Utilisation des contraintes (m,k)-firm pour la gestion de la QdS[1] dans les SGBD temps réel

Emna Bouazizi et Claude Duvallet

LITIS, Université du Havre,
25 rue Philippe Lebon, BP 540
76 058 Le Havre Cedex, FRANCE.
MIRACL, Institut Supérieur d'Informatique et de Multimédia de Sfax,
Université de Sfax,5000 Tunisie
emna.bouazizi@gmail.com, claude.duvallet@univ-lehavre.fr

RÉSUMÉ. *Dans les systèmes temps réel manipulant des tâches périodiques, des travaux introduisant la notion de contraintes (m,k)-firm ont été effectués. Ces contraintes permettent de relaxer les contraintes temps réel des tâches en autorisant que certaines invocations d'une tâche soient perdues. Plus précisément, une tâche sous contraintes (m,k)-firm doit garantir que m invocations parmi k respectent leurs échéances. D'un autre côté, dans les SGBD, notamment temps réel, certaines transactions sont périodiques. Elles permettent, par exemple, de rafraîchir des données temporelles de la base à partir de capteurs. Ces transactions sont appelées "transactions de mise à jour" par opposition aux "transactions utilisateur" qui manipulent des données temporelles ou non. Dans ce papier, nous proposons une adaptation des contraintes (m,k)-firm aux transactions de mise à jour dans les SGBD temps réel. En se basant sur l'exploitation d'une architecture avec contrôle par rétroaction et sur les contraintes temps réel (m,k)-firm, nous proposons d'augmenter cette architecture afin de filtrer l'arrivée des transactions de mise à jour en tenant compte des variations de la charge du système.*

ABSTRACT. *In Real-Time Systems that manage periodic tasks, the notion of (m, k)-firm constraints has been introduced. These constraints can relax real-time constraints of tasks by allowing the lost of some invocations task. Specifically, a task with (m, k)-firm constraints must ensure that m invocations among k meet their deadlines. In Real-Time Database Systems, transactions can be classified into two classes: update transactions and user transactions. Update transactions are used to update the values of real-time data in order to reflect the state of real world. Update transactions execute periodically and have only to write real-time data. User transactions, representing user requests, arrive aperiodically and may read real-time data, and read or write non real-time data. The aim of our work is to maintain a robust RTDBS behavior and to decrease the number of transactions which miss their deadline. We propose an approach based on Feedback Control Scheduling (FCS) that allows also to use (m, k)-firm constraints.*

MOTS-CLÉS : *SGBD temps réel, ordonnancement, transactions de mise à jour, contraintes (m,k)-firm.*

KEYWORDS: *Real-Time Database Systems, Scheduling, Update Transactions, (m,k)-firm constraints.*

1. Qualité de service.

1. Introduction

Les systèmes temps réel distinguent généralement les tâches temps réel souples et dures. Des auteurs ont proposé de relaxer les contraintes temps réel des tâches en distinguant, pour chaque tâche, une partie obligatoire et une partie optionnelle [HAM 95] [WAN 02]. Ceci augmente la capacité d'une tâche à respecter ses contraintes temps réel puisque seule la partie obligatoire doit réellement les respecter. La notion de contraintes temps réel (m,k)-firm a notamment été introduite pour les tâches périodiques [HAM 95]. Cette approche se fonde sur la possibilité de perdre certaines invocations d'une tâche tout en conservant une certaine qualité de service globale. Par exemple, une interpolation des résultats peut permettre de récupérer les invocations perdues.

Pour gérer efficacement les tâches temps réel sous contraintes (m,k)-firm, de nouveaux algorithmes d'ordonnancement ont été proposés [HAM 95] [RAM 99] [WAN 02]. Ces algorithmes sont basés sur des méthodes dynamiques ou statiques permettant de déterminer (en ligne ou hors ligne) les invocations des tâches qui peuvent être considérées comme optionnelles [WAN 02]. Ensuite, en fonction de la charge du système, on peut décider de ne pas exécuter les invocations optionnelles. De même, en cas de conflit (d'accès à un serveur par exemple), les invocations obligatoires seront plus prioritaires que les invocations optionnelles. Nous présentons plus en détail la gestion des tâches (m,k)-firm dans la section 2.

Les applications temps réel qui manipulent de gros volumes de données, sont généralement gérer à l'aide des systèmes appelés SGBDTR[1]. Les transactions qui s'exécutent dans un tel système sont soumises à des échéances [RAM 93] [DUV 99] [RAM 04]. Une transaction temps réel est considérée comme correcte si elle se termine correctement avant son échéance.

Dans la littérature, il a été proposé des modèles de SGBD temps réel qui manipulent, d'une part, des transactions de mise à jour (périodiques) pour rafraîchir les données temporelles et, d'autre part, des transactions utilisateur qui manipulent généralement des données temps réel ou non et/ou des données dérivées [AMI 03b] [AMI 06] [KAN 02a] [BOU 09]. L'architecture proposée est composée de différents modules permettant de gérer efficacement l'exécution à la fois des transactions de mise à jour et des transactions utilisateur. Nous présentons plus précisément cette architecture dans la section 3.

Dans un grand nombre d'applications temps réel basées sur un SGBD temps réel, il n'est pas possible de prévoir les arrivées des transactions utilisateur et des phases d'instabilité peuvent se produire. Pour gérer ces phases d'instabilité du système (phase de surcharge ou phase de sous-utilisation), une méthode basée sur la rétroaction a été présentée [LU 01] [AMI 06]. Elle permet d'influencer le contrôleur d'admission pour écarter des transactions utilisateur lorsque la charge du système augmente ou de laisser plus de transactions entrer dans le système en cas de sous-utilisation [LU 01]. D'autre part, en fonction de la fraîcheur des données, certaines transactions de mise à jour peuvent également être omises (cf. section 3.2).

