même *JVM* que des programmes temps réel, il faudra alors permettre le choix de la politique d'ordonnancement.

5.3 Inversion de priorités

Une inversion de priorité se produit lorsqu'une tâche de forte priorité τ_h est bloquée par une tâche de faible priorité τ_l car elles partagent une ressource. Par exemple dans le scénario suivant : τ_l est la seule tâche activée et commence à utiliser la ressource partagée. Elle obtient un verrou sur la ressource. Lorsque τ_h est activée, elle préempte τ_l, mais lorsqu'elle veut utiliser la ressource, elle est mise en attente car le verrou est détenu par τ_l. On a bien une inversion des priorités puisque τ_h doit attendre la fin de l'exécution d'une tâche moins prioritaire τ_l.

Ce scénario ne pose aucune difficulté particulière, puisque l'on peut borner le temps de blocage. Pourtant, si l'on rajoute une tâche de priorité intermédiaire τ_m activée en même temps que τ_h le problème se complique. Dans ce cas, lorsque τ_h sera bloquée par sa demande de la ressource occupée par τ_l, la tâche active de plus forte priorité sera τ_m, et c'est donc elle qui va occuper le processeur, bien qu'elle ne partage aucune ressource avec τ_h et qu'elle soit moins prioritaire. Dans ce cas il n'y plus aucun moyen de borner le temps d'attente de τ_h : on parle d'inversion de priorité non bornée.

Les solutions possibles Pour résoudre ce problème il convient de mettre en place un mécanisme avancé de gestion des verrous. Nous évoquions déjà ces algorithmes dans le chapitre 3. Java ne gérant en pratique pas les priorités, les plateformes standards ne proposent aucun de ces algorithmes. Par ailleurs, rien dans les *APIs* n'est prévu pour permettre le choix d'un algorithme particulier.

5.4 Pas de contrôle de la charge

Un système temps réel a besoin de prédictibilité. Pour cela, différentes techniques sont utilisées, comme nous l'avons vu dans la première partie de ce manuscrit : l'analyse de faisabilité pour des trafics périodiques, et le contrôle d'admission pour les trafics apériodiques. L'analyse de faisabilité peut se baser sur l'identification d'un scénario pire cas. Si ce scénario est compatible avec les contraintes temporelles, alors tous les scénarios seront valides. En l'absence de contrôle d'admission, il faut au moins pouvoir assurer que les tâches non périodiques ne perturbent pas le fonctionnement des tâches périodiques.

Tâches périodiques Il n'existe pas dans la classe `Thread` de Java de mécanisme permettant au programmeur de spécifier qu'il écrit un processus périodique. Une exécution pseudo périodique peut être obtenue à l'aide des méthodes `Thread.sleep(long)` ou `Object.wait(long)`. En plaçant une de ces instructions à la fin d'une boucle, le *thread* peut être bloqué pour un temps fixé périodiquement. L'activation n'est pas périodique, mais le temps entre la fin d'un traitement et l'activation du suivant sera fixé. Si le coût de la tâche est fixe ou ne varie pas beaucoup, et si le processus est seul, le comportement est périodique. Ce n'est plus le cas dès qu'il peut être préempté (par le *GC* par exemple). Certaines *APIs* de Java, comme les classes `Timer` et `TimerTask` permettent d'écrire du code activé périodiquement. Le mécanisme ne peut toutefois reposer que sur un appel à `Object.wait(long)`, lui même reposant sur une primitive qui rend la main au système en attendant d'être réveillée par celui-ci. Si l'*OS* n'offre pas de garanties temps réel, la *JVM* ne pourra pas faire mieux.

Tâches apériodiques Lorsque l'on écrit un *thread*, il n'est pas possible d'informer la *JVM* que son traitement correspond à un événement apériodique, et donc de ne pas l'exécuter en préférence d'un autre processus dont l'échéance stricte est menacée. Par ailleurs, aucun mécanisme dans la *JVM* ne permet de réagir automatiquement à une augmentation soudaine de la charge, en rejetant les tâches non temps réel dur par exemple.

Tâches sporadiques Enfin, il n'y a dans Java aucun mécanisme de contrôle de flux permettant de reporter l'exécution du traitement d'un événement si cet événement est trop fréquent. Autrement dit, il n'est pas possible d'assurer le comportement sporadique d'une tâche traitant un événement apériodique.

Les solutions possibles Le modèle de *thread* de Java doit être étendu afin de pouvoir spécifier les contraintes temporelles propres à chaque type de trafic temps réel possible. De plus, la *JVM* ne saurait en aucun cas offrir des garanties temporelles que l'*OS* sous-jacent n'offre pas. Il faudra donc exécuter une *machine virtuelle Java temps réel (RTJVM)** sur un *OS temps réel (RTOS)**.

5.5 Les optimisations dynamiques dans Java

L'exécution de Java sur une machine virtuelle lui a apporté, nous l'avons vu, de nombreux avantages. Cependant, l'utilisation d'un *bytecode* interprété induit une surcharge de travail au moment de l'exécution qui a longtemps nui à la réputation des performances du langage. De plus, les optimisations possibles sur des architectures processeur particulières lorsque l'on compile vers un langage machine ne le sont plus.

Pour améliorer les performances du langage, des optimisations sont proposées. Elles reposent sur l'étude dynamique du programme et l'utilisation des informations collectées pour améliorer son fonctionnement.

Par exemple, la compilation à la volée, *Just In Time (JIT)**, introduite par le langage *SmallTalk*, permet de transformer en langage machine natif les parties du programme les plus intensivement exécutées : cela introduit un surcoût au moment de la compilation, mais les performances sur les exécutions suivantes du morceau de code compilé rattraperont ce surcoût.

L'utilisation d'un *JIT* résume bien la philosophie de ces améliorations dynamiques : améliorer les performances, au détriment de la prédictibilité, ce qui est en quelque sorte la démarche opposée à celle de l'analyse temps réel.

Le lecteur intéressé par les optimisations dynamiques dans les langages interprétés pourra consulter [AFG+04].

Les solutions possibles Toutes les optimisations dynamiques qui font perdre de la prédictibilité à l'exécution du programme doivent pouvoir être désactivées pour les *threads* temps réel sur une *RTJVM*.

Il n'est donc pas possible en Java de privilégier le trafic périodique, d'autoriser le trafic apériodique sans risque, ni de traiter sporadiquement une tâche liée à un événement apériodique. Par conséquent, aucun des modèles de tâche présenté au chapitre 1 n'est implantable avec Java. De plus la philosophie de Java de très grande portabilité, dont la perte en performance est compensée par l'utilisation d'optimisations avancées lors de l'exécution, ainsi que sa gestion automatique de la mémoire semble être autant d'obstacles pour son utilisation dans un environnement temps réel.

Pour surmonter ces obstacles, il faudra disposer d'une *JVM* particulère, une *RTJVM*, proposant une gestion alternative de la mémoire, un algorithme d'ordonnancement temps réel, un algorithme permettant de borner les inversions de priorité et des mécanismes de contrôle de charge. Les optimisations dynamiques faisant baisser la prédictibilité de l'exécution doivent par ailleurs être désactivées. La plupart de ces pré requis impliquent également que cette *RTJVM* s'exécute sur un *RTOS*.

Chapitre 6

La spécification Java pour le temps réel (RTSJ)

Vers une spécification

Jusqu'en 1997, écrire une application temps réel entièrement en Java semblait complètement irréaliste, pour toutes les raisons évoquées au chapitre précédent. Cependant, il était déjà possible d'incorporer du Java dans de telles applications. La partie temps réel dur pouvait être programmée en C, tandis que les autre parties de l'application ne nécessitant pas de garanties temps réel pouvait l'être en Java.

À cette époque, l'unique approche explorée pour améliorer les capacités temps réel de Java était son interprétation au niveau matériel. La société *aJile*, par exemple, fournissait (et fournit toujours) un processeur capable d'interpréter directement du *bytecode*. Toutefois, cette solution n'est basée que sur les *APIs* de *Java Micro Edition**.

Jusqu'en 1998, trois groupes d'experts travaillaient de façon indépendante sur un support temps réel pour Java :

- un groupe était mené par Kelvin Nilsen et Lisa Carnahan pour *NewMonics* et le *National Institute for Standards and Technology (NIST)** ;
- Greg Bollella, qui travaillait alors pour *IBM*, dirigeait un projet similaire pour le compte d'un consortium de compagnies intéressées par la technologie Java pour les systèmes temps réel ;
- un troisième effort était mené en interne au sein de *Sun Microsystems*.

Ces trois efforts convergent en 1998 pour aboutir à la constitution du groupe d'experts chargé du tout premier *Java Specification Request (JSR)**, qui vise à clarifier les besoins pour construire une plateforme Java temps réel. Il intègre à l'époque James Gosling (le *père* de Java), qui choisira de se retirer lorsque Greg Bollella rejoindra les équipes de Sun *Microsystems*, et réunit un panel d'entreprises dont *Sun* bien sûr, mais aussi *IBM*, *NewMonics* et bien d'autres acteurs de l'industrie temps réel.

Ce groupe livre un premier document en 1999, que l'on peut considérer comme la

première version de *RTSJ*. La première version officielle est publiée en Juin 2000 [GB00], en même temps qu'une première implantation confiée à *Timesys* appelée *implantation de référence (RI)**. Cette implantation de référence a permis d'identifier un certain nombre de bogues et d'incohérences. La spécification est donc révisée une première fois en Août 2004 (version 1.0.1), puis à nouveau en Juillet 2006 (version 1.0.2).

Ces deux révisions visaient à résoudre des problèmes mineurs, mais à mesure que la spécification gagnait en maturité et que des implantations commerciales voyaient le jour (*Jamaïca VM** de *Aïcas*, *PERC** Pico et *PERC* Raven de *Aonix* etc.), des demandes de modifications plus importantes ont émergé. Celles-ci font actuellement l'objet d'un deuxième *JSR* sur *RTSJ*, le numéro 282. On retrouve de nouveau au sein du groupe d'experts chargé de cette évolution beaucoup d'industriels, mais également des universitaires, comme Andy Wellings (Université de York).

Ce chapitre présente les solutions retenues dans *RTSJ* pour répondre aux problèmes évoqués lors du chapitre précédent. Pour aller plus loin dans la compréhension de cette spécification, le lecteur pourra se référer aux ouvrages d'Andy Wellings [Wel04] et de Peter Dibble [Dib08].

6.1 Les principes directeurs

L'objectif de *RTSJ* est double : étendre les *APIs* de Java pour faciliter la programmation d'applications temps réel et spécifier les caractéristiques d'une *RTJVM*.

Ses principes directeurs sont les suivants :

- *application à tous les environnements Java* ; la spécification ne doit rien inclure qui limiterait son utilisation à un environnement Java particulier, comme *J2ME* ;
- *compatibilité ascendante* ; Les programmes qui respectent la spécification de Java doivent être exécutés correctement sur une implantation de *RTSJ* ;
- *portabilité* ; la spécification doit s'efforcer de respecter le principe *WORA*, c'est à dire la portabilité du *bytecode* produit, mais jamais au prix d'une perte de prédictibilité – *WORA* devient *Write Once Carefully, Run Anywhere Maybe (WOCRAM)** ;
- *évolutivité* ; la spécification n'est pas figée et doit être suffisamment modulaire pour permettre l'implantation d'algorithmes avancés ;
- *prédictibilité* ; Chaque fois qu'un compromis doit être fait, la priorité doit toujours être donnée à la solution offrant la plus forte prédictibilité, même si c'est aux dépens des performances ;
- *syntaxe* ; la syntaxe de Java reste inchangée, pas de nouveau mot clef, un compilateur Java classique doit être capable de produire du *bytecode RTSJ* ;
- *implantation* ; la spécification autorise une certaine flexibilité dans son implantation ; elle définit certaines caractéristiques minimales pour une *RTJVM*, mais également des caractéristiques optionnelles.

RTSJ entend améliorer les capacités temps réel de Java sur les sept points suivants :

1. ordonnancement des *threads* ;
2. gestion de la mémoire ;
3. synchronisation et partage de ressources ;
4. traitements d'événements asynchrones ;
5. transfert de contrôle asynchrone ;
6. arrêt asynchrone d'un *thread* ;
7. accès à la mémoire physique.

Nous détaillons certains de ces points dans la suite de ce chapitre. Toutes les classes proposées par *RTSJ* sont dans le paquetage `javax.realtime`.

6.2 Gestion des Priorités et entités ordonnançables

Le cahier des charges de *RTSJ* spécifie qu'une *RTJVM* doit être capable d'exécuter du code Java non temps réel. Aussi le modèle d'ordonnancement de Java est-il conservé : l'entité ordonnançable de base, le *thread*, est toujours disponible au travers de la classe `Thread` et il est possible de lui assigner dix niveaux de priorité différents. Les `Thread`s possédant une priorité entre 1 et 10 seront alors exécutés selon le modèle traditionnel de Java. Il s'agit des tâches non temps réel.

Pour les tâches nécessitant des garanties temps réel, *RTSJ* introduit une nouvelle entité ordonnançable, définie par l'interface `Schedulable`. Un objet qui implante cette interface verra ses besoins en ressource processeur gérés par un objet `Scheduler` (ordonnanceur en anglais). Les `Schedulable`, qui sont associés à des *threads*, ont tous une priorité supérieure à 10. La spécification demande un minimum de 28 niveaux de priorités suplémentaires pour les *threads* associés à des `Schedulable`s.

RTSJ propose deux classes implantant l'interface `Schedulable` : `RealtimeThread` et `AsyncEventHandler` (*AEH**). La première est une sous classe de `java.lang.Thread`, et son exécution est donc toujours associée à un unique *thread*. La seconde représente un traitement qui peut être associé à un ou plusieurs événements, interne(s) ou externe(s). Un *thread* est construit dynamiquement, ou pris dans un réservoir (*pool*) pour prendre en charge son exécution. Une sous classe, `BoundAsyncEventHandler`, permet de lui dédier un unique *thread*. Le coût en mémoire est alors plus important mais cela évite une pénalité en temps due à la construction dynamique du *thread*. Les événements auxquels peuvent être attachés les *AEHs* sont eux mêmes représentés avec *RTSJ* par la classe `AsyncEvent` (*AE**). S'il est possible d'associer un *AEH* à plusieurs *AEs*, il est également possible d'associer plusieurs *AEHs* à un *AE*. La classe *AE* possède une méthode `fire()` qui provoque l'activation immédiate de tous les *AEHs* qui lui sont associés, à leur priorité respective.

La politique d'ordonnancement pour les priorités temps réel (supérieures à 10) est implantée par défaut par la classe `PriorityScheduler` qui étend `Scheduler`. Elle définit une

politique préemptive non oisive à priorités fixes. D'autres algorithmes peuvent toutefois être implantés et utilisés, mais nous verrons dans la suite de ce manuscrit que l'écriture d'un ordonnanceur indépendant du système d'exploitation sous-jacent pose de nombreux problèmes. Par conséquent, si l'on veut s'assurer de la portabilité de l'application, il est préférable de se limiter à l'ordonnanceur à priorités fixes imposé par la spécification. Notons toutefois que dans [ZW06], les auteurs proposent d'implanter un ordonnanceur *EDF* au dessus de l'ordonnanceur à priorités fixes. Cette approche induit un surcoût pour l'ordonnancement mais permet d'utiliser un ordonnanceur à priorités dynamiques sur toutes les *RTJVMs*.

Deux classes de paramètres doivent être attachées à une entité ordonnançable temps réel : les paramètres nécessaires à l'ordonnanceur pour décider de son éligibilité lorsqu'elle est active (la priorité par exemple), et les paramètres relatifs à son mode d'activation (la période, le pire temps d'exécution et l'échéance pour une tâche périodique par exemple). Les premiers sont modélisés par la classe `SchedulingParameters`, les seconds par `ReleaseParameters`.

Seul un type de `SchedulingParameters` est fourni par *RTSJ* : `PriorityParameters`. Notons que l'utilisation de la méthode `setPriority()` d'un `RealtimeThread`, héritée de la classe `Thread`, doit être équivalente à construire un nouveau `PriorityParameters`.

Les `ReleaseParameters` sont séparés en deux sous classes : `PeriodicParameters` et `AperiodicParameters`. Nous reviendrons sur ces deux classes dans la section 6.4.

6.3 Gestion de la mémoire

Un langage de programmation doit fournir au programmeur un moyen d'accès à la mémoire du système. Une partie de cette mémoire est attribuée au programme chaque fois qu'il en fait la demande (allocation).

La quantité de mémoire disponible n'étant pas illimitée, il doit exister un moyen de rendre la mémoire associée aux structures qui ne seront plus utilisées au système, afin qu'elle puisse être ré-attribuée par la suite.

Trois approches sont possibles :

- laisser le programmeur libérer explicitement la mémoire. C'est le choix des langages comme le C et le C++ ; cette approche est facilement implantable, mais laisse la porte ouverte à des erreurs de programmation difficilement détectables ;
- la plateforme d'exécution libère la mémoire lorsqu'un test logique permet de déterminer qu'elle ne pourra plus être accédée ; cela nécessite que la sémantique du langage de programmation permette de séparer des contextes d'exécution ; lorsqu'une référence n'est plus dans le contexte d'exécution courant, elle n'est plus accessible, la mémoire peut être libérée ;
- la plateforme d'exécution inspecte la mémoire et détermine les zones qui ne sont plus référencées pour les libérer automatiquement ; c'est le mécanisme de *GC* ; l'avantage

sur la solution précédente est de pouvoir libérer des zones pointées par des références toujours dans le contexte d'exécution courant.

Java a retenu la troisième solution, mais celle-ci n'est pas adaptée à l'écriture de systèmes temps réel dur. De nombreuses recherches ont été menées sur les algorithmes de ramasse-miettes temps-réel, mais ces derniers n'étaient pas encore assez mûrs, ou n'ont pas su convaincre, lors de l'élaboration de *RTSJ*. Aussi, même si rien n'empêche une *RTJVM* d'implanter un tel mécanisme, *RTSJ* propose plutôt la deuxième approche. La mémoire est séparée en régions, modélisées par la classe abstraite `MemoryArea`. Lorsque l'on entre dans une zone de mémoire, toutes les allocations effectuées le seront dans cette zone. De nombreux types de zone sont définis, modélisés par les classes (C) et classes abstraites (CA) suivantes :

`HeapMemory` **(C)** Le tas classique de Java, sur lequel le ramasse-miettes (temps réel ou non) peut opérer.

`ImmortalMemory` **(C)** Une zone partagée par tous les *threads* dans laquelle on alloue les objets dont la durée de vie est la même que l'application. Les objets alloués dans la zone de mémoire immortelle ne subissent jamais de temps de latence dû au ramasse-miettes.

`ScopedMemory` **(CA)** Une zone mémoire à laquelle on accède explicitement par un appel de méthode ou en l'attachant à un `Schedulable` lors de sa création. Un compteur est attaché à chaque `ScopedMemory` qui est incrémenté chaque fois qu'un `Schedulable` y entre et décrémenté lorsqu'il en ressort. Si le compteur vaut 0 la zone mémoire peut être détruite. Cette classe est abstraite et autorise plusieurs implantations. Les `ScopedMemory` ne sont pas non plus collectées par le ramasse-miettes.

`VTMemory` **(C)** Une sous catégorie de `ScopedMemory` dans laquelle les allocations peuvent se faire en un temps variable.

`LTMemory` **(C)** Une sous catégorie de `ScopedMemory` dans laquelle les allocations doivent se faire en temps proportionnel à la taille de l'objet à allouer.

`ImmortalPhysicalMemory` **(C) et** `LTPhysicalMemory` **(C)** Ces deux classes étendent respectivement `ImmortalMemory` et `LTMemory` et permettent d'utiliser directement la mémoire de la plateforme, en dehors de la *RTJVM*.

Afin de pouvoir écrire des tâches temps réel dur, *RTSJ* définit une sous classe de `RealtimeThread` : `NoHeapRealtimeThread`. Le code de sa méthode `run()` ne doit contenir aucune référence d'objet présent dans la `HeapMemory`. Cette caractéristique peut être assurée à la compilation, et permet durant l'exécution de préempter sans délai le ramasse-miettes si celui-ci est en cours d'exécution lorsque la tâche est prête pour l'exécution. De la même

```
import javax.realtime.PriorityParameters;
import javax.realtime.PeriodicParameters;
import javax.realtime.RelativeTime;
import javax.realtime.RealtimeThread;

public class PeriodicRealtimeHello1 extends RealtimeThread {

  private void doPeriodic() {
    System.out.println("Hello");
  }

  public void run() {
    do{
      doPeriodic();
    }while (waitForNextPeriod());
  }

  public static void main(String[] args) {
    RelativeTime start = new RelativeTime(10,0);
    RelativeTime period = new RelativeTime(5,0);
    PeriodicRealtimeHello1 rtt = new PeriodicRealtimeHello1(
        new PriorityParameters(15),
        new PeriodicParameters(start, period));
    rtt.start();
  }

}
```

FIG. 6.1 – Écrire une tâche périodique avec un `RealtimeThread`

façon, il est possible de spécifier lors de la construction d'un *AEH* que ce dernier ne doit pas utiliser la `HeapMemory`. Ces deux classes peuvent dans tous les cas se voir associer à une `ScopedMemory` lors de leur construction.

6.4 Contrôle de la charge

Nous présentons dans cette section la façon dont *RTSJ* propose de contrôler les modèles d'arrivée des événements ou tâches. Nous avons vu que les tâches temps réel dur sont, pour la nécessité de l'analyse, des tâches périodiques ou sporadiques, c'est-à-dire que l'on peut restreindre le modèle d'arrivée pire cas à un modèle périodique. Les autres tâches du système peuvent avoir des contraintes plus souples, à condition que l'on puisse garantir que leur exécution ne perturbera pas les tâches temps réel dur.

Nous expliquons ici les paradigmes proposés par *RTSJ* pour écrire les tâches périodiques, sporadiques, et les autres tâches du système, ainsi que pour les intégrer à l'analyse de faisabilité du système.

6.4.1 Écrire une tâche périodique

Le traitement d'une instance d'une tâche périodique est une tâche réactivée périodiquement au cours du temps. En Java, un *thread* qui termine son exécution ne peut pas être redémarré. La solution adoptée est alors d'écrire le traitement périodique dans une boucle, à la fin de laquelle le *thread* suspend son exécution. Il faut alors un mécanisme externe

```
import javax.realtime.AsyncEventHandler;
import javax.realtime.PeriodicParameters;
import javax.realtime.PeriodicTimer;
import javax.realtime.PriorityParameters;
import javax.realtime.RelativeTime;

public class PeriodicRealtimeHello2 extends AsyncEventHandler {

  public PeriodicRealtimeHello(PriorityParameters priority,
      RelativeTime start, RelativeTime period) {
    super(priority, new PeriodicParameters(start, period), null, null,
        null, null);
  }

  public void handleAsyncEvent() {
    doPeriodic();
  }

  private void doPeriodic() {
    System.out.println("Hello");
  }

  public static void main(String[] args) {
    RelativeTime start = new RelativeTime(10, 0);
    RelativeTime period = new RelativeTime(5, 0);
    PeriodicTimer timer = new PeriodicTimer(start, period,
        new PeriodicRealtimeHello2(new PriorityParameters(15),
          start, period));
    timer.start();
  }

}
```

FIG. 6.2 – Écrire une tâche périodique avec un `AsyncEventHandler` et un `PeriodicTimer`

pour réveiller le processus lors de la prochaine occurrence périodique de l'événement dont il assure le traitement.

La classe `RealtimeThread` possède une méthode statique, `waitForNextPeriod()`, dont l'appel est bloquant. C'est la *RTJVM* qui va débloquer la méthode périodiquement. La méthode `run()` doit alors exécuter une boucle infinie, à la fin ou au début de laquelle la méthode `waitForNextPeriod()` est appelée, suspendant l'exécution du *thread* jusqu'à la prochaine période.

La période est spécifiée grâce à l'utilisation de la sous classe de `ReleaseParameters`, `PeriodicParameters`. Si la méthode `waitForNextPeriod()` est appelée sur un `RealtimeThread` qui ne possède pas de référence à un `ReleaseParameters` typé `PeriodicParameters`, l'exception `IllegalThreadStateException` est levée. Un exemple de *thread* temps réel périodique utilisant ce paradigme est proposée dans la figure 6.1.

Un comportement périodique peut également être obtenu avec l'autre entité ordonnançable proposée dans *RTSJ* : la classe *AEH*. Pour cela, il faudra l'associer à une instance de la classe `PeriodicTimer`, qui hérite par l'intermédiaire de `Timer` d '*AE*, et qui appelle sa propre méthode `fire()` périodiquement. Un exemple est donné dans la figure 6.2.

Il est également possible d'attacher l'*AEH* à un événement extérieur à la *RTJVM*, si la périodicité du traitement correspond à une réelle périodicité de l'événement traité.

Par ailleurs, il est possible d'utiliser un `BoundAsyncEventHandler`, dont l'exécution

est attachée à un unique *thread*. Dans ce cas le comportement doit être équivalent à celui obtenu en utilisant un `RealtimeThread`.

6.4.2 Écrire une tâche sporadique

Une tâche sporadique est une tâche apériodique qui peut survenir à plusieurs reprises, et pour laquelle il est possible de borner le temps d'inter arrivée. Cette borne peut être due à la nature de la tâche (cette tâche ne peut pas survenir plus souvent) ou à la capacité de traitement du système (il n'est pas possible de garantir les contraintes temporelles des autres tâches si celle-ci arrive plus souvent).

Pour coder ces tâches sporadiques, *RTSJ* propose d' utiliser les *AEHs* en leur assignant comme `ReleaseParameters` des `SporadicParameters`. Cette classe est une sous-classe de `AperiodicParameters`. En associant un `SporadicParameters` à un *AEH*, il est possible de spécifier une période minimale. Cette valeur a deux fonctions : la première est de permettre l'analyse de faisabilité du `Schedulable`, nous y reviendrons dans la section 6.4.4, la seconde est d'assurer le respect de la fréquence maximale d'inter arrivée.

En effet, plusieurs politiques peuvent être spécifiées en réponse à une arrivée trop fréquente de l'événement. Ces politiques sont appelées les *MIT Violation Policies* et sont au nombre de quatre. Afin de simplifier l'explication de ces politiques, nous allons examiner un exemple.

Tout d'abord, il faut savoir que l'appel de la méthode `fire()` sur un *AE* provoque systématiquement l'ajout d'une requête d'exécution dans une file d'attente pour chaque *AEH* associé à cet événement. Lorsque la ressource processeur est disponible pour le niveau de priorité de l'*AEH*, les requêtes d'exécution sont servies dans l'ordre de la file.

Supposons que la file d'attente d'un *AEH* avec un temps d'inter arrivée minimal fixé à 2 contienne déjà une requête à l'instant X. À l'instant $X + 1$, c'est-à-dire 1 unité de temps trop tôt, la méthode `fire()` d'un événement auquel est associé notre *handler* est invoquée. Si la politique de *MIT Violation* est :

1. *EXCEPT*, une exception de type `MITViolationException` est levée par la méthode `fire()` (quelque soit le nombre de *handler* associé à l'événement) ;
2. *IGNORE*, la file reste inchangée, l'appel est ignoré ; s'il y a d'autre *handlers* associés à l'événement, ils sont traités normalement ;
3. *REPLACE*, si le traitement de la requête précédente n'a pas commencé, elle est remplacée par la nouvelle requête dans la file, sinon la nouvelle est ajoutée à la file mais au temps $X + 2$ et non $X + 1$;
4. *SAVE*, la politique par défaut, la requête est ajoutée à la file, mais au temps $X + 2$.

Enfin, lorsque la file d'attente est pleine et qu'une requête survient, il est également possible de spécifier

- qu'une exception doit être levée, (*EXCEPT*) ;
- que la requête doit être ignorée, (*IGNORE*) ;

– que la dernière requête de la file doit être remplacée (*REPLACE*) ;
– que la taille de la file doit être augmentée (*SAVE*).

6.4.3 Assurer les contraintes temporelles

Les paramètres attachés aux objets ordonnançables permettent de réaliser une analyse de faisabilité d'un système composé uniquement de tâches périodiques ou sporadiques.

Ils permettent également d'automatiser des traitements en cas de non respect des contraintes qu'ils spécifient. Un `Schedulable` qui dépasse son budget ou qui ne respecte pas son échéance est théoriquement rendu inéligible, c'est-à-dire que ses activations futures seront ignorées. Il est également possible de spécifier lors de la création des `ReleaseParameters` deux *AEH* qui seront ordonnancés lors du dépassement du coût pour l'un et lors du dépassement de l'échéance de la tâche pour le second. Il est éventuellement possible dans le code de ces *AEH* de rétablir l'éligibilité de la tâche fautive. Notons que cette dernière n'est pas interrompue, sauf si le *handler* associé possède une plus forte priorité : elle est alors préemptée et peut être interrompue de manière asynchrone (voir section 6.5).

La spécification propose également la classe `ProcessingGroupParameters`, qui permet d'assigner des ressources communes (date de démarrage, période, pire temps d'exécution, échéance) à un groupe d'objets `Schedulable`. Comme pour les `ReleaseParameters`, il est également possible de fournir des *AEH* activés automatiquement en cas de violation de ces paramètres.

Toutefois, dans un cas comme dans l'autre, cela suppose que la *RTJVM* surveille la consommation de temps processeur de chaque tâche. Or, comme cela n'est pas possible avec tous les *OSs*, il s'agit d'une possibilité optionnelle de la spécification, que nous appellerons dans la suite *surveillance du temps CPU ou « temps CPU »* *.

La détection et le traitements des fautes temporelles sur une *RTJVM* qui ne supporte pas le temps *CPU* a fait l'objet d'un travail préliminaire à cette thèse présenté dans [MM05, MM06].

Si la *RTJVM* supporte le temps *CPU*, les `ProcessingGroupParameters` permettent l'écriture, au niveau logique, d'un serveur de tâche. Ce modèle reste très insuffisant car il est trop permissif, il ne permet pas de choisir la politique de service, et ne comporte pas les outils nécessaires à l'intégration du serveur dans l'analyse de faisabilité. Ces points sont expliqués dans [BW03]. Très récemment, Andy Wellings et MinSeong Kim ont proposé de modifier la sémantique de cette classe afin d'en préciser la signification dans un environnement multiprocesseurs, et en ont défini des extensions pour l'écriture de serveurs de tâches apériodiques [WK08].

Cette absence de support portable pour les mécanismes avancés d'intégration de tâches apériodiques a motivé notre travail : fournir des adaptations des mécanismes de service et de vol de temps creux implantés dans l'espace utilisateur afin d'en assurer la portabilité sur l'ensemble des plateformes *RTSJ*.

6.4.4 Analyse de faisabilité

Les méthodes permettant de vérifier la faisabilité du système se trouvent dans la classe `Scheduler`. Toutes les tâches ne sont pas concernées. Le mécanisme repose sur un enregistrement préalable des tâches qui doivent être prises en compte lors de l'analyse. L'analyse en elle-même utilise les propriétés définies par les `SchedulingParameters` et les `ReleaseParameters`. La façon dont sont pris en compte les `ProcessingGroupParameters` n'est pas spécifiée ni d'ailleurs l'algorithme utilisé pour déterminer la faisabilité d'un système de tâches les utilisant. Rien n'indique non plus s'il doit s'agir d'un test suffisant, nécessaire ou nécessaire et suffisant.

Les paramètres mémoire des tâches devraient également entrer en considération, mais il n'est pas expliqué de quelle manière.

Notons également que les `SporadicParameters` ne comportent pas de champs permettant de spécifier la date de la première activation, contrairement aux `PeriodicParameters`. Il faudra donc considérer que la tâche peut être activée dès l'instant initial.

La spécification ne prévoit pas non plus de sous classe à `ReleaseParameters` incluant le paramètre J_i. Il faudra donc utiliser un `SporadicParameters` pour les tâches présentant une gigue d'activation.

Si le support pour écrire les algorithmes d'analyse de faisabilité est bien présent dans *RTSJ*, il semble insuffisamment spécifié et documenté. Nous reviendrons sur ce point dans la section 9.3.

6.5 Interruption et transfert de contrôle asynchrone

Une limitation du modèle de *thread* de Java est l'impossibilité d'interrompre ou de suspendre un autre *thread* que le *thread* courant. Il existe bien les méthodes `suspend()` et `stop()` mais elles sont dépréciées car elle peuvent laisser la mémoire dans un état incohérent. *RTSJ* propose d'utiliser le mécanisme des exceptions pour pallier a cette limitation en introduisant l'exception `AsynchronouslyInterruptedException`.

Un *thread* peut alors explicitement être déclaré comme pouvant être interrompu en utilisant lors de sa conception l'interface `Interruptible`. Cette interface possède deux méthodes :

- `run(AsynchronouslyInterruptedException e)` pouvant lever l'exception d'interruption ;
- `interruptAction(AsynchronouslyInterruptedException e)`.

La première contient le code relatif à l'exécution du *thread*, la seconde le code qu'il faudra éventuellement exécuter avant la terminaison du *thread* s'il est interrompu.

Afin de faciliter l'utilisation de ce mécanisme, la spécification fournit également la classe `Timed` qui étend `AsynchronouslyInterruptedException` et qui possède la méthode `doInterruptible(Interruptible logic)`. Cette méthode exécute la méthode `run(...)`

de l'objet `logic`, et lève l'exception d'interruption asynchrone à l'expiration d'un compte à rebours associé à la classe `Timed`.

Notons enfin qu'un *thread* ainsi interrompu ne peut pas être repris. Ce mécanisme ne permet donc pas de se substituer à l'ordonnanceur pour suspendre momentanément une tâche. Des exemples d'utilisation de ce mécanisme sont fournis dans les annexes page clxix.

