

---

# Evaluation de requêtes adaptable dans les environnements pervasifs : une approche à base de composants récursifs

**Hocine Grine – Thierry Delot – Sylvain Lecomte**

*Laboratoire LAMIH, UMR CNRS 8530  
Université de Valenciennes - Le Mont Houy,  
59313 Valenciennes cedex 9 - France  
{hocine.grine, thierry.delot, sylvain.lecomte}@univ-valenciennes.fr*

---

*RÉSUMÉ. Les environnements pervasifs imposent de nouvelles contraintes (connexion réseau, autonomie, mémoire, etc.) sur l'évaluation de requêtes. Les évaluateurs de requêtes actuels basés sur une architecture monolithique et relativement statique doivent par ailleurs s'adapter dynamiquement à ces contraintes. Nous proposons dans cet article une architecture flexible basée sur des composants Fractal. Cette architecture permet l'ajout de nouvelles fonctionnalités à l'évaluateur de requêtes et l'adaptation de celles-ci à l'environnement.*

*ABSTRACT. Pervasive environments imply new constraints (network connection, autonomy, memory, etc) on query evaluation. Current query processors based on a monolithic and relatively static architecture must dynamically adapt to these constraints. In this article, we propose a flexible component based architecture using the Fractal model. This architecture allows adding new functionalities to query processor and adapting them to the environment.*

*MOTS-CLÉS: Evaluation de requêtes adaptive, informatique pervasive, modèle à composants.*

*KEYWORDS: Adaptive query evaluation, pervasive computing, component model.*

---

## 1. Introduction

L'évolution des réseaux mobiles et des terminaux nomades a rendu l'information de plus en plus distribuée, mobile, et hétérogène. Chaque terminal stockant des informations est aujourd'hui considéré comme une source de données. L'ensemble du système peut lui être vu comme une base de données mobile et distribuée. Dans les environnements pervasifs, l'information peut ainsi être partagée entre les utilisateurs et consultée depuis n'importe quel terminal, à n'importe quel endroit.

L'évaluation de requêtes, nécessaire à la recherche de ces informations, intervient ici dans un contexte fortement distribué, mais aussi particulièrement

dynamique. Ceci est principalement dû à la mobilité des utilisateurs, des changements du système (qualité de service, type du réseau, disponibilité de ressources) ou des besoins de l'utilisateur. Un évaluateur de requêtes déployé dans un environnement pervasif doit être conscient de son contexte et capable de s'adapter à un large champ de paramètres systèmes (bande passante, localisation, énergie etc.) ou applicatifs. Pour permettre une telle adaptabilité, l'architecture de l'évaluateur de requêtes doit être aussi flexible que possible afin de modifier les techniques d'évaluation utilisées, voire d'ajouter ou retirer des fonctionnalités en fonction des changements dans l'environnement. Pour définir un tel évaluateur de requêtes, nous proposons d'utiliser le modèle de conception par composants. Il est ainsi possible de construire l'évaluateur de requêtes sous forme de compositions de composants. Les différents composants correspondant aux éléments de l'évaluateur peuvent alors être assemblés et remplacés à souhait afin de supporter les changements de contexte d'exécution tant au niveau des préférences utilisateur, des besoins de l'application ou des contraintes de l'environnement d'exécution.

La suite de cet article est organisée comme suit. La section 2 décrit les besoins d'adaptabilité d'un évaluateur de requêtes dans un environnement pervasif et les travaux connexes dans l'évaluation adaptable des requêtes. La section 3 décrit notre vision de l'adaptabilité et son mode de gestion. Un évaluateur de requêtes à base de composants est proposé dans la section 4 en montrant un exemple de reconfiguration. Enfin, en guise de conclusion, nous rappelons les points essentiels de notre travail et énumérons les travaux futurs possibles.

## **2. Contexte et problématique**

Avec le développement de la téléphonie mobile et les réseaux ad hoc, les utilisateurs ont la possibilité d'accéder à l'information quelque soit leur localisation. Comme les entités sont mobiles, leurs proximités changent dynamiquement. Par conséquent, suivant la localisation et le temps où la requête a été envoyée, l'utilisateur peut obtenir des réponses différentes. En outre, l'entité envoyant la requête ne peut pas dépendre sur un schéma global qui permet de router la requête vers la destination voulue. La seule information toujours garantie pour une entité mobile est celle qui réside localement. L'évaluation de requêtes est donc très affectée par l'ajout de la mobilité. Celle-ci introduit quatre classes de requêtes (Marsit *et al*, 2005) à savoir : des requêtes dépendantes de la localisation, des requêtes continues, de requêtes spatio-temporelles, et des requêtes dépendantes de la position et de la direction des objets.

