Tolérance aux fautes et amélioration du comportement d'un système Java temps-réel

Damien Masson

Tolérance aux fautes temporelles et amélioration du comportement d'un système Java temps réel

Damien MASSON

Institut Gaspard-Monge, Université de Marne-la-Vallée, Cité Descartes - 5, bd Descartes, Champs-sur-Marne, France.

Après avoir dressé un état de l'art sur la faisabilité théorique d'un système de tâches périodiques ordonnancé par un algorithme à priorités fixes préemptif, nous montrons dans cet article que des fautes temporelles peuvent tout de même se produire au sein d'un système théoriquement faisable en raison d'une mauvaise évaluation des caractéristiques des tâches, que ces fautes peuvent aboutir à une défaillance du système et que l'on peut utiliser les données calculées durant le contrôle d'admission pour mettre en place des détecteurs de fautes et définir un facteur de tolérance. Nous présentons ensuite les résultats obtenus sur un système de tâches périodiques codées avec Java temps réel et exécuté avec la machine virtuelle *jRate*. Ces résultats démontrent que la mise en place des détecteurs et de la tolérance aux fautes apporte une amélioration du comportement du système en présence de fautes.

Keywords: Real-Time Java, Fault Tolerance, Feasibility Analysis

Introduction

Les systèmes temps réel occupent une place de plus en plus importante dans l'industrie et s'étendent à des domaines de plus en plus variés. Le style de programmation nécessaire à la mise en place de contraintes temporelles impose pour l'heure l'utilisation de langages de bas niveau, comme l'assembleur ou le langage C. Les applications temps réel cherchent aujourd'hui a bénéficier des avantages du langage Java : un haut niveau d'abstraction (une approche orienté objet) et une indépendance vis a vis des plates-formes d'exécution (portabilité). Ceci a conduit à la spécification Java pour le temps réel (*RTSJ*) [4] qui a vu le jour en 2001. Elle décrit un ensemble de classes et d'interfaces et définit les contraintes imposées aux machines virtuelles Java temps réel (gestion de la mémoire, algorithmes d'ordonnancement, mécanismes de synchronisation).

Cette spécification est encore en constante évolution (une nouvelle version est parut au mois de Juin 2005). Il existe déjà de nombreuses machines virtuelles *RTSJ*, certaines commercialisées comme *Jamaïca VM* de la société *AICAS*, d'autres destinées à la recherche, comme *RI* (pour *reference implementation*) qui visait à montrer la faisabilité de l'aventure Java temps réel, *jRate* [7, 6], un projet open source basé sur une extension du *GNU GCJ compiler* ou encore *flex* [8] développée par le *MIT*. Quant à Sun microsystems, ils ont récemment annoncé la préparation de leur propre implantation, baptisée projet *Mackinac* [13, 3].

Si la spécification propose aux programmeurs des méthodes permettant de réaliser un contrôle d'admission de tâches périodiques temps réel, les machines que nous avons testées n'en fournissent pas encore une implantation satisfaisante :

- pour *RI* on peut facilement construire un exemple de système de tâches non faisable pour lequel le contrôle d'admission répondra *faisable* ;
- dans le cas de *jRate*, il suffit de consulter le fichier `PriorityScheduler.java` pour s'apercevoir que les méthodes concernées ne sont pas implantées.

Nous avons alors commencé à faire un état de l'art concernant les algorithmes de contrôle d'admission dans le cadre d'un système de tâches périodiques ordonnancées par un algorithme à priorités fixes préemptif, qui nous a permis d'implanter les méthodes déficientes de *RI* et manquantes dans *jRate*. Toutefois, si ces dernières garantissent la faisabilité théorique du système de tâche, en pratique, des fautes peuvent survenir pour de nombreuses raisons. Pour prendre en compte ces fautes, une approche couramment rencontrée dans la littérature est de mettre en place des mécanismes de détection et de traitement de la surcharge [12, 9, 5]. Notre approche, que nous présentons dans cet article, a consisté au contraire à utiliser un contrôle d'admission couplé à des mécanismes de détection et de traitement de faute, afin d'éviter une surcharge.