Dans cet article, nous nous intéressons aux transactions de mise à jour et nous étudions l'application des contraintes (m,k)-firm à ces transactions. Nous étudions la façon dont la charge du système (en combinaison avec la fraîcheur des données) peut être utilisée pour influencer l'admission des transactions de mise à jour. Pour cela, nous rappelons tout d'abord l'utilisation des contraintes (m,k)-firm pour les tâches temps réel (cf. section 2). Ensuite, nous présentons le modèle de SGBD temps réel que nous utilisons et nous rappelons le principe de la rétroaction dans un tel système (cf. section 3). Enfin, basé sur ces deux notions, nous présentons notre contribution pour gérer efficacement les transactions de mise à jour (cf. section 4). Enfin, nous concluons cet article sur l'originalité de notre approche et quelques perspectives à notre travail.

2. Contraintes (m,k)-firm dans les systèmes temps réel

La récurrence des tâches périodiques dans les systèmes temps réel permet d'ignorer certaines invocations (ou jobs) en utilisant les contraintes (m,k)-firm. Ces contraintes spécifient qu'au moins m tâches dans une fenêtre de k invocations ($0 \leq m \leq k$) doivent respecter leur échéance [HAM 95]. Autrement dit, parmi k invocations, m sont obligatoires et $k - m$ sont optionnelles. On peut remarquer que les tâches *firm* traditionnelles, c'est à dire strictes non critiques, sont un cas particulier des tâches (m,k)-firm où $m = k$ ($\neq 0$). En effet, dans ce cas, toutes les invocations d'une même tâche doivent respecter leur échéance.

1. SGBD Temps Réel.

Bernat a montré par le biais d'un exemple, pourquoi il est préférable d'utiliser deux paramètres pour définir ce type de contraintes [BER 98]. En effet, il explique pourquoi le ratio de succès ne donne pas suffisamment d'informations sur les exécutions des tâches : un ratio de succès de 90% est équivalent à une instance perdue parmi 10 ou à 100 parmi 1000. Si les tâches qui manquent leur échéance sont consécutives, alors le deuxième cas peut amener à des résultats très imprécis, voire erronés. On utilise alors deux paramètres : le premier indique le nombre d'instances qui doivent respecter leur échéance et le second limite la fréquence à laquelle les invocations peuvent échouer. Dans ce cas, une tâche ayant des contraintes (9,10)-firm est plus critique qu'une tâche sous contraintes (900,1000)-firm.

Une tâche sous contrainte (m,k)-firm peut se trouver dans deux états différents : un "état normal" et un "état d'échec dynamique". Dès que $(k-m+1)$ invocations d'une tâche ne peuvent respecter leur échéance, la tâche passe dans un état d'échec dynamique. Le but d'un système sous contraintes (m,k)-firm est donc de limiter le nombre de tâches qui passent dans un état d'échec dynamique.

De plus, il a été montré que la notion de contraintes (m,k)-firm est appropriée pour la gestion (la spécification) de la Qualité de Service (QdS) d'une application temps réel [WAN 02]. En effet, un système soumis à des contraintes (m,k)-firm fournit différents niveaux de QdS correspondant aux différentes valeurs comprises entre m et k ($0 \leq m \leq k$). En d'autres termes, la QdS augmente en fonction du nombre d'invocations optionnelles qui s'exécutent. D'autre part, il est clair qu'un système sous contraintes (m,k)-firm nécessite moins de ressources qu'un système classique puisque certaines invocations peuvent être écartées.

Pour gérer efficacement l'exécution des tâches (m,k)- firm, de nouveaux algorithmes d'ordonnancement ont été proposés [HAM 95] [RAM 99] [WAN 02]. Ils se décomposent en deux grandes familles :

• Les algorithmes dynamiques : la priorité de chaque tâche est déterminée en fonction de l'état du système. Par exemple, le protocole DBP (*Distance Based Priority*) calcule la probabilité pour une tâche de tomber dans un état d'échec à partir de l'historique de la tâche [HAM 95]. La distance correspond au nombre de tâches qui peuvent être écartées tout en respectant les contraintes (m,k)-firm.

• Les algorithmes statiques : la priorité est déterminée hors ligne en utilisant un paramètre fixe, par exemple le ratio de succès m/k. Un exemple de protocole statique est ERM (*Enhanced Rate Monotonic*) qui détermine au préalable les invocations obligatoires et les invocations optionnelles [RAM 99].

En bref, les algorithmes statiques fournissent une vision déterministe du système alors que les algorithmes dynamiques fournissent plutôt une vision probabiliste. En contrepartie, les algorithmes dynamiques tiennent compte des éventuels changements du système.

3. Modèle de SGBDTR basé sur la rétroaction

Dans une application reposant sur l'utilisation de SGBD temps réel, des transactions en provenance des utilisateurs arrivent à des fréquences variables. Lorsque la fréquence augmente de façon considérable, l'équilibre du SGBD temps réel est mis en péril. Durant ces périodes de surcharge, le SGBD temps réel va potentiellement disposer de ressources moins importantes et les transactions temps réel vont alors manquer leur échéance en plus grand nombre. Des travaux basés sur une approche Qualité de Service (QdS) [KAN 02b] [AMI 03b] tentent de rendre les SGBD temps réel plus robustes face aux périodes d'instabilité (périodes de sous-utilisation et périodes de surcharge). Ces travaux s'appuient sur des techniques de contrôle avec rétroaction[2] [LU 01] et autorisent la manipulation de résultats imprécis [LIU 94] [LIU 91]. Dans cette section, nous allons détailler ces travaux sur lesquels s'appuie notre proposition et plus particulièrement nous allons présenter le modèle de SGBD temps réel que nous considérons.

3.1. *Les transactions temps réel*

Les transactions temps réel utilisées par notre modèle possèdent des échéances de type *firm* (à échéances strictes non critique [DUV 99]). Par conséquent lorsqu'elles dépassent leur échéance, elles deviennent in-

2. Feedback Control Scheduling (FCS)

utiles pour le système et sont abandonnées. Dans notre cas, nous considérons deux types de transactions temps réel :

• Les transactions de mise à jour : elles ont pour tâche de mettre à jour régulièrement les données acquises auprès de capteurs. Ces transactions sont exécutées périodiquement pour rafraîchir la valeur des données temps réel.

• Les transactions utilisateur : elles effectuent des opérations de lecture/écriture de données non temps réel et/ou des lectures de données temps réel. La non prévisibilité de leur arrivée dans le système et de la charge qu'elles suscitent rend adéquate l'utilisation d'une architecture basée sur le retour d'expérience (ou rétroaction) pour gérer les variations importantes de charge [LU 01].