Comme les terminaux communiquent principalement par une liaison sans fil, les techniques d'optimisation doivent tenir compte des déconnexions, de la bande passante, de la topologie du réseau et des préférences utilisateurs. La limitation de ces techniques devient évidente car elles essaient de résoudre un problème spécifique comme : les fluctuations de mémoire, la charge du serveur,

l'indisponibilité des sources de données. En outre, les optimiseurs de requêtes réalisent un des deux critères d'optimisation qui sont : la réduction du temps de réponse ou la réduction de la consommation totale de ressource. Or, dans un environnement aussi contraint, l'objectif est de minimiser les communications réseau et la consommation d'énergie afin de maximiser le temps de connexion du terminal dans le réseau.

Une des solutions possibles pour faire face à ces contraintes est de recourir à l'évaluation de requêtes adaptable. Celle-ci a été encouragée par la nature des applications Internet interrogeant des données hétérogènes et distribuées sur des réseaux à grande échelle. Initialement, l'optimiseur était incapable d'estimer précisément le coût d'une requête en raison du manque des histogrammes, des statistiques et de métadonnées des différentes sources de données. L'évaluateur de requêtes adaptable a pour objectif de changer l'exécution de l'optimiseur en réponse aux changements de l'environnement au cours de l'évaluation d'une requête. Cette adaptabilité a reçu une très grande attention ces dernières années. Des travaux comme (Li *et al.*, 2004) proposent un algorithme de jointure dans un environnement mobile et ad hoc. Le modèle de coût proposé est intéressant car il tient compte des coûts de communication mais il est codé directement dans le middleware. Dans (Perich *et al.*, 2001), les auteurs proposent un Framework pour un évaluateur de requêtes dans un environnement ad hoc. Celui-là ne traite que de simples requêtes, et ne prend pas les problèmes d'optimisation en considération. D'autres évaluateurs de requêtes utilisent des opérateurs adaptables comme le XJoin (Urhan *et al.*, 1999) pour s'adapter aux flux de données. Eddie (Avnur *et al.*, 2000), décrit un opérateur qui s'adapte dynamiquement aux flux de données. Pourtant, dans un environnement peu dynamique cette approche est moins efficace que les approches classiques. Les requêtes parachutes (Bonnet *et al.*, 1998) peuvent, quant à elles, être utilisées pour faire face aux indisponibilités des sources de données, mais imposent un schéma global des sources de données. Une approche pour le développement d'évaluateurs de requêtes adaptables est proposée par (Vu *et al.*, 2004). La solution proposée décentralise le contrôle et les modifications des plans d'exécution et améliore le coût du traitement local d'une requête en tenant compte des changements des ressources (CPU, E/S et mémoire). Cependant, cette solution ne considère pas les contraintes imposées par les ressources réseau et l'énergie des terminaux.

L'adaptabilité décrite précédemment est généralement codée directement dans l'application (modèle de données, opérateurs, stratégie d'optimisation, etc.) et il est difficile d'ajouter de nouvelles fonctionnalités ou de modifier celles qui existaient avant sans manipuler le code source de l'évaluateur de requêtes.

### **3. Adaptabilité de l'évaluateur de requêtes**

L'adaptabilité d'un évaluateur de requêtes est définie par sa capacité de modifier son mode de fonctionnement en fonction des changements de l'environnement

d'exécution (Hellerstain *et al.*, 2000). Nous considérons que cette adaptabilité se définit aussi par sa capacité à changer la structure interne de l'évaluateur afin d'offrir plusieurs personnalités (Hérault *et al.*, 2002) fonctionnant dans des environnements hétérogènes. Le concept d'une personnalité représente une version d'un évaluateur de requêtes et signifie qu'un évaluateur peut être rendu de plusieurs façons différentes. Par exemple, un évaluateur de requêtes dépendantes de la localisation et un évaluateur de requêtes continues représentent deux personnalités d'un évaluateur de requêtes.