Nous commençons par présenter le cadre théorique de notre travail et les notations utilisées dans la section 1. Nous présentons l'algorithme de contrôle d'admission dans la section 2. Dans la section 3 nous définissons les fautes temporelles et nous expliquons comment nous pouvons les détecter en surchargeant des méthodes de la *Realtime Specification for Java (RTSJ)*. La section 4 présente les différents traitements des fautes que l'on a implantés. Nous expliquons quels mécanismes de mesure nous avons mis en place dans la section 5 et présentons les résultats de ces dernières dans la section 6. Enfin nous présentons en conclusion les perspectives de travail offertes par ces résultats.

1 Cadre de travail et notations

Nous limitons le cadre de notre travail à un système de N tâches périodiques ordonnancées par une politique préemptive à priorité fixe. Une tâche, notée τ_i, est caractérisée par :

- son coût noté C_i ;
- son échéance relative notée D_i ;
- sa période notée T_i ;
- sa priorité notée P_i.

Les mesures présentées sont obtenues avec le compilateur *RTSJ jRate* [7] car il est le seul parmi les implantations de cette norme que nous avons testées (*RI*, *JamaicaVM*, ...) à respecter certain aspects essentiels pour notre travail, comme la signature de la méthode `RealtimeThread.waitForNextPeriod()`. Elles ont été réalisées sur une machine équipée d'un processeur Intel pentium 4 cadencé à 2 GHz, de 500 Mo de mémoire et du noyau temps réel Timesys *2.4.18-timesys-4.0.243*.

2 Contrôle d'admission

La théorie de l'ordonnancement appliquée aux systèmes temps réels a été largement étudiée ces vingt dernières années. De nombreux résultats ont été publiés [11, 10, 1, 2]. Dans ce travail nous nous intéressons à l'ordonnancement à priorités fixes car toutes les implantations *RTSJ* doivent au moins proposer cette politique, ceci en raison de la facilité d'implantation de cette dernière.

Nous présentons dans cette section les résultats théoriques décris par Liu et Layland [11], et généralisés par Lehoczky [10] au cas où l'échéance de la tâche peut être supérieur à sa période.

2.1 Condition de charge

La charge du système, notée U s'exprime par :

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

Deux cas de figure se présentent alors :

- si $U > 1$, le système n'est pas faisable ;
- si $U \leq 1$, la condition de charge ne permet pas de décider de la faisabilité du système.

	Priorité (P_i)	Echéance relative (D_i)	Période (T_i)	Coût (C_i)
τ_1	20	6	6	3
τ_2	15	2	4	2

TAB. 1: Données du système de tâche

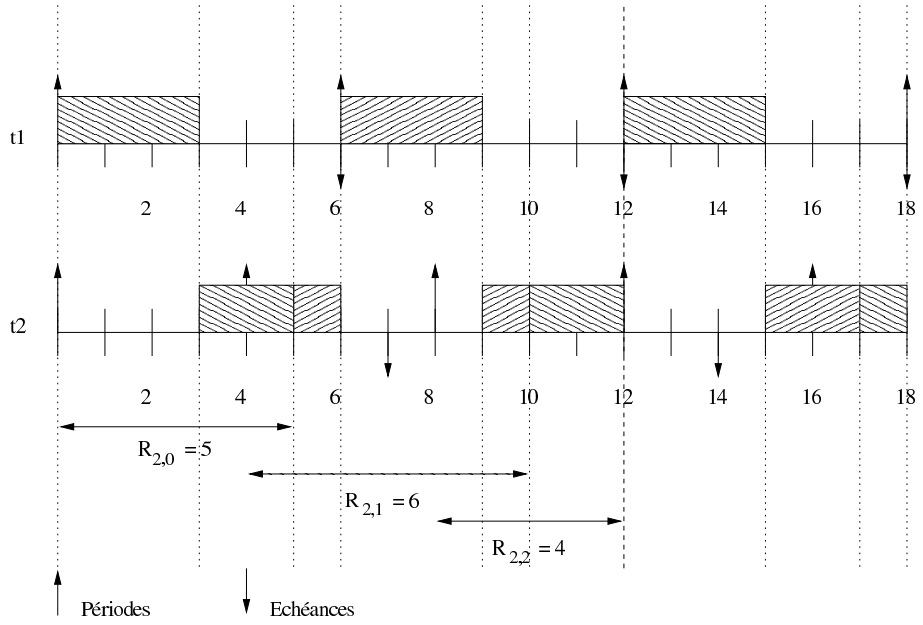

FIG. 1: Temps de réponses

2.2 Calcul des temps de réponses pire cas

Le temps de réponse pire cas d'une tâche est le temps maximal que prendra la tâche entre l'activation de l'une de ses instances et la fin de celle-ci. Si l'on prend pour hypothèse $D_i < T_i$, et que l'on se place dans le pire cas, c'est à dire que toutes les tâches sont activées au même instant, ce temps est atteint dès la première instance, mais cela n'est plus vrai dans le cas général. Ceci est mis en évidence par l'exemple présenté par le tableau 1 et la figure 1. Le calcul du temps de réponse pire cas dans le cas général nécessite par conséquent une étude plus approfondie.