Troisième partie

Gestionnaire d'événements pour RTSJ

Dans cette partie nous proposons des solutions pour intégrer la gestion des événements apériodiques dans *RTSJ*. Ces solutions doivent fonctionner au dessus de n'importe quelle *RTJVM* respectant la spécification. Aussi considérons nous que toutes les possibilités optionnelles de *RTSJ*, comme le *temps CPU*, ou les `ProcessingGroupParameters` ne sont pas disponibles.

Nous présentons d'abord en détails les restrictions apportées pour que l'environnement reste portable, et les solutions que nous apportons pour contourner certaines de ces restrictions.

Nous présentons ensuite les modifications qui doivent être apportées aux algorithmes *PS* et *DS* pour être utilisés dans ce contexte.

Nous proposons un algorithme approché dynamique de calcul des temps creux disponibles et nous montrons comment il peut être utilisé pour intégrer dans *RTSJ* un ordonnanceur de tâches apériodiques au niveau utilisateur.

Nous présentons de nombreux résultats de simulations pour évaluer ces algorithmes modifiés et les comparer aux solutions existantes non implantables avec *RTSJ*.

Chapitre 7

Serveurs de tâches

Nous avons passé en revue dans le chapitre 6 les approches prévues par la spécification pour traiter les événements apériodiques :

- ces événements peuvent être modélisés par des *AEs* et leurs traitements par des *AEHs* ;
- si un `AperiodicParameters` est associé à l'*AEH*, la seule solution pour pouvoir garantir la faisabilité des autres tâches est de lui attribuer la plus faible priorité du système ;
- si un `SporadicParameters` est associé à l'*AEH*, l'algorithme d'analyse de faisabilité pourra considérer le pire scénario (l'événement arrive à sa fréquence maximale) car la *RTJVM* propose des mécanismes pour forcer le comportement sporadique ;
- les `ProcessingGroupParamers` permettent en théorie de créer un serveur, mais la spécification reste trop floue sur leur sémantique, et le mécanisme repose de toute façon sur le *temps CPU*.

Nous présentons dans ce chapitre l'implantation de l'approche par serveurs de tâches d'un gestionnaire d'événements pour *RTSJ*. Une partie de ce travail a été publiée dans [MM07].

7.1 Contraintes et limitations

Nous souhaitons proposer un mécanisme portable pour la gestion des événements apériodiques dans *RTSJ*. Nous souhaitons réduire au maximum ses effets de bord sur l'exécution du système et l'intégrer à l'analyse de faisabilité.

Puisque l'on veut un mécanisme portable, nous nous reposons exclusivement sur la spécification minimum pour une *RTJVM*. Cela implique que notre gestionnaire ne peut se reposer sur les mécanismes optionnels de *RTSJ* tels que le *temps CPU* ou les `ProcessingGroupParameters`. De plus, on ne s'autorise pas à modifier l'ordonnanceur. Nous disposons d'un ordonnanceur à priorité fixe préemptif avec vingt huit niveaux de priorité temps réel.

Les tâches chargées d'exécuter le code lié à la gestion des événements, et les tâches codant le mécanisme de gestion sont donc obligatoirement des tâches temps réel ordinaires,

sans privilège particulier, ordonnancées par le même algorithme que les tâches utilisateur du système.

Nous examinons dans ce chapitre le cas de la gestion des événements déléguée à un serveur. Un serveur est une tâche pour laquelle il est possible de calculer ou de borner l'interférence sur les autres tâches, ce qui permet son intégration à l'analyse de faisabilité du système.

L'algorithme de service a deux objectifs : gérer la capacité du serveur, c'est-à-dire le nombre d'unités de temps allouées pour traiter les événements ; et maintenir une file d'attente d'événements qu'il devra ordonnancer (sans être l'ordonnanceur du système). La gestion de cette file d'attente sera détaillée dans la section 7.4. Le coût associé à l'activation d'un événement et à sa prise en compte (ajout dans la file) est limité mais non nul. Comme la fréquence d'arrivée des événements peut être bornée par le mécanisme de contrôle de charge de *RTSJ*, ce coût peut être prise en compte dans l'analyse de faisabilité du système.

7.2 Gestion de la capacité du serveur

Le serveur possède une capacité qui est renouvelée selon une politique propre à l'algorithme de service utilisé. Le point commun à tous les serveurs est qu'ils perdent de la capacité pendant qu'ils exécutent une tâche. Le serveur doit donc être capable de savoir combien chaque tâche servie utilise de temps *CPU* afin de mettre à jour sa capacité. Il doit en outre être capable d'interrompre la tâche en cours d'exécution lorsque sa capacité est épuisée.

Si le serveur s'exécute à la plus haute priorité, ces deux problèmes sont résolus trivialement. En effet, pour savoir combien de temps la tâche la plus prioritaire a consommé à un instant donné, il suffit de mesurer le temps écoulé depuis son démarrage. De même, pour l'interrompre lorsque la capacité du serveur est épuisée, il suffit de mettre en place un compte à rebours initialisé avec la capacité restante du serveur avant de démarrer son service, et de l'annuler lorsque son exécution arrive à son terme. Dans l'éventualité où le compte à rebours arriverait à expiration, le mécanisme d'interruption asynchrone de *RTSJ* permet d'interrompre la tâche. Notons que pour que le compte à rebours puisse interrompre la tâche, il faut qu'il possède une priorité plus forte. Il est donc nécessaire de réserver pour le serveur les deux priorités les plus fortes : la plus élevée pour les comptes à rebours, la seconde pour l'exécution des tâches apériodiques.

Si *RTSJ* permet d'interrompre une tâche, il est toutefois impossible de reprendre son exécution là où elle a été interrompue. Ce problème sera développé dans la section 7.3. Un compte à rebours levant automatiquement une exception asynchrone peut être aisément implanté grâce à la classe `Timed` de *RTSJ*.

Si le serveur n'est pas la tâche de plus haute priorité, alors les tâches servies peuvent être préemptées, ce qui complique le suivi de la consommation du temps par le serveur, et donc la gestion de sa capacité. D'une façon générale, le problème du temps *CPU* revient

fréquemment lorsque l'on utilise *RTSJ*. En effet, connaître la consommation d'une tâche à un instant donné est une possibilité optionnelle dans la spécification, car elle requiert que le système sur lequel s'exécute la *RTJVM* fournisse cette information. Il est cependant possible, grâce à une modification simple de la classe `RealtimeThread`, d'implanter cette fonctionnalité au niveau utilisateur. Une partie du prix à payer est une augmentation de la complexité en temps des changements de contexte, qui peut toutefois être intégré à l'analyse de faisabilité. Cela nécessite également de gérer et de stocker en mémoire une pile dont la hauteur est bornée par le nombre de tâches. Nous détaillons ce mécanisme de surveillance du temps *CPU* dans la section 9.1. Malheureusement, il n'est utilisable que sur des systèmes sans partages de ressources, pour lesquels on suppose qu'aucune tâche ne se suspend elle-même, et dans un environnement où il n'existe pas de tâche plus prioritaire que la machine virtuelle. Aussi, nous ne pouvons l'utiliser pour écrire nos serveurs.

En conclusion, une première restriction doit être apportée aux algorithmes de service : un serveur doit être la tâche de plus forte priorité. Ceci implique également qu'un seul serveur ne peut exister dans le système.

7.3 Ordonnancement des tâches au niveau utilisateur

Les traitements des événements pourront tous être exécutés par un *thread* dédié connu du serveur, ou dans des *threads* propres. Dans les deux cas, lorsque la capacité du serveur se trouve épuisée, les tâches de traitement qui ont été démarrées doivent être suspendues. Grâce au transfert de contrôle asynchrone de *RTSJ*, on peut interrompre un *thread* temps réel mais cette interruption est définitive. Par conséquent, un traitement ne pourra être démarré que si la capacité restante du serveur est supérieure au pire temps d'exécution de ce traitement. Les traitements ne peuvent pas être morcelés, ils doivent être exécutés entièrement ou pas du tout. Nous dirons dans la suite que nous nous sommes restreints à une exécution semi préemptive pour le traitement des événements, car nous ne pouvons pas forcer leur suspension, même si cela n'empêche pas l'ordonnanceur du système de les préempter.

Cela peut amener à des situations où, bien qu'il y ait des événements à traiter, et bien qu'il y ait du temps libre dans le système, il ne soit pas possible de servir les événements en attente. Pour diminuer l'impact de ce phénomène, nous avons expérimenté plusieurs politiques pour gérer la file d'attente des événements à traiter.

7.4 Politiques de gestion de la file

Lorsqu'un événement survient, il est ajouté instantanément dans une file d'attente. Ceci se fait en temps constant si la politique de file retenue est *FIFO* ou *LIFO*. En revanche si la politique retenue nécessite de maintenir une file triée, le coût de l'insertion

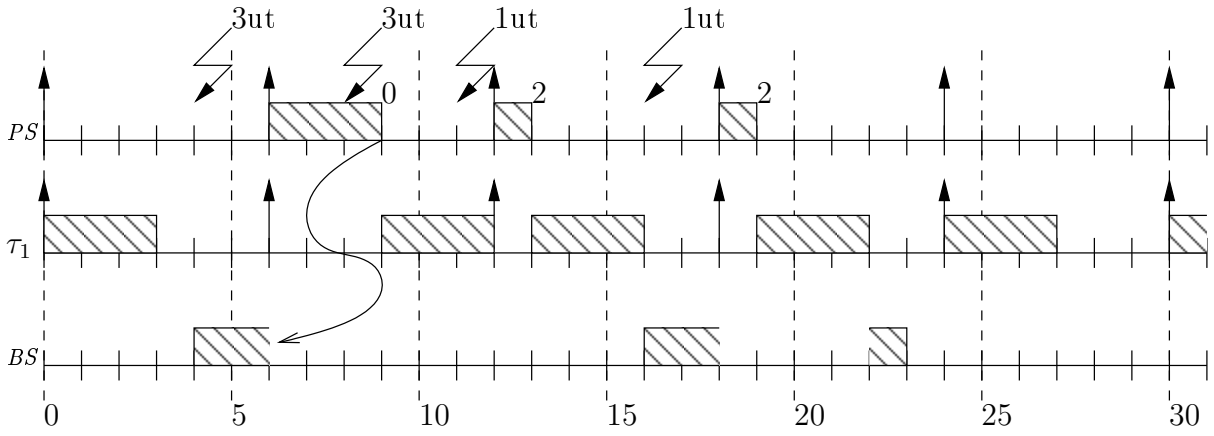


FIG. 7.1 – Intérêt de la duplication BS

devient linéaire en la taille de la file. La taille de la file et la fréquence d'arrivée des événements peuvent toutefois être bornées, ce qui permet d'intégrer ce coût à l'analyse de faisabilité.

En plus de *FIFO* et *LIFO*, nous avons testé une politique qui consiste à trier la file dans l'ordre croissant des pire temps d'exécution des tâches : *Lower Cost First (LCF)**.

Dans tous les cas, excepté pour *LCF*, si le temps disponible n'est pas suffisant pour traiter la tâche en tête de file, la file est parcourue jusqu'à trouver une tâche avec un coût inférieur à la capacité restante.

La politique qui offre les meilleurs résultats pour toutes les politiques de service (et également pour le vol de temps creux) est *LCF*. Cela ne suffit pourtant pas à garantir qu'un événement soit systématiquement traité si le système dispose de suffisamment de temps pour ce traitement. Pour cela, nous proposons une politique de duplication. Cela consiste à dupliquer le traitement de chaque événement. La première réplique est traitée en *BS*, ce qui assure que s'il y a du temps disponible, il finira par terminer son exécution. La seconde réplique est prise en charge par le serveur. La première réplique qui termine son exécution interrompt le traitement de l'autre grâce au mécanisme de transfert de contrôle asynchrone fourni par *RTSJ*.

La figure 7.1 illustre la duplication en *BS*. Dans cet exemple, la capacité du serveur (un *PS*) est de $3ut$ et sa période de $6ut$. Un événement dont le traitement nécessite $3ut$, e_1, survient à l'instant 4. Le serveur n'a alors pas de capacité, le traitement démarre donc en *BS*. Au temps 6, alors que le traitement n'a pu s'exécuter que $2ut$, le serveur récupère sa capacité et traite donc e_1 à la plus haute priorité, mais en le reprenant du début. En effet, il n'est pas possible de changer dynamiquement et instantanément la priorité d'une tâche avec les implantations standards de *RTSJ*. Au temps 9, le traitement se termine, et celui qui avait démarré en *BS* est donc interrompu. Entre temps, un autre événement dont le traitement nécessite $3ut$, e_2, est survenu, ainsi qu'un autre, e_3, dont le traitement nécessite $1ut$. À l'instant 12, le serveur récupère sa capacité, et traite l'événement avec le coût le plus faible : e_3. Au temps 13, il n'y a donc plus assez de capacité pour traiter

e_2. Cette configuration peut se répéter entre chaque période du serveur, empêchant ainsi indéfiniment le service de e_2. Grâce à la duplication de e_2 en *BS*, son traitement pourra tout de même se terminer à l'instant 23.

Dans la suite, on notera *BS*-X^* la politique de gestion de file X mise en œuvre conjointement avec une duplication *BS*.

7.5 Implantation

Nous détaillons ici les modifications des deux politiques de service principale *PS* et *DS* en vue de leur implantation avec *RTSJ* ainsi que les classes nécessaires pour cette implantation. Des propositions de modification de la spécification pour mieux y intégrer les gestionnaires d'événements seront décrites dans le chapitre 9.

7.5.1 Événements et file d'attente

Nous avons vu qu'un événement pouvait être implanté avec *RTSJ* grâce à la classe `AsyncEvent` (*AE*), et que son traitement pouvait être implanté dans un objet de type `AsyncEventHandler` (*AEH*). Plusieurs événements peuvent être associés à un même traitement, et des traitements différents peuvent être attachés à un seul événement. On dit que la relation entre ces événements est de type (n, m). L'objet exécutable par l'ordonnanceur du système, qui implante l'interface `Schedulable` est le traitement, et non l'événement.

En nous basant sur ce modèle, nous proposons la classe `ManageableAsyncEvent-Handler` (*MAEH**), qui modélise un traitement pouvant être pris en charge par un serveur. Cette classe, contrairement à son inspiratrice, n'implante pas l'interface `Schedulable`, puisque le code qu'elle contient n'est pas destiné à être pris en charge par l'ordonnanceur de la *RTJVM*, mais par notre serveur. Nous avons alors besoin d'un objet particulier pour modéliser les événements dont certains traitements pourront être délégués à un ou des serveurs : nous proposons la nouvelle classe `ManageableAsyncEvent` (*MAE**) qui étend `AsyncEvent`. Grâce au mécanisme d'héritage de Java, nos événements peuvent toujours se voir associer des traitements ordinaires. La classe est étendue pour pouvoir se voir affecter des *MAEHs*.

Le serveur lui même est un objet de type `AbstractUserLandTaskServer` qui implante `Schedulable`. Cette classe abstraite regroupe les méthodes nécessaires à la gestion de la file d'attente.

Lorsque la méthode `fire()` est appelée sur un *MAE*, les *AEH* qui lui sont attachés sont activés à leurs priorités respectives, puis chacun de ses *MAEHs* sont ajoutés dans la file d'attente du serveur auquel ils sont attachés.

La duplication en *BS* peut se faire lorsque l'événement est déclenché. Nos *MAEHs* encapsulent quatre objets de visibilité privée :

– un `Interruptible` qui contient le code du traitement à effectuer ;

- une `AsynchousInterruptedException` qui sert à interrompre la duplication si l'événement a pu être traité par le serveur ;
- un `AsyncEvent` qui nous sert à déclencher la duplication ;
- un `AsyncEventHandler` associé à la plus faible priorité du système, qui utilise l'exception d'interruption asynchrone pour exécuter la méthode run() de l'objet interruptible.

On ajoute également un booléen permettant d'activer ou de désactiver la duplication. Lorsqu'un *MAEH* est ajouté à la file d'un serveur, la méthode `fire()` de l'*AE* est appelée, ce qui active l'*AEH* de faible priorité. Celui-ci utilise la méthode `doInterruptible()` de l'exception pour exécuter le code de l'`Interruptible`. Si cette méthode retourne normalement, le *MAEH* est retiré de la liste du serveur. Une méthode publique permet de déclencher l'exception depuis le serveur si celui-ci parvient à traiter le *MAEH* en premier.

Pour résumer, voici les classes que nous proposons pour modéliser un serveur, un événement, et son traitement (handler) :

ManageableAsyncEvent Cette classe hérite de *AE*. Nous y ajoutons une liste d'objets de type *MAEH* et nous surchargeons la méthode `addHandler(AsyncEventHandler h)` pour pouvoir ajouter des *MAEHs*. Enfin nous surchargeons la méthode `fire()` pour que les *MAEHs* soient ajoutés à la file d'attente du serveur auquel ils sont attachés. Le code de cette classe est fourni par l'annexe C.7.

ManageableAsyncEventHandler Cette classe contient un objet de type `Interruptible`. Elle possède également une référence sur le serveur auquel ce traitement est attaché. Lorsqu'un *MAE* auquel elle est attachée survient, sa référence est ajoutée dans la file d'attente du serveur, et si la duplication *BS* est activée, un *AEH* est activé à la plus faible priorité du système. Le code de cette classe est fourni par l'annexe C.6.

AbstractUserLandTaskServer Implantation abstraite fournissant les méthodes de gestion de la file. Le code de cette classe est fourni par l'annexe C.4.

7.5.2 Serveur à scrutation

Le *PS* est une tâche périodique. À chacune de ses activations, il récupère la totalité de sa capacité et peut commencer à traiter les événements présents dans sa file d'attente, jusqu'à épuisement de sa capacité. Si la file d'attente vient à être vide, il perd le reste de sa capacité.

Le serveur à scrutation peut donc être implanté dans une classe `UserLandPollingTaskServer`. Comme l'héritage multiple des classes n'est pas permis en Java, cette classe ne peut pas hériter de `RealtimeThread`. En revanche, nous pouvons utiliser un champ privé de classe contenant une instance de `RealtimeThread` à laquelle on associe des

`PeriodicParameters`. Les méthodes de `Schedulable` non implantées par la classe `AbstractTaskServer` peuvent alors être déléguées à ce champ.

Le code de cette classe est fourni par l'annexe C.10.

7.5.3 Serveur ajournable

Le *DS* peut être intégré à l'analyse de faisabilité, mais son comportement n'est pas périodique. Sa capacité est effectivement restaurée périodiquement, mais il peut se déclencher n'importe quand puisqu'il conserve sa capacité tant qu'elle n'est pas consommée.

Son implantation sera donc assez semblable à celle du *PS*, à la différence que l'objet `Schedulable` auquel on va déléguer le comportement ne sera plus un `RealtimeThread` mais un `AsyncEventHandler`. Cet objet sera attaché à un événement particulier qui servira à réveiller le serveur chaque fois qu'un événement survient.

Une particularité du *DS* est qu'il peut disposer à un instant t d'une capacité supérieure à sa capacité rafraichie périodiquement. En effet s'il lui reste x unités de temps de capacité, alors $x ut$ avant le rafraichissement de sa capacité, il dispose en fait de $x + C_s$ ut. Si des tâches sont en attente, avec un coût plus fort que la capacité restante du serveur, il sera donc nécessaire de réveiller le serveur x unités de temps avant le rafraichissement de sa capacité.

Le code de cette classe est fourni par l'annexe C.10.

7.6 Synchronisations

Nous étudions ici l'influence des synchronisations sur nos serveurs de tâche.

Partage de ressources entre tâches temps réel dur Le partage de ressource entre tâches temps réel dur n'affecte pas le serveur, puisqu'il s'exécute à une priorité plus élevée que les autres tâches du système.

Partage de ressources entre tâches temps dur et tâches traitées par le serveur Le serveur peut être bloqué si une tâche temps réel souple apériodique demande un verrou sur une ressource déjà obtenu par une autre tâche. Ce temps de blocage peut être borné si l'on connait a priori les ressources utilisées par les tâches apériodiques et le protocole de synchronisation utilisé. L'analyse de faisabilité pourra alors conclure sur la possibilité d'autoriser ce partage.

En revanche, le serveur ne peut théoriquement pas bloquer d'autre tâche : comme les tâches qu'il sert sont exécutées en une seule fois, elles disposent de suffisamment de temps processeur pour exécuter leur section critique entièrement. Cela suppose bien sûr que la tâche n'est pas interrompue par l'épuisement de la capacité. Dans le cas contraire, il faudra veiller à relâcher les verrous possédés par une tâche apériodique interrompue, ce

qui ne peut se faire que dans la méthode `interruptAction()` de l'objet `Interruptible` qu'il encapsule. Ceci nécessite la sémantique de l'interface `Lock` apportée par Java 1.5.

Par ailleurs, si les tâches temps réel souple peuvent accéder à des ressources, la duplication *BS* doit être désactivée.

7.7 Surcoût et intégration à l'analyse de faisabilité

Le surcoût de ce mécanisme pour le système dépend de l'implantation de la *RTJVM*. Nous utilisons un `Schedulable` pour le seveur (un `RealtimeThread` pour le *PS* et un *AEH* pour le *DS*), ainsi qu'un *AEH* pour chaque tâche apériodique afin d'implanter la duplication en *BS*.

Il peut donc sembler au premier abord que le surcoût augmente avec le nombre de tâches apériodiques à traiter. Cependant, la spécification précise qu'un *AEH* n'est pas forcément attaché à un *thread*. Comme toutes les répliques sont traitées à la même priorité, une implantation optimisée de la *RTJVM* devrait en principe n'utiliser qu'un seul *thread* pour tous les traitements dupliqués.

Chapitre 8

Voleurs de temps creux

Nous présentons dans ce chapitre un algorithme de vol de temps creux dynamique, implantable, aussi bien du point de vue de la complexité algorithmique que de sa réalisation avec *RTSJ* sur une plate-forme standard. Pour cela, nous nous basons sur une approximation, une borne minimale, des temps creux disponibles. Nous montrons comment obtenir cette borne minimale dynamiquement en un temps constant, grâce à des calculs de complexité linéaire en mémoire intégrables à l'analyse de faisabilité. Les résultats présentés ici ont été publiés en partie dans [MM08b], [MM08c] et [MM08a].

8.1 Utilisation de DASS au niveau utilisateur

Davis propose un algorithme approché, *DASS*, présenté dans la section 4.7.2. Cet algorithme permet de borner la quantité de temps creux disponible au niveau de priorité i à l'instant t, $S_i(t)$, par la durée de l'intervalle $[t, d_i(t)[$ à laquelle il faut retrancher l'interférence maximale causée par les niveaux de priorité supérieurs ou égaux à i. Cette interférence est bornée par la somme des interférences de chaque niveau de priorité. Comme l'équation 4.17 nous donne l'interférence d'un niveau de priorité durant un intervalle donné en temps constant, on obtient une borne sur l'interférence totale, et donc sur $S_i(t)$, en temps linéaire. Pour calculer la quantité de temps creux du système, il faut alors prendre le minimum des $S_i(t)$, ce qui donne une complexité totale en temps quadratique. Toutefois, $S_i(t)$ n'a besoin d'être recalculé que lorsque τ_i complète une instance. Le reste du temps il peut être mis à jour en temps constant à condition de savoir combien de temps le processeur a été inoccupé ou occupé avec des tâches moins prioritaires depuis le précédent calcul. Si les $S_i(t)$ sont à jour, la quantité de temps creux s'obtient en temps linéaire ($S(t) = \min_{\forall i} S_i(t)$).

Le support du temps *CPU* semble donc nécessaire pour implanter *DASS*.

Temps CPU pour l'approximation de l'interférence Nous rappelons ici l'équation 4.17 qui donne l'interférence d'une tâche τ_i sur une tâche τ_j de plus faible priorité dans

un intervalle [a,b] :

$$I_i^j(a,b) = \bar{\mathbf{c}}_\mathbf{i}(\mathbf{a}) + f_i(a,b)C_i + \min(C_i, (b - x_i(a) - f_i(a,b)T_i)_0)$$

Le temps d'exécution restant $\bar{c}_i(t)$, doit être connu. En réalité, si ce calcul n'est effectué que lorsque τ_j termine une instance, cela implique $\bar{c}_i(t) = 0$, car dans le cas contraire c'est τ_i qui serait entrain de s'exécuter. Ceci est vrai même dans le cas où les tâches partagent des ressources à condition de supposer qu'elles relâchent leurs verrous avant la fin de leur exécution.

Temps CPU pour mettre à jour $\mathbf{S_i(t)}$ Lorsqu'à un instant quelconque on désire connaître $S_i(t)$, il suffit de retrancher à la dernière valeur calculée le temps consommé depuis par toutes les tâches de priorité inférieure, incluant le temps pendant lequel le processeur est resté inoccupé.

Si l'*OS* ne fournit pas cette information, il est toujours possible de surveiller la consommation de chaque tâche. Le principe est détaillé dans la section 9.1. La solution consiste à garder en mémoire une pile contenant les tâches démarrées et non terminées. Une tâche est empilée lorsqu'elle démarre, dépilée lorsqu'elle termine. Avant d'empiler une tâche, on met à jour le temps consommé par la tâche se trouvant au sommet de la pile. Si les partages de ressources sont autorisés, les changements de contexte liés aux inversions de priorité ne sont pas détectables (sauf dans le cas de *PCE*) : le temps mesuré est alors le temps passé à chaque niveau de priorité et non plus le temps consommé par chaque tâche.

On peut alors, à chaque fois qu'une tâche démarre, mettre à jour le $S_i(t)$ de toutes les tâches plus prioritaires que celle qui s'exécutait précédemment. De la même façon, à n'importe quel instant t, il est possible de mettre à jour les $S_i(t)$ de toutes les tâches plus prioritaires que celle qui s'exécute actuellement.

L'erreur dans le temps mesuré si le partage de ressources est autorisé n'est pas problématique. Supposons en effet qu'une tâche de priorité élevée τ_h est restée bloquée car elle désire accéder à une ressource détenue par une tâche moins prioritaire τ_b. Cette dernière a alors vu sa priorité augmenter et s'est exécutée jusqu'à relâcher le verrou. Ce temps supplémentaire qui a été nécessaire ne peut pas être retranché à S_h. Cependant, l'erreur maximale est égale à B_h. Or B_i est précisément la valeur que l'on retranche à tous les S_i lorsque l'on désire savoir si des temps creux sont volables (équation 4.15).

Il est donc possible d'implanter *DASS* au niveau utilisateur, à condition d'ajouter au début de chaque tâche une mise à jour en temps linéaire de tous les $S_i(t)$ et à la fin de chaque tâche un calcul, en temps linéaire également, du $S_i(t)$ concerné. Ces opérations doivent de plus être protégées contre toute interruption. Lorsque l'on a besoin de calculer la quantité de temps creux, il est de nouveau nécessaire de mettre à jour tous les $S_i(t)$ et de déterminer la valeur minimale. Ceci se fait également en temps linéaire.

Par ailleurs, une approximation moins fine de la quantité de temps creux peut être

obtenue en temps constant. Puisque l'on effectue déjà un parcours partiel des tâches à la fin de chaque instance pour mettre à jour le temps consommé et calculer l'interférence, il est possible d'en profiter pour calculer le minimum des S_i. Le reste du temps, dans le pire scénario, la valeur a diminué du temps écoulé depuis le dernier calcul.

8.2 Un algorithme de vol de temps creux minimal

Afin de réduire le coût, et donc l'impact sur l'ordonnançabilité du système, des opérations à ajouter en début et fin d'instance de chaque tâche, nous proposons un autre algorithme, *Minimal Approximate Slack Stealing (MASS)**. Comme pour la version légèrement modifiée de *DASS* décrite dans la section précédente, la quantité de temps creux disponible à un instant quelconque est obtenue en temps constant en se basant sur le pire scénario dans lequel elle a diminué du temps écoulé depuis son dernier calcul.

Le dernier calcul en question est effectué à chaque fois qu'une tâche termine son exécution. Il s'effectue comme pour *DASS* en temps linéaire car l'interférence causée par chaque tâche plus prioritaire doit toujours être estimée, afin d'en faire la somme. En revanche le nombre d'opérations nécessaire pour estimer chaque interférence est beaucoup plus petit, car on l'approxime au lieu de le calculer.

Afin de contrôler la consommation des tâches, il est également nécessaire d'ajouter des opérations au début de l'exécution des tâches. Cette fois, ceci peut être fait en temps constant, car il n'est plus nécessaire d'exploiter cette donnée instantanément.

Notations Nous reprenons les notations utilisées pour décrire le voleur de temps creux dynamique dans la section 4.6.1.

On note α_i la i^{e} tâche apériodique. On note la plus haute priorité 1. Une activité de niveau 0 correspond à une inactivité du système. La priorité la plus forte (1) est réservée pour la tâche qui implante le mécanisme de gestion des tâches apériodiques. Une plage de priorité commençant à 2 est réservée pour l'exécution des tâches apériodiques prises en charge par le mécanisme. On note *hrtp* l'ensemble des priorités temps réel dur, c'est-à-dire les valeurs de priorité supérieures.

8.3 Données à collecter et à calculer

Nous rappelons que la quantité de temps creux au niveau i notée $S_i(t)$ est le délai applicable au trafic en attente à la priorité i sans que les requêtes composant ce trafic ne dépassent leur échéance. Ceci ne tient pas compte des tâches de priorités inférieures.

Nous décomposons $S_i(t)$ en deux parties afin d'en faciliter le calcul. Ces deux parties sont la quantité de travail qui peut être exécutée à la priorité i avant la prochaine échéance d'une requête de τ_i, et la quantité de travail qui est demandée au niveau i. On note $\bar{w}_i(t)$ la quantité de travail disponible et $\bar{c}_i(t)$ la quantité de travail demandée au niveau i.

On a alors $S_i(t) = \bar{w}_i(t) - \bar{c}_i(t)$.

La quantité de temps creux disponible à la priorité la plus haute du système à l'instant t pour qu'aucune tâche périodique ne rate sa prochaine échéance est alors le minimum parmi les $S_i(t)$, comme rappelé par l'équation 8.1.

$$S(t) = \min_{\forall i \in hrtp} S_i(t) = \min_{\forall i \in hrtp} (\bar{w}_i(t) - \bar{c}_i(t)) \tag{8.1}$$

Comme les temps creux ne sont calculés que lorsqu'une tâche termine une instance, les valeurs monitorées des $\bar{c}_i(t)$ ne doivent être égales ou inférieures à leur valeurs réelles qu'à ce moment là.

8.4 Initialisation des données

Dans le cas d'une activation synchrone des tâches à l'instant t_0, l'équation 8.2 permet de calculer pour chaque tâche une borne maximale pour l'interférence des tâches plus prioritaires sur τ_i.

$$I_i(t_0) \leq \sum_{\forall k<i} \left\lceil \frac{D_i}{T_k} \right\rceil C_k \tag{8.2}$$

Le calcul de cette borne consiste à sommer les interférences causées par chaque tâche plus prioritaire. Cette interférence est une valeur pessimiste plus forte que l'interférence de τ_k sur τ_i donnée par l'équation 4.17 page 67. Toutefois, l'interférence comptabilisée ici trop tôt permettra d'estimer plus facilement l'interférence plus tard dans l'exécution du système.

Dans le cas d'une première activation non synchrone des tâches, le nombre d'activations de τ_k entre le démarrage de τ_i et sa première échéance absolue doit être calculé. L'interférence de τ_k sur τ_i est alors égale à ce nombre multiplié par C_k.

On a par conséquent au démarrage du système :

$$\forall i \begin{cases} \bar{w}_i(t_0) &= D_i - I_i(t_0) \\ \bar{c}_i(t_0) &= C_i \end{cases} \tag{8.3}$$

L'initialisation a une complexité quadratique en temps car il faut calculer l'interférence maximale de chaque tâche. Ceci n'est pas problématique car ces calculs peuvent être exécutés avant le démarrage du système, et n'interfèrent donc pas sur son fonctionnement.

8.5 Mise à jour des données

Nos indices évoluent à chaque pas de l'horloge du système. Soit δ_k la durée d'occupation du processeur par la tâche τ_k durant un intervalle $[t_1, t_2]$. Entre t_1 et t_2, $\bar{c}_k$ a diminué de δ_k, et pour toutes les tâches τ_l plus prioritaire de τ_k, $\bar{w}_l$ a diminué de δ_k.

Nous n'avons toutefois besoin des valeurs des $\bar{w}_i$ et $\bar{c}_i$ que lorsqu'une tâche termine une instance. Nous montrons ici comment il est possible de maintenir une borne minimale de $\bar{w}_i$ pour toutes les tâches en ajoutant des opérations en temps linéaire à la fin de chaque instance, et une borne maximale de $\bar{c}_i$ pour toutes les tâches en ajoutant des opérations en temps constant au début et à la fin de chaque tâche.

Notations pour le reste de la section : Nous notons t_d la dernière date à laquelle une tâche a commencé une instance, t_f la dernière date à laquelle une tâche a terminé une instance, t la date courante, $dt_f = (t - t_f)$ et $dt_d = (t - t_d)$.

8.5.1 Maintien d'une borne minimale pour les $\bar{w}_i$

Considérons que la tâche τ_l, une tâche périodique temps réel dur quelconque, termine une instance à la date t. La dernière mise à jour de la valeur des $\bar{w}_i$ date de l'instant t_f. Il faut donc étudier l'intervalle $[t_f, t]$. On cherche à calculer pour toutes les tâches τ_k la valeur $\bar{w}_k(t)$ en fonction de $\bar{w}_k(t_f)$.