Pour réaliser l'adaptation, nous proposons deux niveaux d'adaptabilité : une adaptabilité à grains fins et une adaptabilité à gros grains. L'adaptabilité à grains fins concerne les opérateurs de requêtes, le choix des algorithmes correspondants, ainsi que les stratégies d'évaluation. Ces stratégies définissent l'ordonnement des différentes opérations nécessaires à l'évaluation d'une requête. Par exemple, un terminal peut utiliser une stratégie pour rediriger toutes ses requêtes vers d'autres terminaux afin d'économiser ses ressources. L'adaptabilité à gros grains concerne plus la reconfiguration de l'évaluateur de requêtes. Cette reconfiguration permet de modifier la personnalité de l'évaluateur de requêtes suivant les changements du contexte et offre une nouvelle personnalité à l'évaluateur de requêtes.

Afin de réaliser l'adaptation de l'évaluateur de requêtes, une définition des différents facteurs affectant l'évaluation d'une requête (c'est-à-dire son contexte) est nécessaire. Ces facteurs représentent l'ensemble des caractéristiques de l'environnement d'exécution, de l'application et de l'utilisateur et sont présentés dans la section suivante.

### **3.2. Description du contexte**

Notre définition du contexte repose sur les travaux menés au sein du projet MOSAIQUES, où le contexte est représenté par trois parties appelées *ContextPart* (Caron *et al.*, 2006) : *EnvironmentCtxt*, *ApplicationCtxt*, *UserPreferences*. L'*EnvironmentCtxt* regroupe les caractéristiques de l'environnement d'exécution et correspond aux capacités matérielles du terminal comme la mémoire, la charge du CPU. L'*ApplicationCtxt* regroupe les besoins de l'application en termes d'évaluation de requêtes comme le type de requêtes à supporter par exemple : requêtes dépendantes de la localisation, requêtes relatives à la localisation ou requêtes continues. Les préférences utilisateur *UserPreferences* regroupent les contraintes imposées par l'utilisateur sur l'évaluation de requêtes comme le temps maximal alloué à la recherche du résultat, le coût financier (lié par exemple aux réseaux de téléphonie mobile) ou le type du résultat (partiel ou complet).

#### **3.1. Le noyau de l'évaluateur de requêtes**

Nous proposons l'utilisation du modèle à composants pour développer des évaluateurs de requêtes. Ces derniers, possèdent les mêmes caractéristiques d'un composant applicatif. L'intérêt d'utiliser ce modèle est la réutilisabilité du code ainsi

que la modularité. Cette modularité permet, de concevoir des évaluateurs dynamiquement reconfigurables pour s'adapter au mieux aux évolutions de leur environnement d'exécution et à l'ajout dynamique de nouvelles fonctionnalités par ajout/retrait ou remplacement de composants. L'idée de notre solution est de démarrer d'une configuration minimale appelée « noyau de l'évaluateur de requêtes » puis l'étendre vers une configuration plus complexe. Le noyau de l'évaluateur de requêtes est un composant décomposé en plusieurs composants élémentaires offrant les fonctionnalités de base : l'analyse, l'optimisation, l'exécution de la requête et la composition des résultats. Pour mettre à niveau l'évaluateur de requêtes, la structure du noyau reste inchangée. Par contre, d'autres composants peuvent être ajoutés pour répondre à un besoin donné. A titre d'exemple, considérons une application utilisant un évaluateur de requêtes classiques. Si l'utilisateur a besoin de localiser « le restaurant Italien le plus proche », un composant assurant l'évaluation de requêtes dépendantes de la localisation est nécessaire. L'évaluateur de requêtes résultant utilisera un composant de localisation géographique pour évaluer la position de l'utilisateur. Un composant permettant l'évaluation de requêtes continues est également nécessaire dans le cas où l'utilisateur est mobile et souhaite recevoir « la liste des prochaines stations service » au fur et à mesure de son déplacement.