Ce calcul a fait l'objet de nombreux travaux qui ont abouti à l'algorithme présenté dans la figure 2 page suivante. Le principe repose sur la constatation suivante : le traitement d'une instance de la tâche ne peut être retardé que par l'exécution de tâches plus prioritaires ou par ses instances précédentes, et son exécution ne peut être suspendue que par l'activation d'une tâche plus prioritaire. La formule présenté permet de calculer le temps de réponse de chaque instance d'une tâche en fonction du temps de réponse de la précédente par une formule de récurrence. Il reste alors à itérer ce calcul sur toutes les instances de la tâche jusqu'à en trouver une dont le temps de réponse est inférieur à la période, ce qui assurera qu'il n'y aura plus d'influence sur les instances suivantes : le temps de réponse pire cas sera alors le plus grand temps de réponse calculé durant ces itérations.

```
[Renvoit le pire temps de réponse en nano-secondes]
Fonction WCResponseTime( S : Schedulable) : entier long
    R_q : entier long
    R_max : entier long ← 0
    q : entier ← 0
    R_tmp : entier long
    Tant que ( vrai) faire
        R_q ← 0
        R_tmp ← S.cost * (q + 1)
        Tant que (R_q ≠ R_tmp) faire
            HP(S) : Ensemble des Schedulable de priorité supérieur ou égale à celle de S
            R_q ← R_tmp
            R_tmp ← S.cost * (q + 1)
            Pour tout S_j Dans HP(S) Faire R_tmp ← R_tmp + ⌈R_q / S_j.period⌉ * S_j.cost
        Fait
        R_q ← R_tmp
        Si (R_max < R_q − q * S.period) Alors
            R_max ← R_q − q * S.period
        Fin Si
        Si (R_q ≤ (q + 1) * S.period) Alors
            break ;
        Fin Si
        q ← q + 1
    Fait
    Retourner R_max
Fin
```

FIG. 2: Calcul du temps de réponse pire cas d'une tâche dans un système préemptif à priorités fixes

2.3 Implantation

Nous proposons aux développeurs un nouveau paquet, `javax.realtime.extended`, dans lequel il peut trouver la classe `RealtimeThreadExtended`. Cette classe étend `RealtimeThread` et surcharge différentes méthodes.

Les méthodes concernées par le contrôle d'admission sont `addToFeasibility()` et `removeFromFeasibility()` qui sont surchargées pour déléguer le travail à l'une de nos classe. Cette classe appelée `FeasibilityAnalysis` implante l'algorithme de la figure 2.

3 Détection de fautes

Le système est dit défaillant lorsque une (ou plusieurs) tâche dépasse son échéance (D_i). Si l'on réalise systématiquement un contrôle d'admission avant de démarrer un système de tâche, il ne devrait en principe pas y avoir de défaillance. Cependant, les algorithmes présentés dans la section précédente reposent sur l'hypothèse que les caractéristiques des tâches sont exactes et respectées à chaque instance, notamment leurs coûts (C_i). Or pour obtenir ce coût il faut mener une étude statistique de façon expérimentale. Il arrive

alors qu'une tâche dépasse ce coût, soit parce que ce dernier a été mal évalué, soit parce qu'un évènement extérieur au système a provoqué un ralentissement de ce dernier. La tâche commet alors une *faute*, pouvant entraîner sa défaillance, ou celle d'une tâche moins prioritaire.

La détection d'un dépassement de coût d'une tâche s'avère être un problème délicat, car sous-entend que l'on puisse mesurer le temps de possession des ressources CPU d'une tâche à tout moment. En revanche, le calcul des temps de réponses pires cas effectué lors du contrôle d'admission fourni pour chaque tâche la date à laquelle elle doit avoir fini son traitement. Cette date est obtenue en ajoutant le temps de réponse pire cas de la tâche à la date d'activation de l'instance.