Cas des tâches plus prioritaires que τ_l Pour toutes les tâches τ_k de plus forte priorité que τ_l, on a $\bar{w}_k(t_f) = \bar{w}_k(t_d) - dt_f$ car dans l'intervalle $[t_f, t]$, seules des tâches de priorité inférieure ou égale à l ont été exécutées. En effet, si une tâche de plus forte priorité avait été exécutée dans cet intervalle, elle aurait terminé son exécution avant τ_l, étant plus prioritaire. On fait ici toutefois l'hypothèse qu'une tâche relâche les verrous qu'elle pourrait posséder avant la fin de son exécution périodique.

$$\forall k \in hrtp \backslash k < l, \bar{w}_k(t) = \bar{w}_k(t_f) - dt_f$$

Cas de τ_l La tâche τ_l vient de terminer une instance. Depuis la dernière fin d'instance d'une tâche, le système n'a pu exécuter que τ_l ou des tâches de plus faible priorité (même argument que dans le paragraphe précédent). Par conséquent, $\bar{w}_l$ a diminué de dt_f. Cependant, l'intervalle considéré pour le calcul des temps creux au niveau l augmente d'une période de τ_l. La valeur de $\bar{w}_l$ augmente donc de T_l mais diminue de l'interférence que va subir la tâche de la part des tâches plus prioritaires qui seront activées entre sa prochaine activation, $x_l(t_f)$, et sa prochaine échéance $d_l(t_f)$. Rappelons que l'interférence qu'elle subit de la part des tâches activées avant $x_l(t_f)$ a déjà été prise en compte dans $\bar{w}_l(t_f)$. Cette interférence est bornée par la somme des interférences provenant de chacune des tâches plus prioritaires. Nous donnons dans la section 8.6 une procédure pour borner l'interférence d'une tâche sur une autre en temps constant qui ne nécessite qu'un seul test (la valeur exacte peut être obtenue en temps constant par l'équation 4.17, mais nécessite plus de calculs).

$$\bar{w}_k(t) = \bar{w}_k(t_f) - dt_f + T_k - I_k(t)$$

Cas des tâches moins prioritaires que τ_l Pour les tâches de priorité plus faible que τ_l, le problème est plus compliqué. En effet, la quantité de travail disponible n'a diminué que pour celles d'entre elles qui ont vu s'exécuter dans l'intervalle d'étude des tâches moins prioritaires. Soit il faut mettre à jour en temps linéaire les $\bar{w}_i$ pour tout i à chaque début d'instance (comme dans l'algorithme *DASS* légèrement modifié), soit il faut accepter ici encore une perte temporaire de précision. Nous allons en effet considérer que la quantité de travail disponible a diminué de dt_f pour toutes les tâches moins prioritaires. On leur rajoute toutefois immédiatement $\bar{c}_l(t)$, qui correspond au temps processeur occupé par τ_l dans l'intervalle, temps qui n'a donc pas été perdu aux niveaux de priorité inférieurs. Le reste de l'erreur pourra être rattrapé lorsque les tâches s'étant réellement exécutées dans cet intervalle termineront leur exécution, en augmentant le travail disponible du temps d'exécution de ces tâches.

$$\forall k > l, \bar{w}_k(t) = \bar{w}_k(t_f) - dt_f + \bar{c}_l^*$$

où $\bar{c}_l^*$ est le temps consommé par par τ_l. Il est égal à $C_l - (\bar{c}_l(t) - \min(dt_f, dt_d))$ si les tâches peuvent consommer moins que leur pire temps d'exécution, C_l sinon.

8.5.2 Maintien d'une borne maximale pour les $\bar{c}_i$

Pour maintenir à jour les $\bar{c}_i(t)$ il faut intervenir chaque fois qu'une tâche démarre ou complète une instance. La procédure est la même que pour *DASS*, mais on ne met à jour que le $\bar{c}_i(t)$ de la tâche qui occupait le processeur précédemment, ce qui se fait en temps constant, la mise à jour des $\bar{w}_i(t)$ étant reportée jusqu'à la prochaine fin d'exécution d'une tâche.

Lorsqu'une tâche τ_l commence une instance à l'instant t, si k est le niveau de priorité qui occupait le processeur précédemment (si τ_k est la tâche occupant le sommet de la pile d'exécution), si k appartient à $hrtp$, $\bar{c}_k$ diminue de $\min(dt_f, dt_d)$.

Lorsqu'une tâche τ_l termine une instance à l'instant t, $\bar{c}_l(t) = C_l$.

8.5.3 Bilan des opérations à effectuer

Lorsqu'une tâche temps réel dur périodique quelconque τ_l termine une instance : Soit $k \in hrtp$,

$$\begin{array}{rl} \forall k < l, & \bar{w}_k(t) = \bar{w}_k(t_f) - dt_f \\ \forall k > l, & \bar{w}_k(t) = \bar{w}_k(t_f) - dt_f + \bar{c}_k^* \\ k = l, & \left\{ \begin{array}{rcl} \bar{w}_k(t) & = & \bar{w}_k(t_f) - dt_f + T_k - I_k(t) \\ \bar{c}_k(t) & = & C_k \end{array} \right. \end{array} \tag{8.4}$$

L'équation 8.4 récapitule les opérations à effectuer lorsqu'une tâche temps réel dur complète une instance. Ces opérations se font en un temps linéaire en nombre de tâches.

Le calcul de $\bar{c}_i^*$ n'est nécessaire que si les temps d'exécution des tâches peuvent varier.

Lorsqu'une tâche tâche temps réel dur périodique quelconque τ_l commence une instance : Soit τ_k la tâche au sommet de la pile d'exécution (précédemment exécutée),

$$\text{Si } k \in hrtp, \bar{c}_k(t) = \bar{c}_k(\max(t_f, t_d)) - \min(dt_f, dt_d) \tag{8.5}$$

L'équation 8.5 récapitule l'opération à effectuer chaque fois qu'une tâche temps réel dur débute une instance. Cette opération a une complexité en temps constant.

8.6 Approximation de l'interférence

On considère une tâche τ_j temps réel dur périodique quelconque qui termine une instance à l'instant t_2. On connait $\bar{w}_j(t_1)$, t_1 étant la date du dernier instant de mise à jour de $\bar{w}_j$. On note sa prochaine échéance $d_j(t_2)$ et la suivante $d_j^+(t_2)$. On a $d_j(t_2) \geq t_2$ et $d_j^+(t_2) = d_j(t_2) + T_j$. On sait qu'entre t_1 et t_2, le travail disponible à la priorité j a diminué de $dt = t_2 - t_1$. À l'instant t_2, il augmente de T_j moins $I_j(t_2)$, l'interférence subie par τ_j venant de l'exécution des tâches plus prioritaires activées entre $d_j(t_2)$ et $d_j^+(t_2)$. En effet, l'interférence subit avant $d_j(t_2)$ à déjà été prise en compte lors du calcul de $\bar{w}_j(t_1)$.

Les deux inéquations suivantes nous permettent de trouver une borne supérieure pour cette interférence :

$$I_j(t) \leq \sum_{k<j} I_k^j(t) \tag{8.6}$$

$$I_i^j(t) \leq Nba_i^j(t).C_i \tag{8.7}$$

où $I_i^j(t)$ est l'interférence des instances de τ_i sur τ_j qui commencent entre $d_j(t)$ et $d_j^+(t)$, et $Nba_i^j(t)$ est le nombre d'activations de τ_i entre $d_j(t)$ et $d_j^+(t)$.

La première indique que l'interférence subie de la part de l'ensemble des tâches plus prioritaires est bornée par la somme des interférences de chaque tâche plus prioritaire, la seconde que l'interférence d'une tâche est bornée par le produit de son nombre d'activations et de son coût.

Nous montrons maintenant que $Nba_i^j(t)$, et donc $I_i^j(t)$, ne peut prendre que deux valeurs, qui peuvent être calculées avant le démarrage des tâches temps réel dur, et que l'on peut déterminer laquelle en temps constant,

8.6.1 Valeurs possibles de $\mathbf{Nba_i^j(t)}$

Soit q et r le quotient et le reste dans la division euclidienne de T_j par T_i. On a $T_j = qT_i + r$. Pour chaque activation de τ_j, on note u le temps avant la prochaine activation

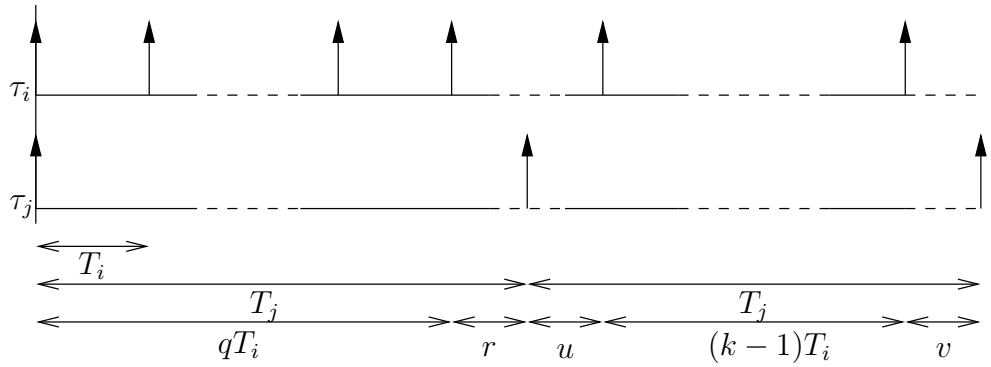


FIG. 8.1 – Nombre d'activations - notations

de τ_i, $k = Nba_i^j(t)$ le nombre d'activations de τ_i avant la prochaine activation de τ_j et v le nombre tel que $v = T_j - (k-1)T_i - u$. On a $T_j = u + (k-1)T_i + v$, $u < T_i$ et $v < T_i$.

La figure 8.1 récapitule ces notations.

Theoreme 1. *Il n'y a que deux valeurs possibles pour k, qui sont $q+1$ et q.*

Preuve de $k > q-1$.
Supposons $k \leq q-1$,
$k \leq q-1 \Rightarrow k-1 \leq q-2 \Rightarrow$
$u + (k-1)T_i + v < T_i + (q-2)T_i + T_i$
car $u < T_i$ et $v < T_i$.
Donc, puisque nous avons $u + (k-1)T_i + v = T_j$, on obtient $T_j < qT_i$.
Ce qui est en contradiction avec $T_j = qT_i + r$, $r \geq 0$.

□

Preuve de $k < q+2$.
Supposons $k \geq q+2$,
$k \geq q+2 \Rightarrow k-1 \geq q+1 \Rightarrow$
$u + (k-1)T_i + v \geq u + (q+1)T_i + v \Rightarrow$
$T_j \geq (q+1)T_i$, car $u \geq 0$ et $v \geq 0$. Ce qui est en contradiction avec $T_j = qT_i + r$, $q \geq 1$ et $r < T_i$. □

8.6.2 Choix de la valeur de $\mathbf{Nba_i^j(t)}$

Le nombre d'activations de τ_i dans l'intervalle $[d_j(t), d_j^+(t)]$ est donc égal à q ou à $q+1$. Nous allons montrer que pour savoir quelle est la bonne valeur, il faut comparer u et r. Rappelons que u est la durée entre la prochaine échéance de $d_j(t)$ et la prochaine activation de τ_i.

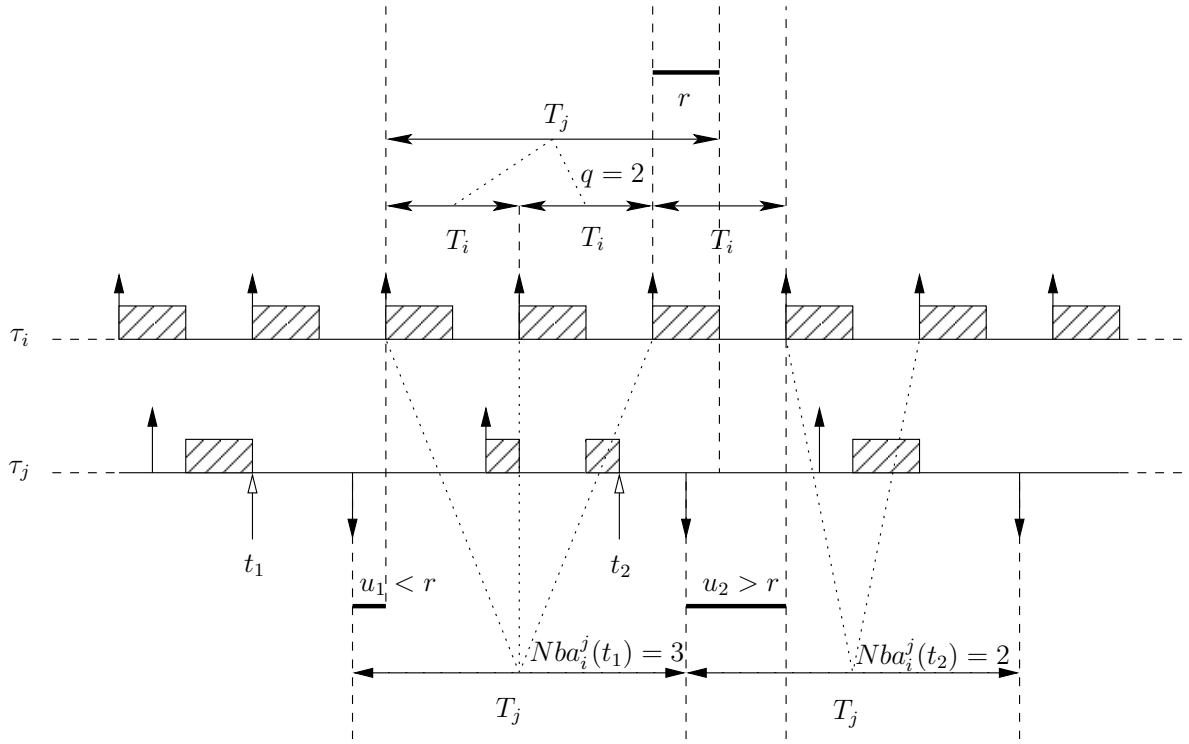


FIG. 8.2 – Nombre d'activations - exemple

Theoreme 2.

$$u \leq r \quad \Rightarrow \quad k = q + 1 \tag{8.8}$$

$$u > r \quad \Rightarrow \quad k = q \tag{8.9}$$

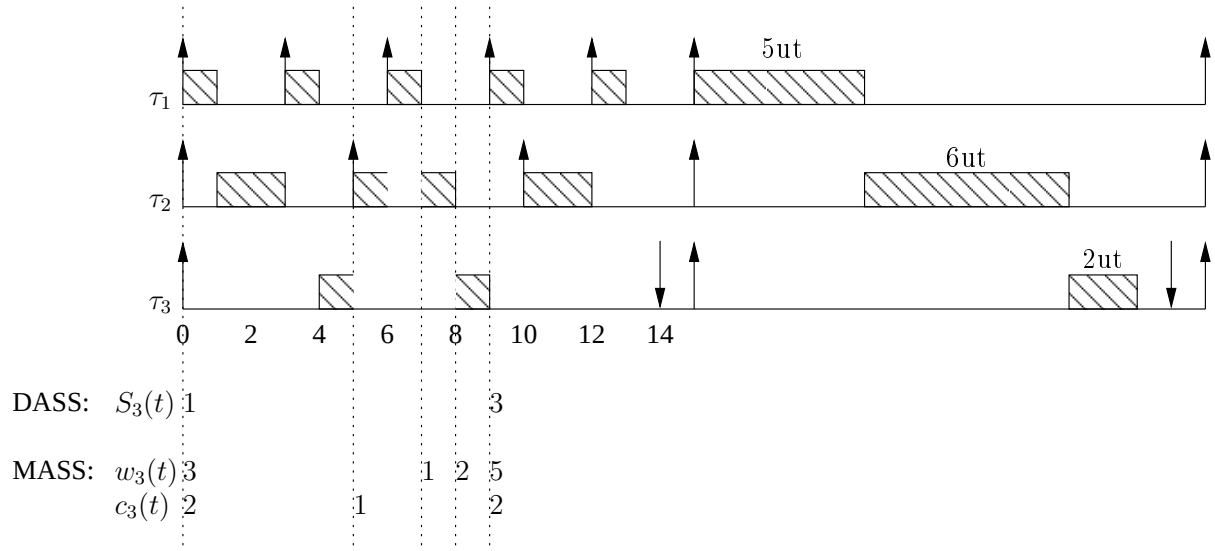
Preuve de l'équation 8.8.
$k = q + 1 \Rightarrow k - 1 = q \Rightarrow$
$u + (k-1)T_i + v = qT_i + u + v \Rightarrow$
$u + v = r \Rightarrow$
$u \leq r$ car $u \geq 0$ et $v \geq 0$. □

Preuve de l'équation 8.9.
$k = q \Rightarrow$
$u + (k-1)T_i + v = (q-1)T_i + u + v \Rightarrow$
$u + v - T_i = r \Rightarrow$
$u > r$ car $v < T_i$. □

La figure 8.2 illustre ce théorème. Pour déterminer $I_i^j(t)$, il suffit donc de comparer u et r. Or r peut être calculé avant le début du traitement des tâches périodiques, ou obtenu par l'opérateur *modulo* qui correspond à une instruction dans le *bytecode* Java. Pour obtenir u, il faut soustraire $x_j(t)$ à $x_i(x_j(t))$: $x_j(t)$ est connu et $x_i(x_j(t))$ s'obtient avec une division et une multiplication ($x_i(t) = \lceil t/T_i \rceil T_i$). La complexité est donc la même

	Période	Coût	Échéance
τ_1	3	1	3
τ_2	5	2	5
τ_3	15	2	14

(a) Données du système



(b) Diagramme d'exécution sans tâches apériodiques

FIG. 8.3 – Calcul de la laxité de niveau 3 avec DASS et avec MASS

que pour obtenir l'interférence exacte dans un intervalle quelconque donnée par l'équation 4.17 page 67, mais le nombre d'opérations nécessaire est beaucoup moins important.

8.7 Exemple comparatif de *DASS* et *MASS*

Nous proposons ici un exemple numérique permettant de mieux comprendre les différences d'approche dans les algorithmes *DASS* et *MASS*.

Le système étudié comporte trois tâches. Les paramètres des tâches et l'exécution du système en l'absence de tâches apériodiques est donné par la figure 8.3.

Nous nous intéressons au calcul de la laxité au niveau 3.

Avec *DASS* À l'instant 0, $S_3(0)$ est égal à l'échéance de τ_3, à laquelle on retranche son coût, et l'interférence de τ_1 et de τ_2 dans l'intervalle $[0, 14]$. On a donc $S_3(0) = 14 - 5 * 1 - 3 * 2 - 2 = 1$. C'est en effet le nombre d'unités de temps inutilisé au niveau 3 dans l'intervalle [0,14]. Cette quantité n'évolue pas avant l'instant 9, car seules les tâches de priorité inférieure ou égale à celle de τ_3 sont exécutées. Remarquons en revanche qu'aux instants 3 et 5, il est nécessaire de retrancher respectivement 2 *ut* à $S_1(t)$ et 1 *ut* aussi bien à $S_1(t)$ qu'à $S_2(t)$, d'où la complexité en temps linéaire des opérations à effectuer au début de chaque tâche.

À l'instant 9, τ_3 termine l'exécution de sa première instance. L'intervalle considéré pour le calcul de $S_3(t)$ n'est plus $[t, 14]$ mais $[t, 29]$. La durée de cet intervalle à l'instant 9 est égale à 20 *ut*, auxquelles il faut retrancher l'interférence de 4 activations de τ_2, de 7 activations de τ_1 et d'une activation de τ_3. On a donc $S_3(9) = 20 - 8 - 7 - 2 = 3$. $S_3(9)$ est recalculée entièrement sans tenir compte de l'information déjà connue qu'il reste une unité de temps libre jusqu'à l'instant 14.

Avec *MASS* La valeur de $\bar{w}_3(0)$ est égale à l'échéance de τ_3 diminuée des interférences de τ_1 et τ_2 sur l'intervalle $[0, 14]$. On a donc $\bar{w}_3(0) = 14 - 6 - 5 = 3$. Comme $\bar{c}_3(0) = 2$, cela nous donne $S_3(0) = 1$, soit la même valeur que celle donnée par *DASS*.

Aux instants 1 et 3, les tâches τ_1 et τ_3 complètent une instance. $\bar{w}_3(t)$ est alors décrémenté du temps écoulé, mais immédiatement incrémenté du coût des tâches qui terminent, et ne varie donc pas avant l'instant 7.

En revanche, $\bar{c}_3(t)$ est mis à jour à l'instant 5. En effet, τ_2 démarre alors une instance, et la tâche qui s'exécutait précédemment, τ_3 est mise à jour. On a $\bar{w}_3(5) = 3$ et $\bar{c}_3(5) = 1$, ce qui donne $S_3(5) = 2$. Cette valeur est fausse et, plus grave, est supérieure à la valeur exacte ($S_3(5) = 1$). Cette erreur n'a pourtant pas d'influence car les temps creux ne sont évalués que lorsqu'une tâche termine son exécution. La prochaine évaluation aura donc lieu à l'instant 7.

Or, à l'instant 7, $\bar{w}_3$ et $\bar{w}_2$ sont décrémentés de 3 *ut*, car c'est le temps écoulé depuis la dernière terminaison d'une tâche périodique, et incrémenté du coût de τ_1, c'est-à-dire seulement 1 *ut*. On a donc $S_3(7) = 3 - 3 + 1 - 1 = 0$, soit une valeur inférieure à celle retournée par *DASS*.

Au temps 8, τ_2 complète à son tour une exécution, $\bar{w}_3(8)$ est donc décrémenté de 1 *ut* et incrémenté de 2 *ut*. On a donc $S_3(8) = 1 + 2 - 1 - 1 = 1$, soit la même valeur que celle retournée par *DASS*.

Enfin au temps 9, c'est τ_3 qui complète son instance. L'interférence qu'elle va subir dans l'intervalle $[9, 15]$ de la part de τ_1 et de τ_2 est déjà intégrée à $\bar{w}_3$, car on a borné l'interférence par le nombre d'activation multiplié par le coût, et que la prochaine activation aussi bien de τ_1 que de τ_2 n'intervient qu'au temps 15. La valeur de $\bar{w}_3$ augmente d'une période ($29 - 14 = 15$) puisque l'on considère la prochaine échéance, et diminue de l'interférence de τ_1 et de τ_2 dans l'intervalle $[15, 30]$. On a $S_3(9) = (2 + 15 - 6 - 5) - 2 = 3$.

Remarques sur le calcul de l'interférence Dans cet exemple, le calcul de l'interférence de τ_1 et de τ_2 sur τ_3 n'est pas très compliqué car lors de l'échéance de τ_3, toutes les instances commencées sont terminées. Autrement dit, l'interférence causé par une tâche est un multiple de son nombre d'activation.

Si ce n'était pas le cas, *DASS* calculerait l'interférence exacte, d'où la constante du temps de calcul nécessaire plus grande, tandis que *MASS* utiliserait la borne supérieure donnée par le nombre d'activations multiplié par le coût. Cette approximation permet à

```
public final void run() {
  do {
    computeBeforePeriodic();
    logic.run();
    computeAfterPeriodic();
    waitForNextPeriod();
  } while (!slackStealer.flagEnd);
}
```

FIG. 8.4 – Méthode run() d'un Schedulable compatible avec un voleur de temps creux

l'instance suivante de ne considérer que les activations de tâches de plus forte priorité qui surviennent à un instant supérieur ou égal à l'échéance.

8.8 Implantation

Les implantations de *DASS* et de *MASS* ne diffèrent pas beaucoup de celles des serveurs de tâches. La partie « ordonnancement » est exactement la même. Un voleur de temps creux est finalement un serveur de tâches dont la capacité est égale à chaque instant à la quantité de temps creux dans le système.

Nous proposons donc la classe `AbstractUserLandEventManager`. Cette classe regroupe le code commun aux serveurs et au voleur de temps creux. La classe `AbstractUserLandTaskServer` est alors modifiée pour étendre `AbstractUserLandEventManager`. Une nouvelle classe, AbstractUserLandSlackStealer est proposée pour mutualiser le code commun à tous les voleurs de temps creux.

La plus grosse difficulté dans l'implantation consiste dans le calcul des temps creux. Nous l'avons vu, il est nécessaire d'ajouter des opérations au début et à la fin de chaque tâche périodique. Pour cela, il n'y a pas d'autre solution avec la spécification actuelle que d'utiliser notre propre sous classe de `RealtimeThread` : `AbstractSlackCompatibleSchedulable`. Cette classe prend en paramètre un `Runnable` encapsulant le code à exécuter périodiquement.

Sa méthode `run()`, déclarée `final`, est donnée par la figure 8.4. Les méthodes `computeBeforePeriodic()` et `computeAfterPeriodic()` sont déclarées abstraites.

Chaque voleur de temps creux devra alors se voir associer une implantation de `AbstractSlackCompatibleSchedulable` implantant ces méthodes.

Dans le cas de *DASS* et de *MASS*, les opérations doivent être effectuées à la plus haute priorité et protégées par un verrou. Pour cela, nous avons choisi d'ajouter dans les classes implantant ces algorithmes deux *AEs* `wakeAfterPeriodic` et `wakeBeforePeriodic` auxquels nous avons attaché deux *AEHs* associés à la plus haute priorité. Lorsque la méthode `computeBeforePeriodic()`, respectivement `computeAfterPeriodic()` d'un

```
void computeAfterPeriodic() {
  synchronized (slackStealer.lock) {
    slackStealer.caller = this;
    slackStealer.wakeAfterPeriodic.fire();
  }
}

void computeBeforePeriodic() {
  synchronized (slackStealer.lock) {
    slackStealer.caller = this;
    slackStealer.wakeBeforePeriodic.fire();
  }
}
```

FIG. 8.5 – Méthodes computeBeforePeriodic() et computeAfterPeriodic() d'un Schedulable compatible avec MASS et DASS

`SlackCompatibleSchedulable` est appelée, celui-ci, possédant une référence sur le voleur de temps creux, peut appeler la méthode `fire()` de l'événement concerné et mettre à jour une référence permettant au voleur de temps creux de savoir qui l'a réveillé. La figure 8.5 illustre ces propos.

Cette solution retenue pour l'implantation des voleurs de temps creux présente un inconvénient important : si un autre type de `RealtimeThread` est utilisé dans le système, il ne sera pas intégré à l'analyse pour calculer la quantité de temps creux. Nous proposons donc dans le chapitre 9 de modifier la spécification pour ajouter plus facilement du code exécuté automatiquement à la fin et au début de chaque tâche.

Le code de ces classes est disponible dans les annexes.

Chapitre 9

Propositions d'ajouts à la spécification RTSJ

Ce chapitre présente les modifications à *RTSJ* qui permettraient de mieux y intégrer notre gestionnaire d'événements, ainsi que d'autres propositions pour améliorer l'utilisation portable au niveau utilisateur de la spécification. Les propositions faites dans ce chapitre ont été publiées dans [MM08a].

9.1 Obtenir le temps CPU au niveau utilisateur

La surveillance du temps *CPU* est une fonctionnalité qui fait défaut dans *RTSJ*, et est l'objet du point sept du *JSR* 282 :

> *7. Add a method to* `Schedulable` *and* `ProcessingGroupParameters`*, that will return the elapsed CPU time for that schedulable object (if the implementation supports CPU time)*
>
> 7. Ajouter une méthode à `Schedulable` et à `ProcessingGroupParameters` qui retourne le temps processeur pour cet objet ordonnançable (si l'implémentation supporte la surveillance du temps *CPU*).

L'exécution d'une tâche τ_i temps réel dur périodique peut être vu comme une succession de périodes pendant lesquelles elle occupe le processeur, et de périodes pendant lesquelles le processeur est occupé avec des tâches plus prioritaires. En effet, comme la politique d'ordonnancement est non oisive, si le processeur n'exécute pas τ_i alors que celle-ci est activée, c'est qu'il est occupé à une priorité plus élevée. Ainsi, pour connaitre le temps *CPU* de τ_i, il faut sommer les durées des périodes pendant lesquelles elle occupe le processeur. La première période d'occupation du processeur débute forcément lorsque la tâche démarre son traitement. Les périodes suivantes démarrent chaque fois qu'une tâche qui a préempté τ_i termine son exécution, rendant ainsi la main à τ_i. De même, la dernière période d'occupation termine lorsque τ_i termine son traitement, et les précédentes chaque fois que τ_i est préemptée, c'est-à-dire chaque fois qu'une tâche de plus forte priorité débute

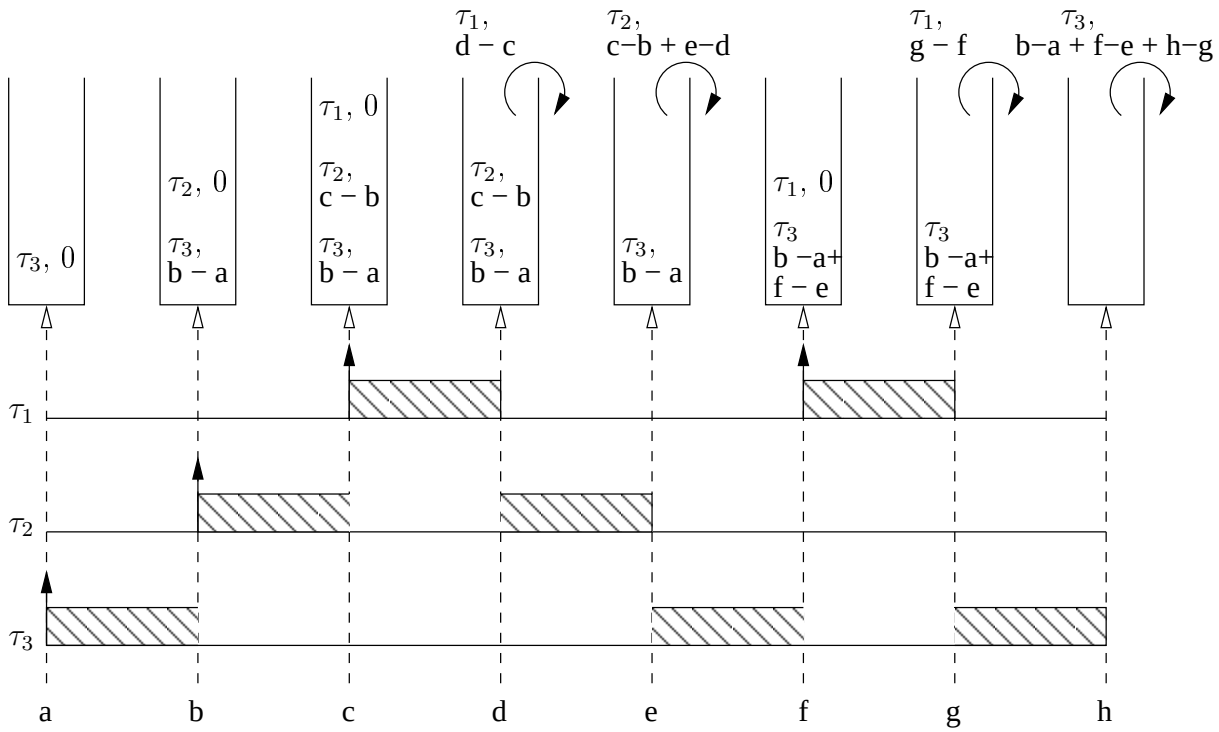


FIG. 9.1 – Obtention du temps CPU au niveau utilisateur

son exécution.

Autrement dit, les instants à surveiller correspondent tous au début du traitement d'une tâche ou à la fin de ce traitement. Si on ajoute au sommet d'une pile une tâche qui commence son exécution, et que le sommet de la pile est enlevé chaque fois qu'une tâche termine son exécution, la tâche actuellement traitée se trouve toujours en haut de cette pile. Si, en plus, la date du dernier événement est sauvegardé avant chaque opération sur la pile, on peut mettre à jour le temps consommé par chaque tâche.

La figure 9.1 illustre cette procédure : la pile y est représentée au dessus du diagramme. À l'instant a, la tâche τ_3 commence son exécution. Elle est donc empilée et le temps qu'elle a consommé est nul. À l'instant b, la tâche τ_2 commence son exécution, préemptant τ_3. Le temps consommé par τ_3 est donc maintenant de $b-a$ et τ_2 est empilée. Le même scénario se produit à l'instant c lorsque τ_1 préempte τ_2. τ_1 termine son exécution à l'instant d, elle est donc dépilée. Lorsque τ_2 termine à son tour son exécution à l'instant e, elle a consommé $c - b + e - d$ unités de temps.

Il est donc nécessaire de pouvoir ajouter du code au début et à la fin de chaque tâche. Nous avons vu dans la section 8.8 que cela était possible.

Il est par conséquent possible d'ajouter la fonctionnalité de la surveillance du temps *CPU* même si l'implémentation ne la supporte pas, à condition d'accepter d'en payer le prix en mémoire et en temps. Le prix en temps est ajouté au coût du changement de contexte, et le coût en mémoire est borné par le nombre de tâches (taille maximale de la pile).

Un inconvénient majeur de ce mécanisme est qu'il ne fonctionne que si toutes les tâches composant le système sont codées avec une sous-classe modifiée de `RealtimeThread`. Si une autre classe est utilisée pour coder une tâche, rien ne peut plus être déduit du temps séparant deux événements de type début ou fin d'une instance d'une tâche codée avec la classe modifiée. De plus il faut veiller à ce que les opérations ajoutées en fin et début d'instance soient atomiques, ou exécutées à la plus haute priorité.

Par ailleurs, si des tâches plus prioritaires que la machine virtuelle s'exécutent sur le système, les temps mesurés perdront en précision. Dans l'optique d'une approche pire cas, cela n'est pas catastrophique. En effet, les temps mesurés seront toujours plus grands que les temps réellement consommés.