### **3.3. La gestion de l'adaptabilité**

Pour développer un évaluateur de requêtes adaptable, nous distinguons mécanisme d'adaptation et politiques d'adaptation (Efstratiou *et al.*, 2001). L'adaptation est donc à la charge d'un module nommé « gestionnaire d'adaptation » (Grine *et al.*, 2006), qui, suivant les changements dans l'environnement, exécute la politique d'adaptation adéquate. Ce module est composé de six sous-modules : *Un gestionnaire de profil* responsable de la collecte des informations sur les préférences utilisateur et ses priorités. *Un moniteur* a pour fonction de surveiller l'environnement d'exécution. *Un gestionnaire de contexte* a pour rôle de collecter, filtrer les informations sur le profil de l'utilisateur et sur l'environnement d'exécution de manière à attribuer pour chaque élément du contexte une valeur. *Un gestionnaire de règles* reçoit les événements et exécute les règles spécifiées suivant le paradigme ECA (Événement-Condition-Action). *Un service de courtage* gère les multiples personnalités de l'évaluateur de requêtes adaptées au contexte en se basant sur le principe du courtage sémantique. *Un moteur d'adaptation* est responsable de la prise des décisions pour changer le comportement de l'évaluateur de requêtes. L'adaptation de l'évaluateur de requêtes se fait en mettant à jour sa personnalité. Pour ce faire, la mise à jour doit respecter les propriétés transactionnelles ACID. Le processus d'adaptation se déclenche après la détection d'un changement de contexte. Une liste d'actions est envoyée au gestionnaire d'adaptation qui sera classée par ordre de priorité. L'action se traduit soit par un changement de stratégie d'évaluation soit par une reconfiguration de l'évaluateur de requêtes. En cas d'échec de l'action, une action *rollback* est exécutée afin de défaire l'action précédemment sélectionnée.

### 3.4. Le contrôle de l'adaptabilité

L'adaptabilité de l'évaluateur de requêtes est contrôlée par des politiques d'adaptation. Une politique d'adaptation se compose d'un trigger pour la règle, qui se déclenche lorsqu'un événement est reçu, d'une condition qui doit être vérifiée, et une action à exécuter. L'utilisation d'une politique d'adaptation permet de spécifier, d'une manière déclarative, la manière d'adapter l'évaluateur de requêtes aux changements du contexte d'exécution. La partie « événements » d'une politique décrit la circonstance qui déclenche cette règle particulière. L'événement peut être un changement du contexte d'exécution, détecté par le gestionnaire de contexte, ou un événement se produisant à l'intérieur de l'application elle-même. Nous avons identifié cinq types d'événements relatifs : à la charge du CPU, à la mémoire libre, à l'état du réseau, à l'état de la batterie, et à la mobilité. Chaque événement possède un poids, qui est un nombre entier compris entre 0 et 10. Ce poids définit le degré d'importance de chaque événement. Les événements choisis sont principalement liés au réseau, à la batterie, à la mémoire, à la mobilité et à la charge du CPU. Une valeur par défaut de « 5 » est attribuée au poids de chaque événement. Ces poids peuvent être modifiés lorsque l'utilisateur ajoute plus de précision sur le comportement de l'évaluateur de requêtes. Par exemple, l'utilisateur peut choisir entre deux modes d'exécution : meilleures performances, et meilleures économies d'énergie. Dans le premier cas, les poids maximums sont attribués au CPU et à la mémoire, tandis que dans le second cas, les poids maximums sont attribués à la batterie et au réseau.

```
<?xml version="1.0" encoding="UTF-8"?>
<policy name="disable-caching">
  <rule>
    <event name="memory-change" weight="6"/>
    <condition context="memory-free" op="LT" value="10Mb"/>
    <action>
      <fscript>
        dispatcher:=$handler/child::request-dispatcher;
        if(name($dispatcher/#handler/component:*)"=cache')
        then{
          unbind($dispatcher/#handler);
          request-handler:=$handler/child::request-handler;
          bind($dispatcher/#handler,request-handler/interface::request-handler);
        }
      </fscript>
      <java>
        simple.rules.actions.ForwardStrategy1
      </java>
    </action>
  </rule>
</policy>
```

**Figure 1.** Un exemple de politique d'adaptation

La partie condition est une expression booléenne qui est employée en tant que garde. La partie action est une classe Java à exécuter et/ou un programme FScript (David *et al.*, 2006). FScript est un langage de script utilisé pour programmer les reconfigurations des composants Fractal permettant l'adaptation de l'évaluateur de