La détection d'un dépassement de temps de réponse pire cas indique qu'une tâche a consommé plus de ressources que prévu lors de son admission.

Les instants critiques à surveiller correspondent à des dates relatives à chaque activation, c'est pourquoi un détecteur est une tâche périodique, de période identique à celle du thread surveillé, avec un décalage de phase avec ce dernier égal à son pire temps de réponse. Cette approche par tâches périodiques permet d'éviter la création d'un détecteur à chaque activation et ainsi de réduire le coût induit dans le système car une seule tâche temps réel est nécessaire pour chaque thread.

3.1 Implantation

Toujours dans notre classe `RealtimeThreadExtended`, nous surchargeons la méthode `start()` pour qu'un détecteur périodique soit lancé après un temps de latence égal au pire temps de réponse. Ce détecteur est implanté par un `PeriodicTimer` qui à son réveil vérifie l'état d'un booléen et d'un compteur d'instance ajouté à notre classe : si l'instance n'est pas terminée, un traitement de faute peut alors être déclenché. Le booléen et le compteur sont mis à jour en surchargeant la méthode `waitForNextPeriod()` qui est appelée à la fin de chaque traitement périodique et qui est bloquante jusqu'au démarrage de l'instance suivante :

```
public boolean waitForNextPeriod(){
  computeAfterPeriodic();
  boolean returnValue =
    super.waitForNextPeriod();
  computeBeforePeriodic();
  return returnValue;
}
```

Grâce à cette approche, nous sommes en mesure d'intervenir avant et après chaque traitement périodique de la tâche.

Remarquons que cela suppose que la méthode `waitForNextPeriod()` soit appelée une fois avant la première instance, ce qui est souvent le cas afin que le coût de la méthode `start` ne perturbe pas la première instance de la tâche.

4 Traitements

Désormais, nous sommes capable de détecter d'éventuelles fautes au sein de notre système de tâche. La question du traitement de ces fautes se pose alors. Les conséquences d'un dépassement de temps de réponse peuvent être de trois types :

- pas de défaillance du système, on observe simplement un retard éventuel dans les temps de réponse des tâches moins prioritaires ;
- la tâche fautive dépasse son échéance et devient donc défaillante, entraînant probablement des défaillances en cascade au sein des tâches moins prioritaires ;
- la tâche fautive se termine normalement avant son échéance, mais le retard qu'elle entraîne pour les tâches moins prioritaires provoque une série de défaillances.

Nous souhaitons empêcher que les tâches fautives de fortes priorités ne provoquent la défaillance de tâches non fautives de plus basses priorités. Pour cela, nous avons envisagé trois traitements que nous présentons dans les sous-sections suivantes. Nous comparons ces traitements et nous mesurons l'amélioration du comportement du système de façon pratique sur un exemple de trois tâches dans la section 6.

4.1 Arrêt immédiat des tâches fautives

La première idée de traitement consiste à simplement arrêter les tâches qui commettent des fautes. Cette approche est très pessimiste, car commettre une faute peut ne pas avoir de conséquence sur la faisabilité du système.

Notons par ailleurs que, en Java, on ne peut pas arrêter brutalement un thread. La solution théorique consisterait à baisser la priorité de la tâche afin que celle-ci se fasse préempter, mais cette possibilité n'est pas implantée avec jRate. Pour les besoins de notre analyse, nous avons donc ajouté un booléen dans la classe `RealtimeThreadExtended` qui est vérifié après chaque instruction de la boucle constituant le traitement périodique. Si ce booléen passe dans l'état `vrai`, alors on sort de la boucle et le thread est arrêté. Cela provoque un effet de bord embêtant : le thread doit dans sa méthode `run()` vérifier l'état du booléen, et donc faire appel à la méthode `RealtimeThread.currentRealtimeThread()`, dont le coût n'est pas borné . En raison de cela, la tâche va régulièrement commettre de petits dépassements de coût, de l'ordre de quelques millisecondes. Ces dépassements restants inférieurs à la précision de nos détecteurs, cela ne gêne pas notre analyse.

4.2 Arrêt après dépassement de la tolérance accordée à la tâche

Afin d'avoir une approche moins pessimiste, et permettre aux tâches fautives de continuer aussi longtemps qu'elles le peuvent sans gêner les tâches moins prioritaires, il convient de calculer ce que nous appellerons la tolérance du système, c'est à dire le coût supplémentaire que l'on peut accorder à chaque tâche sans que cela n'entraîne théoriquement de défaillance.