Enfin, ce mécanisme ne peut fonctionner que si les tâches ne partagent pas de ressources. En cas d'inversion de priorité dû à un accès concurrent, il n'y pas pas de possibilité de maintenir la pile à jour. Les temps mesurés ne seront plus les temps consommés par chaque tâche, mais le temps passé pour chaque niveau de priorité.

Dans l'éventualité d'une *RTJVM* tournant sur un *OS* non temps réel, ou sur lequel le temps *CPU* n'est pas implanté nativement, il parait toutefois raisonnable de permettre au programmeur d'activer ce mécanisme au niveau utilisateur, sachant qu'il devra en payer le prix en mémoire et en temps.

Nous proposons donc d'intégrer ce mécanisme dans la spécification, par l'ajout de méthodes dans la classe `Scheduler` :

- `setUserlandCPUTime(boolean value);`
- `RelativeTime getCPUTime(Schedulable s)`
  `throws UnsupportedOperationException;`

La première permet d'activer ou de désactiver la surveillance du temps *CPU* au niveau utilisateur. La seconde retourne le temps *CPU* consommé par l'instance courante d'un `Schedulable` passé en paramètre. Ce temps peut être fourni par le système si cette fonction est disponible ou par la surveillance au niveau utilisateur si elle activée. Une exception est levée si le temps *CPU* n'est ni disponible au niveau du système, ni activé au niveau utilisateur.

Comme les opérations sont faites dans la classe `Scheduler`, les autres types de changements de contextes peuvent être pris en charge.

La pile nécessaire au calcul du temps *CPU* peut de plus être stockée dans la classe `Scheduler`, qui peut réutiliser pour cela la structure que l'implémentation utilise pour définir la politique d'ordonnancement, amortissant ainsi le surcoût en mémoire. Un autre intérêt de déléguer le mécanisme à l'objet représentant l'ordonnanceur est qu'il va pouvoir exécuter les opérations sur la pile à la plus haute priorité, assurant ainsi leur atomicité. Le coût du mécanisme n'a plus besoin d'être ajouté aux pires temps d'exécution des tâches utilisateur, mais est simplement ajouté au coût des changements de contexte.

Afin de pouvoir intégrer ces coûts dans l'analyse de faisabilité, nous proposons l'ajout d'une troisième méthode dans `Scheduler`, la méthode `RelativeTime getContextSwitchCost()`.

D'une part cette méthode permet une meilleure modularité pour l'analyse de faisabilité (nous reviendrons sur ce point dans la section 9.3), d'autre part le type de mécanisme utilisé pour la surveillance du temps *CPU* n'influe plus en dehors de la classe `Scheduler`.

9.2 Ajout de code en début et fin d'instance

Nous avons vu qu'il pouvait s'avérer très utile d'ajouter automatiquement du code au début et à la fin des instances de chaque tâche : pour calculer le temps *CPU* au niveau utilisateur, mais aussi pour implanter des algorithme tels que *MASS*.

Le temps d'exécution du code ajouté doit alors être pris en considération lors de l'analyse de faisabilité, ce qui implique que la classe `Scheduler`, responsable de l'analyse de faisabilité, connaisse ce temps d'exécution.

Par ailleurs, comme pour le calcul du temps *CPU*, certains traitements ne sont utiles que s'ils sont réellement exécutés par toutes les tâches, et pas seulement par celles qui sont codées avec une classe particulière. Il est parfois également nécessaire de rendre non préemptif ces traitements. Enfin, on peut vouloir exécuter du code lors de chaque changement de contexte, et pas seulement lors des préemptions « normales ».

Pour toutes ces raisons, il nous semble préférable de déléguer l'exécution de ce code directement à la classe `Scheduler`, en y ajoutant la méthode :

–
```
void addContextSwitchHandler(
        AsyncEventHandler handler, boolean beforeSchedulable,
        boolean afterSchedulable, boolean afterEachContextSwitch);
```

9.3 Analyse de faisabilité

Les méthodes permettant de réaliser l'analyse de faisabilité sont toutes situées dans les classe `Scheduler` et `RealtimeThread`. En fait, les méthodes de `RealtimeThread` sont des raccourcis faisant appel à celles de `Scheduler`. Par conséquent, pour changer l'algorithme d'analyse de faisabilité utilisé, par exemple pour prendre en considération le cas de l'utilisation d'un serveur ajournable, il faut surcharger l'ordonnanceur.

Ceci ne semble pas être justifié. En effet, on peut vouloir analyser le système d'une façon différente, sans pour autant en changer le comportement. Selon le type d'application visée, un simple test de charge peut suffire, ou encore on peut dans certains cas se contenter de l'assurance que k échéances sur m seront respectées (modèle $(m, k)firm$ [RH95]). On peut bien sûr surcharger la classe codant l'ordonnanceur en ne réécrivant que les méthodes de faisabilité, mais ce n'est pas une approche très optimisée en matière de réutilisation du code.

Nous proposons plutôt la création d'une interface `FeasibilityAnalysis`, et d'un champ de ce type dans la classe `Scheduler`, avec les méthodes `setFeasiblityAnalysis( FeasibilityAnalysis fa)` et `getFeasiblityAnalysis(FeasibilityAnalysis fa)` associées.

Les méthodes concernant la faisabilité dans la classe `Scheduler` peuvent alors déléguer leur comportement à l'objet référencé par ce champ.

On peut alors imaginer deux sous interfaces pour la classe `FeasibilityAnalysis` qui seraient `NecessaryTest` et `SufficientTest` et une interface `SufficientAndNecessaryTest` qui hériterait de ces deux dernières. Nous pouvons également fournir l'implémentation pour les classes `FixedPriorityPreemptiveLoadCondtion`, `FixedPriorityPreemptiveWorstCaseResponseTimeAnalysis`.

Chapitre 10

Évaluation des performances

Afin de comparer les algorithmes modifiés pour permettre leur implantation au niveau utilisateur dans *RTSJ*, entre eux, mais surtout avec les algorithmes non implantables, nous avons réalisé un ensemble de simulations dont nous présentons ici les résulats.

Ces simulations ont été réalisées à l'aide d'un simulateur événementiel écrit en Java que nous diffusons sous la licence *GPL*. Ce simulateur est présenté dans l'annexe B.

10.1 Méthodes de génération et de simulation

La métrique utilisée pour comparer les performances des différents algorithmes est le *temps de réponse* moyen des tâches apériodiques. Ces temps moyens sont mesurés en fonction de la charge totale, et les expériences sont répétées pour différentes charges périodiques. De plus, nous réitérons les simulations en faisant varier le nombre de tâches périodiques qui composent le trafic.

Nous générons aléatoirement des systèmes de 2, 4, 6, 8, 10, 20, 40, 80 et 100 tâches périodiques. Nous calculons alors leurs charges, et nous conservons pour chaque famille, caractérisée par le nombre de tâches périodiques, dix systèmes pour chacune des charges suivantes : 30, 50, 70 et 90%.

Les périodes des tâches sont générées selon une loi exponentielle dans l'intervalle $[40, 2560]$ *ut*. Les coûts sont ensuite générés selon une loi de distribution exponentielle dans l'intervalle $[1, T_i]$. Nous verrons dans les sections suivantes que nous simulons des systèmes avec des échéances égales aux périodes ainsi que des systèmes avec des échéances inférieures ou égales aux périodes. Dans la seconde éventualité, nous générons les échéances selon une loi exponentielle dans l'intervalle $[C_i, T_i]$. Les priorités sont ensuite assignées selon la politique *DM*.

La faisabilité des systèmes est alors vérifiée, puis leur charge calculée. Les systèmes non faisables et ceux avec une charge différant de plus de 1% de celle recherchée sont alors rejetés.

Nous générons ensuite dix ensembles de tâches apériodiques pour un grand nombre de charges apériodiques (de 1 à 60%). Ceci nous permet de constituer une réserve de

systèmes avec un large éventail de charges totales (apériodiques plus périodiques). Les coûts des tâches apériodiques sont générés aléatoirement suivant une loi de distribution exponentielle dans l'intervalle $[1, 16]$ et les dates d'arrivée suivant une loi uniforme dans l'intervalle $[1, 100000]$. Les simulations s'arrêtent lorsque toutes les tâches apériodiques ont terminé leur exécution, lorsque cela est possible, ou lorsque 100000 *ut* se sont écoulées sans qu'aucune tâche non périodique n'ait terminé.

Une fois les simulations effectuées, nous calculons les *temps de réponse* moyens des tâches apériodiques, soit en considérant l'ensemble des simulations, soit en regroupant les systèmes par nombre de tâches périodiques, ou encore par charges périodiques.

10.2 Service en tâche de fond

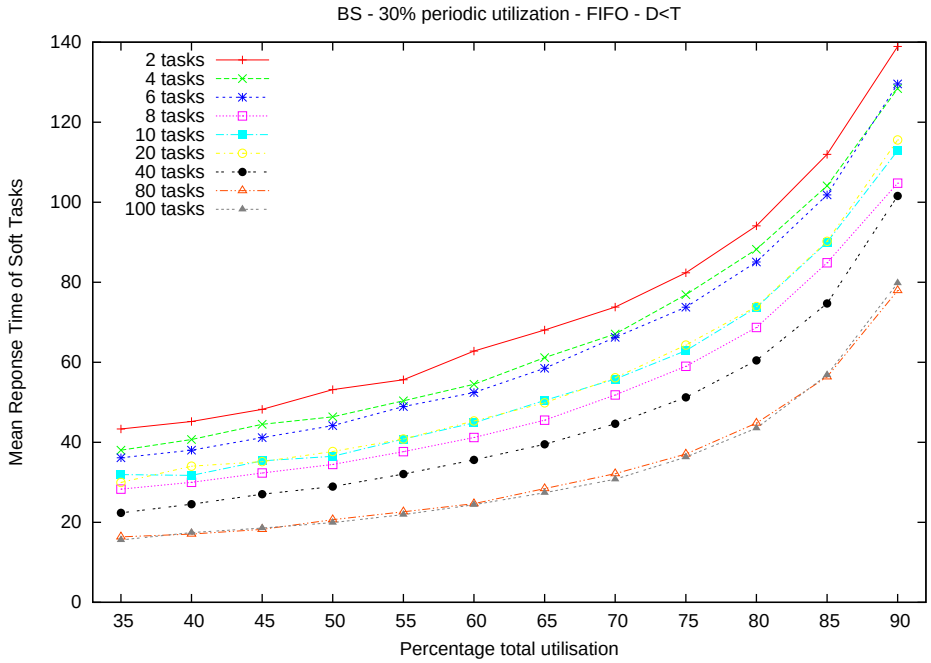
Le *BS* consiste à ordonnancer les tâches apériodiques à la plus faible priorité, dès qu'elles arrivent. L'implantation de ce mécanisme avec *RTSJ* peut se faire selon deux approches : de façon implicite en affectant aux tâches apériodiques la plus faible priorité, ou de façon explicite en utilisant l'architecture de classe que nous proposons dans le chapitre 7.

10.2.1 BS implicite

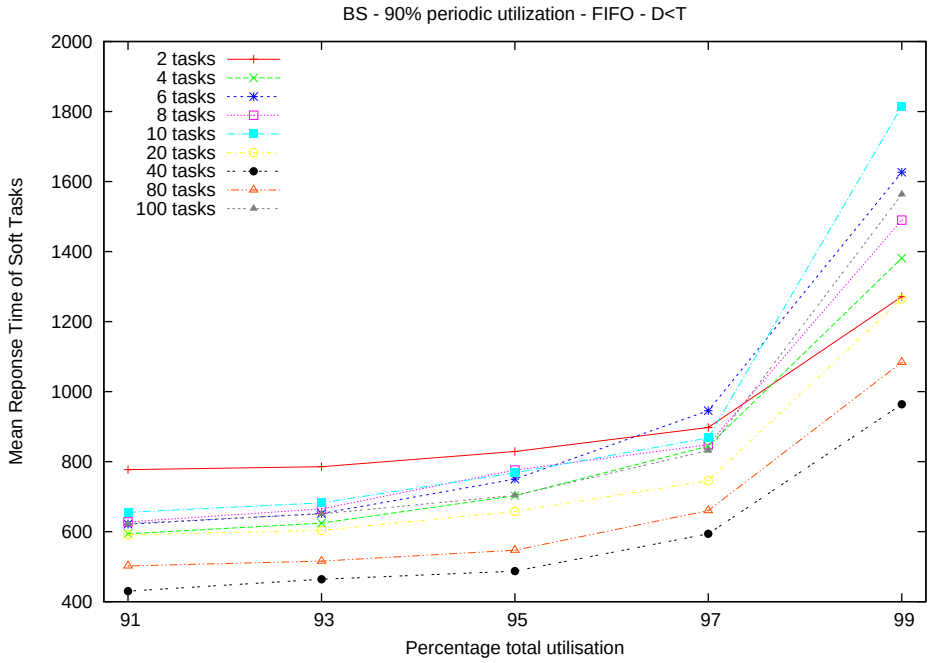
Les classes *AE* et *AEH* de la spécification permettent d'implanter un *BS* simplement en affectant la priorité temps réel la plus basse (10) à l'*AEH*. La politique de file est alors a priori *FIFO*, sans toutefois de garanties sur ce point. En effet, *RTSJ* ne spécifie pas le comportement de l'ordonnanceur face à deux tâches de même priorité.

Influence du nombre de tâches La figure 10.1 montre que le nombre de tâches composant le trafic périodique n'a pas d'influence directe sur les performances. La sous figure 10.1(a) présente les résultats obtenus pour chaque composition de trafic périodique pour une charge périodique de 30% avec une politique de *BS* où les tâches sont traitées en *FIFO*. La sous figure 10.1(b) présente les mêmes résultats pour une charge périodique de 90%.

Lorsque la charge périodique est faible, les *temps de réponse* pour un système comportant plus de tâches ont tendance toutefois à être meilleurs. Pour comprendre ce phénomène, il faut prendre l'exemple d'un cas extrême : une même charge composée soit d'une seule tâche, soit uniquement de tâches avec un coût de $1ut$. Les tâches apériodiques qui arrivent pendant l'exécution de la seule tâche périodique sont retardées dans le premier cas de figure jusqu'à la fin de son traitement. Dans le second cas de figure, elles subissent uniquement une interférence d'$1ut$ pour chaque tâche périodique activée durant leur exécution. Lorsque la charge augmente, la probabilité qu'un grand nombre de tâches périodiques soient activées durant l'exécution d'une tâche apériodique augmente et la différence entre les deux cas de figure devient moins sensible.



(a) BS - 30% - FIFO - D<T



(b) BS - 90% - FIFO - D<T

FIG. 10.1 – Influence du nombre de tâches composant le trafic périodique sur un *BS*

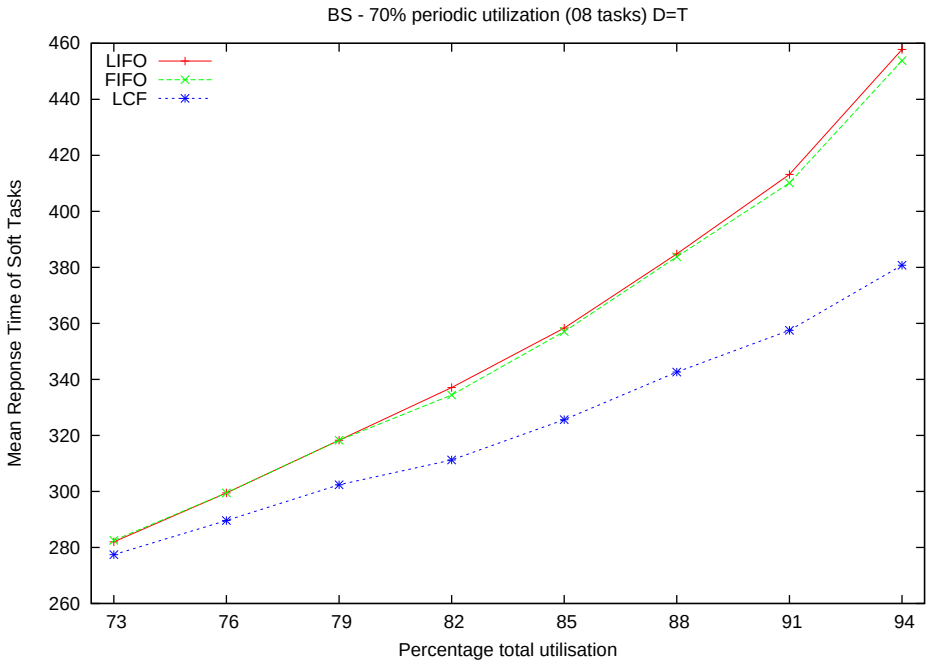


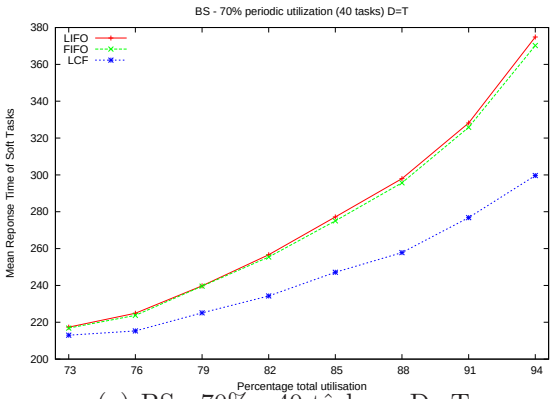
FIG. 10.2 – Influence de la politique de gestion de la file sur un *BS*

Les résultats obtenus pour des systèmes générés avec $D_i = T_i$, ou avec d'autres politiques de file amènent aux mêmes conclusions.

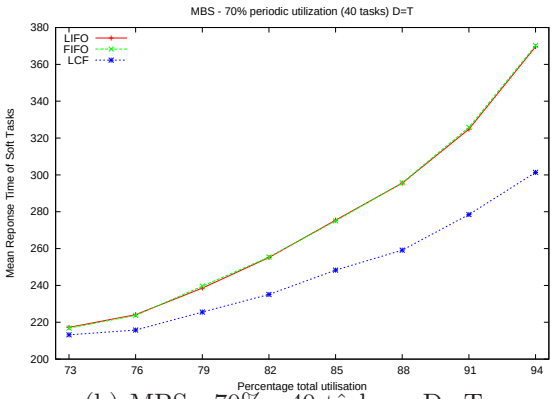
Influence de la charge apériodique Comme nous pouvons le constater sur la figure 10.1, plus la charge apériodique augmente, plus les *temps de réponse* des apériodiques gérées en *BS* augmentent. Ceci est vrai pour toutes les politiques de traitement, toutes les politiques de files, toutes les charges périodiques et toutes les possibilités de composition du trafic périodique : plus la charge apériodique est élevée, plus le temps disponible pour chaque tâche apériodique diminue.

Influence de la charge périodique Plus la charge périodique est grande, plus les *temps de réponse* moyens des tâches apériodiques sont grands. Sur la figure 10.1, on peut constater que les *temps de réponse* pour la plus mauvaise courbe dans la situation de charge périodique 30%, obtenue avec une composition de 2 tâches, sont compris entre 40 et 140*ut*. En revanche, pour une charge périodique de 90%, la meilleure courbe, obtenue avec une composition de 40 tâches, prend ses valeurs entre 400 et 1000*ut*.

Influence de la politique de gestion de la file Même si seule la politique *FIFO* est implantable, nous avons simulé les autres politiques. Il en ressort que la politique qui donne les *temps de réponse* moyens les plus petits, pour tous les algorithmes de traitement de tâches apériodiques, toutes les situations de charges et toutes les compositions de trafic périodique est la politique *LCF*. Ceci est illustré sur la figure 10.2.



(a) BS - 70% - 40 tâches - D=T



(b) MBS - 70% - 40 tâches - D=T

Charge	FIFO	LIFO	LCF
73%	216,83	217,34	212,969
76%	223,73	224,91	215,298
79%	239,6	239,77	225,123
82%	255,41	256,65	234,306
85%	275,03	277,19	247,163
88%	295,7	297,98	257,803
91%	325,84	328,19	276,810
94%	370,22	374,87	299,625
Moyenne	275,29	277,11	246,137

(c) BS - temps de réponse

Charge	FIFO	LIFO	LCF
73%	216,83	217,23	213,211
76%	223,73	224,14	215,760
79%	239,6	238,58	225,526
82%	255,41	255,15	235,098
85%	275,03	275,43	248,299
88%	295,7	295,52	259,158
91%	325,84	324,86	278,477
94%	370,22	369,60	301,350
Moyenne	275,29	275,06	247,110

(d) MBS - temps de réponse

FIG. 10.3 – Comparaison de *BS* et *MBS*

10.2.2 BS explicite

Pour pouvoir utiliser d'autres politiques de file, nous pouvons également utiliser l'ensemble de classe proposées dans le chapitre 7. Dans ce cas là, il faudra créer une nouvelle sous classe de `TaskServer`, que l'on appellera `BackgroundTaskServer`. Cette dernière pourra déléguer sa méthode `run()` à un `RealtimeThread` non périodique, de priorité 10. Ce *thread* exécutera une boucle infinie durant laquelle il traitera les événements de sa file d'attente si elle n'est pas vide. Comme pour les autres serveurs, il est impossible de démarrer une autre tâche si une première a déjà commencé et n'a pas terminé son exécution. Concrètement, pour la politique *LCF* par exemple, si une tâche de plus faible coût est activée alors qu'une première tâche a déjà démarré son exécution, le *BS* « normal » donnera lieu à une préemption (si aucune tâche plus prioritaire n'est active), alors que l'algorithme implantable avec notre mécanisme continuera le traitement de la première tâche.

Nous appelons l'algorithme modifié *Modified Background Scheduling (MBS)**. Si pour la politique *FIFO* les simulations montrent que les algorithmes *BS* et *MBS* sont strictement équivalents[1], de très légères différences sont observées pour les autres politiques. Les tableaux 10.3(d) et 10.3(c) donnent les *temps de réponse* moyens représentés par les

[1] comme attendu puisqu'il n'y a pas de préemption en *FIFO*

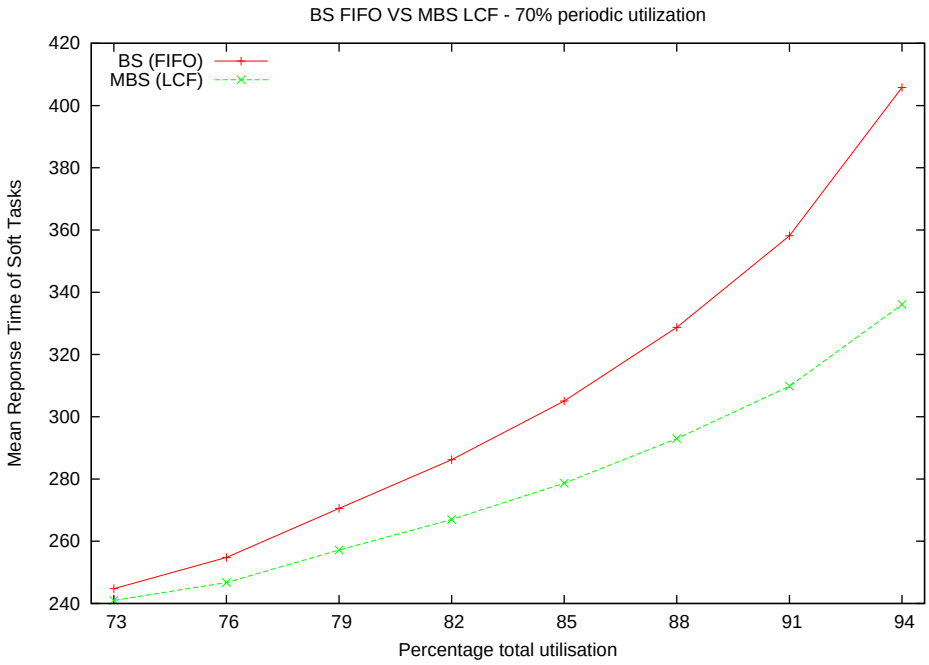


FIG. 10.4 – *BS-FIFO* et *MBS-LCF* : Moyenne des temps de réponse des apériodiques sur toutes les charges périodiques de 70%

courbes des figures 10.3(a) et 10.3(b), obtenus avec les politiques *BS* et *MBS* pour un système possédant une charge périodique de 70% composée de 40 tâches avec $D_i = T_i$. Nous constatons que les performances de *MBS* sont très légèrement supérieures pour *LIFO*, mais la tendance est inversée pour *LCF*.

Les résultats de simulations de la politique *MBS* sont semblables à ceux de la politique *BS*. Les mêmes tendances sur l'influence des différentes caractéristiques du système sur les *temps de réponse* sont observées.

10.2.3 Comparaison de BS et MBS

Dans le cas du *BS* implicite, la seule politique de gestion de file implantable est *FIFO*. Pour un *BS* explicite, la politique de gestion la plus performante est *LCF*. La figure 10.4 présente les *temps de réponse* moyens des tâches apériodiques pour une charge périodique de 70%, toutes compositions de charges périodiques confondues (nombre de tâches et génération de l'échéance). Elle illustre la supériorité de *MBS* associée à *LCF* sur *BS* associée à *FIFO*. Les résultats observés indépendamment pour les différentes compositions de charges périodiques sont similaires.

Conclusion Il est possible d'implanter avec *RTSJ* le *BS* de deux façons. La première est implicite et consiste à réserver la plus faible priorité pour tous les traitements apériodiques : dans ce cas seule la politique *FIFO* est possible pour gérer la file d'attente. La seconde, explicite, consiste à utiliser une tâche de faible priorité chargée de gérer une file

d'attente et d'exécuter les éléments de cette file : plusieurs politiques sont alors possibles pour la gestion de la file, et celle qui offre les meilleures performances est *LCF*. Il ressort des simulations que nous avons effectuées que la politique implantable la plus efficace pour traiter les tâches apériodiques en tâche de fond consiste à utiliser l'algorithme *MBS* associé à la politique de gestion de file *LCF*.

10.3 Serveur à scrutation

Le *PS* est une tâche périodique qui possède une file à laquelle sont ajoutées les tâches apériodiques lorsqu'elles surviennent. Le *PS* traite les tâches apériodiques de sa file dans les limites de sa capacité. Enfin le *PS* perd sa capacité si la file se trouve vide et cette capacité est remise à sa valeur maximale à chaque période du serveur.

L'usage de *RTSJ* pour l'implantation de cette politique apporte deux restrictions :

- le *PS* doit être la tâche la plus prioritaire du système ;
- les tâches apériodiques ne peuvent être démarrées que si le serveur dispose encore de suffisamment de capacité pour finir leur exécution.

Différentes politiques peuvent être appliquées pour la gestion de la file, mais ce n'est pas le seul facteur qui influencera les performances du serveur. En effet, ces dernières dépendent également de sa période et de sa capacité.

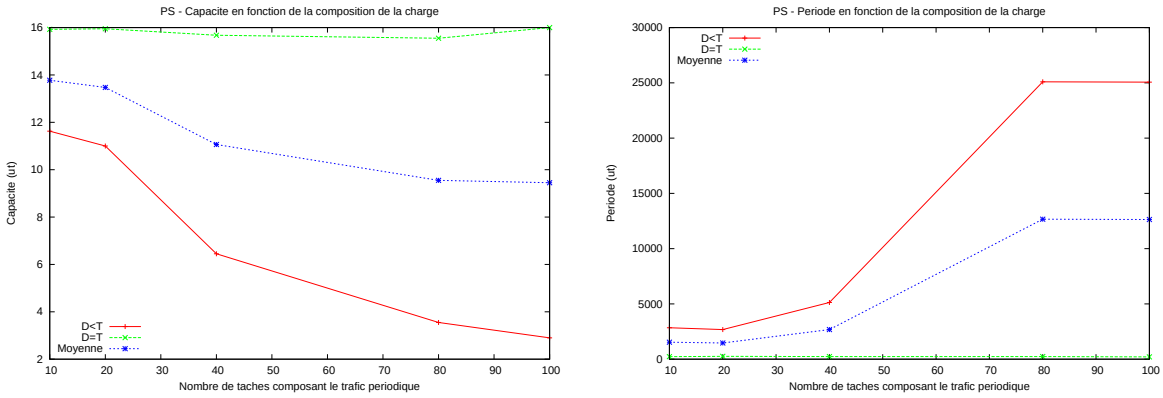
Pour obtenir les meilleures performances possibles du *PS*, nous nous proposons de maximiser la charge du système. Cette charge, notée U, se compose de la charge périodique U_T et de celle du serveur U_S.

$$\begin{aligned} U &= U_T + U_S \\ U &= \sum \frac{C_i}{T_i} + \frac{C_s}{T_s} \end{aligned}$$

La charge d'un système faisable étant bornée par 1, on déduit de cette équation la valeur maximale de C_s/T_s.

$$\frac{C_s}{T_s} \leq 1 - \sum \frac{C_i}{T_i} \tag{10.1}$$

Pour trouver une borne minimale C_s^{min} sur C_s, nous fixons la période du serveur à 2560 qui est la borne supérieure de l'intervalle dans lequel sont générées les périodes des tâches périodiques. Nous sommes alors assurés que le serveur n'interfère qu'une seule fois sur chaque tâche périodique dans le pire cas (activation synchrone). Nous recherchons alors la plus grande valeur de C_s dans l'intervalle $[1, 16]$ pour que le système soit faisable. Nous limitons C_s à 16 car il ne sert à rien que le *PS* possède une grande capacité. Mieux vaut privilégier sa période car si aucune tâche périodique n'est en attente lorsqu'il se déclenche, il perd sa capacité. Comme 16 est le coût maximal des tâches apériodiques dans nos



(a) Capacite (b) Periode

FIG. 10.5 – Période et capacité du *PS* en fonction du nombre de tâches périodiques

simulations, cette valeur constitue une bonne borne maximale pour la capacité du *PS*. Si aucune valeur de C_s n'est possible avec $T_s = 2560$, alors il n'y a aucun couple (C_s, T_s) qui permette de garder un système faisable : il ne reste qu'à exécuter les tâches apériodiques en *BS*. En effet, une période plus grande du serveur ne changerait rien puisqu'il reste la tâche la plus prioritaire, et une période plus petite ne ferait qu'augmenter son interférence sur les autres tâches.

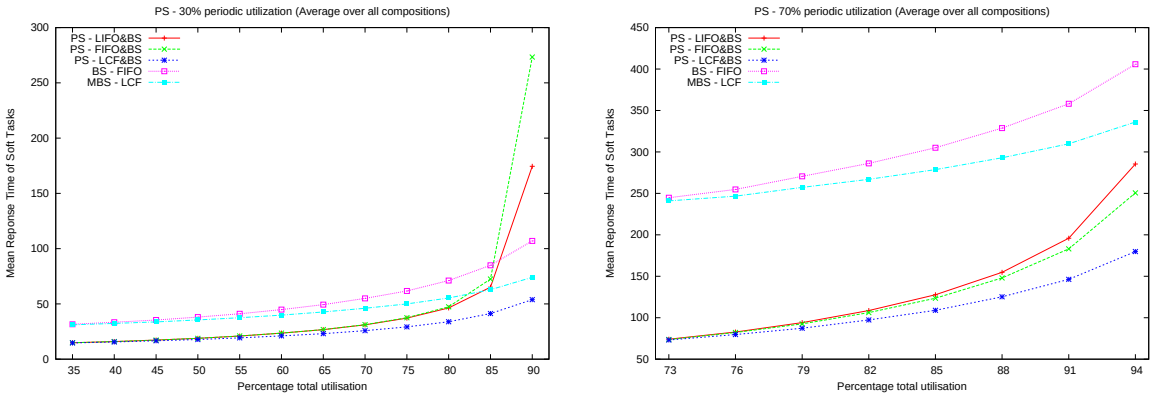
Nous recherchons ensuite par dichotomie la plus petite valeur possible de T_s dans l'intervalle $[C_s^{min}/(1 - \sum C_i/T_i), 2560]$. Pour chaque valeur de T_s considérée, nous testons des valeurs décroissantes de capacité dans l'intervalle $[C_s^{min}, T_s(1 - \sum C_i/T_i)]$.

Toutes les tâches apériodiques de coût supérieur à la capacité sont traitées en *BS* implicite.

10.3.1 PS non modifié

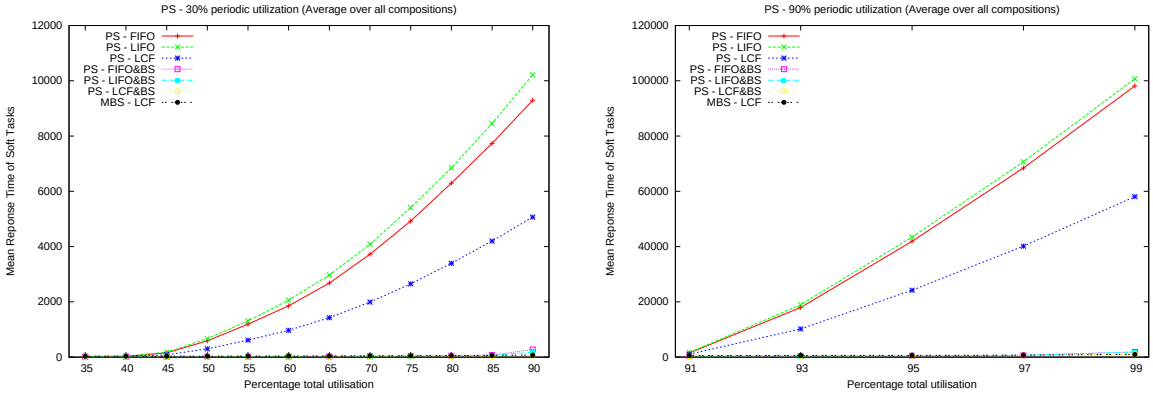
Le serveur s'exécute à la plus haute priorité, par conséquent, pour une capacité et une période fixée, la charge périodique comme le nombre de tâches composant cette charge n'influent pas sur le comportement du serveur. En revanche, plus la charge périodique augmente, et plus le nombre de tâches composant ce trafic augmente, plus on s'attend à voir diminuer la capacité maximale et augmenter la période minimale du serveur pour que le système reste faisable.