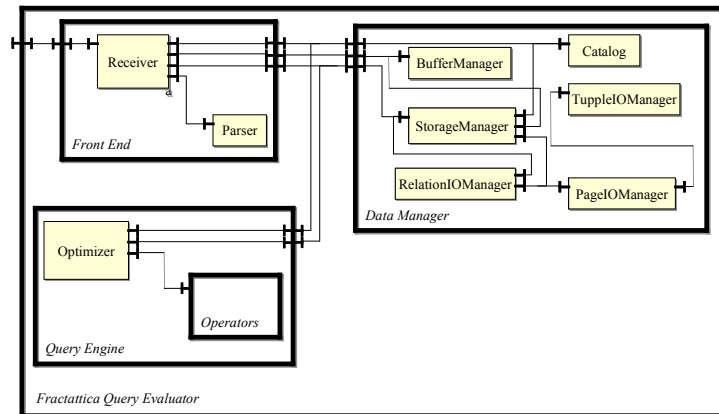
requêtes. FScript a été choisi comme langage de reconfiguration, car il offre certaines « garanties » sur les changements appliqués au composant cible, notamment l'atomicité de la reconfiguration.

La figure 1 décrit une politique, basée sur XML, qui désactive le cache quand un événement de changement de mémoire est détecté. Si la mémoire a atteint un niveau bas inférieur à 10 Mo, une action est déclenchée. La partie action de la politique comporte deux parties : un programme FScript, et une classe Java. La partie FScript de la politique décrit comment l'architecture de l'évaluateur de requêtes est reconfigurée en déconnectant le composant cache. FPath est utilisé pour naviguer dans l'architecture Fractal et sélectionner les éléments correspondant au critère. La partie Java permet le lancement de la redirection des requêtes en utilisant par exemple une stratégie de diffusion adaptée comme celles proposées par (Thilliez *et al.*, 2005) pour évaluer des requêtes dépendantes de la localisation.

#### 4. L'évaluateur de requêtes sous forme de composants Fractal

Nous avons choisi Fractal comme modèle à composants pour implémenter l'évaluateur de requêtes adaptable. Fractal est un modèle à composants hiérarchique, réflexif qui permet le partage de composants (Bruneton, 2006). Ce modèle permet de développer et de maintenir des systèmes logiciels complexes. A l'aide de ce modèle, un service ou une application peuvent, en outre, être vus comme une composition de composants. Cette composition n'est pas figée mais peut être modifiée de façon dynamique (ajout ou retrait de composant(s), remplacement d'un ou plusieurs composants).

La figure 2 montre l'architecture du noyau de l'évaluateur de requêtes sous forme d'un composant composite Fractal. Le noyau est constitué de trois principaux sous-composants : un composant « FrontEnd » qui représente la partie réception et analyse des requêtes. Le composant « DataManager » représente la partie gestion de données avec six sous-composants : le composant « BufferManager » récupère des pages depuis le disque vers la mémoire principale. Ce composant est paramétrable, en précisant la taille du buffer ainsi que la politique de remplacement. Le composant « Catalog » décrit les données stockées dans le système. Le composant « PageIOManager » écrit/lit des pages à partir du disque. Les relations sont manipulées par le composant « RelationIOManager » et les tuples sont gérés par le composant « TupleIOManager ». Le composant « StorageManager » est la couche la plus basse d'attica qui gère l'espace sur disque, où les données sont stockées. Le composant « Query Engine » représente le moteur d'exécution avec un sous-composant « Optimizer », qui construit un plan d'évaluation d'une requête à partir d'une collection d'opérateurs algébriques. Le composant composite « Operators » regroupe une liste d'opérateurs de requêtes.