Dans cet article, nous allons surtout considérer les transactions de mise à jour pour contribuer à la stabilisation de la charge du système lorsque cela est nécessaire (phase de surcharge ou phase de sous-utilisation).

3.2. *Les données temps réel et la qualité des données*

Les données temps réel représentent la capture de l'état du monde réel. Par exemple, dans une centrale nucléaire, on peut trouver des capteurs de température qui sont utilisés pour contrôler le système et détecter les éventuelles anomalies (surchauffe du système ou autre). Ces données doivent être remises à jour régulièrement afin de refléter au plus près le monde réel. Elles possèdent donc une durée de validité qui représente la période pendant laquelle elles peuvent être utilisées.

Amirijoo et al. ont introduit une notion de la Qualité des Données (QdD) [AMI 03b] qui permet de considérer qu'une donnée stockée dans la base peut posséder une certaine déviation par rapport à sa valeur dans le monde réel. On appelle cette déviation "erreur sur la donnée" (notée DE[3]) et on la calcule en faisant la différence entre la donnée en base et la valeur du monde réel. Par exemple, on peut admettre que la donnée température lorsqu'elle vaut "37°2" en base est très proche de "37°3" qui est la donnée réelle et que, par conséquent, il n'est pas nécessaire de mettre à jour cette donnée. Cette déviation possède un seuil (MDE[4]) qui permet de déterminer si la transaction qui souhaite mettre à jour une donnée temps réel peut être écartée (DE $<$ MDE) ou pas (DE $\geq$ MDE). Ce travail est effectué par le *contrôleur de précision*.

3.3. *Les critères de performance*

Trois principales mesures de performance sont considérées dans les travaux relatifs au modèle de SGBD temps réel basé sur le principe de rétroaction [AMI 03a] [AMI 06] : $MissRatio$ (MR), $DataFreshness$ (DF) et $DataError$ (DE).

1) $MissRatio$: ce paramètre est défini comme suit :

$$MR = 100 \times \frac{\#Tardy}{\#Submitted}(\%) \quad (1)$$

où $\#Tardy$ représente le nombre de transactions qui ratent leur échéance, et $\#Submitted$ est le nombre total des transactions.

2) $DataFreshness$: dans une base de données temps réel, une donnée peut devenir obsolète (non fraîche) avec le passage du temps. Pour mesurer la fraîcheur d'une donnée d_i, on utilise son paramètre AVI. Une version de donnée possède une estampille qui indique la valeur de la dernière observation de la donnée dans le monde réel. La donnée d_i est considérée comme fraîche si : $TempsCourant - Estampille(d_i) \leqslant AVI(d_i)$. La fraîcheur de la base de données peut ainsi être mesurée. Elle représente le rapport entre les données fraîches et l'ensemble des données de la base.

3) $DataError$: représente l'écart entre la valeur courante de la donnée (*CurrentValue(d_i)*) et sa valeur mise à jour (*UpdateValue(d_i)*). La limite supérieure de l'erreur est représentée par le paramètre MDE,

3. Data Error
4. Maximum Data Error

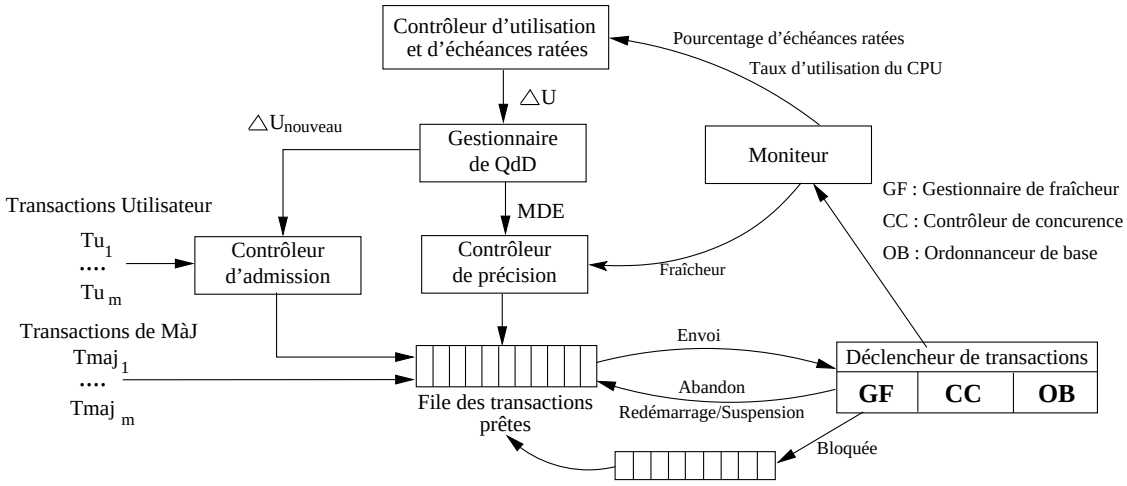

Figure 1. *Architecture de SGBDTR basée sur la rétroaction.*

erreur maximale sur la donnée (voir paragraphe 2.1). Le paramètre DE d'une version de donnée d_i est défini comme :

$$DE_i = 100 \times |\frac{CurrentValue(d_i) - UpdateValue(d_i)}{CurrentValue(d_i)}|(\%) \quad (2)$$

Notons que la qualité de donnée (QdD) dépend de sa fraîcheur et de la valeur du paramètre DE.

3.4. *Le modèle global*

Nous allons présenter en détail dans cette section les parties du modèle qui nous intéressent plus particulièrement dans cet article. En reprenant la figure 1 [AMI 03b], nous allons considérer le modèle général et donner une brève description de chacun des composants de ce modèle.

Le *contrôleur d'admission* a pour tâche de contrôler les transactions utilisateur qui sont acceptées dans le système. Il effectue ce contrôle en fonction de la charge d'utilisation calculée et des paramètres de qualité de service spécifiés par les DBA. Son fonctionnement est contrôlé par la boucle de rétroaction qui lui fournit ses paramètres de fonctionnement.