La tolérance correspond alors à la valeur que l'on peut ajouter au coût de chaque tâche de sorte que le système reste faisable (au sens de l'analyse décrite dans la section précédente). Pour la calculer, nous réalisons une recherche dichotomique de cette valeur entre 0 et `Integer.MAX_VALUE`. Ce calcul pourrait être optimisé en choisissant de meilleures bornes, mais puisqu'il est réalisé de façon statique avant le lancement du système de tâche, nous ne nous sommes pas intéressé à sa complexité.

Au lieu d'arrêter les tâches après le dépassement de leur pire temps de réponse, nous les arrêtons après le dépassement du pire temps de réponse calculé avec le coût incrémenté de la tolérance.

4.3 Arrêt après dépassement de la tolérance accordée à l'ensemble du système

Le traitement décrit précédemment considère que toutes les tâches ont les mêmes possibilités de commettre des fautes. La tolérance du système y est équitablement répartie entre toutes les tâches.

La dernière approche que nous avons implantée consiste à considérer que plus une tâche est prioritaire, plus elle a le droit de commettre une faute : cette fois les tâches sont arrêtées après un dépassement du pire temps de réponse égal à la tolérance maximale. Si la première tâche fautive termine avant d'avoir consommé toute la tolérance, le reste est attribué à une éventuelle seconde tâche fautive. Cette tolérance est obtenue en cherchant le dépassement de coût maximum que peut faire chaque tâche. Pour savoir quelle est la tolérance d'une tâche il faut alors retrancher à cette valeur celles des dépassements effectués par des tâches plus prioritaires.

5 Mesures

Afin d'effectuer nos mesures, nous avons développer différents outils. Le premier permet de parser un fichier qui décrit le système de tâche à tester. Les tâches sont construites et démarrées automatiquement. Durant leurs exécution, des données sont collectées. Un deuxième outils permet ensuite d'obtenir une représentation graphique de ces données sous forme de diagramme temporel.

Les données collectées correspondent aux dates clés de la vie du système qui sont :

– le début du traitement d'une instance (l'instant où la méthode `computeBeforePeriodic()` est appelée) ;
– la fin du traitement d'une instance (l'instant où la méthode `computeAfterPeriodic()` est appelée ;
– le déclenchement d'un détecteur.

Afin d'avoir une précision maximale, nous utilisons pour ces mesures une particularité des processeurs Intel, qui possèdent une instruction `RDTSC` (*ReaD Time-Stamp Counter*) permettant d'obtenir le nombre de cycles réalisés depuis la mise en route du système.

Nous avons donc écrit une librairie en Java natif (*JNI*) permettant d'exploiter cette instruction, afin d'obtenir des durées précises à la nanoseconde. Ces temps sont sauvegardés dans un champs de type `StringBuffer` afin de ne pas ralentir le système par des opérations d'entrées sorties. A la fin de l'exécution du système, ces tampons sont écrit dans un fichier de résultat qui peut alors être interprété par notre outils de représentation graphique.

Les figures présentées dans la section 6 ont été obtenues grâce à ces outils, et présentent donc des résultats expérimentaux obtenus sur de véritables systèmes de tâches codées en Java temps réel, et non pas des résultats obtenus par une simulation.

6 Résultats

Nous présentons dans cette section les résultats obtenus sur le système de tâches décrit dans le tableau 2. Le temps de réponse pire cas de la tâche τ_i y est noté $WCRT_i$ et sa tolérance A_i. Un dépassement de coût a volontairement été ajouté pour la tâche la plus prioritaire, ce qui représente le cas le plus défavorable. Nous comparerons les diagrammes temporels obtenus dans les cas suivants :

– pas de mécanisme de détection ;
– détecteurs actifs, aucun traitement en cas de faute ;
– détecteurs actifs, arrêt immédiat des tâches fautives ;
– détecteurs actifs, arrêt après dépassement d'un facteur de tolérance réparti équitablement entre toutes les tâches ;
– détecteurs actifs, arrêt après dépassement d'un facteur de tolérance attribué entièrement à la première tâche fautive.