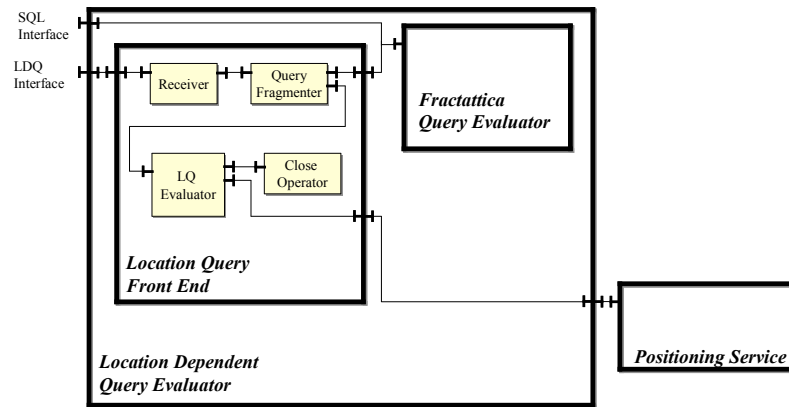


**Figure 2.** L'architecture du noyau de l'évaluateur de requêtes

Nous avons implémenté le noyau de l'évaluateur en nous basant sur un système de gestion de base de données ouvert appelé « Attica » (Attica, 2004). Nous avons donc fait un découpage en composants Fractal reconfigurables d'Attica de manière à respecter la spécification que nous avons définie. Fractattica est donc le nom du noyau de l'évaluateur de requêtes basé sur Attica. Il est développé en utilisant l'implémentation Java du modèle à composants Fractal appelée Julia. Il existe d'autres évaluateurs de requêtes ouverts implémentés en Java comme Hypersonic SQL (Hypersonic, 2006), Mckoi SQL (Mckoi SQL, 2004). Cependant, par raison de simplicité, nous avons fait le choix d'Attica comme noyau d'évaluateur de requêtes. En effet, il répond aux besoins que nous avons précédemment prédéfinis notamment : il est implémenté en Java, il offre la fonctionnalité de base pour l'évaluation de requêtes (l'analyse, l'optimisation, et l'exécution d'une requête et le renvoi du résultat), il offre plusieurs types de jointures comme le Merge Join et le Nested Loop Join, il offre également la possibilité de distinguer les trois principaux composants précédemment définis.

#### 4.2. La reconfiguration de l'évaluateur de requêtes

Afin de proposer une version particulière d'un évaluateur de requêtes, un nouvel assemblage de composants est défini. L'idée est de réutiliser le code du noyau et d'ajouter un autre composant permettant l'évaluation de requêtes dépendantes de la localisation. Un exemple d'une requête dépendante de la localisation est : « Quelle est la station de métro la plus proche de moi? ». La requête peut être divisée en deux parties : une requête simple évaluant les stations de métro et leur position, et une jointure entre le résultat obtenu et la position de l'utilisateur.



**Figure 3.** L'architecture d'un évaluateur de requêtes dépendantes de la localisation

La figure 3 montre l'architecture d'un évaluateur de requêtes dépendantes de la localisation en réutilisant le noyau défini précédemment. Le modèle Fractal nous a permis d'étendre Fractattica en lui ajoutant un autre composant « Location Query Front End » pour obtenir deux personnalités : l'un permet l'exécution des requêtes SQL, et l'autre permet l'exécution des requêtes dépendantes de la localisation. Le composant « Location Query Front End » est constitué de quatre sous-composants : un composant « Receiver » qui reçoit la requête LDQ, un composant « Query Fragmenter » qui décompose la requête reçue en une requête simple et extrait l'opérateur de localisation, et un composant « Close Operator » qui représente l'opérateur de la localisation. On note que ce composant est paramétrable, c'est-à-dire qu'on peut lui attribuer une distance comme par exemple 50 mètres. Le composant « LQ Evaluator » récupère la localisation de l'utilisateur à partir du composant « Positioning Service » et exécute une jointure avec les résultats obtenus à partir du noyau de l'évaluateur de requêtes.

## 5. Conclusion

Dans cet article, une architecture flexible d'évaluateur de requêtes adaptable basée sur le modèle à composants Fractal a été présentée. Celle-ci permet la reconfiguration dynamique de l'évaluateur de requêtes en se basant sur la notion du noyau et des différentes personnalités. L'adaptabilité est gérée par un gestionnaire d'adaptation qui détecte les événements et déclenche l'adaptation en utilisant des politiques d'adaptation. Le noyau de l'évaluateur de requêtes est actuellement opérationnel. Des travaux sont en cours pour développer de nouvelles personnalités et permettre l'évaluation de différents types de requêtes comme des requêtes dépendantes de la localisation et des requêtes continues.

Lors de la définition des politiques d'adaptation, nous avons constaté que l'occurrence d'un événement peut déclencher plusieurs actions qui peuvent résulter des conflits. Ces conflits peuvent être des conflits statiques (au moment de la conception) et des conflits dynamiques (au moment de leur exécution). De ce fait, un système de gestion des conflits entre les politiques d'adaptation est nécessaire. L'utilisation de FScript nous a permis une reconfiguration de l'évaluateur de requêtes simple et dynamique. Toutefois, cette reconfiguration est perdue après le redémarrage de l'application. Un système permettant de rendre la reconfiguration persistante nous paraît nécessaire.