Les transactions qui sont admises dans le système sont placées dans une file d'attente avant d'être envoyées au *déclencheur de transactions*. Ce *déclencheur de transactions* possède pour fonction de gérer l'exécution des transactions. Il dispose de plusieurs modules complémentaires :

• Un *gestionnaire de fraîcheur (GF)* : il vérifie la fraîcheur des données qui vont être accédées par une transaction. Si les données sont obsolètes (non fraîches) alors la transaction est mise en attente dans une file.

• Un *contrôleur de concurrence (CC)* : il est chargé de gérer les conflits d'accès aux données qui apparaissent entre les transactions. Dans la plupart des travaux [KAN 02b, AMI 03b], il s'agit du protocole 2PL-HP (*Two Phase Locking High Priority*) [ABB 88].

• Un *ordonnanceur de base (OB)* : il s'agit bien souvent d'EDF (*Earliest Deadline First*) [BUT 97] qui ordonnance les transactions selon le principe que la transaction qui possède l'échéance la plus proche doit être exécutée en priorité.

Les différents modules du *déclencheur de transactions* peuvent être remplacés par des modules équivalents. C'est ainsi que l'on peut modifier la politique de contrôle de concurrence ou modifier la politique d'ordonnancement.

Un *moniteur* permet de mesurer les performances du système en inspectant l'exécution des transactions (quantité de transactions terminées, abandonnées, qui ont ratées leur échéance, ...). Les valeurs ainsi mesurées permettent d'alimenter le *contrôleur de qualité de service* et font parties de la boucle de contrôle par rétroaction qui va contribuer à stabiliser le système.

Le *contrôleur d'utilisation et d'échéances ratées* (ou *contrôleur de qualité de service*) permet de réajuster les paramètres de QdS en fonction des valeurs déterminées par le *moniteur* et des paramètres de référence[5]. Les valeurs ainsi obtenues sont transmises au *contrôleur d'admission* et au *gestionnaire de qualité des données*.

Le *gestionnaire de qualité des données* permet de réajuster la valeur du paramètre MDE qui constitue le paramètre de QdD. La valeur de MDE est calculée en fonction de l'utilisation du système. Le paramètre MDE est ensuite fourni au *contrôleur de précision* qui écarte les transactions de mise à jour lorsque les données à mettre à jour sont suffisamment représentatives du monde réel en considération de la valeur de MDE.

3.5. *La boucle de contrôle par rétroaction*

La boucle de rétroaction possède pour tâche de stabiliser le système durant les phases d'instabilité. Pour cela, elle s'appuie sur le principe d'observation puis d'auto-adaptation. L'auto-adaptation a lieu tout au long du fonctionnement du système car les demandes des utilisateurs sont imprévisibles et la charge doit être ajustée en permanence. L'observation consiste à prendre en compte l'état de fonctionnement du système et à déterminer s'il correspond aux paramètres de qualité de service initialement spécifiés. Cette observation se fait via le *moniteur*. À partir de l'observation effectuée, le système adapte ses paramètres, via le *contrôleur de qualité de service*, afin d'augmenter ou de diminuer les transactions acceptées dans le système. Cette adaptation va provoquer une modification du comportement du système et ainsi des valeurs observées par le *moniteur*.

Le fonctionnement de la boucle de rétroaction doit tendre vers une stabilité du système autour d'une valeur de référence, fixée par le DBA (par exemple, il peut s'agir d'un taux d'utilisation de 80%). Le but est donc de réduire l'oscillation du système autour de cette valeur de référence.

4. Transactions de mise à jour et contraintes (m,k)-firm

Dans cette section, nous nous proposons d'appliquer les contraintes (m,k)-firm en combinaison avec la boucle de rétroaction pour pouvoir écarter certaines instances des transactions de mise à jour lorsque la charge du système varie.

4.1. *Problématique*

Le contrôle par rétroaction présenté dans la section précédente permet de filtrer les transactions utilisateur suivant la charge observée du système. Si la charge du système augmente, alors on écarte de plus en plus de transactions utilisateur pour rétablir cette charge. Pour les transactions de mise à jour, le contrôle s'effectue sur la fraîcheur des données. Autrement dit, on peut écarter des transactions si la valeur à mettre à jour est similaire à la précédente. Cela implique que, dans un premier temps, toutes les transactions de mise à jour sont acceptées par le système et insérées dans la file d'attente. Dans le cas où la charge du SGBD temps réel devient trop importante, on ajuste la valeur du paramètre MDE et on vérifie la fraîcheur des données à mettre à jour. En d'autres termes, toutes les transactions de mise à jour sont considérées comme étant de même niveau (de même priorité) ; seule la périodicité des mises à jour permet de spécifier la criticité des données temps réel.

Le contrôle d'admission des transactions utilisateur est plus "sévère" puisque dès leur arrivée dans le système, elles doivent répondre aux critères du contrôleur d'admission pour être insérées dans la file d'at-

5. fournit par le DBA.

tente. Ainsi, on considère que les transactions de mise à jour sont plus prioritaires que les transactions utilisateur : il est difficilement concevable de laisser passer des transactions utilisateur alors que les données qu'elles doivent accéder ne sont pas fraîches. Cependant, nous pensons qu'un contrôle des transactions de mise à jour dès leur arrivée permet de rétablir plus rapidement la stabilité de la charge du système tout en garantissant une certaine QdS.

Dans la suite de cet article, nous nous intéressons au contrôle d'admission des transactions de mise à jour en utilisant les contraintes (m,k)-firm proposées pour la gestion des tâches dans les systèmes temps réel en combinaison avec la méthode de rétroaction.

4.2. *Définition des transactions de mise à jour sous contraintes (m,k)-firm*

Avant de montrer les modifications apportées au système pour intégrer le contrôle des transactions de mise à jour, nous allons définir le concept de "transactions de mise à jour (m,k)-firm". Le but de ce contrôle est de pouvoir écarter des transactions de mise à jour avant même qu'elles n'entrent dans le système tout en garantissant qu'elles respectent certaines contraintes.

L'une des caractéristiques intéressantes des transactions de mise à jour est la périodicité. En effet, les contraintes (m,k)-firm sont basées sur la périodicité des tâches temps réel pour pouvoir écarter certaines invocations en limitant la fréquence de perte. Nous proposons d'appliquer les contraintes (m,k)-firm aux transactions de mise à jour de la façon suivante.