	P_i	T_i	D_i	C_i	$WCRT_i$	A_i
τ_1	20	200	70	29	29	11
τ_2	18	250	120	29	58	11
τ_3	16	1500	120	29	87	11

TAB. 2: Système de tâches testé

Les figures présentées sont obtenues avec l'outil que nous avons mis au point, et présentent le comportement du système lors de la cinquième activation de la tâche τ_1, qui coïncide avec l'activation d'une instance de τ_2 et de τ_3.

Les flèches vers le haut représentent les périodes, les flèches vers le bas les échéances. Les doubles flèches en gras matérialisent les détecteurs, et les chevrons fermant indiquent les pires temps de réponses.

6.1 Exécution sans détecteurs

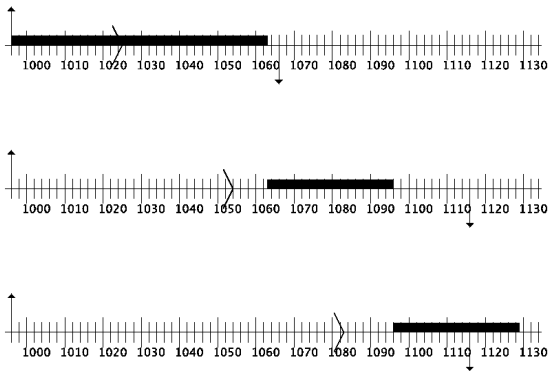

FIG. 3: Exécution sans les détecteurs

La figure 3 nous montre que la tâche τ_1 commet une faute temporelle à l'instant 1020*ms*. Elle termine tout de même avant son échéance, de même que la tâche τ_2 mais la tâche τ_3 subit une défaillance. C'est exactement le type de phénomène que nous souhaitons éviter.

6.2 Exécution avec détecteurs, sans traitement

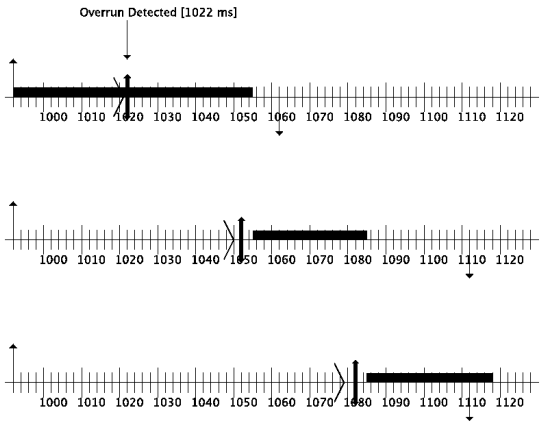

FIG. 4: Exécution sans traitement

L'exécution présenté dans la figure 4 est similaire à celle de la figure précédente. On constate que les détecteurs ont un petit retard. Ce retard ne peut pas excéder dix millisecondes. Ce phénomène est du à l'implantation des `PeriodicTimer` dans *jRate* : si la valeur donnée pour le premier déclenchement n'est pas un multiple de dix, la précision n'est pas bonne. Nous avons donc volontairement arrondi les valeurs de déclenchement des détecteurs. C'est pourquoi le détecteur de la tâche τ_1 a $30-29=1$ milliseconde de retard, celui de τ_2 $60-58=2$ millisecondes et celui de τ_3 $90-87=3$ millisecondes, ce que l'on peut vérifier sur le diagramme.

Le surcoût engendré dans le système par la présence de nos détecteurs est celui d'une préemption, plus une valeur non bornée correspondant à la vérification du booléen, comme expliqué dans la section 4.1 page 6. On peut estimer ce surcoût comme inférieur à la précision de la machine temps réel sous-jacente et

donc négligeable. Il faut tout de fois garder à l'esprit que plus il y a de tâches dans le système, plus il y a de détecteurs, et plus ce surcoût aura d'influence sur son exécution.