## 6. Remerciements

Ce travail est effectué dans le cadre du projet MOSAIQUES financé par le FEDER et la région «Nord Pas-de-Calais »

## 7. Références

- Attica database system, <http://www.inf.ed.ac.uk/teaching/courses/adbs/attica/>, 2004.
- Avnur R., Hellerstein J. M. "Eddies: Continuously Adaptive Query Processing.", *SIGMOD Rec.*, 29(2):261–272, 2000.
- Babu S., Bizarro P., "Adaptive Query Processing In The Looking Glass.", *CIDR*, p. 238–249, 2005.
- Bonnet P., Tomasic A., Parachute Queries In The Presence Of Unavailable Data Sources. Technical Report RR-3429, INRIA Rocquencourt, France, 1998.
- Bruneton E., Coupaye T., Leclercq M., Quéma V., Stefani J.-B. "The Fractal Component Model and Its Support in Java.", *Software Practice and Experience*, special issue on Experiences with Auto-adaptive and Reconfigurable Systems. 36(11-12), 2006.
- Caron O., Carré B., Gransart C., Le Pallec X., Lecomte S., Marvie R., Nebut M., Seriai A., Vanwormhoudt G., "Propositions pour la modélisation d'applications ubiquitaires.", *Actes de les 3e Journées Francophones Mobilité et Ubiquité*, Paris, septembre 2006.
- David P.-C., Ledoux T., "Safe Dynamic Reconfigurations of Fractal Architectures with FScript.", in: *Proceeding of Fractal CBSE Workshop, ECOOP'06*, Nantes, France, July 2006
- Efstratiou C., Cheverst K., Davies N., Friday A. "An Architecture for the Effective Support of Adaptive Context-Aware Applications", *Proceedings of 2nd International Conference in Mobile Data Management (MDM'01)*, Springer, Lecture Notes in Computer Science Volume 1987, p.15-26, 2001.
- Grine H., Delot T., Lecomte S., "Un évaluateur de requêtes adaptable pour les environnements pervasifs". *3èmes Journées Francophones: Mobilité et Ubiquité 2006, ACM International Conference Proceeding Series*, Paris, septembre.

- Hellerstein J. M., Franklin M. J., Chandrasekaran S., Deshpande A., Hildrum K., Madden S., Raman V., Shah M. A. "Adaptive Query Processing: Technology in Evolution.", *IEEE Data Engineering Bulletin*, 23(2):7-18, 2000.
- Hérault C., Bennani N., Delot T., Lecomte S., Thilliez M., "Adaptability of Non-Functional Services for Component Model, Application to the M-Commerce.", *IEEE International Symposium on Advanced Distributed Computing (ISADS)*, Guadalajara, Jalisco, Mexico, p. 93–104, 2002.
- Hypersonic Lightweight 100% Java SQL Database Engine, <http://hsqldb.org/> , 2006
- Li J., Li. J., "Query Processing in Mobile Ad Hoc Wireless Networks", *The Fourth International Conference on Computer and Information Technology (CIT'04)*, p. 633-638, Washington, DC, USA , 2004.
- Thilliez M., Delot T., Lecomte S., "An Original Positioning Solution to Evaluate Location-Dependent Queries in Wireless Environments", *Special Issue on Distributed Data Management in Journal of Digital Information Management*. ISSN 0972-7272, Volume 3, Issue 2, Juin 2005.
- Marsit N., Hameurlain A., Mammeri Z., Morvan F., "Query Processing in Mobile Environments: A Survey and Open Problems.", *DFMA*, p.150-15, 2005.
- Mckoi SQL Database, <http://www.mckoi.com/database/> , 2004.
- Perich F., Avancha S., Josh A., Yesha Y., Joshi K., Query Routing And Processing In Mobile Ad-Hoc Environments. Technical Report, UMBC, November 2001.
- Urhan T., Franklin M. J., "XJoin: Getting Fast Answers From Slow And Bursty Networks.", Technical Report CS-TR-3994, University of Maryland, February 1999.
- Vu T-T., Collet Ch.. "Adaptable Query Evaluation using QBF.", Proc. Of the IDEAS, July 2004.