Chaque transaction (de mise à jour) se voit assigner deux paramètres m et k : le premier indique le nombre minimum d'instances qui doivent respecter leur échéance et le second limite la période durant laquelle des instances peuvent être perdues. Autrement dit, la définition des contraintes (m,k)-firm pour les transactions de mise à jour est identique à celle des contraintes (m,k)-firm pour les tâches dans les systèmes temps réel.

On attribue également un autre paramètre, m_c (m courant), permettant de balayer les différentes valeurs comprises entre m et k. Au début, la valeur de m_c est fixée à k. Lorsqu'une surcharge du système survient, on tente de diminuer la valeur du paramètre m_c tout en respectant la contrainte $m_c \geq m$. Inversement, si on constate une sous-utilisation du système, alors la valeur de m_c peut être incrémentée en respectant la contrainte $m_c \leq k$.

Il reste à déterminer dans quelle mesure l'invocation courante d'une transaction de mise à jour peut être écartée du système en respectant ses contraintes (m,k)-firm. Pour cela, on utilise une méthode similaire au protocole d'ordonnancement DBP proposé pour les tâches temps réel. En fait, on vérifie, à partir de l'historique de la transaction, si l'invocation courante est une invocation obligatoire ou optionnelle tout en respectant les contraintes (m_c,k)-firm de la transaction. Si l'invocation courante est optionnelle, alors elle peut être écartée du système.

Il est intéressant de noter que même une fois acceptées par le contrôleur (m,k)-firm, les transactions de mise à jour peuvent à nouveau être écartées par le *contrôleur de précision*.

4.3. *Contraintes (m,k)-firm et criticité des transactions*

Les contraintes (m,k)-firm permettent de gérer différents niveaux de criticité des transactions. En effet, plus l'écart entre m et k est important, plus la transaction peut écarter des instances et moins elle est critique. Dans la pratique, les valeurs lues par les capteurs ne sont pas toutes de la même importance. Par exemple, dans un système temps réel embarqué tel qu'un avion, les informations concernant la température des moteurs sont plus importantes que les informations concernant l'air conditionné de la cabine. Dans ce cas, pour les capteurs de la cabine, on permettra à plus d'instances de la transaction de mise à jour d'être perdues que pour la transaction qui gère la température des moteurs. On pourra, par exemple, attribuer des contraintes (9,10)-firm pour la température des moteurs et des contraintes (10,100)-firm pour la température de la cabine. Les contraintes (m,k)-firm devront donc être fixées par le concepteur de l'application pour chaque transaction de mise à jour en fonction de la criticité des données auxquelles elles accèdent.

5. Architecture FCSA utilisant un contrôleur (m,k)-firm des transactions de mise à jour : (m,k)-firm-FCSA

L'architecture (m,k)-firm-FCSA est une extension de l'architecture FCSA classique. Pour manipuler les transactions de mise à jour (m,k)- firm, nous avons apporté quelques modifications à l'architecture générale FCSA. Nous avons notamment ajouté le contrôleur d'admission des transactions de mise à jour basé sur les contraintes (m,k)-firm des transactions. Nous le nommons "contrôleur (m,k)-firm". Ce contrôleur nécessite des informations provenant d'autres composants pour gérer l'admission des transactions. En effet, nous souhaitons écarter plus ou moins d'invocations des transactions lorsque la charge du système varie. C'est pourquoi, comme pour le contrôleur d'admission des transactions *utilisateur*, un paramètre provenant de la boucle de rétroaction lui est transmis (ΔU_{maj}). L'algorithme employé au niveau du *gestionnaire de qualité des données* est ainsi modifié afin de répartir l'effort à fournir pour chacun des composants (*contrôleur d'admission, contrôleur (m,k)-firm, contrôleur de précision*).

La seconde information nécessaire au contrôleur (m,k)-firm est l'historique des transactions de mise à jour. Il permet en effet de savoir si une instance peut être rejetée tout en respectant les contraintes (m,k)-firm. Pour cela, le *déclencheur de transactions* reporte la réussite (ou non) des transactions de mise à jour au contrôleur (m,k)-firm.

Pour maintenir un historique qui reflète de manière fidèle le déroulement des transactions de mise à jour, il est nécessaire de considérer que les instances écartées par le contrôleur de précision sont des instances ayant respecté leur échéance. En effet, les opérations du *contrôleur de précision* ne doivent pas influencer le fonctionnement du contrôleur (m,k)-firm. Du point de vue du contrôleur (m,k)-firm, la transaction s'est exécutée correctement et elle a réécrit la même valeur dans la base de données.

La figure 2 illustre le modèle du SGBD temps réel basé sur la rétroaction et tenant compte des contraintes (m,k)-firm des transactions de mise à jour. Le contrôleur (m,k)-firm permet de gérer l'admission des transactions de mise à jour en fonction de la charge du système et de l'historique des transactions. Il permet ainsi de résoudre les problèmes posés durant les phases d'instabilité du système en permettant à plus ou moins de transactions de mise à jour de s'exécuter en exploitant des renseignements fournis par la boucle de rétroaction.