Influence du nombre de tâches périodiques Les figures 10.5(a) et 10.5(b) présentent respectivement l'évolution de la capacité et de la période des *PS* en fonction du nombre de tâches périodiques. La période augmente globalement avec le nombre de tâches, tandis que la capacité diminue. Le phénomène est moins prononcé pour les ensembles de tâches générés avec $D_i = T_i$.



(a) Charge Périodique de 30% (b) Charge Périodique de 70%

FIG. 10.6 – *PS* avec duplication : Résultats moyens pour l'ensemble des compositions de trafic périodique, charges périodiques de 30 et 70%



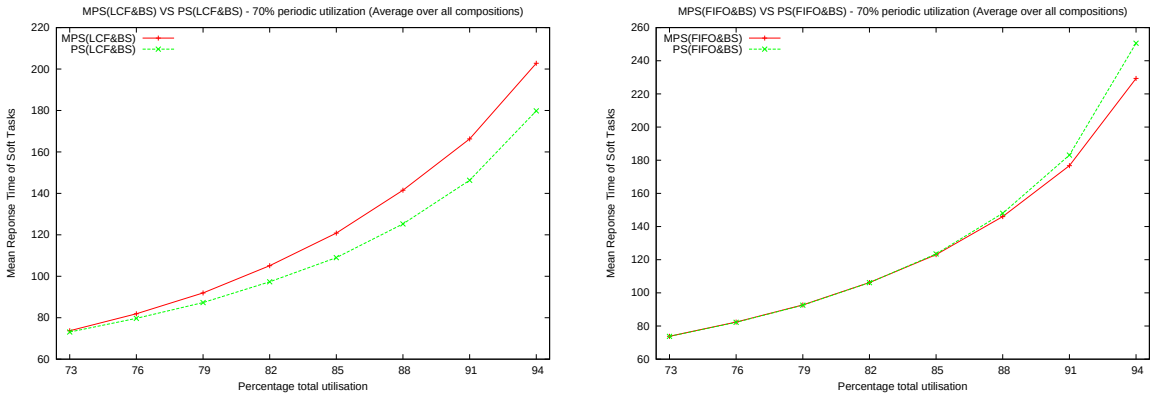
(a) Charge périodique de 30% (b) Charge Périodique de 90%

FIG. 10.7 – *PS* : Résultats moyens pour l'ensemble des compositions de trafic périodique, charges périodiques de 30 et 90%

Influence de la charge périodique Si l'on considère les résultats moyens sur l'ensemble des compositions possibles de trafic périodique (génération de l'échéance et nombre de tâches), on observe comme attendu une baisse des performances corrélée avec l'augmentation de la charge périodique. Ces résultats sont illustrés pour les charges de 30 et 70% dans la figure 10.6.

Influence de la charge apériodique Les *temps de réponse* augmentent avec la charge apériodique. Les artefacts constatés pour le *BS* ne sont plus présents.

Influence de la politique de gestion de la file La meilleure politique de gestion est sans contestes *LCF*, comme l'illustrent les figures 10.6 et 10.7. De plus, notons que sans la duplication *BS*, les performances sont très largement inférieures aux politiques *BS*, pour toutes les situations de charges, comme l'illustre la figure 10.7.



(a) LCF&BS (b) FIFO&BS

FIG. 10.8 – Comparaison de *PS* et *MPS*, résultats moyens pour une charge périodique de 70%

Comparaison avec *BS* En revanche, lorsque la duplication est activée, les performances du *PS* avec la politique *LCF* sont toujours meilleures que le *BS*. Ceci est déjà illustré par la figure 10.6 et vérifié pour toutes les situations de charges et de compositions de trafic périodique.

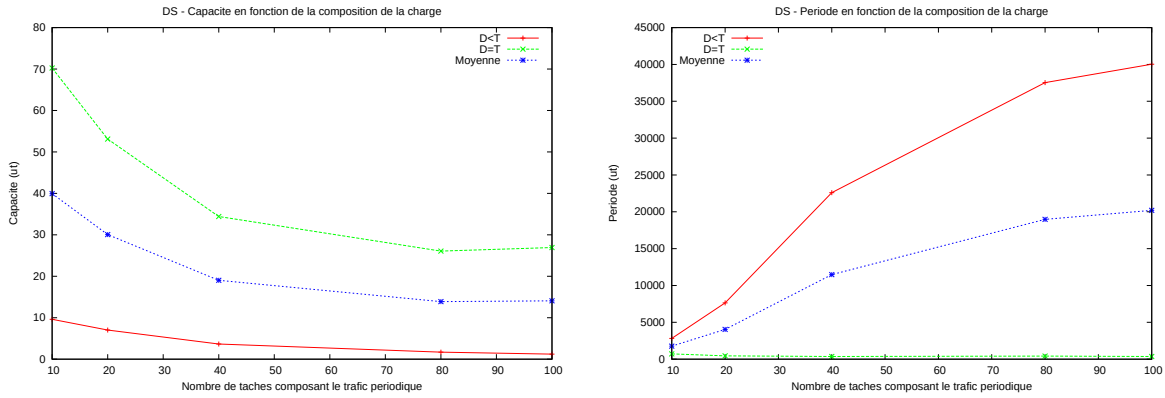
10.3.2 PS modifié

Les remarques faites sur *PS* sont vérifiées de la même façon pour *Modified Polling Server (MPS)**.

Le *PS* modifié ne peut démarrer le traitement d'une tâche que si celle-ci peut être complétée durant la période courante. On s'attend donc à une dégradation des *temps de réponse*, puisque plus de capacité risque d'être gâchée. C'est en pratique ce que l'on observe pour la politique *LCF*. La figure 10.8 l'illustre. Nous remarquons toutefois que les comportements des deux politiques sont comparables. Nous remarquons également que pour la politique *FIFO*, le *MPS* peut s'avérer plus performant que le *PS*. Ceci peut être expliqué par la duplication : pour *PS*, le répliqua pris en charge par le serveur peut démarrer son exécution, être interrompu, puis lorsque la capacité du serveur est restaurée reprendre son exécution, alors que le répliqua traité en *BS* a pu entre temps s'exécuter ; au contraire pour *MPS*, si le répliqua traité par le serveur commence, il termine forcément dans cette instance du serveur. L'interférence entre les deux répliquas existe dans les deux cas, mais elle est moins importante pour le *MPS*.

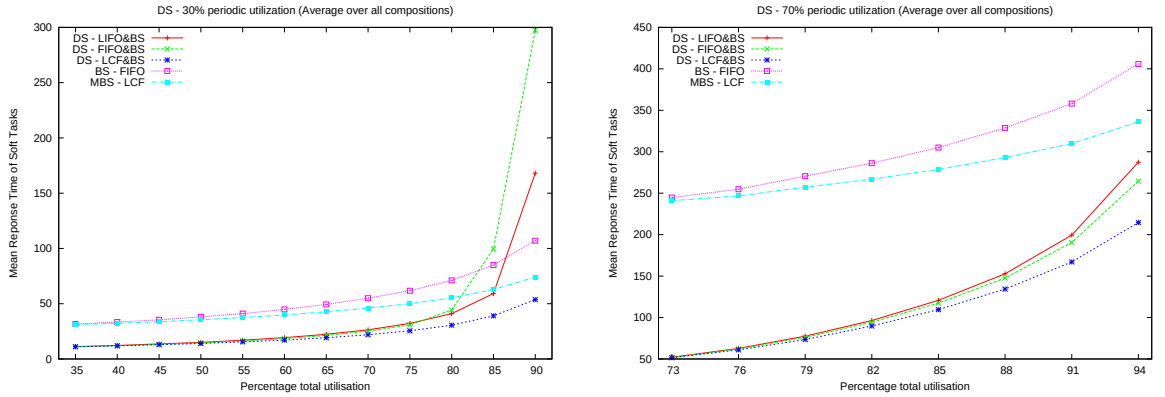
10.4 Serveur ajournable

Comme pour le *PS*, on a cherché à générer des *DS* avec des caractéristiques permettant de maximiser la charge totale du système. Contrairement au *PS*, le *DS* conserve sa capacité. Il n'est donc plus nécessaire d'essayer de maximiser sa période si c'est au détriment



(a) Capacite (b) Periode

FIG. 10.9 – Période et capacité du *DS* en fonction du nombre de tâches périodiques



(a) Charge Périodique de 30% (b) Charge Périodique de 70%

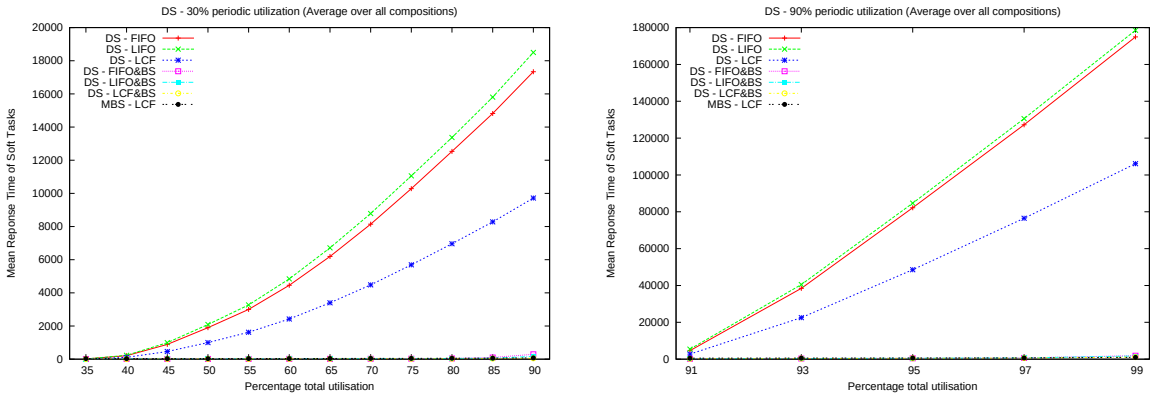
FIG. 10.10 – *DS* avec duplication : Résultats moyens pour l'ensemble des compositions de trafic périodique, charges périodiques de 30 et 70%

de sa capacité. Pour générer les caractéristiques des *DS*, on recherche donc par dichotomie la plus petite période possible pour une capacité de 2560. Si aucune période ne permet d'obtenir un système faisable, on diminue progressivement la capacité.

10.4.1 DS non modifié

Influence du nombre de tâches Les figures 10.9(a) et 10.9(b) présentent respectivement l'évolution de la capacité et de la période des *DS* en fonction du nombre de tâches périodiques. Comme pour le *PS*, la période augmente globalement avec le nombre de tâches, tandis que la capacité diminue.

Influence de la charge périodique Comme pour le *PS*, les performances se dégradent quand la charge périodique augmente, comme l'illustre la figure 10.10 qui présente les résultats moyens du *DS* sur l'ensemble des compositions de trafic périodique donnant des



(a) Charge périodique de 30%

(b) Charge Périodique de 90%

FIG. 10.11 – *DS* : Résultats moyens pour l'ensemble des compositions de trafic périodique, charges périodiques de 30 et 90%

charges de 30 et 70%.

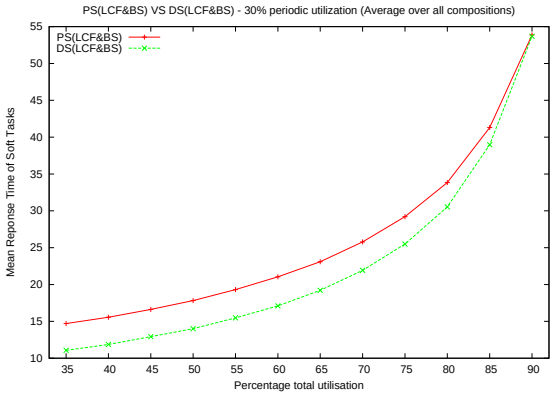
Influence de la charge apériodique Très logiquement, plus la charge apériodique augmente, plus *temps de réponse* moyens augmentent.

Influence de la politique de gestion de la file Comme pour les autres gestionnaires de tâches apériodiques, la politique de gestion de file *LCF* s'avère être la plus efficace, ce que l'on peut constater sur les figures 10.10 et 10.11. De plus, la duplication *BS* s'avère indispensable puisque sans elle les performances sont très largement inférieures a celles des politiques *BS*, pour toutes les situations de charges, comme l'illustre la figure 10.11.

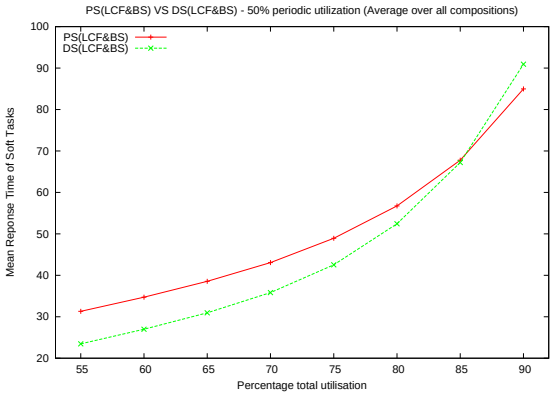
Comparaison avec *PS* La figure 10.12 présente une comparaison des résultats moyens pour le *PS* et le *DS* pour les différentes charges périodiques. Le *PS* semble bien mieux supporter la montée en charge. Pour une charge périodique de 30%, le *PS* reste meilleur que le *DS* quelque soit la charge apériodique. En revanche, pour les autres conditions de charges périodiques, les courbes se croisent. Plus la charge périodique est élevée, plus les courbes se croisent tôt. Même si théoriquement le *DS* est plus performant que le *PS*, ce dernier permet une borne d'utilisation processeur plus élevée, à cause de l'effet *double hit* qui change l'analyse de faisabilité d'un système en présence d'un *DS*. Cette baisse de performance constatée lors de la montée en charge correspond à une augmentation des cas où il n'a pas été possible de trouver des paramètres corrects de priorité et de période pour le serveur.

10.4.2 DS modifié

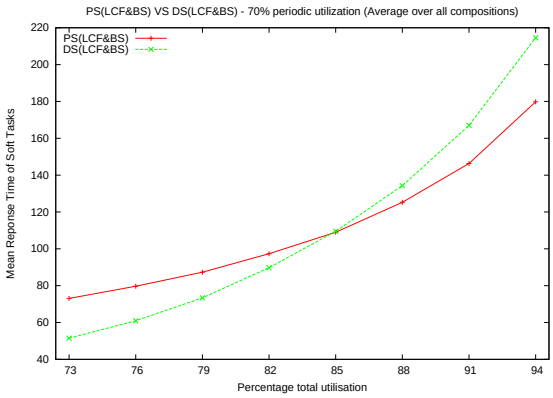
Toutes les observations faites pour le *DS* sont vérifiées pour le *Modified Deferrable Server (MDS)**.



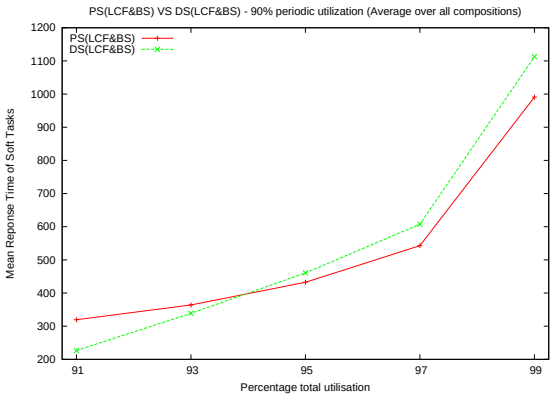
(a) Charge périodique de 30%



(b) Charge périodique de 50%

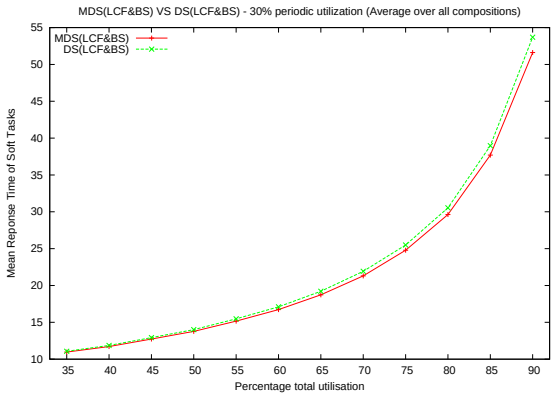


(c) Charge périodique de 70%

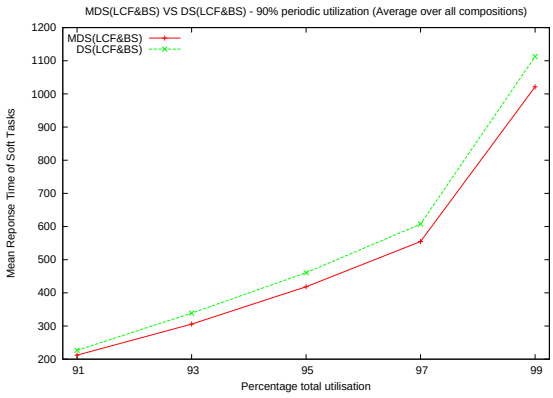


(d) Charge périodique de 90%

FIG. 10.12 – Comparaison de *PS* et *DS*



(a) Charge périodique de 30%



(b) Charge périodique de 90%

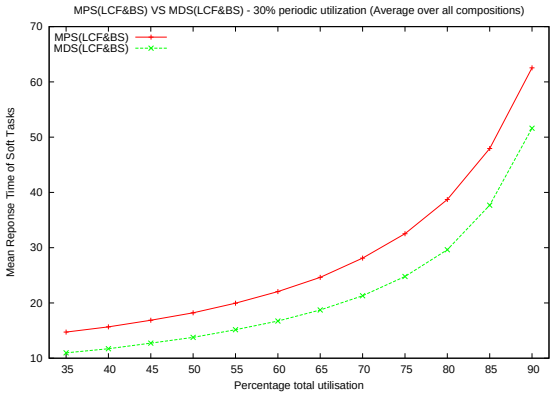
FIG. 10.13 – Comparaison de *DS* et *MDS*

Comparaison DS et MDS Les deux politiques se comportent de la même façon (voir figure 10.13). Pour la politique *LCF* avec la duplication activée, on constate de meilleures performances avec *MDS*. Comme pour le *PS* en *FIFO*, cela peut s'expliquer par le fait que lorsque l'on autorise la préemption des tâches servies, il peut arriver qu'une tâche commence son exécution dans le serveur, mais que cela soit finalement le dupliqua exécuté en *BS* qui termine l'exécution. On a alors passé du temps à exécuter une partie de la tâche à la plus haute priorité, alors que dans le cas non préemptif cette capacité est conservée pour traiter éventuellement une autre tâche plus tard.

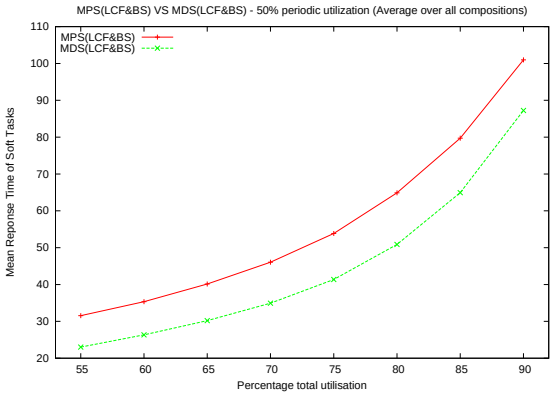
Comparaison MPS et MDS Le *MPS* offre des performances dégradées par rapport au *PS*, tandis que le *MDS* a plutôt tendance à offrir de meilleures performances que le *DS*. La politique *MDS* est de ce fait meilleure que *MPS* dans presque tous les cas, malgré une borne d'utilisation du *CPU* plus faible. La figure 10.14 présente les moyennes des résultats obtenus pour les différentes compositions de charges périodiques.

10.5 Vol de temps creux

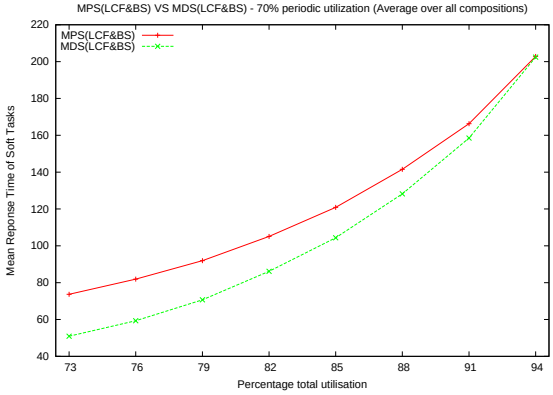
Pour chaque méthode de calcul ou d'approximation des temps creux, nous comparons les résultats d'une exécution préemptive à ceux d'une exécution non préemptive. Les algorithmes évalués sont *Exact Slack Stealing (ESS)**, *DASS* modifié pour ne calculer les temps creux que lorsqu'une tâche périodique termine et *MASS*. Les acronymes *OESS**, *ODASS** et *OMASS** désignent respectivement une version préemptive de *ESS*, *DASS* et *MASS*.



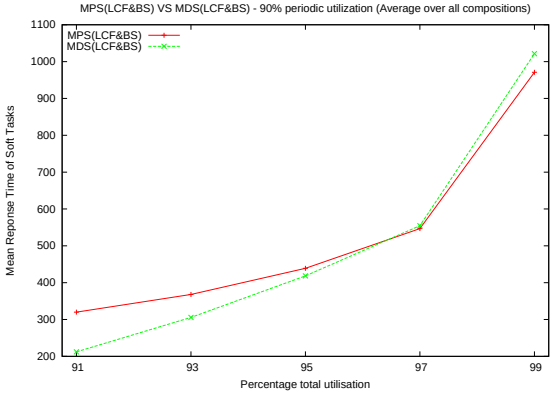
(a) Charge périodique de 30%



(b) Charge périodique de 50%

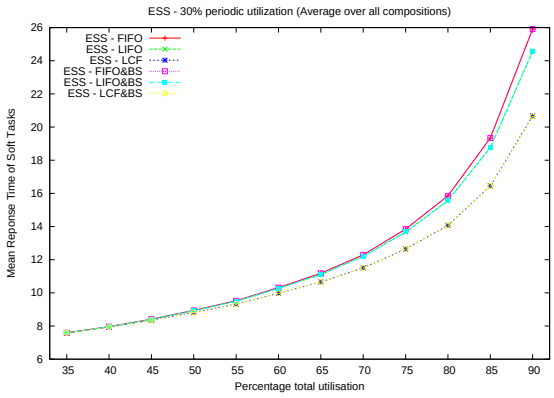


(c) Charge périodique de 70%

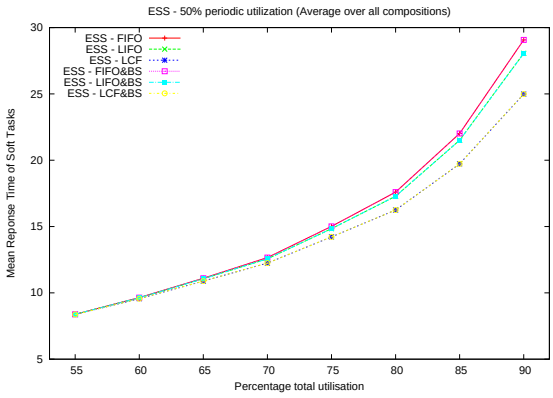


(d) Charge périodique de 90%

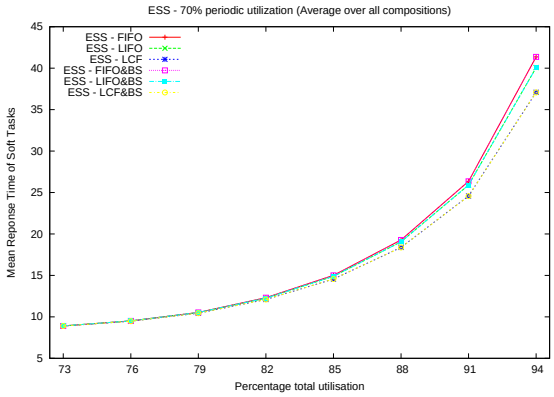
FIG. 10.14 – Comparaison de *MPS* et *MDS*



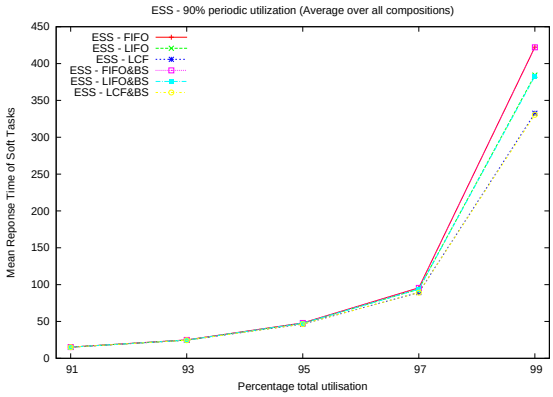
(a) Charge périodique de 30%



(b) Charge périodique de 50%



(c) Charge périodique de 70%



(d) Charge périodique de 90%

FIG. 10.15 – Temps de réponse moyens pour l'ensemble des compositions possibles de charges périodiques pour *ESS*

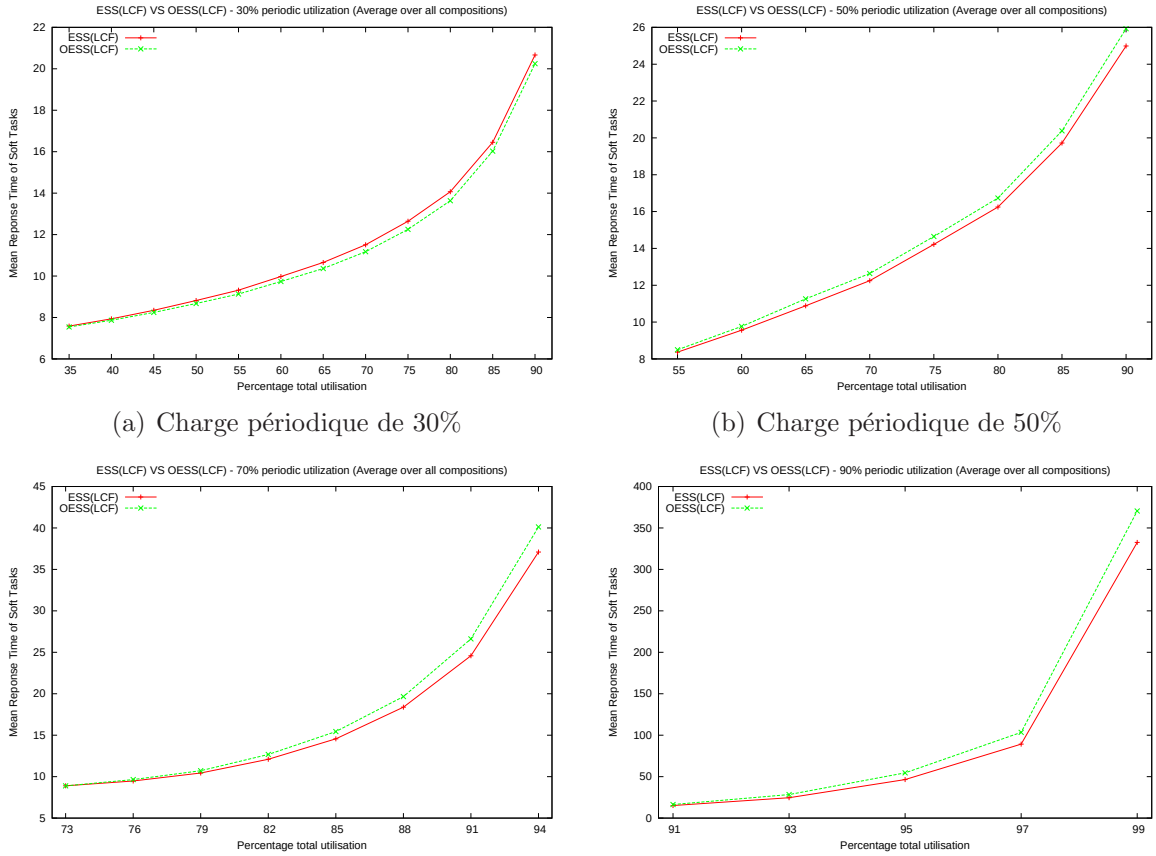
10.5.1 Valeurs exactes des temps creux

Les résultats des simulations sur l'algorithme *ESS* sont présentés dans la figure 10.15. Comme pour les autres politiques, les meilleures performances sont obtenues avec la politique *LCF*. La duplication ne change pas les résulats moyens obtenus : on voit sur la figure que les courbes sont confondues et une comparaison des valeurs obtenues le confirme. Cela signifie que peu ou pas de tâches ont un coût supérieur à la quantité de temps creux maximale dans le système, et que la très grande majorité d'entre elles sont prises en charge par *ESS*, même si le début de leur exécution est reporté en raison de l'ordonnancement non préemptif.

Ces résultats nous serviront d'étalon pour mesurer les performances de *DASS* et de *MASS*.

La figure 10.16 présente une comparaison avec les résultats obtenus pour une exécution préemptive des tâches apériodiques.

Il est très intéressant de noter que pour des charges périodiques supérieures ou égales à 50%, l'utilisation non préemptive des temps creux (*ESS*) donne de meilleurs résultats que l'exécution immédiate avec préemption (*OESS*).



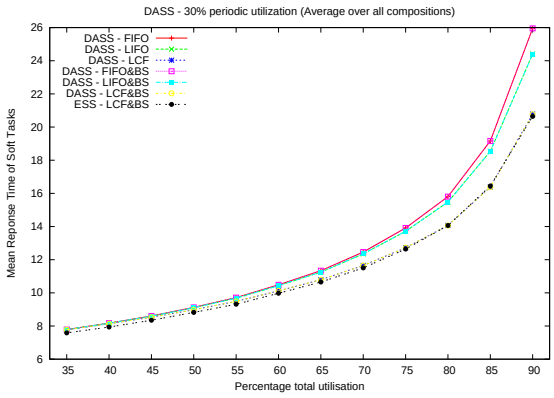
(a) Charge périodique de 30%

(b) Charge périodique de 50%

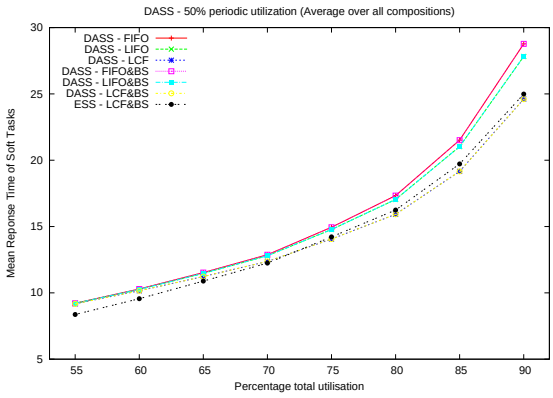
(c) Charge périodique de 70%

(d) Charge périodique de 90%

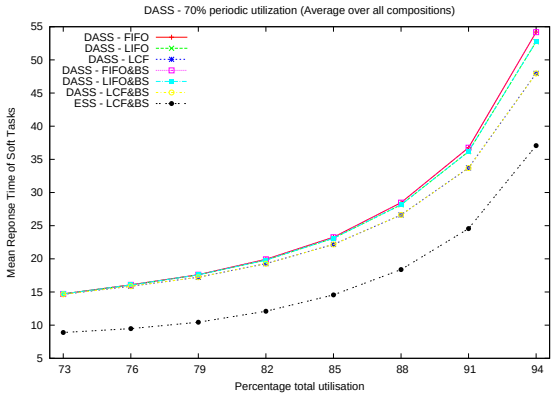
FIG. 10.16 – Comparaison des temps de réponse moyens obtenus pour toutes les compositions de charges périodiques avec *ESS* et *OESS*



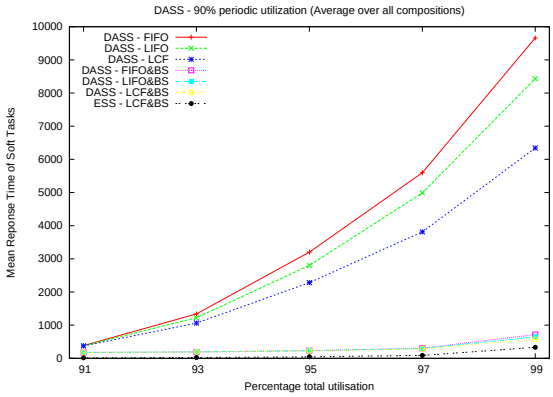
(a) Charge périodique de 30%



(b) Charge périodique de 50%



(c) Charge périodique de 70%



(d) Charge périodique de 90%

FIG. 10.17 – Temps de réponse moyens pour l'ensemble des compositions possibles de charges périodiques pour *DASS*

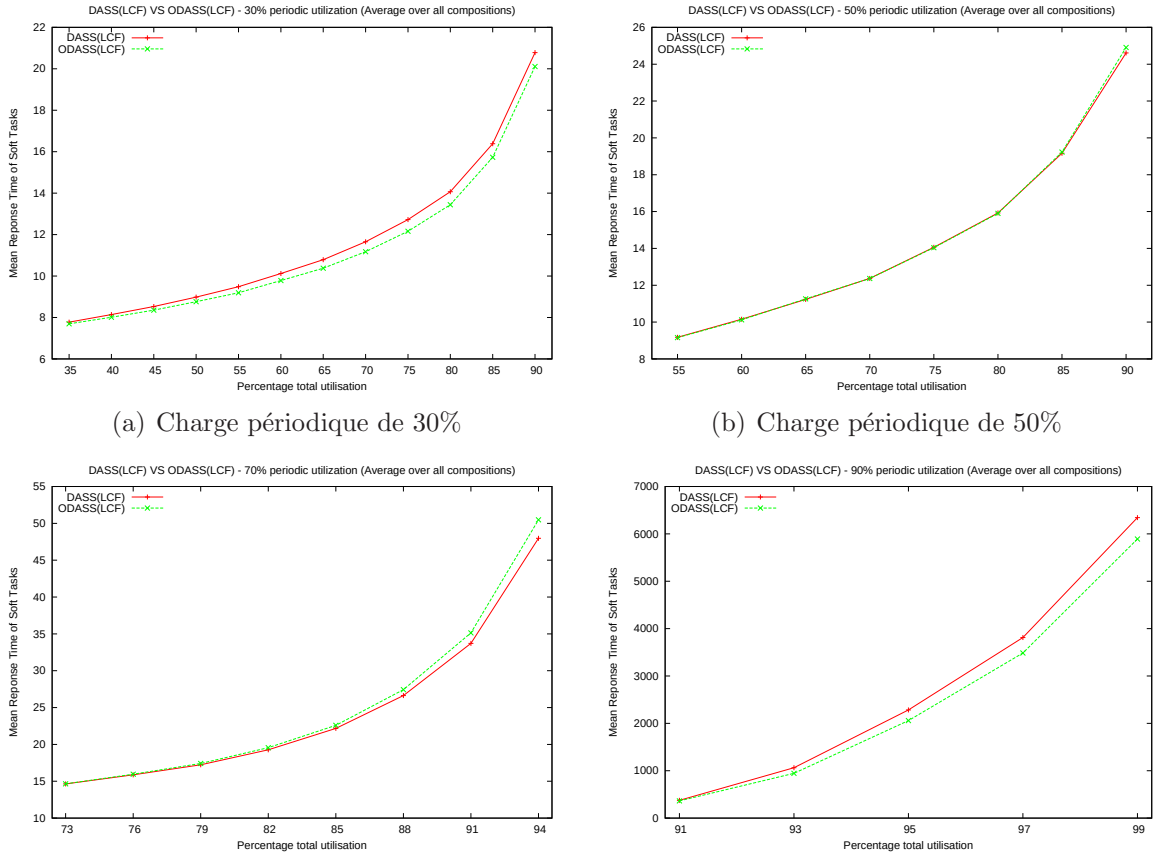
Rappelons en effet la non optimalité du vol de temps creux pour résoudre le problème de minimisation des *temps de réponse*, illustrée par la figure 4.3 page 60.