6.3 Arrêt immédiat des tâches fautives

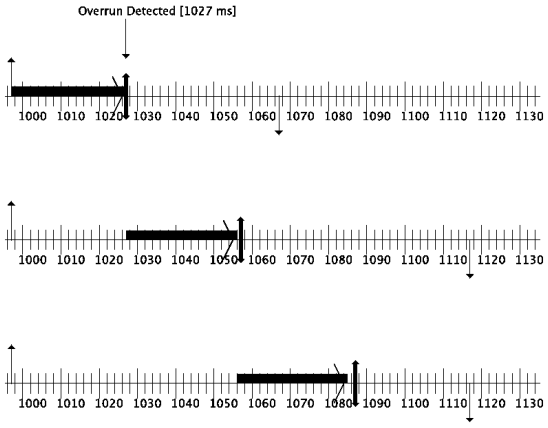

FIG. 5: Exécution sans tolérance

Dans l'exécution présentée par la figure 5, les tâches sont arrêtées dès qu'elles commettent des fautes. On constate que la seule tâche à commettre une défaillance est la tâche τ_1. Cependant après la fin de l'exécution de la tâche τ_3, le système est inoccupé, et il reste du temps avant son échéance. On peut alors penser que la tâche τ_1 aurait put avoir le temps de terminer, ou au moins de s'exécuter plus longuement sans cause la défaillance de τ_2 ou/et de τ_3.

6.4 Tolérance répartie sur toutes les tâches

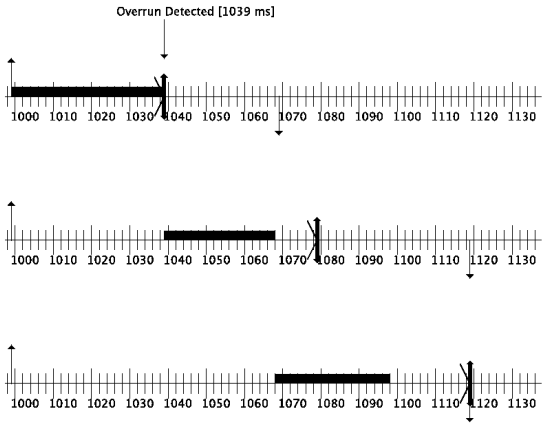

FIG. 6: Exécution avec tolérance répartie

La figure 6 est issue d'une exécution où nous avons laissé à toutes les tâches la même tolérance : onze millisecondes, leurs donnant ainsi le droit de toutes commettre une faute. Les pires temps de réponse indiqués par les chevrons fermant représentent maintenant le pire temps de réponse en prenant en compte un dépassement de coût de la tâche de onze millisecondes.

Tâche	Temps de réponse pire cas avec faute
τ_1	$WCRT_1 + A_1 = 11$
τ_2	$WCRT_2 + A_1 + A_2 = 22$
τ_2	$WCRT_3 + A_1 + A_2 + A_3 = 33$

TAB. 3: Temps de réponse pire cas avec faute

Le tableau 3 donne ces nouveaux temps de réponse pire cas. Nous constatons que seule la tâche τ_1 est arrêtée, et qu'elle a eu plus de temps pour s'exécuter que dans le cas précédent. Cependant, nous remarquons également qu'il reste du temps inoccupé, car les tâche τ_2 et τ_3 n'ont pas consommé leur tolérance, puisqu'elles n'ont pas commis de faute.

6.5 Tolérance accordée aux premières tâches fautives

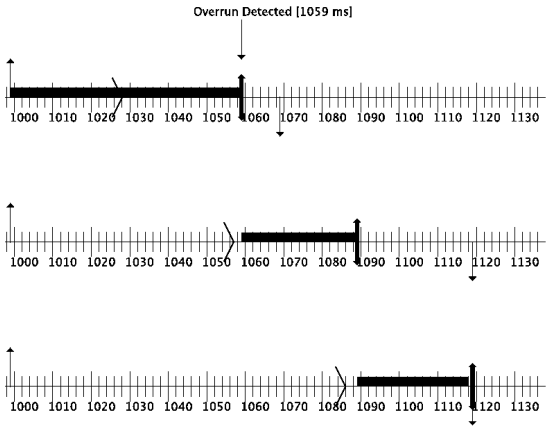

FIG. 7: Exécution avec tolérance

La figure 7 correspond au dernier traitement envisagé : tout le temps disponible du système dans le pire cas d'exécution, soit trente-trois millisecondes, est accordé à la première tâche fautive. Si cette tâche ne consomme pas toute cette tolérance, et qu'une autre tâche commet une faute, elle profitera de ce qui reste de la tolérance. La tâche τ_1 est donc arrêtée trente-trois millisecondes après son pire temps de réponse et comme ni τ_2 ni τ_3 ne commettent de faute, elles terminent toutes les deux juste avant leurs échéances.

Conclusions et travaux futurs

L'étude des résultats théoriques de l'analyse de faisabilité d'un système nous a permis de fournir une implantation correcte des méthodes spécifiées par la norme *RTSJ* pour le contrôle d'admission, ce que nous avons pu vérifier concrètement sur la machine *jRate*.