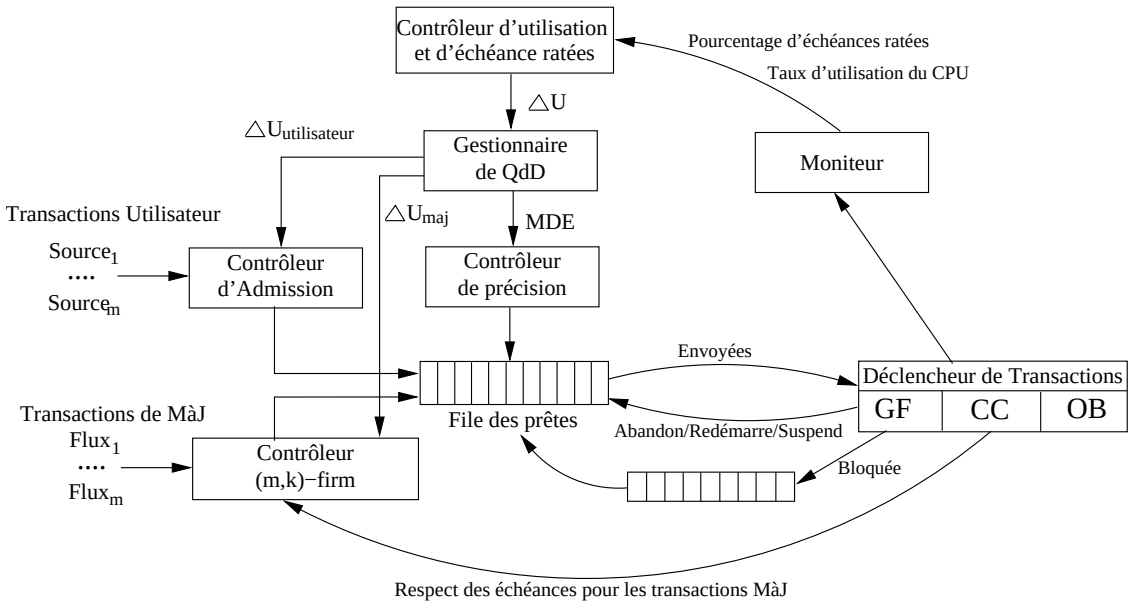

Figure 2. *Le contrôleur (m,k)-firm et l'architecture FCSA*

6. Simulations et résultats

6.1. *Description des modules du simulateur et du principe de la simulation*

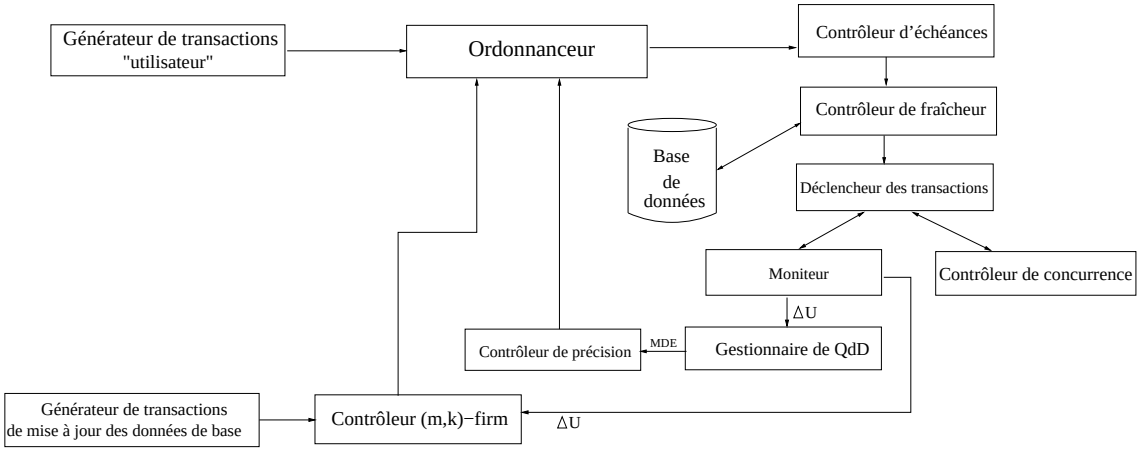

Figure 3. *Architecture globale du simulateur (m,k)-firm-FCSA* .

Nous avons développé un simulateur basé sur l'architecture (m,k)-firm-FCSA présentée dans la section 5 (cf. Figure 2). Le modèle du simulateur est illustré par la figure 3. Il simule le fonctionnement d'un SGBD temps réel.

Le contrôle d'admission des transactions de mise à jour sous contraintes (m,k)-firm nécessite d'utiliser le ratio m/k comme paramètre pour la spécification des transactions de mise à jour. Dans notre simulateur, la valeur du ratio m/k est comprise entre 0 et 1 et k = 10.

Dans ce modèle, une fois générée, la transaction va subir plusieurs tests, effectués par les composants du simulateur. Le simulateur est composé des modules suivants :

1) Générateur de transactions : il est composé lui-même de deux parties :

- Générateur de transactions *utilisateur* : c'est l'élément responsable de la génération des transactions *utilisateur* selon une distribution aléatoire. Par la suite, ces transactions sont placées dans la file de l'ordonnanceur.

- Générateur de transactions de mise à jour : génère les transactions selon leur périodicité. Les transactions ainsi générées sont envoyées au contrôleur (m,k)-firm.

2) Contrôleur (m,k)-firm : le rôle de ce contrôleur est de gérer l'admission des transactions de mise à jour générées en fonction de la charge du système (taux de manquement des échéances > taux spécifié dans les paramètres de référence de la QdS) et de l'historique des transactions provenant du déclencheur.

3) Contrôleur de précision : permet d'écarter les transactions de mise à jour admises par le contrôleur (m,k)-firm lorsque les données à mettre à jour sont suffisamment représentatives du monde réel, en fonction de la valeur de MDE fournie par le gestionnaire de QdD. Cela veut dire qu'on peut écarter des transactions si la valeur à mettre à jour est *similaire* à celle qui se trouve dans la base. On considère que deux données sont similaires lorsque la différence entre la valeur contenue dans la base et la nouvelle valeur est inférieure à la valeur du paramètre MDE.

4) Gestionnaire de QdD : en fonction de la charge du système (reportée par le moniteur), il réajuste la valeur de MDE.

5) Ordonnanceur : il reçoit les transactions fournies par le générateur de transactions *utilisateur* et le contrôleur (m,k)-firm des transactions de mise à jour. Il les ordonnance en fonction de leurs priorités.

6) Contrôleur d'échéances : il vérifie que les transactions vont pouvoir être exécutées avant leur échéance sinon elles sont immédiatement écartées (abandonnées).

7) Contrôleur de fraîcheur : il vérifie la fraîcheur des données qui vont être accédées par la transaction. Si les données sont fraîches, alors la transaction peut être exécutée et elle est envoyée au déclencheur de transactions. Sinon, elle est mise en attente dans une file des transactions bloquées. Elle est ensuite réinsérée dans la file des transactions prêtes à être exécutées dès que la mise à jour des données est effectuée.

8) Déclencheur : il permet l'exécution des transactions. En cas de conflit, il fait appel au contrôleur de concurrence.

9) Contrôleur de concurrence : c'est le composant responsable de la résolution des conflits d'accès aux données selon le protocole 2PL-HP.