10.5.2 Temps creux calculés avec DASS

Les résultats des simulations sur l'algorithme *DASS* sont présentés dans la figure 10.17. Comme pour les autres politiques, les meilleures performances sont obtenues avec la politique *LCF* et la duplication en *BS* activée. On peut voir que les performances restent proches de celles obtenues avec *ESS*. Plus la charge périodique augmente, plus la différence augmente, mais même pour le cas extrême d'une charge de 90%, l'ordre de grandeur reste le même.

Il est également intéressant de noter que la duplication n'apporte pas d'amélioration sensible, sauf lorsque la charge périodique devient réellement très importante (90%), c'est-à-dire lorsque les performances de *DASS* décroissent.

La figure 10.18 présente une comparaison avec les résultats moyens obtenus pour une exécution préemptive des tâches apériodiques pour toutes les situations de charges périodiques. Comme avec *ESS*, la politique non préemptive apporte de meilleurs résultats



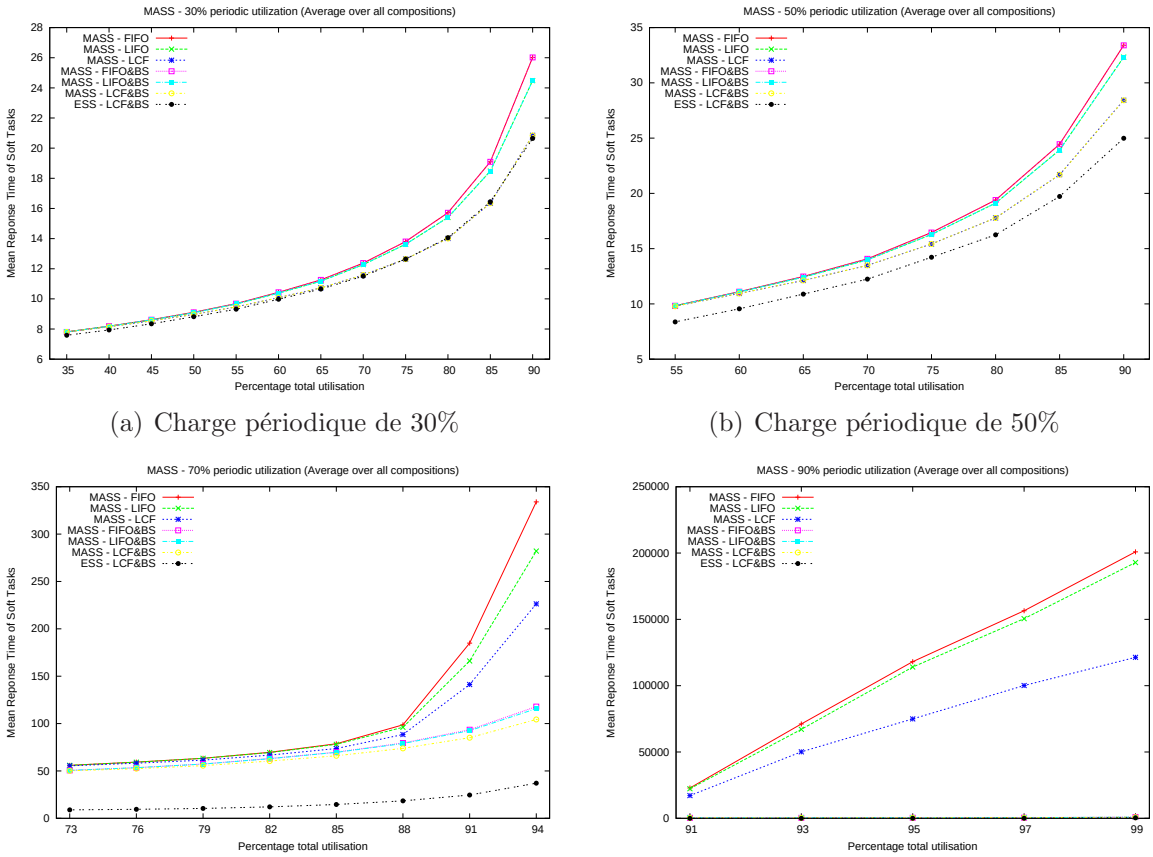
(a) Charge périodique de 30%

(b) Charge périodique de 50%

(c) Charge périodique de 70%

(d) Charge périodique de 90%

FIG. 10.18 – Comparaison des temps de réponse moyens obtenus pour toutes les compositions de charges périodiques avec *DASS* et *ODASS*



(a) Charge périodique de 30%

(b) Charge périodique de 50%

(c) Charge périodique de 70%

(d) Charge périodique de 90%

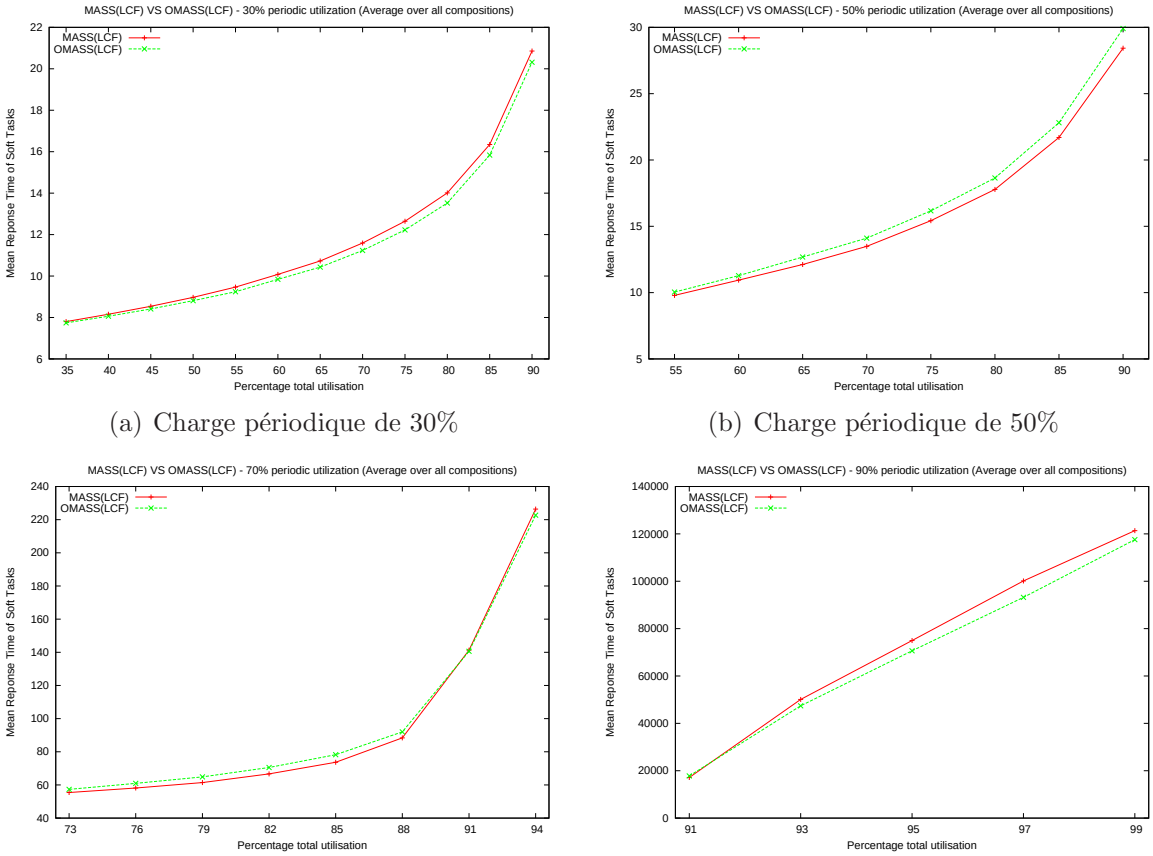
FIG. 10.19 – Temps de réponse moyens pour l'ensemble des compositions possibles de charges périodiques pour *MASS*

dès que la charge périodique dépasse 50%. Contrairement à *ESS*, la situation s'inverse à nouveau lorsque la charge périodique passe à 90%.

10.5.3 Temps creux calculés avec MASS

Les résultats des simulations sur l'algorithme *MASS* sont présentés dans la figure 10.19. Pour des charges périodiques de 30% et de 50%, les mêmes remarques que pour *DASS* peuvent être faites : la meilleure politique est *LCF* avec la duplication activée, les performances de toutes les politiques restant dans le même ordre de grandeur. Pour une charge périodique de 70%, lorsque la charge apériodique augmente, la duplication commence à apporter un gain sensible de performance, ce que nous ne constations pas avec *DASS*. Pour une charge périodique de 90%, comme pour *DASS*, la duplication devient obligatoire pour garder des performances raisonnables.

La figure 10.20 vient renforcer les résultats déjà obtenus sur les simulations de *ESS* et *DASS* : l'ordonnancement non préemptif des tâches apériodiques avec un voleur de temps creux ne dégrade pas forcément les performances.



(a) Charge périodique de 30%

(b) Charge périodique de 50%

(c) Charge périodique de 70%

(d) Charge périodique de 90%

FIG. 10.20 – Comparaison des temps de réponse moyens obtenus pour toutes les compositions de charges périodiques avec *MASS* et *OMASS*

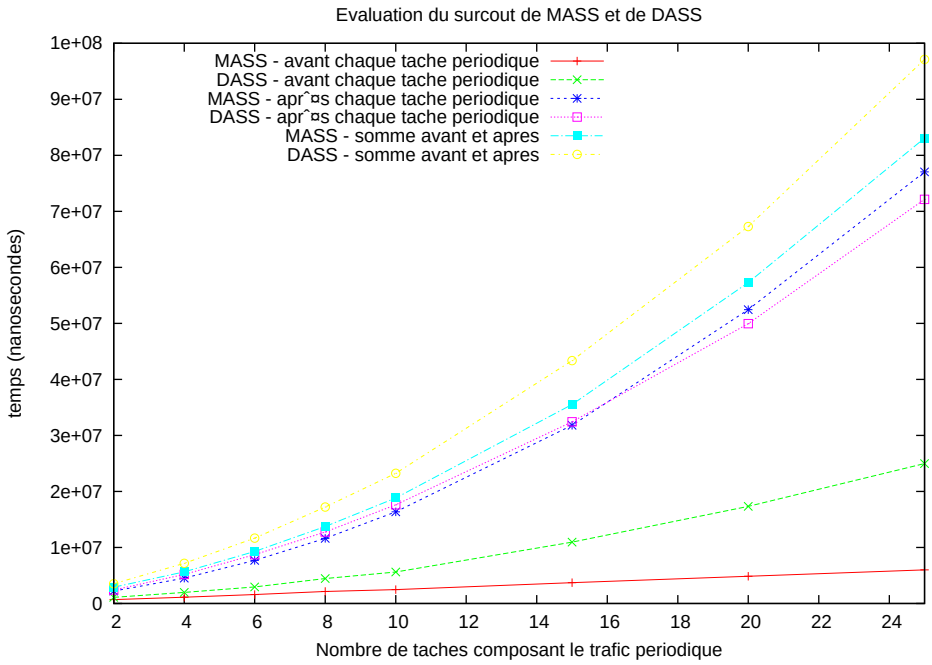


FIG. 10.21 – Surcoûts des ajouts avant et après chaque instance pour *DASS* et *MASS*

10.6 Surcoûts des politiques DASS et MASS

Les implantations de *DASS* et *MASS* que nous proposons ne diffèrent que par le code ajouté au début et à la fin de chaque instance de chaque tâche périodique.

Nous avons donc pour chacun de ces deux algorithmes mesuré le temps passé a exécuter ce code. Pour effectuer nos mesures, nous avons utilisé la machine virtuelle *Jamaïca VM*, un noyau *real-time linux free* 2.6.9 sur un processeur Intel(R) Pentium(R) M cadencé à 1,2GHz.

Comme la complexité du code inséré à la fin de chaque instance est linéaire en nombre de tâches pour les deux algorithmes, et que c'est également le cas pour le code inséré en début d'instance pour *DASS*, nous avons effectué les mesures en faisant varier le nombre de tâches de 2 à 25.

Chaque tâche effectue cinq instances. La période T_i de chaque tâche τ_i est générée aléatoirement dans l'intervalle $[5, 10]$ et son coût dans l'intervalle $[2, T_i]$. La première tâche générée se voit attribuer la priorité *RTSJ* 11, la seconde 12 et ainsi de suite.

La figure 10.21 présente les résultats obtenus. Le temps cumulé passé dans les deux méthodes est plus importants pour *DASS*. Plus le nombre de tâches augmente, plus le temps passé augmente, et plus la différence entre les temps obtenus avec les deux algorithmes augmente.

Même pour un faible nombre de tâches (2), ces coûts sont observables et doivent donc être intégrés à l'analyse de faisabilité. On constate que jusqu'à 10 tâches, le temps passé dans le code ajouté en fin d'instance est plus important pour *DASS* que pour *MASS*, mais que cela s'inverse ensuite. Le code effectué pour *DASS* comporte en effet plus d'instructions

pour chaque passage dans la boucle, mais bien que la complexité soit la même pour les deux algorithmes, dans le cas de *MASS* toutes les tâches sont parcourues, alors que pour *DASS* seules les tâches avec une priorité plus forte que la tâche qui vient de terminer le sont.

Le surcoût total en temps nécessaire pour chaque instance de tâche périodique étant plus important avec l'algorithme *DASS* qu'avec l'algorithme *MASS* , la borne maximale d'utilisation du processeur d'un système faisable sera également plus basse pour *DASS* que pour *MASS*. Cela justifie l'utilisation de *MASS* plutôt que *DASS*, même si nos simulations montrent que les performances théoriques de *DASS* sur des systèmes avec une charge périodique très élevée sont moins dégradées qu'avec *MASS*. C'est en effet précisément les systèmes les plus chargés qui souffriront le plus d'un surcoût élevé à prendre en compte dans l'analyse de faisabilité.

10.7 Conclusions

La figure 10.22 page suivante montre la courbe obtenue avec la meilleure politique de gestion de file pour chacune des politiques implantables au niveau utilisateur. Nous avons aussi ajouté la courbe donnée par *ESS*, car elle nous sert d'étalon.

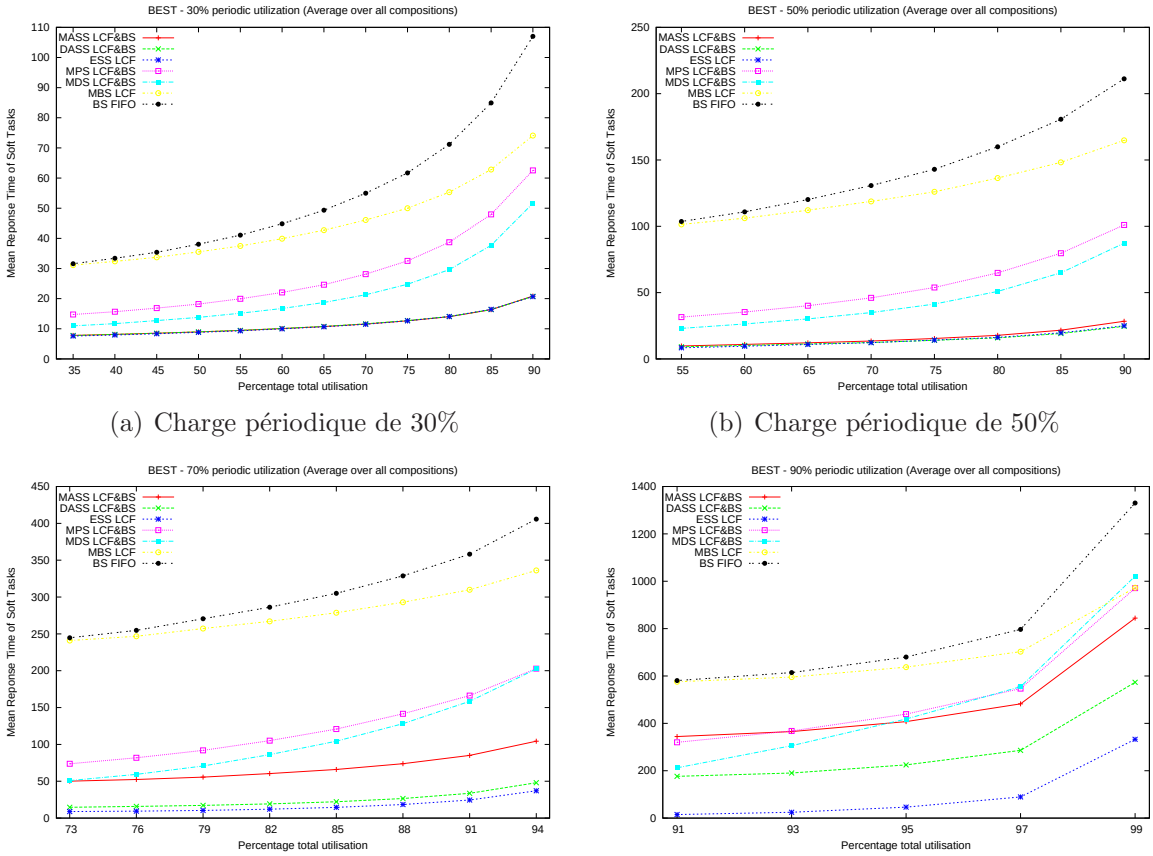
Nous remarquons que pour toutes les situations de charge, le classement par performances des algorithmes reste le même : *BS–FIFO* < *BS–LCF* < *PS&BS–LCF* < *DS&BS–LCF* < *MASS&BS–LCF* < *DASS&BS–LCF* < *ESS–LCF*.

La duplication en *BS* devient obligatoire dès que la charge périodique augmente. Sans elle, les performances des serveurs et des voleurs de temps creux sont même moins bonnes que celles du *BS*.

Pour des charges périodiques de 30% et de 50%, les trois politiques de vol de temps creux ont des performances tout à fait comparables. Lorsque la charge augmente, les performances de *MASS* décroissent plus vite que celles de *MASS* qui décroissent plus vite que celles de *ESS*.

Les trois algorithmes de vol de temps creux réagissent de la même façon à l'augmentation du trafic apériodique. Les performances des serveurs décroissent plus vite que celles des voleurs de temps creux lorsque la charge apériodique augmente.

Enfin, la restriction de l'ordonnancement non préemptif des tâches apériodiques apportée par la volonté d'implantabilité au niveau utilisateur ne dégrade pas vraiment les performances. Dans le cas du *PS*, la duplication *BS* limite fortement la baisse de performance (nous avons même constaté que pour un ordonnancement *FIFO*, *MPS* pouvait être meilleur que *PS*). Pour le *DS*, le report de l'ordonnancement des tâches apériodiques permet même d'améliorer les performances moyennes car il conserve sa capacité. Dans le cas des voleurs de temps creux, le fait de retarder le début d'un traitement n'implique pas un *temps de réponse* plus grand du fait de la non optimalité de l'utilisation des temps creux au plus tôt pour minimiser les *temps de réponse* (voir figure 4.3 page 60).



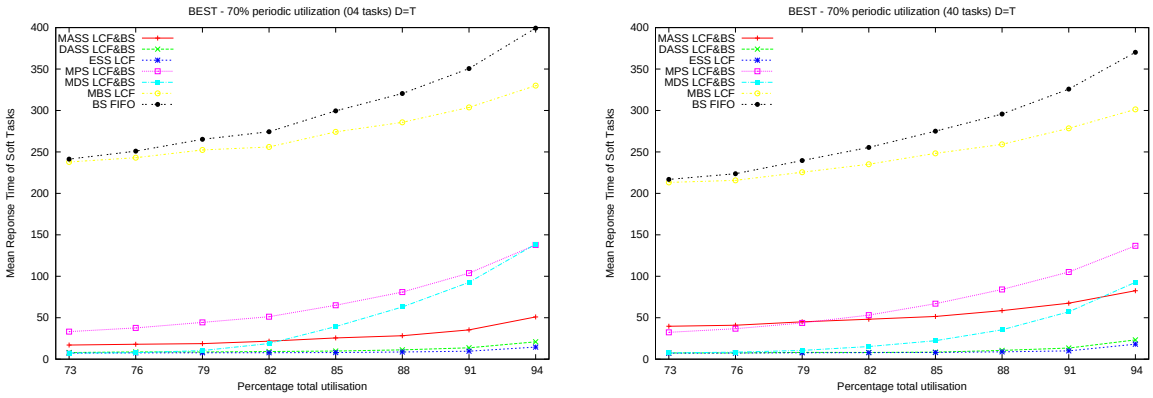
(a) Charge périodique de 30%

(b) Charge périodique de 50%

(c) Charge périodique de 70%

(d) Charge périodique de 90%

FIG. 10.22 – Temps de réponse moyens pour l'ensemble des compositions possibles de charges périodiques pour chaque politique associée à *LCF&BS*



(a) Trafic périodique de 70% composée de 4 tâches (b) Trafic périodique de 70% composée de 40 tâches

FIG. 10.23 – Comparaison des temps de réponse moyens pour une charge périodique de 70% composée de 4 tâches et de 40 tâches

L'algorithme *MASS* se révèle être une bonne solution pour répondre au problème de l'ordonnancement mixte de tâches périodiques avec des contraintes strictes et de tâches apériodiques avec des contraintes souples.

Il offre en effet des performances tout à fait comparables avec *MASS* et *ESS* pour des charges périodiques allant jusqu'à 50%.

Pour des systèmes avec de fortes charges périodiques, 70 ou 90%, les performances de *MASS* décroissent plus rapidement que celles de *DASS*. Rappelons cependant que la surcharge de travail pour le calcul des temps creux est alors plus importante, et que celle-ci n'est pas prise en compte dans nos simulations.

De plus, *MASS* supporte moins bien l'augmentation du nombre de tâches composant le trafic périodique que *DASS*, comme le montre la figure 10.23. Ceci est dû à la perte de précision des valeurs des $\bar{w}_i$ qui est rattrapée progressivement chaque fois qu'une tâche termine son exécution périodique. Plus il y a de tâches, plus longtemps la valeur reste sous estimée. Cependant, il faut encore une fois relativiser cette baisse et considérer le surcoût plus important de *DASS*, surtout que celui-ci dépend justement du nombre de tâches.

Conclusions

Les systèmes temps réel sont étudiés depuis de nombreuses années. De nombreux résultats ont été démontrés sur l'analyse de faisabilité des systèmes de tâches périodiques possédant des contraintes temps réel dur, c'est-à-dire la capacité à garantir le respect des échéances strictes de ces tâches.

Cependant, si l'on considère l'ordonnancement de tâches apériodiques à contraintes temps réel souple, la problématique n'est plus de garantir le respect des échéances, mais de minimiser les *temps de réponse*. Des solutions ont été recherchées pour résoudre ce problème sans remettre en question les résultats connus concernant les tâches périodiques.

L'approche par serveurs de tâches fut la première proposée. Elle consiste à utiliser une tâche périodique spéciale pour traiter les événements apériodiques. Il a ensuite été proposé de mettre à profit les temps creux du système (les instants d'inactivité) pour retarder le plus possible l'exécution des tâches périodiques au profit des apériodiques.

Par ailleurs, le langage de programmation Java a permis dans de nombreux secteurs de l'informatique de réduire les coûts de développement en minimisant les risques d'erreurs difficilement détectables avant l'exécution. Ceci a naturellement conduit à poser la question de son utilisation pour le développement d'applications temps réel dès la fin des années 90.

Le langage n'ayant pas été conçu au départ pour répondre aux besoins particuliers du temps réel, plusieurs consortiums se sont créés pour mettre à plat les différents problèmes et proposer des solutions. Ces consortiums ont fini par fusionner et par donner naissance à la spécification Java pour le temps réel (*RTSJ*).

Cette spécification encore jeune ne proposant pas de solutions intégrées pour répondre au problème de l'ordonnancement conjoint de tâches temps réel dur périodiques et temps réel souple apériodiques, nous avons étudié l'implantabilité et l'intégration des algorithmes proposés par la communauté temps réel.

Ces résultats algorithmiques sont proposés dans un cadre général où l'on s'autorise à discuter directement avec le matériel et ne sont donc pas implantables dans le contexte de Java. Nous avons donc cherché à adapter les solutions existantes en nous abstrayant du matériel et en ne nous appuyant que sur quelques pré requis minimaux, afin d'assurer la plus grande portabilité possible de la solution proposée.

Nous nous sommes par exemple restreints dans nos travaux à l'utilisation d'un ordon-

nanceur préemptif à priorité fixe, car c'est le seul algorithme d'ordonnancement imposé par la spécification à toutes les *RTJVMs*.

Dans cette thèse, nous montrons comment adapter les mécanismes existants de service et de vol de temps creux pour permettre leur implantation non plus dans un système d'exploitation, mais au niveau utilisateur au moyen de tâches temps réel ne disposant pas de droits particuliers (sinon celui de s'attribuer la plus haute priorité).

Nous avons dans une première partie détaillé le fonctionnement des algorithmes *PS* *DS* et *DASS*, trois algorithmes permettant de répondre efficacement au problème de l'ordonnancement conjoint évoqué ci-dessus.

Nous avons décrit dans la deuxième partie les difficultés liées à l'utilisation de Java pour programmer des applications temps réel, les différentes approches possibles pour répondre à ces difficultés, et détaillé les solutions proposées par *RTSJ*.

Nous avons dans une troisième partie proposé les algorithmes *MPS* et *MDS*, des versions modifiées de *PS* et *DS*. Nous montrons également comment il est possible d'implanter une version légèrement modifiée de *DASS*. Enfin nous proposons *MASS*, un algorithme original de calcul de temps creux au niveau utilisateur dont la complexité de mise en œuvre est très faible et se prête mieux à un environnement utilisateur.

Nous avons comparé les performances de ces algorithmes au moyen de simulations et nous proposons un ensemble de classes permettant de les implanter.

Il en résulte que l'algorithme que nous proposons, *MASS*, permet d'obtenir des *temps de réponse* moyens pour les tâches apériodiques du même ordre de grandeur que ceux obtenus avec *DASS* ou même avec un algorithme basé sur un calcul exact de la quantité de temps creux. Lorsque la charge périodique augmente, jusqu'à des situations extrêmes (90%), les performances se dégradent en revanche plus vite que celles de *DASS*. Il faut tout de même noter qu'elles restent supérieures à celles des serveurs modifiés.

Nos simulations ne tennant pas compte du surcoût des algorithmes lors de l'exécution, nous avons exécuté les deux algorithmes *DASS* et *MASS* et mesuré le temps passé dans le code nécessaire à l'estimation de la quantité de temps creux utilisable. Il s'avère que ce temps est moins important pour *MASS* car l'algorithme nécessite moins d'opérations. Ces premiers résultats confirment que dans les deux cas il faut intégrer le surcoût du code ajouté à l'analyse de faisabilité du système, ce qui réduit le nombre de systèmes ordonnançables. Comme ce surcoût est plus important pour *DASS*, la borne d'ordonnançabilité d'un système utilisant *DASS* sera plus faible qu'un système utilisant *MASS*. Autrement dit, il existe des systèmes non faisables avec *DASS* mais faisables avec *MASS*.

Ce travail a également donné lieu à la proposition d'extensions de la spécification *RTSJ* actuelle. Pour mieux intégrer ces extensions, nous proposons également des modifications,

notamment pour la classe `Scheduler`. Ces modifications vont être proposées au groupe d'experts chargés de maintenir la spécification.

Nous travaillons actuellement sur une utilisation de notre implantation du vol de temps creux pour gérer des structures de données concurrentes. Il existe en effet des structures de données « auto-balancées » dont la contrainte d'équilibrage après une opération de mise à jour est relâchée, c'est-à-dire peut être reportée à un instant ultérieur. On peut citer les arbres « chromatiques » qui sont des arbres rouge-noirs relâchés. Dans [FM08] Fauberteau et Midonnet proposent d'utiliser les temps creux pour rééquilibrer de telles structures. Nous prévoyons d'implanter la solution qu'ils proposent en utilisant l'algorithme *MASS*.

Bibliographie

[ABD+95] Neil Audsley, Alan Burns, Robert Ian Davis, K. W. Tindell et Andy Wellings : Fixed priority pre-emptive scheduling : an historical perspective. *Real-Time Systems : The International Journal of Time-Critical Computing Systems*, 8:173–198, 1995.

[ABR+93] Neil Audsley, Alan Burns, Mike Richardson, Ken Tindell et Andy Wellings : Applying new scheduling theory to static priority pre-emptive scheduling. *Software Engineering Journal*, 8:284–292, 1993.

[ABRW92] Neil Audsley, Alan Burns, Mike Richardson et Andy Wellings : Deadline monotonic scheduling theory. *In Real-Time Programming*, pages 55–60, 1992.

[AFG+04] Matthew Arnold, Stephen J. Fink, David Grove, Michael Hind et Peter F. Sweeney : A survey of adaptive optimization in virtual machines. Rapport technique, IBM Research, 2003 (updated 2004).

[Aud93] Neil Audsley : *Flexible Scheduling in Hard Real-Time Systems*. Thèse de doctorat, Department of Computer Science, University of York, UK, 1993.

[Bur91] Alan Burns : Scheduling hard real-time systems : a review. *Software Engineering Journal*, 6(3):116–128, 1991.

[But04] Giorgio C. Buttazzo : *Hard Real-time Computing Systems : Predictable Scheduling Algorithms And Applications*. Real-Time Systems. Springer-Verlag TELOS, Santa Clara, CA, USA, 2004.

[BW03] Alan Burns et Andy Wellings : Processing group parameters in the real-time specification for java. *In On the Move to Meaningfull Internet Systems 2003 : Workshop on Java Technologies for Real-Time and Embedded Systems*, volume LNCS 2889, pages 360–370. Springer, 2003.

[CC89] H. Chetto et M. Chetto : Some results of the earliest deadline scheduling algorithm. *IEEE Transactions on Software Engineering*, 15(10), 1989.

[Dav95] Robert Ian Davis : *On Exploiting Spare Capacity in Hard Real-Time Systems*. Thèse de doctorat, University of York, 1995.

[Dib08] Peter Dibble : *Real-Time Java Platform Programming : Second Edition*. BookSurge Publishing, 2008.

[DSZ90] A. Dix, R.F. Stone et H. Zedan : Design issues for reliable time-critical systems. *In* H. Zedan, éditeur : *Proceedings of 1989 conference "'Real Time Systems : Theory and Applications"*, pages 305–322, 1990.

[DTB93] Robert Ian Davis, Ken Tindell et Alan Burns : Scheduling slack time in fixed priority pre-emptive systems. *In Proceedings of the 14th IEEE Real-Time Systems Symposium (RTSS '93)*, pages 222–231, 1993.

[EJ00] Cecilia Ekelin et Jan Jonsson : Solving embedded system scheduling problems using constraint programming. *In Proceedings of the Real-Time Systems Symposium*, 2000.

[FM08] Frédéric Fauberteau et Serge Midonnet : Worst case analysis of TreeMap data structure. *In Poster session of 2nd Junior Researcher Workshop on Real-Time Computing*, pages 33–36, Rennes, France, octobre 2008.