Nous avons vu que la faisabilité théorique reposait sur des données comme le coût des tâches qui peuvent être éventuellement dépassées durant une exécution réelle à cause de facteurs extérieurs, provoquant des fautes au sein du système de tâches.

Grâce à l'étude des pires temps de réponse, nous avons pu définir un type de faute dont la détection ne nécessite pas une surveillance complète de l'utilisation du CPU. Nous en avons déduit un mécanisme de détection que nous avons codé et testé.

Enfin nous avons montré que l'étude théorique précédente permettait d'obtenir un facteur de tolérance qui maximise le temps d'exécution d'une tâche avant que sa faute ne provoque une défaillance du système.

Cependant, notre étude considère un système assez statique, dans lequel toutes les tâches sont connues avant le lancement, permettant de mettre en place des algorithmes coûteux en temps pour calculer les temps de réponse et la tolérance du système.

Notre objectif dans la suite de ce travail sera d'arriver aux mêmes résultats dans un système plus dynamique où l'on peut ajouter ou enlever des tâches « en temps réel » en adaptant le comportement de nos détecteurs.

Par ailleurs, nous ne nous sommes pas posé les problèmes liés a la précédence et au partage des ressources entre les tâches du système. Il conviendra dans ce dernier cas d'étudier l'influence de la tolérance sur la détermination du temps de blocage (b_i).

Un autre axe de notre recherche consistera en l'étude de l'admission et de la détection des fautes dans le cas de tâches apériodiques.

Remerciement

Je tiens à remercier tout spécialement Serge Midonnet pour tous ses conseils et relectures.

Références

[1] N. C. Audsley, A. Burns, M. F. Richardson, and A. J. Wellings. Hard Real-Time Scheduling : The Deadline Monotonic Approach. In *8th IEEE Workshop on Real-Time Operating Systems and Software*, Atlanta, 1991.

[2] Enrico Bini, Giorgio C. Buttazzo, and Giuseppe M. Buttazzo. Rate monotonic analysis : The hyperbolic bound. *IEEE Transactions on Computers*, 52(07) :933–942, July 2003.

[3] Greg Bollella, Bertrand Delsart, Romain Guider, Christophe Lizzi, and Frederic Parain. Mackinac : Making hotspot(tm) real-time. In *Eighth IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC'05)*, pages 45–54, 2005.

[4] Greg Bollella and James Gosling. *The Real-Time Specification for Java*, volume 33. Addison-Wesley Publishing, 2000.

[5] G. C. Buttazzo and J. Stankovic. Red : A robust earliest deadline scheduling algorithm. In *Proceedings of Third International Workshop on Responsive Computing Systems*, 1993.

[6] Angelo Corsaro and Douglas C. Schmidt. The design and performance of the jrate real-time java implementation. In *On the Move to Meaningful Internet Systems, 2002 - DOA/CoopIS/ODBASE 2002 Confederated International Conferences DOA, CoopIS and ODBASE 2002*, pages 900–921, London, UK, 2002. Springer-Verlag.

[7] Angelo Corsaro and Douglas C. Schmidt. Evaluating real-time java features and performance for real-time embedded systems. In *RTAS '02 : Proceedings of the Eighth IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS'02)*, page 90, Washington, DC, USA, 2002. IEEE Computer Society.

[8] Catalin A. Francu. Real-time scheduling for java. Master's thesis, MIT Dept. of Electrical Engineering and Computer Science, 2002.

[9] G. Koren and D. Shasha. Rt-0138 - d-over : an optimal on-line scheduling algorithm for overloaded real-time systems. Technical report, INRIA, february 1992.

[10] J.P. Lehoczky. Fixed priority scheduling of periodic task sets with arbitrary deadline. In *11th IEEE Real-Time System Symposium*, pages 201–209, December 1990.

[11] C. L. Liu and James W. Layland. Scheduling algorithms for multiprogramming in a hard real time environment. *Journal of the Association for Computing Machinery*, 20(1) :46–61, January 1973.

[12] Carey Douglass Locke. *Best-effort decision-making for real-time scheduling*. PhD thesis, Computer Science Department, Carnegie Mellon University, Pittsburgh, PA, May 1986.

[13] Sun microsystems. *The Real-Time Java Platform, A Technical White Paper*, June 2004.