10) Base de données temps réel : elle est construite par un générateur de données qui génère aléatoirement les données de façon à ce qu'il n'y ait pas d'information double. La cohérence dans la base est assurée par les transactions de mise à jour. Cela permet de simuler un mouvement de mise à jour des données comme dans les bases de données réelles. Une fois fixé pour l'expérience, le paramètre "taille" ,qui exprime le nombre de données de la base, reste inchangé.

11) Moniteur : il scrute l'état du système (charge et degré de concurrence) et transmet les informations fournies au gestionnaire de QdD et au contrôleur (m,k)-firm.

6.2. *Résultats et commentaires*

6.2.1. *Expérience 1 : Résultats des simulations pour les transactions de mise à jour*

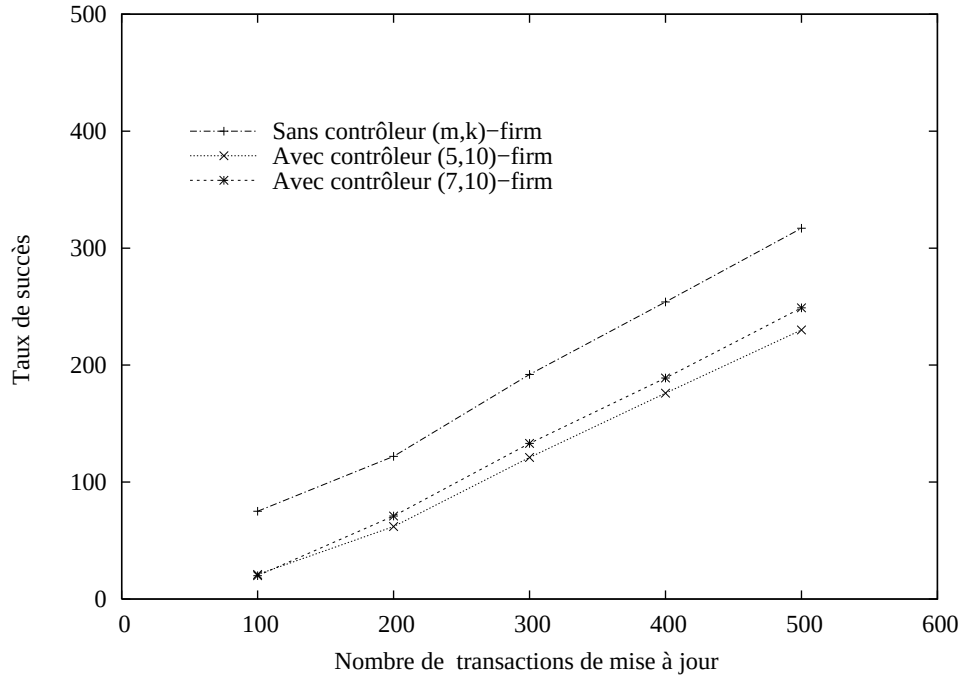

Figure 4. *Résultats des simulations pour les transactions de mise à jour.*

Comme montré par la figure 4, le nombre de transactions de mise à jour qui respectent leurs échéances en utilisant l'architecture FCSA de base est plus important que celui offert en utilisant l'architecture (m,k)-firm-FCSA. Cela s'explique par le fait que dans l'architecture FCSA, les transactions générées sont écartées uniquement par le contrôleur de précision alors que dans l'architecture (m,k)-firm-FCSA, elles sont écartées non seulement par le contrôleur de précision mais aussi par le contrôleur (m,k)-firm.

La figure 4 montre aussi que le contrôleur (5,10)-firm écarte plus de transactions de mise à jour par rapport au contrôleur (7,10)-firm.

6.2.2. *Expérience 2 : Résultats des simulations pour les transactions utilisateur*

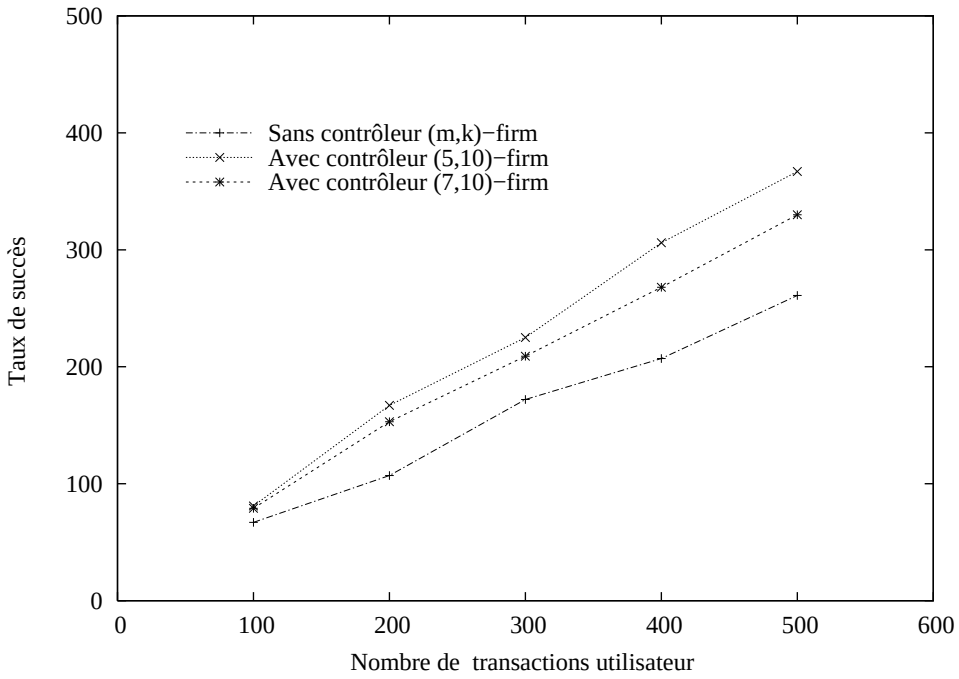

Figure 5. *Résultats des simulations pour les transactions utilisateur.*

La figure 5 montre que le nombre de transactions *utilisateur* réussies observé croît avec la diminution de la valeur du ratio m/k. En effet, l'utilisation de l'architecture (m,k)-firm-FCSA avec un contrôleur (5,10)-firm offre un taux de réussite des transactions nettement plus élevé par rapport à celui fourni par l'utilisation d'un contrôleur (7,10)-firm. L'architecture FCSA de base, fournit les performances les moins bonnes. En effet, les transactions *utilisateur* ont plus de chance de respecter leur échéance puisque le contrôleur (m,k)-firm permet d'écarter des transactions de mise à jour en fonction de la charge du système, ce qui entraîne la diminution du nombre de conflits.