[Foh94] Gerhard Fohler : *Flexibility in Statically Scheduled Hard Real-Time Systems*. Thèse de doctorat, Technische Universität Wien, Institüt fur Technische Informatik, 1994.

[GB00] James Gosling et Greg Bollella : *The Real-Time Specification for Java*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2000.

[JD90] Xu J. et Parnas D. : Scheduling processes with release times, deadlines, precedence, and exclusion, relations. *IEEE Transactions on Software Engineering*, 16(3):360–369, 1990.

[KFG+92] H. Kopetz, G. Fohler, G. Grunsteidl, H. Kantz, G. Pospischil, P. Puschner, J. Reisinger, R. Schlatterbeck, W. Schutz, A. Vrchoticky et R. Zainlinger : The programmer's view of mars. *In Proceedings of the Real-Time Systems Symposium*, 1992.

[KRP+94] Mark Klein, Thomas Ralya, Bill Pollak, Ray Obenza et Michael González Harbour : *A practitioner's handbook for real-time analysis : guide to rate monotonic analysis for real-time systems*. Kluwer Academic Publishers, 1994.

[Leh90] John P. Lehoczky : Fixed priority scheduling of periodic task sets with arbitrary deadline. *In Proceedings of the 11th IEEE Real-Time System Symposium*, pages 201–209, décembre 1990.

[LL73] Chung Laung Liu et James W. Layland : Scheduling algorithms for multiprogramming in a hard real time environment. *Journal of the Association for Computing Machinery*, 20(1):46–61, janvier 1973.

[LRT92] John P. Lehoczky et Sandra Ramos-Thuel : An optimal algorithm for scheduling soft-aperiodic tasks fixed priority preemptive systems. *In proceedings of the 13th IEEE Real-Time Systems Symposium*, pages 110–123, Phoenix, Arizona, décembre 1992.

[LSD89] John P. Lehoczky, Lui Sha et Y. Ding : The rate monotonic scheduling algorithm : Extact characterization and average case behavior. *In Proceedings of the Real-Time Systems Symposium*, pages 166–171, 1989.

[LSS87] John P. Lehoczky, Lui Sha et Jay K. Strosnider : Enhanced aperiodic responsiveness in hard real-time environments. *In Proceedings of the Real-Time Systems Symposium*, pages 110–123, San jose, California, décembre 1987. IEEE Computer Society.

[LW82] J. Y. T. Leung et J. Whitehead : On the complexity of fixed-priority scheduling of periodic real-time tasks. *Performance Evaluation*, 2(4):237–250, décembre 1982.

[MM05] Damien Masson et Serge Midonnet : Tolérance aux fautes temporelles et amélioration du comportement d'un système java temps réel. *In Journées Doctorales Informatique et Réseaux (JDIR'05)*, décembre 2005.

[MM06] Damien Masson et Serge Midonnet : Fault Tolerance with Real-Time Java. *In 14th International Workshop on Parallel and Distributed Real-Time Systems (WPDRTS'06)*, page 172, Rhodes Island, Greece, avril 2006. IEEE Computer Society Press. In Proceedings of the 20th International Parallel and Distributed Processing Symposium (IPDPS'08) (8 pp.).

[MM07] Damien Masson et Serge Midonnet : The Design and Implementation of Real-time Event-based Applications with the Real-time Specification for Java. *In 15th International Workshop on Parallel and Distributed Real-Time Systems (WPDRTS'07)*, page 148, Long Beach, Califonia, USA, mars 2007. IEEE Computer Society Press. In Proceedings of the 21th International Parallel and Distributed Processing Symposium (IPDPS'07) (8 pp.).

[MM08a] Damien Masson et Serge Midonnet : RTSJ Extensions : Event Manager and Feasibility Analyzer. *In 6th Java Technologies for Real-Time and Embedded Systems (JTRES'08)*, volume 343 de *ACM International Conference Proceeding*, pages 10–18, Santa Clara, California, septembre 2008. Association for Computing Machinery.

[MM08b] Damien Masson et Serge Midonnet : Slack time evaluation with RTSJ. *In Proceedings of the 23rd Annual ACM Symposium on Applied Computing (SAC'08)*, pages 322–323, Fortaleza, Ceará, Brazil, mars 2008. Short paper and poster.

[MM08c] Damien Masson et Serge Midonnet : Userland Approximate Slack Stealer with Low Time Complexity. *In 16th Real-Time and Network Systems International Conference (RTNS'08)*, pages 29–38, Rennes, France, octobre 2008. (10 pp.).

[Mok83] A. K. Mok : Fundamental design problems of distributed systems for the hard real-time environment. Rapport technique, Massachusetts Institute of Technology, Cambridge, MA, USA, 1983.

[RH95] Parameswaran Ramanathan et Moncef Hamdaoui : A dynamic priority assignment technique for streams with (m, k)-firm deadlines. *IEEE Trans. Comput.*, 44(12):1443–1451, 1995.

[RTL93] Sandra Ramos-Thuel et John P. Lehoczky : On-line scheduling of hard deadline aperiodic tasks in fixed-priority systems. *In Proceedings of the 14th IEEE Real-Time Systems Symposium (RTSS '93)*, 1993.

[SB94] M. Spuri et Giorgio C. Buttazzo : Efficient aperiodic service under earliest deadline scheduling. *In Proceedings of the IEEE Real-Time Systems Symposium (RTSS'94)*, pages 2–11, San Juan, Puerto Rico, décembre 1994. IEEE Computer Society.

[SB96] M. Spuri et Giorgio C. Buttazzo : Scheduling aperiodic tasks in dynamic priority systems. *Journal of Real-Time Systems*, 10(2), 1996.

[Ser72] O. Serlin : Scheduling of time critical processes. *In Spring Joint Computers Conference*, pages 925–932, 1972.

[Sie02] Fridtjof Siebert : *Hard Realtime Garbage Collection in Modern Object Oriented Programming Languages.* AICAS book, 2002. ISBN 3-8311-3893-1.

[SLS88] Brinkley Sprunt, John P. Lehoczky et Lui Sha : Exploiting unused periodic time for aperiodic service using the extended priority exchange algorithm. *In Proceedings of the Real-Time Systems Symposium*, pages 251–258, Huntsville, AL, USA, décembre 1988.

[SLS95] Jay K. Strosnider, John P. Lehoczky et Lui Sha : The deferrable server algorithm for enhanced aperiodic responsiveness in hard real-time environments. *IEEE Trans. Comput.*, 44(1):73–91, 1995.

[Spr90] Brinkley Sprunt : *Aperiodic Task Scheduling for Real-Time Systems.* Thèse de doctorat, Department of Electrical and Computer Engineering, Carnegie Mellon University, Pittsburgh, PA, août 1990.

[Spu96] Marco Spuri : Analysis of deadline scheduled real-time systems. Rapport technique 2772, Institut national de recherche en informatique et en automatique (INRIA), 1996.

[SRL90] Lui Sha, R. Rajkumar et John P. Lehoczky : Priority inheritance protocols : An approach to real-time synchronisation. *IEEE Transactions on Computers*, 39(9):1175–1185, 1990.

[SSL89] Brinkley Sprunt, Lui Sha et John P. Lehoczky : Aperiodic task scheduling for hard real-time systems. *Real-Time Systems : The International Journal of Time-Critical Computing Systems*, 1:27–60, 1989.

[Sta88] John A. Stankovic : Misconceptions about real-time computing. *IEEE Computer*, 21(10):10–19, 1988.

[TLS96] Too-Seng Tia, Jane W.-S. Liu et Mallikarjun Shankar : Algorithms and optimality of scheduling soft aperiodic requests in fixed-priority preemptive systems. *Real-Time Systems : The International Journal of Time-Critical Computing Systems*, 10(1):23–43, 1996.

[Wel04] Andy Wellings : *Concurrent and Real-Time Programming in Java*. John Wiley & Sons, 2004.

[WK08] Andy Wellings et MinSeong Kim : Processing group parameters in the real-time specification for java. *In proceedings of JTRES 2008*, 2008.

[ZW06] Alexandros Zerzelidis et Andy Wellings : Getting more flexible scheduling in the rtsj. *In ISORC '06 : Proceedings of the Ninth IEEE International Symposium on Object and Component-Oriented Real-Time Distributed Computing*, pages 3–10, Washington, DC, USA, 2006. IEEE Computer Society.

Acronymes

AE

`AsyncEvent`, classe *RTSJ* modélisant un événement asynchrone. 81, 85, 86, 95, 99, 100, 114, 124, 159

AEH

`AsyncEventHandler`, classe *RTSJ* modélisant un traitement pouvant être associé à un *AE*. 81, 84–87, 95, 99, 100, 102, 114, 124

API

Application Programming Interface, ensemble de librairies facilitant l'écriture d'un programme informatique. 75, 76, 79, 80

BS

Ordonnancement en tâche de fond, *Background Scheduling*. 49–51, 98–100, 102, 124, 126–128, 130–132, 134, 136, 140, 145, 159

BS-X

Politique de gestion de file *X* associée à une duplication en *BS*. 99

CPU

Processeur, *Compute Process Unit*, unité centrale de traitement. 41, 87, 96, 97, 103, 117–120, 136, 165

DASS

Dynamic Approximate Slack Stealing, Algorithme utilisant une approximation dynamique des temps creux pour ordonnancer des tâches apériodiques. clxxv, 68, 103–105, 108, 112–114, 136, 138, 140, 142, 144, 145, 147, 150

DM

Assignation de priorité en fonction de l'échéance, *Deadline Monotonic*. 40, 44, 45, 123

DS

Serveur ajournable, *Deferrable Server*. 24, 51, 54–56, 93, 99, 101, 102, 132–134, 136, 145, 150

EDF

Politique d'ordonnancement dynamique en fonction de l'échéance, *Earliest Deadline First*. 41, 42, 44, 47, 82

EPES
Serveur à échange de priorité étendu, *Extended Priority Exchange Server*. 51

ESS
Exact Slack Stealing, Algorithme utilisant un calcul exact des temps creux pour ordonnancer des tâches apériodiques. 136, 138, 140, 142, 145, 147

FIFO
First In First Out, « premier arrivé premier servi », politique de gestion d'une file d'attente qui consiste a servir en premier l'élément qui attend depuis le plus de temps. 51, 97, 98, 124, 126–128, 132, 136, 145

GC
Ramasse-miettes, *Garbage Collector*, mécanisme de recyclage automatique de la mémoire. 71, 73–76, 82

GPL
GNU Public Licence, licence logiciel garantissant la mise a disposition du code source. clxxiii, 24, 71, 123, 163

ICPP
Protocole à priorité plafond immédiate, *Immediate Ceiling Priority Protocol*, modification de *PCP* où une tâche voit sa priorité augmenter dès qu'elle obtient un verrou. 48, 161, 162

IDE
Integrated Development Environement, environnement de développement intégré, programme regroupant un éditeur de texte, un compilateur, des outils automatiques de fabrication, et souvent un débogueur. 71

IEEE
Institute of Electrical and Electronics Engineers. 162

J2ME
Java 2 Micro Edition, édition de Java minimale destinée aux environnements avec des ressources restreintes. 79, 80

JIT
La compilation à la volée, *Just In Time* , consiste à compiler juste avant de l'exécuter un morceau de code qui semble très fréquemment utilisé lors de l'exécution d'un programme. 77

JSR
Requète pour une spécification Java, *Java Specification Request*, système normalisé ayant pour but de faire évoluer la plateforme Java. 79, 80, 117, 163

JVM
Machine virtuelle Java, *Java Virtual Machine*. 71, 74–78, 163, 165

LCF

Lower Cost First, « faible coût en premier », politique de gestion d'une file d'attente qui consiste a servir en premier l'élément avec le plus faible pire temps d'exécution. 98, 126–129, 131, 132, 134, 136, 138, 140, 142, 145

LIFO

Last In First Out, dernier arrivé premier servi, politique de gestion d'une file d'attente qui consiste a servir en premier l'élément qui attend depuis le moins longtemps. 51, 97, 98, 128

LLF

Politique d'ordonnancement dynamique en fonction de la laxité, *Least Laxity First*. 41, 42

MAE

`ManageableAsyncEvent`, classe Java que nous proposons pour implanter un événement dont certains des traitements peuvent être effectués par un gestionnaire d'événements. 99, 100

MAEH

`ManageableAsyncEventHandler`, classe Java que nous proposons pour implanter un traitement pouvant être effectué par un gestionnaire d'événements. 99, 100

MASS

Minimal Approximate Slack Stealing, Algorithme utilisant une approximation dynamique des temps creux pour ordonnancer des tâches apériodiques. clxxv, 105, 112–114, 120, 136, 138, 142, 144, 145, 147, 150, 151

MBS

Ordonnancement en tâche de fond non preemptif, *Modified Background Scheduling*. 127–129

MDS

Serveur ajournable modifié, *Modified Deferrable Server*. 134, 136, 150

MPS

Serveur à scrutation modifié, *Modified Polling Server*. 132, 136, 145, 150

NIST

National Institute for Standards and Technology (NIST). 79

OCPP

Protocole à priorité plafond d'origine, *Original Ceiling Priority Protocol*, acronyme désignant la version originale de *PCP*, par opposition à *ICPP*. 48

ODASS

Original Dynamic Approximate Slack Stealing, Algorithme utilisant une approximation dynamique des temps creux pour ordonnancer des tâches apériodiques de façon préemptive. 136

OESS

Original Exact Slack Stealing, Algorithme utilisant un calcul exact des temps creux pour ordonnancer des tâches apériodiques de façon préemptive. 136, 138

OMASS

Original Minimal Approximate Slack Stealing, Algorithme utilisant une approximation dynamique des temps creux pour ordonnancer des tâches apériodiques de façon préemptive. 136

OS

Système d'exploitation, *Operating System*, logiciel chargé de faire le lien entre les applications utilisateur et l'architecture matérielle. 24, 75–77, 87, 104, 119, 163, 165

PCE

Émulation de priorité plafond, *Priority Ceiling Emulation*, nom donné à *ICPP* dans *RTSJ*. 48, 104

PCP

Protocole à priorité plafond, *Priority Ceiling Protocol*, protocole pour résoudre les inversions de priorités non bornées. 48, 65, 160, 161

PES

Serveur à échange de priorité, *Priority Exchange Server*. 51

PIP

Protocole d'héritage de priorité, *Priority Inheritance Protocol*, protocole pour résoudre les inversions de priorités non bornées. 48

POSIX

famille de standards pour les plateformes *UNIX** définie depuis 1988 par l'*IEEE** (Portable Operating System Interface for uniX). 48, 162

PPP

Protocole de protection de priorité, *Priority Protect Protocol*, nom donné à *ICPP* dans *POSIX*. 48

PS

Serveur à scrutation, *Polling Server*. 24, 51, 54–56, 93, 98–102, 129, 130, 132–134, 136, 145, 150

RI

Implantation de référence de *RTSJ*, *Reference Implementation*. 80

RM

Assignation de priorité en fonction de la période, *Rate Monotonic*. 39, 40, 44

RR

Tourniquet, *Round Robin*, algorithme d'ordonnancement à temps partagé très courant. 37, 75

RT-GC
Ramasse-miettes temps réel, *Real-Time Garbage Collector*, mécanisme de recyclage automatique de la mémoire temps réel. 74

RTJVM
JVM temps réel, *Real-Time Java Virtual Machine*. 77, 78, 80–83, 85, 87, 93, 95, 97, 99, 102, 119, 150, 165

RTOS
OS temps réel, *Real-Time Operating System*. 77, 78

RTSJ
Spécification pour Java issue des *JSR*–01 et *JSR*–282, *Real-Time Specification for Java*. 23–25, 42, 48, 50, 80–88, 93, 95–99, 103, 117, 123, 124, 128, 129, 144, 149, 150, 159, 162, 163, 165

RTSS
Real-Time System Simulator, Simulateur de événementiel de systèmes temps réel écrit en Java sous licence *GPL*. clxxiii

SASS
Static Approximate Slack Stealing, Algorithme utilisant une approximation statique des temps creux pour ordonnancer des tâches apériodiques. 66

SS
Serveur sporadique, *Sporadic Server*. 51

UNIX
Système d'exploitation multitâche et multi-utilisateur créé en 1969. 162

ut
unité de temps. 39, 40, 63, 98, 101, 112, 113, 123, 124, 126

WOCRAM
Write Once Carefully, Run Anywhere Maybe (écrivez une seule fois en faisant attention, exécutez peut être partout) est le slogan détourné de *WORA* utilisé pour résumer les préceptes de *RTSJ*. 80

WORA
Write Once, Run Anywhere (écrivez une seule fois, exécutez partout) est le slogan utilisé pour résumer la philosophie de portabilité qui a fait naitre le langage Java. 71, 80, 163

Vocabulaire

Analyse de faisabilité
: Étude statique du système de tâches temps réel afin de déterminer si les contraintes de temps et de mémoire peuvent être respectées. 43

Bytecode
: Code assembleur simplifié produit par compilation d'un code source Java et destiné à être interprété par une *JVM*. 71, 77, 79, 80, 111

Hyperpériode
: Plus petit commun multiple des périodes. 59, 61

Jamaïca VM
: Implantation d'une *RTJVM* compatible *RTSJ* par *Aïcas*. 80, 144

PERC
: Projet de machine virtuelle Java de la société *AONIX*, divisé en trois produits selon les applications visées : PERC Ultra, PERC Pico et PERC Raven. Les deux derniers sont des *profils*, c'est-à-dire des sous-ensemble, de *RTSJ*. 80

Periode occupée de niveau i
: une période occupée de niveau i, *i-level busy period*, est une période pendant laquelle le processeur n'exécute que des tâches de priorités supérieures ou égales à P_i. 46, 47

Sporadique
: Une tâche sporadique est une tâche apériodique dont on peut borner le temps minimal d'inter arrivée, ce qui permet de la considérer comme une tâche périodique dans l'étude du pire scénario. 22

Temps CPU
: Surveillance du temps *CPU*, fonctionnalité de certains *OSs* permettant d'obtenir le temps déjà passé à exécuter une tâche donnée. 87, 93, 95

Temps de réponse
: Temps séparant l'activation d'une tâche (à distinguer du début de son démarrage) de la fin de son exécution. 22, 32, 43, 45–49, 54, 55, 59, 60, 123, 124, 126–128, 131, 132, 134, 140, 145, 149, 150

Thread

Processus léger. Alors qu'un processus dispose de sa propre mémoire virtuelle, un processus léger partage la mémoire virtuelle de son processus père avec d'autres processus légers.. clxx, clxxi, 75–77, 81, 83–86, 88, 89, 97, 102, 127

Annexes

Annexe A

Exemples

A.1 Utilisation des interruptions asynchrones de RTSJ

A.1.1 Exemple en utilisant `Timed`

C'est la manière la plus directe pour utiliser le mécanisme de transfert de contrôle.

```
import javax.realtime.*;

public class ATC_exemple1{

 // nombre d instructions necessaires
 // pour une boucle d'une unite de temps
 public static final int NBIT = 50000000;

 public static void main(String[] args){
  new RealtimeThread(){
   public void run(){
    Interruptible t1 = new Interruptible(){
     public void run(AsynchronouslyInterruptedException e)
      throws AsynchronouslyInterruptedException{
      for(int i  = 0 ; i< 10 ; i++){
       // Boucle d une unite de temps
       for(int j = 0 ; j < NBIT ; j++);
       System.out.println(
         "T1 : "+(i+1)+" unites de temps consommees");
      }
     }
     public void interruptAction(
       AsynchronouslyInterruptedException e){
      System.out.println("T1 : J'ai ete interrompu !");
       }
    };
    new Timed(new RelativeTime(5000,0)).doInterruptible(t1);
   }
  }.start();
 }
}
```

Donne le résultat :

```
>$ /opt/timesys/rtsj/bin/tjvm
 > -Xbootclasspath:/opt/timesys/rtsj/lib/foundation.jar
```

```
 > -Djava.class.path=. ATC_exemple1
T1 : 1 unités de temps consommées
T1 : 2 unités de temps consommées
T1 : 3 unités de temps consommées
T1 : 4 unités de temps consommées
T1 : 5 unités de temps consommées
T1 : J'ai été interrompu !
```

A.1.2 Exemple en utilisant `interrupt()`

Ici, on utilise la méthode `interrupt()` d'un *thread* temps réel depuis un autre *thread*. Si le *thread* que l'on veut interrompre exécute du code explicitement déclaré comme pouvant être interrompu, une exception d'interruption générique lui est propagée.

```
import javax.realtime.*;

public class ATC_exemple2{

 // nombre d instructions necessaires pour
 // une boucle d une unite de temps
 public static final int NBIT = 50000000;

 public static void main(String[] args){

  new RealtimeThread(new PriorityParameters(18)){

   public void run(){

    // T1 va essayer de s executer pendant 10 unites de temps
    final RealtimeThread t1 = new RealtimeThread(new PriorityParameters(16)){

     public void interruptibleMethod() throws AsynchronouslyInterruptedException{
      for(int i  = 0 ; i< 10 ; i++){
       for(int j = 0 ; j < NBIT ; j++);
       System.out.println(
         "T1 : "+(i+1)+" unites de temps consommees");
      }
     }

     public void run(){

      try{
       interruptibleMethod();
      }catch(AsynchronouslyInterruptedException e){
       System.out.println("T1 : J'ai ete interrompu !");
      }

     }
    };

    // T2 est plus prioritaire, mais ne demarre qu apres cinq unites de temps
    RealtimeThread t2 = new RealtimeThread(
      new PriorityParameters(17),new PeriodicParameters(
       new RelativeTime(5000,0),new RelativeTime(10000,0))){
     public void run(){
      System.out.println("T2 : Je vais interrompre T1");
      t1.interrupt();
```

```
     }
    };

    t1.start();
    t2.start();
   }

  }.start();
 }
}
```

Donne le résultat :

```
>$ /opt/timesys/rtsj/bin/tjvm
 > -Xbootclasspath:/opt/timesys/rtsj/lib/foundation.jar
 > -Djava.class.path=. ATC_exemple2
T1 : 1 unités de temps consommées
T1 : 2 unités de temps consommées
T1 : 3 unités de temps consommées
T1 : 4 unités de temps consommées
T1 : 5 unités de temps consommées
T2 : Je vais interrompre T1
T1 : J'ai été interrompu !
```

A.1.3 Exemple en utilisant `doInterruptible() / fire()`

Maintenant, on utilise une instance particulière d'exception d'interruption. Tous les *threads* qui exécutent sa méthode `doInterruptible()` se verront propager cette instance de l'exception lors de l'appel à `fire()`.

```
import javax.realtime.*;

public class ATC_exemple3{

 // nombre d instructions necessaires pour
 // une boucle d une unite de temps
 public static final int NBIT = 50000000;

 public static void main(String[] args){

  final AsynchronouslyInterruptedException theAIE =
   new AsynchronouslyInterruptedException();

  new RealtimeThread(new PriorityParameters(18)){

   public void run(){

    RealtimeThread t1 = new RealtimeThread(new PriorityParameters(16)){

     public void run(){

      theAIE.doInterruptible(new Interruptible(){
       public void interruptAction(AsynchronouslyInterruptedException exception){
        System.out.println("T1 : J ai ete interrompu !");
       }
```

```
        // Attention a ne pas oublier la clause << throws >> !
        public void run(AsynchronouslyInterruptedException exception)
          throws AsynchronouslyInterruptedException{
         for(int i  = 0 ; i< 10 ; i++){
          for(int j = 0 ; j < NBIT ; j++);
          System.out.println(
           "T1 : "+(i+1)+" unites de temps consommees");
         }
        }
       });


     }
    };

    // T2 est plus prioritaire, mais ne demarre qu apres cinq unites de temps
    RealtimeThread t2 = new RealtimeThread(
      new PriorityParameters(17),new PeriodicParameters(
       new RelativeTime(5000,0),new RelativeTime(10000,0))){
     public void run(){
      System.out.println("T2 : Je vais interrompre T1");
      theAIE.fire();
     }
    };

    t1.start();
    t2.start();
   }

  }.start();
 }
}
```

Donne le résultat :

```
>$ /opt/timesys/rtsj/bin/tjvm
 > -Xbootclasspath:/opt/timesys/rtsj/lib/foundation.jar
 > -Djava.class.path=. ATC_exemple3
T1 : 1 unités de temps consommées
T1 : 2 unités de temps consommées
T1 : 3 unités de temps consommées
T1 : 4 unités de temps consommées
T1 : 5 unités de temps consommées
T2 : Je vais interrompre T1
T1 : J'ai été interrompu !
```

Annexe B

RTSS : Un simulateur événementiel de systèmes temps réel

Afin de pouvoir comparer nos algorithmes de prise en charge d'événements apériodiques modifiés pour être implantés en Java avec les algorithmes de la littérature, nous avons écrit un simulateur de système temps réel. Ce programme, baptisé *Real-Time System Simulator (RTSS)**, est distribué sous la licence *GPL*. Il est écrit en Java et repose sur le principe de simulation événementielle. Il peut être récupéré à l'adresse `http://dajam.fr/RTSS`.

B.1 Simulation événementielle

Un simulateur est un programme qui représente une entité extérieure sous la forme d'un modèle. Pour un simulateur événementiel, le modèle ne peut être modifié que par l'arrivée d'événements qui le font évoluer.

Ici, un événement est associé à une date. Une file d'attente d'événements, triée par date, est gérée par le simulateur. Le simulateur traite les événements dans l'ordre de la file, le temps courant étant chaque fois la date associée au dernier événement traité.

Un événement peut agir de trois façons sur la simulation (en plus de faire avancer la date courante) :

- il peut réveiller l'ordonnanceur ;
- il peut modifier la trace de la simulation ;
- il peut générer d'autres événements.

Par exemple, pour modéliser l'exécution d'une tâche périodique, il suffira de créer un seul événement : la première activation de la tâche. Lorsque le simulateur traite cet événement :

1. l'ordonnanceur est prévenu et peut ainsi ajouter la tâche aux tâches actuellement actives ; ceci peut conduire à la création par l'ordonnanceur d'autres événements (début de la tâche, préemption d'une autre etc.), en fonction de la politique d'or-

donnancement ;

2. l'activation à l'instant courant de la tâche est enregistrée dans la trace de simulation ;
3. un autre événement d'activation au temps courant incrémenté de la période de la tâche est ajouté à la file d'événements du simulateur.

La simulation s'arrête lorsque la file d'événements est vide ou lorsque la date limite de fin de simulation est atteinte (si une telle date a été fixée).

B.2 Architecture du programme

Le projet est séparé en quatre paquetages :

- `gui` regroupe toutes les classes nécessaires à l'interface graphique – celle-ci n'est pas à jour et ne permet pas encore de simuler les algorithmes d'ordonnancement mixte de tâches périodiques temps réel dur et apériodiques temps réel souple ;
- `realtime` contient l'implantation des algorithmes d'ordonnancement, les classes qui modélisent les tâches périodiques ou apériodiques, etc. ;
- `simulator` englobe les classes concernant la simulation, comme l'objet qui contrôle la date courante, mais aussi les différents types d'événements ;
- `tools` renferme les classes qui nous ont servies à lancer les simulations dont nous avons présenté les résultats dans ce manuscrit.

Annexe C

Code de notre gestionnaire d'événements pour RTSJ

Nous donnons ici le code des principales classes composants notre paquetage `fr.upe.masson.realtime.eventManager`. Ces classes sont :

- `AbstractUserLandEventManager` est une classe abstraite représentant un gestionnaire d'événements ;
- `ManageableAsyncEvent` est une sous classe de `javax.realtime.AsyncEvent` qui représente un événement auquel peuvent être attachés des traitements compatibles avec un gestionnaire d'événements ;
- `ManageableAsyncEventHandler` est la classe qui représente ces traitements ;
- `AbstractUserLandTaskServer` est une sous classe abstraite de `AbstractUserLandEventManager` qui représente un serveur de tâche ;
- `UserLandPollingTaskServer` est une sous classe concrète de la précédente qui modélise un serveur à scrutation ;
- `UserLandDeferrableTaskServer` est une autre sous classe concrète de `AbstractUserLandTaskServer` qui modélise un serveur ajournable ;
- `AbstractUserLandSlackStealer` est une autre sous classe abstraite de `AbstractUserLandEventManager` qui représente un voleur de temps creux ;
- `AbstractSlackCompatibleSchedulable` est la classe abstraite dont doivent hériter toutes les tâches utilisateur auxquelles on veut voler du temps creux ;
- `DASSUserLandSlackStealer` est une sous classe concrète de `AbstractUserLandSlackStealer` utilisant l'algorithme *DASS* modifié et fournissant également une sous classe interne concrète de `AbstractSlackCompatibleSchedulable` à utiliser avec ce voleur de temps creux ;
- `MASSUserLandSlackStealer` est une sous classe concrète de `AbstractUserLandSlackStealer` utilisant l'algorithme *MASS* et fournissant également une sous classe interne concrète de `AbstractSlackCompatibleSchedulable` à utiliser avec ce voleur de temps creux.

C.1 AbstractSlackCompatibleSchedulable

```
package fr.upe.masson.realtime.eventManager;

import javax.realtime.PeriodicParameters;
import javax.realtime.PriorityParameters;
import javax.realtime.RealtimeThread;

/**
 * All Schedulable in a application using a subclass of
 * AbstractUserLandSlackStealer should inherits from this class
 *
 * @author dm
 *
 */
public abstract class AbstractSlackCompatibleSchedulable extends RealtimeThread {

  /**
   * Hard tasks are periodic tasks
   *
   * @param priorityParameters
   * @param periodicParameters

   * @param logic
   *            the logic to be performed periodically
   */
  protected AbstractSlackCompatibleSchedulable(
      PriorityParameters priorityParameters,
      PeriodicParameters periodicParameters,
       Runnable logic) {
    super(priorityParameters, periodicParameters);
    this.logic = logic;
    this.periodicParameters = (PeriodicParameters) getReleaseParameters();

  }

  protected final PeriodicParameters periodicParameters;

  private final Runnable logic;

  /**
   * executes the run() method of the logic in a loop ending by a call to
   * waitForNextPeriod() methods computeBeforePeriodic and
   * computeAfterPeriodic are called respectively after and before the
   * waitForNextPeriod invocation. Loop can terminates either if the flagEnd
   * of the slackStealer is positioned to true.
   */
  public final void run() {
    for(int i =0 ; i < 5 ; i++){
      waitForNextPeriod();
      computeBeforePeriodic();
      logic.run();
      computeAfterPeriodic();
    }
  }

  abstract void computeBeforePeriodic();

  abstract void computeAfterPeriodic();

}
```

C.2 AbstractUserLandEventManager

```
package fr.upe.masson.realtime.eventManager;

import java.util.Iterator;
import java.util.LinkedList;
import java.util.List;

import javax.realtime.AbsoluteTime;
import javax.realtime.PriorityParameters;
import javax.realtime.PriorityScheduler;
import javax.realtime.RelativeTime;
import javax.realtime.Schedulable;
import javax.realtime.Scheduler;
import javax.realtime.Timed;

import fr.upe.masson.realtime.eventManager.test.Tools;

/**
 * An EventManager is a Schedulable Object that schedules in user land event,
 * represented by the interface Manageable.
 *
 * @author dm
 *
 */
public abstract class AbstractUserLandEventManager implements Schedulable {

  /**
   * Event list to be managed. Event are Manageable objects
   */
  private final List events = new LinkedList();

  protected final PriorityParameters priorityParameters;

  protected AbstractUserLandEventManager() {

    timedException = new Timed(new RelativeTime());
    remainingCapacity = new RelativeTime();
    time = new AbsoluteTime();
    elapsed = new RelativeTime();
    lastUpdate = new AbsoluteTime();
    priorityParameters = new PriorityParameters(
        ((PriorityScheduler) Scheduler.getDefaultScheduler())
            .getMaxPriority());
  }

  /**
   * allow the manager to properly terminates
   */
  public void setFlagEnd() {
    flagEnd = true;
  }

  protected boolean flagEnd = false;

  /**
   * Queue Policy available for en event manager
   *
   * @author dm
   *
   */
```

```
public static class QueuePolicy {
  static final int FIFO_POLICY = 0;
  static final int LIFO_POLICY = 1;
  static final int LCF_POLICY = 2;

  static boolean contains(int queuePolicy) {
    return (queuePolicy == FIFO_POLICY || queuePolicy == FIFO_POLICY
        || queuePolicy == FIFO_POLICY);
  }
}

/**
 * method called when final start method in EventManager is called on a
 * reference of this
 */
protected abstract void performStart();

private static AbstractUserLandEventManager started = null;
private int flag = 0;

/**
 * Start this server. This method can only be called once. As soon as there
 * is one task server started, it is impossible to start another one.
 */
public final void start() {
  if (flag == 0) {
    flag = 1;
    started = this;
    performStart();
  }
}

/**
 *
 * @return return a reference on the started event manager if there is any,
 *         null otherwise
 */
public static AbstractUserLandEventManager getStarted() {
  return started;
}

/**
 *
 * @param manageableEventHandler
 *            the handler to be removed. Semantic is similar to
 *            java.util.List one.
 */
public final void remove(ManageableAsyncEventHandler manageableEventHandler) {
  synchronized (lock) {
    events.remove(manageableEventHandler);
  }
}

private final Object lock = new Object();

/**
 * Enqueue an handler in the event list. Position depends of the current
 * queue policy
 *
 * @param handler
 *            the handler to be enqueued.
```

```
 */
public void enqueue(ManageableAsyncEventHandler handler) {
  switch (queuePolicy) {
  case QueuePolicy.LCF_POLICY:
    addIncreasing(handler);
    break;
  case QueuePolicy.LIFO_POLICY:
    synchronized (lock) {
      events.add(0, handler);
    }
    break;
  default:// FIFO
    synchronized (lock) {
      events.add(handler);
    }
  }
}

private void addIncreasing(ManageableAsyncEventHandler meh) {
  for (int i = 0; i < events.size(); i++) {
    ManageableAsyncEventHandler current = (ManageableAsyncEventHandler) events
        .get(i);
    if (current.getCost().compareTo(meh.getCost()) < 0) {
      synchronized (lock) {
        events.add(i, meh);
      }
      return;
    }
  }
  synchronized (lock) {
    events.add(meh);
  }
}

/**
 * should be one of EventManage.QueuePolicy.FIFO_POLICY, LIFO_POLICY and
 * LCF_Policy
 */
private int queuePolicy = QueuePolicy.FIFO_POLICY;

/**
 * @return an integer which is one of EventManage.QueuePolicy.FIFO_POLICY,
 *         LIFO_POLICY and LCF_Policy
 */
public int getQueuePolicy() {
  return (this.queuePolicy);
}

/**
 * @return an integer which is one of EventManage.QueuePolicy.FIFO_POLICY,
 *         LIFO_POLICY and LCF_Policy
 */
public void setQueuePolicy(int queuePolicy) {
  if (QueuePolicy.contains(queuePolicy))
    this.queuePolicy = queuePolicy;
  else
    throw new IllegalArgumentException();
}

private final AbsoluteTime lastUpdate;
private final AbsoluteTime time;
```

```
  private final RelativeTime remainingCapacity;
  private final RelativeTime elapsed;

  private void updateRemainingCapacity() {

    Tools.rtClock.getTime(time);
    time.subtract(lastUpdate, elapsed);
    remainingCapacity.subtract(elapsed, remainingCapacity);
    lastUpdate.set(time);

  }

  /**
   * Schedule tasks in the queue in the limit of capacity, skipping tasks with
   * a higher cost than the remaining budget. Do not modify budget.
   *
   * @param budget
   *            this method never modifies budget
   * @return the unused part of budget
   */
  protected RelativeTime manage(final RelativeTime budget) {

    remainingCapacity.set(budget);
    Tools.rtClock.getTime(lastUpdate);

    for (Iterator it = events.iterator(); it.hasNext()
        && remainingCapacity.getMilliseconds() > 0;) {

      final ManageableAsyncEventHandler handler = (ManageableAsyncEventHandler) it
          .next();
      if (handler.getCost().compareTo(remainingCapacity) <= 0) {
        timedException.resetTime(remainingCapacity);
        final FlaggedCloneableInterruptible logic = handler
            .getRegularLogic();
        timedException.doInterruptible(logic);
        if (logic.isInterrupted()) {
          remainingCapacity.set(0, 0);
          break;
        }
        remove(handler);
        handler.asynchronouslyInterruptBSDuplicate();
        updateRemainingCapacity();
        return manage(remainingCapacity);

      }
    }
    return remainingCapacity;

  }

  private final Timed timedException;

}
```

C.3 AbstractUserLandSlackStealer

```
package fr.upe.masson.realtime.eventManager;

import java.util.Comparator;
import java.util.SortedSet;
```

```
import java.util.TreeSet;

import javax.realtime.AbsoluteTime;
import javax.realtime.AsyncEvent;
import javax.realtime.AsyncEventHandler;
import javax.realtime.Clock;
import javax.realtime.MemoryParameters;
import javax.realtime.ProcessingGroupParameters;
import javax.realtime.RelativeTime;
import javax.realtime.ReleaseParameters;
import javax.realtime.Scheduler;
import javax.realtime.SchedulingParameters;

import fr.upe.masson.realtime.eventManager.test.Tools;

public abstract class AbstractUserLandSlackStealer extends
    AbstractUserLandEventManager {

  /**
   *
   * @return the exact value of available slack time or a lower bound
   */

  private final RelativeTime elapsedTime = new RelativeTime();
  private final AbsoluteTime aTime = new AbsoluteTime();

  private final RelativeTime returnValue = new RelativeTime();

  protected RelativeTime getSlack() {

    Clock.getRealtimeClock().getTime(aTime).subtract(lastSlackComputation,
        elapsedTime);

    if (elapsedTime.compareTo(slack) <= 0)
      slack.subtract(elapsedTime, returnValue);
    else
      returnValue.set(0, 0);

    Tools.LOG.append(returnValue).append("\n");

    return returnValue;
  }

  protected final AbsoluteTime lastSlackComputation = new AbsoluteTime();
  protected final RelativeTime slack = new RelativeTime();
  protected final AsyncEvent wakeBeforePeriodic = new AsyncEvent();
  protected AsyncEventHandler beforePeriodicHandler;

  protected final AsyncEvent wakeAfterPeriodic = new AsyncEvent();
  protected AsyncEventHandler afterPeriodicHandler;

  protected final AbsoluteTime start;

  protected AbstractUserLandSlackStealer(AbsoluteTime start) {
    this.start = start;
    realSchedulable = new AsyncEventHandler() {
      private final RelativeTime budget = new RelativeTime();

      public void handleAsyncEvent() {
        if (flagEnd)
          return;
```

```
        budget.set(getSlack());
        manage(budget);
      }
    };
    realSchedulable.setSchedulingParameters(priorityParameters);
    wakeUp = new AsyncEvent();
    wakeUp.addHandler(realSchedulable);
  }

  private final AsyncEvent wakeUp;
  private final AsyncEventHandler realSchedulable;

  protected void wakeUp() {
    wakeUp.fire();
  }

  public void enqueue(ManageableAsyncEventHandler meh) {
    super.enqueue(meh);
    wakeUp.fire();
  }

  /**
   * Ordered set of PeriodicTask. HeadSet(t) should return the tasks with an
   * higher priority than t
   */
  protected SortedSet hardTasks = new TreeSet(new Comparator() {

    /**
     * Compare object assuming they are Schedulable with priorityParameters
     * according to their priority
     *
     * @param o1
     * @param o2
     * @return
     */
    public int compare(Object o1, Object o2) {
      AbstractSlackCompatibleSchedulable s1 = (AbstractSlackCompatibleSchedulable) o1;
      AbstractSlackCompatibleSchedulable s2 = (AbstractSlackCompatibleSchedulable) o2;
      return s2.getPriority() - s1.getPriority();
    }

  });

  void register(AbstractSlackCompatibleSchedulable hardTask) {
    hardTasks.add(hardTask);

  }

  protected abstract void initDataStructure();
  protected abstract void computeSlack();

  protected void performStart() {
    initDataStructure();
  }

  /************** delegate to realSchedulable */

  public void run() {
    realSchedulable.run();
  }
```

```
  /** throw new UnsupportedOperationException(); **/
  ...

}
```

C.4 AbstractUserLandTaskServer

```
package fr.upe.masson.realtime.eventManager;

import javax.realtime.PeriodicParameters;

public abstract class AbstractUserLandTaskServer extends
    AbstractUserLandEventManager {

  protected AbstractUserLandTaskServer(PeriodicParameters periodicParameters) {
    this.periodicParameters = (PeriodicParameters) periodicParameters
        .clone();
  }

  protected final PeriodicParameters periodicParameters;


}
```

C.5 DASSUserLandSlackStealer

```
package fr.upe.masson.realtime.eventManager;

import java.util.Iterator;
import java.util.Stack;

import javax.realtime.AbsoluteTime;
import javax.realtime.AsyncEventHandler;
import javax.realtime.Clock;
import javax.realtime.PeriodicParameters;
import javax.realtime.PriorityParameters;
import javax.realtime.RelativeTime;

public class DASSUserLandSlackStealer extends AbstractUserLandSlackStealer {

  public static class DASSCompatibleSchedulable extends
      AbstractSlackCompatibleSchedulable {

    private long si = 0;

    /**
     *
     * @param priorityParameters
     * @param periodicParameters
     *            start time is assumed to be 0
     * @param slackStealer
     * @param logic
     * @param isLast
     *            set true if this is the last Schedulable to register, then
     *            data structure can be initialized
     */
    public DASSCompatibleSchedulable(PriorityParameters priorityParameters,
        PeriodicParameters periodicParameters,
        DASSUserLandSlackStealer slackStealer, Runnable logic,
```

```
    boolean isLast) {
  super(priorityParameters, periodicParameters, logic);
  slackStealer.register(this);
  this.slackStealer = slackStealer;
}

private final DASSUserLandSlackStealer slackStealer;

void computeAfterPeriodic() {
  synchronized (slackStealer.lock2) {
    slackStealer.caller = this;
    slackStealer.wakeAfterPeriodic.fire();
  }
}

void computeBeforePeriodic() {
  synchronized (slackStealer.lock2) {
    slackStealer.caller = this;
    slackStealer.wakeBeforePeriodic.fire();
  }
}

public void computeSi(long date) {
  date -= periodicParameters.getStart().getMilliseconds();
  long nd = nextDeadline(date);
  si = nd - date - interference(date, nd);
}

private long previousPeriod(long date, boolean inclusive) {
  long periodMillis = periodicParameters.getPeriod()
      .getMilliseconds();

  long previousPeriod = (date / periodMillis) * periodMillis;
  if (!inclusive && previousPeriod == date)
    previousPeriod -= periodMillis;
  return previousPeriod;
}

private long nextPeriod(long date, boolean inclusive) {
  long previousPeriod = previousPeriod(date, !inclusive);
  return previousPeriod
      + periodicParameters.getPeriod().getMilliseconds();
}

private long nextDeadline(long date) {
  return nextPeriod(date, true)
      + periodicParameters.getDeadline().getMilliseconds();
}

private long interference(long a, long b) {
  long interference = 0;
  for (Iterator it = slackStealer.hardTasks.headSet(this).iterator(); it
      .hasNext();) {
    DASSCompatibleSchedulable s = (DASSCompatibleSchedulable) it
        .next();
    interference += s.inter(a, b);
  }
  interference += inter(a, b);
  return interference;
}
```

```
  private long inter(long a, long b) {
    long truc = b - nextPeriod(a, true);
    long periodMillis = periodicParameters.getPeriod()
        .getMilliseconds();
    long costMillis = periodicParameters.getCost().getMilliseconds();
    long f = Math.max(0, truc / periodMillis);
    return f
        * costMillis
        + Math
            .min(costMillis, Math.max(0, truc - f
                * periodMillis));
  }

}

// LOG FIELDS
public final static StringBuffer LOG = new StringBuffer();
private final static AbsoluteTime ABS_TIME_1 = new AbsoluteTime();
private final static AbsoluteTime ABS_TIME_2 = new AbsoluteTime();
private final static RelativeTime REL_TIME = new RelativeTime();
public static long timeInAft = 0;
public static long timeInBef = 0;

public DASSUserLandSlackStealer(AbsoluteTime start) {
  super(start);
  beforePeriodicHandler = new AsyncEventHandler() {
    public void handleAsyncEvent() {
      synchronized (lock) {

        Clock.getRealtimeClock().getTime(ABS_TIME_1); // LOG ACTION

        Clock.getRealtimeClock().getTime(time);
        time.subtract(lastEventDate, elapsedTime);
        lastEventDate.set(time);

        Object executing = null;

        if (!executionStack.isEmpty())
          executing = executionStack.peek();

        for (Iterator it = hardTasks.iterator(); it.hasNext();) {
          DASSCompatibleSchedulable s = (DASSCompatibleSchedulable) it
              .next();
          if (s.equals(executing))// OK if executing == null
            break;
          s.si -= elapsedTime.getMilliseconds();
        }
      }
      executionStack.push(caller);

      Clock.getRealtimeClock().getTime(ABS_TIME_2);// LOG ACTION
      LOG.append("bef ").append(
          ABS_TIME_2.subtract(ABS_TIME_1, REL_TIME)).append("\n");// LOG
                                                                 // ACTION
      timeInBef += REL_TIME.getNanoseconds();

    }
  };
  beforePeriodicHandler.setSchedulingParameters(priorityParameters);
  wakeBeforePeriodic.addHandler(beforePeriodicHandler);
```

```
    afterPeriodicHandler = new AsyncEventHandler() {
      public void handleAsyncEvent() {

        synchronized (lock) {

          Clock.getRealtimeClock().getTime(ABS_TIME_1);// LOG ACTION

          Object top = executionStack.pop();
          if (!top.equals(caller))
            throw new AssertionError(
                "The caller must be on top of the execution top");

          Clock.getRealtimeClock().getTime(time);
          time.subtract(lastEventDate, elapsedTime);
          lastEventDate.set(time);

          // strictly less than caller
          for (Iterator it = hardTasks.headSet(caller).iterator(); it
              .hasNext();) {
            DASSCompatibleSchedulable s = (DASSCompatibleSchedulable) it
                .next();
            s.si -= elapsedTime.getMilliseconds();
          }

          caller.computeSi(time.getMilliseconds());
          computeSlack();

          Clock.getRealtimeClock().getTime(ABS_TIME_2);// LOG ACTION
          LOG.append("aft ").append(
              ABS_TIME_2.subtract(ABS_TIME_1, REL_TIME)).append(
              "\n");// LOG ACTION
          timeInAft += REL_TIME.getNanoseconds();
        }//end sync lock
      }
    };
    afterPeriodicHandler.setSchedulingParameters(priorityParameters);
    wakeAfterPeriodic.addHandler(afterPeriodicHandler);
  }

  private final AbsoluteTime time = new AbsoluteTime();
  private final RelativeTime elapsedTime = new RelativeTime();

  private final AbsoluteTime lastEventDate = new AbsoluteTime();

  private Stack executionStack = new Stack();

  private DASSCompatibleSchedulable caller;
  private final Object lock = new Object();
  private final Object lock2 = new Object();

  protected void computeSlack() {
    Clock.getRealtimeClock().getTime(lastSlackComputation);
    long si = Long.MAX_VALUE;
    for (Iterator it = hardTasks.iterator(); it.hasNext();) {
      DASSCompatibleSchedulable task = (DASSCompatibleSchedulable) it
          .next();
      si = Math.min(si, task.si);
    }

    slack.set(si);
  }
```

```
  protected void initDataStructure() {
    for (Iterator it = hardTasks.iterator(); it.hasNext();) {
      DASSCompatibleSchedulable task = (DASSCompatibleSchedulable) it
          .next();
      task.computeSi(start.getMilliseconds());
    }
    computeSlack();
    lastSlackComputation.set(start);
    lastEventDate.set(start);
  }

}
```

C.6 ManageableAsyncEventHandler

```
package fr.upe.masson.realtime.eventManager;

import javax.realtime.AperiodicParameters;
import javax.realtime.AsyncEvent;
import javax.realtime.AsyncEventHandler;
import javax.realtime.AsynchronouslyInterruptedException;
import javax.realtime.PriorityParameters;
import javax.realtime.PriorityScheduler;
import javax.realtime.RelativeTime;
import javax.realtime.Scheduler;

public class ManageableAsyncEventHandler {

  private final AsynchronouslyInterruptedException interruptBS;
  private final AsyncEventHandler duplicationBSHandler;
  private final AsyncEvent duplicationBS;
  private boolean duplicate;

  /**
   * Invocation of this method will fire an AsynchronouslyInterruptedException
   * to the AsyncEventHandler that is in charge to execute the BS duplication
   */
  public void asynchronouslyInterruptBSDuplicate() {
    if (duplicate){
      interruptBS.fire();
      isInterrupted = true;
    }
  }

  private boolean isInterrupted = false;

  /**
   * this(aperiodicParameters, logic, false);
   */
  public ManageableAsyncEventHandler(AperiodicParameters aperiodicParameters,
      final FlaggedCloneableInterruptible logic) {
    this(aperiodicParameters, logic, false);
  }

  /**
   *
   * @param aperiodicParameters
   * @param logic
   *            the logic to associate to this handler
```

```
   * @param duplicate
   *            true to activate the BS duplication
   */
  public ManageableAsyncEventHandler(AperiodicParameters aperiodicParameters,
      final FlaggedCloneableInterruptible logic, boolean duplicate) {

    this.aperiodicParameters = (AperiodicParameters) aperiodicParameters.clone();
    this.regularLogic = logic;
    this.duplicatedLogic = (FlaggedCloneableInterruptible) logic.clone();
    this.duplicate = duplicate;

    if (duplicate) {
      duplicationBSHandler = new AsyncEventHandler() {
        public void handleAsyncEvent() {
          if (!isInterrupted)
            interruptBS.doInterruptible(duplicatedLogic);
          if(isInterrupted || !duplicatedLogic.isInterrupted())
            try {
              AbstractUserLandEventManager.getStarted().remove(
                  ManageableAsyncEventHandler.this);
            } catch (NullPointerException e) {
              System.err.println("An event manager should be started now");
              System.exit(1);
            }
        }
      };
      duplicationBSHandler
          .setSchedulingParameters(new PriorityParameters(
              ((PriorityScheduler) Scheduler
                  .getDefaultScheduler()).getMinPriority()));
      duplicationBSHandler.setReleaseParameters(aperiodicParameters);
      duplicationBS = new AsyncEvent();
      duplicationBS.addHandler(duplicationBSHandler);
      interruptBS = new AsynchronouslyInterruptedException();

    } else {
      interruptBS = null;
      duplicationBS = null;
      duplicationBSHandler = null;
    }

  }

  private final AperiodicParameters aperiodicParameters;

  /**
   *
   * @return a newly allocated RelativeTime which represents the cost of the
   *         logic encapsulated in this handler
   */
  public RelativeTime getCost() {
    return new RelativeTime(aperiodicParameters.getCost());
  }

  private final FlaggedCloneableInterruptible regularLogic;
  private final FlaggedCloneableInterruptible duplicatedLogic;

  /**
   *
   * @return the Interruptible logic which this handler encapsulates
   */
```

```
  public FlaggedCloneableInterruptible getRegularLogic() {
    return regularLogic;
  }

  public FlaggedCloneableInterruptible getDuplicatedLogic() {
    return duplicatedLogic;
  }

  /**
   * Calling this method will register this handler in the started
   * EventManager queue An {@link IllegalStateException} is thrown is there is
   * non currently started EventManager
   */
  public void handleAsyncEvent() {
    AbstractUserLandEventManager em = AbstractUserLandEventManager
        .getStarted();
    if (em == null)
      throw new IllegalStateException();
    em.enqueue(this);
    if (duplicate)
      duplicationBS.fire();
  }

}
```

C.7 ManageableAsyncEvent

```
package fr.upe.masson.realtime.eventManager;

import java.util.Iterator;
import java.util.LinkedList;
import java.util.List;

import javax.realtime.AsyncEvent;

public class ManageableAsyncEvent extends AsyncEvent {

  private List mehs = new LinkedList();

  private Object lock = new Object();

  public void addHandler(ManageableAsyncEventHandler meh) {
    synchronized (lock) {
      mehs.add(meh);
    }
  }

  public void removeHandler(ManageableAsyncEventHandler meh) {
    synchronized (lock) {
      mehs.remove(meh);
    }
  }

  public void fire() {
    super.fire();
    synchronized (lock) {
      for (Iterator it = mehs.iterator(); it.hasNext();) {
        ManageableAsyncEventHandler h = (ManageableAsyncEventHandler) it
            .next();
        h.handleAsyncEvent();
```

```
      }
    }
  }
}
```

C.8 MASSUserLandSlackStealer

```
package fr.upe.masson.realtime.eventManager;

import java.util.Iterator;
import java.util.Stack;

import javax.realtime.AbsoluteTime;
import javax.realtime.AsyncEventHandler;
import javax.realtime.Clock;
import javax.realtime.PeriodicParameters;
import javax.realtime.PriorityParameters;
import javax.realtime.RelativeTime;

public class MASSUserLandSlackStealer extends AbstractUserLandSlackStealer {

  public static class MASSCompatibleSchedulable extends
      AbstractSlackCompatibleSchedulable {

    /**
     *
     * @param priorityParameters
     * @param periodicParameters
     *            start time is assumed to be the same for each
     *            MASSCompatibleSchedulable
     * @param slackStealer
     * @param logic
     * @param isLast
     *            set true if this is the last Schedulable to register, then
     *            data structure can be initialized
     */
    public MASSCompatibleSchedulable(PriorityParameters priorityParameters,
        PeriodicParameters periodicParameters,
        MASSUserLandSlackStealer slackStealer, Runnable logic,
        boolean isLast) {
      super(priorityParameters, periodicParameters, logic);
      slackStealer.register(this);
      this.slackStealer = slackStealer;
    }

    private final MASSUserLandSlackStealer slackStealer;
    private long ciMillis = periodicParameters.getCost().getMilliseconds();
    private long ciNanos = periodicParameters.getCost().getMilliseconds();

    private long wiMillis = 0;
    private long wiNanos = 0;

    private void setWi(RelativeTime time) {
      wiMillis = time.getMilliseconds();
      wiNanos = time.getNanoseconds();
    }

    long getSi() {
      long nbMillis = wiMillis - ciMillis;
      long nbNanos = wiNanos - ciNanos;
```

```
    return nbMillis + nbNanos / 1000000;
  }

  void computeAfterPeriodic() {
    synchronized (slackStealer.lock2) {
      slackStealer.caller = this;
      slackStealer.wakeAfterPeriodic.fire();
    }
  }

  void computeBeforePeriodic() {
    synchronized (slackStealer.lock2) {
      slackStealer.caller = this;
      slackStealer.wakeBeforePeriodic.fire();
    }
  }

  private long getInterference(AbsoluteTime date) {

    long interference = 0;
    for (Iterator it = slackStealer.hardTasks.iterator(); it.hasNext();) {
      MASSCompatibleSchedulable s = (MASSCompatibleSchedulable) it
          .next();
      if (equals(s))
        break;
      interference += getInterference(s, date);

    }
    return interference;
  }

  /**
   * Warning : Millisecond precision only
   *
   * @param s
   * @param date
   * @return
   */
  private long getInterference(MASSCompatibleSchedulable s,
      AbsoluteTime date) {

    long u = nextDeadline(date.getMilliseconds(), true);
    long v = s.nextPeriod(u, true);

    long delta = v - u;

    long thisPeriod = this.periodicParameters.getPeriod()
        .getMilliseconds();
    long sPeriod = s.periodicParameters.getPeriod().getMilliseconds();

    long r = thisPeriod % sPeriod;
    long nba = thisPeriod / sPeriod;

    if (delta < r)
      nba++;

    return nba * s.periodicParameters.getCost().getMilliseconds();

  }

  private long previousPeriod(long date, boolean inclusive) {
```

```
    long periodMillis = periodicParameters.getPeriod()
        .getMilliseconds();
    long previousPeriod = (date / periodMillis) * periodMillis;
    if (!inclusive && previousPeriod == date)
      previousPeriod -= periodMillis;
    return previousPeriod;
  }

  private long nextPeriod(long date, boolean inclusive) {
    long previousPeriod = previousPeriod(date, !inclusive);
    return previousPeriod
        + periodicParameters.getPeriod().getMilliseconds();
  }

  private long nextDeadline(long date, boolean inclusive) {
    return previousPeriod(date, false)
        + periodicParameters.getDeadline().getMilliseconds();
  }

}

// LOG FIELDS
public final static StringBuffer LOG = new StringBuffer();
private final static AbsoluteTime ABS_TIME_1 = new AbsoluteTime();
private final static AbsoluteTime ABS_TIME_2 = new AbsoluteTime();
private final static RelativeTime REL_TIME = new RelativeTime();
public static long timeInAft = 0;
public static long timeInBef = 0;

public MASSUserLandSlackStealer(AbsoluteTime start) {
  super(start);

  beforePeriodicHandler = new AsyncEventHandler() {
    public void handleAsyncEvent() {

      synchronized (lock) {

        Clock.getRealtimeClock().getTime(ABS_TIME_1); // LOG ACTION
        Clock.getRealtimeClock().getTime(time);
        time.subtract(lastCiUpdate, elapsedTimeCi);

        lastCiUpdate.set(time);

        if (!executionStack.isEmpty()) {
          MASSCompatibleSchedulable top = (MASSCompatibleSchedulable) executionStack
              .peek();
          top.ciMillis -= elapsedTimeCi.getMilliseconds();
          top.ciNanos -= elapsedTimeCi.getNanoseconds();
        }
        executionStack.push(caller);

        Clock.getRealtimeClock().getTime(ABS_TIME_2);// LOG ACTION
        LOG.append("bef ").append(
            ABS_TIME_2.subtract(ABS_TIME_1, REL_TIME)).append(
            "\n");// LOG ACTION
        timeInBef += REL_TIME.getNanoseconds();

      }
    }
  };
  beforePeriodicHandler.setSchedulingParameters(priorityParameters);
```

```
wakeBeforePeriodic.addHandler(beforePeriodicHandler);

afterPeriodicHandler = new AsyncEventHandler() {

  public void handleAsyncEvent() {

    synchronized (lock) {

      Clock.getRealtimeClock().getTime(ABS_TIME_1);// LOG ACTION

      Object top = executionStack.pop();
      if (!top.equals(caller))
        throw new AssertionError(
            "The caller must be on top of the execution top");

      Clock.getRealtimeClock().getTime(time);
      time.subtract(lastCiUpdate, elapsedTimeCi);
      time.subtract(lastWiUpdate, elapsedTimeWi);

      lastWiUpdate.set(time);
      lastCiUpdate.set(time);

      if (elapsedTimeCi.compareTo(elapsedTimeWi) > 0)
        elapsedTimeCi.set(elapsedTimeWi);


      Iterator it = hardTasks.iterator();

      while (it.hasNext()) {
        MASSCompatibleSchedulable s = (MASSCompatibleSchedulable) it
            .next();

        s.wiMillis -= elapsedTimeWi.getMilliseconds();
        s.wiNanos -= elapsedTimeWi.getNanoseconds();

        if (s.equals(caller)) {

          long nextInterference = caller
              .getInterference(lastCiUpdate);



          s.wiMillis = s.wiMillis
              + s.periodicParameters.getPeriod()
                  .getMilliseconds()
              - nextInterference;
          s.ciMillis = s.periodicParameters.getCost()
              .getMilliseconds();
          break;
        }
      }
      while (it.hasNext()) {// task with a lower priority than
        // caller
        MASSCompatibleSchedulable s = (MASSCompatibleSchedulable) it
            .next();
        s.wiMillis += caller.periodicParameters.getCost()
            .getMilliseconds();
      }

      computeSlack();
```

```
          Clock.getRealtimeClock().getTime(ABS_TIME_2);// LOG ACTION
          LOG.append("aft ").append(
              ABS_TIME_2.subtract(ABS_TIME_1, REL_TIME)).append(
              "\n");// LOG ACTION
          timeInAft += REL_TIME.getNanoseconds();

        }// end lock
      }
    };
    afterPeriodicHandler.setSchedulingParameters(priorityParameters);
    wakeAfterPeriodic.addHandler(afterPeriodicHandler);

  }

  private Stack executionStack = new Stack();

  private MASSCompatibleSchedulable caller;
  private final Object lock = new Object();
  private final Object lock2 = new Object();

  /**
   * Really compute a bound on the available slack time
   */
  protected void computeSlack() {

    Clock.getRealtimeClock().getTime(lastSlackComputation);


    long slacktmp = Long.MAX_VALUE;
    for (Iterator it = hardTasks.iterator(); it.hasNext();) {
      MASSCompatibleSchedulable s = (MASSCompatibleSchedulable) it.next();
      slacktmp = Math.min(slacktmp, s.wiMillis);

    }
    slack.set(slacktmp);

  }

  private final AbsoluteTime lastCiUpdate = new AbsoluteTime();
  private final AbsoluteTime lastWiUpdate = new AbsoluteTime();
  private final AbsoluteTime time = new AbsoluteTime();
  private final RelativeTime elapsedTimeCi = new RelativeTime();
  private final RelativeTime elapsedTimeWi = new RelativeTime();

  protected void initDataStructure() {

    for (Iterator it = hardTasks.iterator(); it.hasNext();) {

      MASSCompatibleSchedulable t = (MASSCompatibleSchedulable) it.next();

      t.setWi(t.periodicParameters.getDeadline());

      for (Iterator it2 = hardTasks.iterator(); it2.hasNext();) {
        MASSCompatibleSchedulable t2 = (MASSCompatibleSchedulable) it2
            .next();
        if (t2.equals(t))
          break;
        RelativeTime rightTerm = HighResolutionTimeTools.multiply(
            HighResolutionTimeTools.ceil(t.periodicParameters
                .getDeadline(), t2.periodicParameters
                .getPeriod()), t2.periodicParameters.getCost());
```

```
        t.wiMillis -= rightTerm.getMilliseconds();
        t.wiNanos -= rightTerm.getNanoseconds();

      }

    }
    computeSlack();
    lastCiUpdate.set(start);
    lastWiUpdate.set(start);
  }

}
```

C.9 UserLandDeferrableTaskServer

```
package fr.upe.masson.realtime.eventManager;

import javax.realtime.AbsoluteTime;
import javax.realtime.AsyncEvent;
import javax.realtime.AsyncEventHandler;
import javax.realtime.MemoryParameters;
import javax.realtime.PeriodicParameters;
import javax.realtime.PeriodicTimer;
import javax.realtime.ProcessingGroupParameters;
import javax.realtime.RelativeTime;
import javax.realtime.ReleaseParameters;
import javax.realtime.Scheduler;
import javax.realtime.SchedulingParameters;

import fr.upe.masson.realtime.eventManager.test.Tools;

public class UserLandDeferrableTaskServer extends AbstractUserLandTaskServer {

  private final PeriodicTimer priorityTimer;

  private final Object lock;

  public UserLandDeferrableTaskServer(final PeriodicParameters parameters) {
    super(parameters);
    lock = new Object();
    priorityTimer = new PeriodicTimer(parameters.getStart(), parameters
        .getPeriod(), new AsyncEventHandler(priorityParameters,
        periodicParameters, null, null, null, null) {
      public void handleAsyncEvent() {

        Tools.LOG.append(
            Tools.rtClock.getTime(atime).subtract(Tools.start,
                rtime)).append("␣wtf␣").append(
            Thread.currentThread().getPriority()).append("\n");

        if (flagEnd)
          priorityTimer.stop();

        if (capacityAlreadyRefilled) {
          capacityAlreadyRefilled = false;
          return;
        }
        synchronized (lock) {
          Tools.LOG.append(
              Tools.rtClock.getTime(atime).subtract(Tools.start,
```

```
                rtime)).append(" DS reset capacity\n");
        actualCapacity.set(parameters.getCost());
      }
      wakeUp.fire();// the server should be wake up
    }
  });


  realSchedulable = new AsyncEventHandler(priorityParameters,
      periodicParameters, null, null, null, null) {

    private final RelativeTime budget = new RelativeTime();

    public void handleAsyncEvent() {

      if (flagEnd)
        priorityTimer.stop();

      synchronized (lock) {

        Tools.LOG.append(
            Tools.rtClock.getTime(atime).subtract(Tools.start,
                rtime)).append(" DS begins : capacity = ")
            .append(actualCapacity).append("\n");

        RelativeTime nextPeriod = nextRelativePeriod();

        if (!capacityAlreadyRefilled
            && nextPeriod.compareTo(actualCapacity) <= 0) {

          actualCapacity.add(parameters.getCost(), budget);

          capacityAlreadyRefilled = true;
        } else
          budget.set(actualCapacity);


        actualCapacity.set(manage(budget));

        Tools.LOG.append(
            Tools.rtClock.getTime(atime).subtract(Tools.start,
                rtime)).append(" DS ends : capacity = ")
            .append(actualCapacity).append("\n");
      }

    }

  };

  actualCapacity = new RelativeTime(parameters.getCost());
  wakeUp = new AsyncEvent();
  wakeUp.addHandler(realSchedulable);
}

private boolean capacityAlreadyRefilled = false;

private final AbsoluteTime atime = new AbsoluteTime();
private final RelativeTime rtime = new RelativeTime();

/**
 * called in a synchronized context
```

```
   *
   * @return
   */
  private RelativeTime nextRelativePeriod() {
    try{
    return priorityTimer.getFireTime().subtract(
        Tools.rtClock.getTime(atime), rtime);
    }catch(IllegalStateException e){
      if(flagEnd){
        rtime.set(100000,0);
        return rtime;
      }else throw e;
    }
  }

  /**
   * the one in EventManager is private and we use another one here in order
   * to properly synchronized its access
   */
  private final RelativeTime actualCapacity;

  private final AsyncEvent wakeUp;
  private final AsyncEventHandler realSchedulable;

  public void enqueue(ManageableAsyncEventHandler meh) {
    super.enqueue(meh);
    wakeUp.fire();
  }

  /************** delegate to realSchedulable */

  protected void performStart() {
    priorityTimer.start();
  }

  public void run() {
    realSchedulable.run();
  }

  /** throw new UnsupportedOperationException(); **/
  ...
}
```

C.10 UserLandPollingTaskServer

```
package fr.upe.masson.realtime.eventManager;

import javax.realtime.MemoryParameters;
import javax.realtime.PeriodicParameters;
import javax.realtime.ProcessingGroupParameters;
import javax.realtime.RealtimeThread;
import javax.realtime.ReleaseParameters;
import javax.realtime.Scheduler;
import javax.realtime.SchedulingParameters;

public class UserLandPollingTaskServer extends AbstractUserLandTaskServer {

  public UserLandPollingTaskServer(PeriodicParameters parameters) {
    super(parameters);
```

```
  }

  private RealtimeThread realSchedulable =
    new RealtimeThread(priorityParameters,periodicParameters) {

    public void run() {

      while (!flagEnd) {
        manage(periodicParameters.getCost());
        waitForNextPeriod();
      }

    }

  };

  /************** delegate to  realSchedulable */

  protected void performStart() {
    realSchedulable.start();
  }

  public void run() {
    realSchedulable.run();
  }

  /** throw new UnsupportedOperationException(); **/
  ...
}
```