7. Conclusion et perspectives

Dans cet article, nous avons mis en œuvre un contrôleur (m,k)-firm pour gérer l'admission des transactions de mise à jour dans une architecture de contrôle par rétroaction. Les simulations ont montré l'importance du contrôleur (m,k)-firm pour les transactions de mise à jour. En effet, une fois les transactions admises par ce contrôleur, elles ont de grandes chances de se terminer correctement. Le contrôleur (m,k)-firm a, d'une part, éliminé les transactions de mise à jour qui n'auraient pas respecté leur échéance quoiqu'il arrive, et d'autre part, filtré les transactions en fonction de la charge du système. Autrement dit, une transaction admise dans le système s'exécute dans de "bonnes conditions". Quand aux transactions *utilisateur*, le fait d'écarter quelques transactions de mise à jour augmente les chances des transactions *utilisateur* d'être exécutées avant leur échéance. On peut conclure que l'architecture (m,k)-firm-FCSA permet de diminuer les phases d'instabilité du système et d'en rétablir rapidement la stabilité. Par conséquent, elle offre une meilleure QdS.

Dans cet article, nous n'avons pas tenu compte de la distribution de l'architecture d'ordonnancement avec rétroaction. Nous nous sommes concentrés sur la QdS dans le cas d'un environnement centralisé. Dans nos travaux futurs, nous continuerons d'étudier les possibilités d'appliquer ces travaux aux SGBD distribués temps réel. En effet, la présence de plusieurs sites dans un système distribué pose des problèmes qui ne sont pas présents dans les systèmes centralisés et la performance des systèmes distribués dépend de la distribution de la charge de travail entre les sites.

8. Bibliographie

[ABB 88] ABBOTT R. K., GARCIA-MOLINA H., « Scheduling Real-time Transactions : a Performance Evaluation », 4^{th} *International Conference on VLDB*, Morgan Kaufmann, 1988, p. 1-12.

[AMI 03a] AMIRIJOO M., HANSSON J., SON S. H., « Error-Driven QoS Management in Imprecise Real-Time Databases », *Proceedings of* 15^{th} *Euromicro Conference on Real-Time Systems (ECRTS)*, Portugal, 2003.

[AMI 03b] AMIRIJOO M., HANSSON J., SON S. H., « Algorithms for Managing QoS for Real-Time Data Services Using Imprecise Computation », 9^{th} *International Conference on Real-Time and Embedded Computing Systems and Applications*, Springer, 2003, p. 136-157.

[AMI 06] AMIRIJOO M., HANSSON J., SON S. H., « Specification and Management of QoS in Real-Time Databases Supporting Imprecise Computations », *IEEE Trans. Computers*, vol. 55, n° 3, 2006, p. 304-319.

[BER 98] BERNAT G., « Specification and Analysis of Weakly Hard Real-Time Systems », PhD thesis, Université de les Illes Balears, 1998.

[BOU 09] BOUAZIZI E., « Gestion de la qualité de Services dans les SGBD temps réel », PhD thesis, Université du Havre, 2009, Université du Havre, Thèse soutenue le 9 Avil 2009.

[BUT 97] BUTTAZZO G., *Hard Real-Time Computing Systems*, Kluwer Academic Publishers, 1997.

[DUV 99] DUVALLET C., MAMMERI Z., SADEG B., « Les SGBD temps réel », *Technique et Science Informatiques*, vol. 18, n° 5, 1999, p. 479-517.

[HAM 95] HAMDAOUI M., RAMANATHAN P., « A Dynamic Priority Assignement Technique for Streams with (m, k)-Firm Deadlines », *IEEE Trans. Computers*, vol. 44, n° 12, 1995, p. 1443-1451.

[KAN 02a] KANG K.-D., SON S. H., STANKOVIC J. A., « Service Differentiation in Real-Time Main Memory Databases », *ISORC*, 2002, p. 119-128.

[KAN 02b] KANG K.-D., SON S. H., STANKOVIC J. A., ABDELZAHER T. F., « A QoS-Sensitive Approach for Timeliness and Freshness Guarantees in Real-Time Databases », *Euromicro Conference on Real-Time Systems*, IEEE Computer Society, 19-21 June 2002, p. 203-212.

[LIU 91] LIU J. W.-S., LIN K.-J., SHIH W. K., SHI YU A. C., CHUNG J.-Y., ZHAO W., « Algorithms for Scheduling Imprecise Computations », *IEEE Computer*, vol. 24, n° 5, 1991, p. 58-68, IEEE Computer Society.

[LIU 94] LIU J., SHIH W.-K., K.-J. LIN R. BETTATI J.-Y. C., « Imprecise Computations », *Proceedings of the IEEE*, vol. 82, 1994, p. 83-94.

[LU 01] LU C., « Feedback Control Real-Time Scheduling », PhD thesis, University of Virginia, 2001.

[RAM 93] RAMAMRITHAM K., « Real-Time Databases », *Distributed and Parallel Databases*, vol. 1, n° 2, 1993, p. 199-226.

[RAM 99] RAMANATHAN P., « Overload Management in Real-Time Control Applications Using (m, k)-Firm Guarantee », *IEEE Trans. Parallel Distrib. Syst.*, vol. 10, n° 6, 1999, p. 549-559.

[RAM 04] RAMAMRITHAM K., SON S. H., DIPIPPO L. C., « Real-Time Databases and Data Services », *Real-Time Systems*, vol. 28, n° 2-3, 2004, p. 179-215.

[WAN 02] WANG Z., SONG Y., POGGI E.-M., SUN Y., « Survey of Weakly-Hard Real-Time Schedule Theory and its Application », *Distributed Computing and Applications to Business, Engineering and Science (DCABES)*, 2002, p. 429-437.