

Objets autonomes adaptables

Jean Louis Sourrouille, José Lino Contreras

INSA Lyon, L3i, Bat. B. Pascal,
F69621 Villeurbanne Cedex
E-mail : sou@if.insa-lyon.fr

RÉSUMÉ. Ce document présente le projet ARTO (*Adaptable Real-Time Objects*) dont l'objectif est de réaliser des systèmes capables de modifier dynamiquement leur comportement en fonction des changements intervenant dans leur contexte d'exécution. Ces adaptations sont guidées par l'évaluation de la qualité de service instantanée fournie. La stratégie adoptée comporte des décisions à deux niveaux : au niveau local, chaque objet est autonome et a une entière liberté pour sa politique de décision, tandis qu'au niveau global l'efficacité et l'optimisation sont assurées par une gestion centralisée et collective du partage des ressources. En pratique, une architecture à base de méta-objets a été adoptée pour séparer les aspects fonctionnels des aspects non fonctionnels qui pilotent l'adaptation et sont, par essence, plus volatiles car dépendants du contexte. Le prototype réalisé est brièvement décrit.

MOTS-CLÉS : QoS, adaptation, Objets actifs, temps-réel mou, architecture à méta-objets

1. Introduction

Dans un système fermé, un programme s'exécute dans un environnement prédictible, et en suivant la stratégie prédéfinie atteint avec une probabilité raisonnablement connue les objectifs fixés. A part quelques cas particuliers, ce monde fermé ne peut plus être considéré comme une approximation réaliste du monde réel. Les systèmes s'ouvrent vers l'extérieur et communiquent, devenant ainsi interdépendants et donc d'une certaine manière imprédictibles. Le projet présenté ARTO (*Adaptable Real-Time Object*) se situe dans ce contexte ordinaire d'un système qui reçoit à des moments inconnus des requêtes pour l'exécution de services. Le but de ce projet est de définir un modèle d'objet autonome capable de s'adapter à des changements de contexte dans le domaine particulier du temps-réel "mou". Le mot *mou* signifie que certaines contraintes de temps pourront ne pas être respectées, par opposition au temps-réel *dur* dans lequel les contraintes de temps doivent être impérativement respectées. Ce projet s'intéresse cependant à toute la gamme des contraintes, des plus larges aux plus strictes, mais ces aspects ne seront pas détaillés. Dans une première étape, pratiquement terminée aujourd'hui, le projet s'est limité à l'exécution sur un seul processeur. La deuxième étape étendra le projet aux

2 Objets autonomes adaptables

nombreuses possibilités supplémentaires apportées par une exécution distribuée, certaines ayant déjà été partiellement étudiées.

Depuis les "gluons" qui permettent de connecter dynamiquement des objets [May95] aux composants qui s'adaptent à leur contexte en passant par les couches intermédiaires (*middleware*) de gestion de la qualité de service [Abd99], il existe de très nombreuses visions de l'adaptation. Cette multiplicité n'exclut pas une communauté de mécanismes. La vision de l'aspect "adaptable" du projet est d'abord explicitée en précisant le but de l'adaptation et la stratégie pour atteindre ce but. La section suivante décrit les différents mécanismes d'adaptation en parallèle avec le modèle d'objet actif proposé. Ensuite le mode de description des informations est abordé, puis le prototype réalisé et quelques résultats sont présentés.

2. Adaptation : principes

Tout système est a priori supposé faire au mieux (*best effort*) pour satisfaire les besoins exprimés. Dans l'optique de ce projet, l'adaptation commence au-delà, lorsque tous les besoins ne peuvent être satisfaits et qu'il faut prendre des décisions pour maintenir la satisfaction globale au plus haut niveau possible, ou autrement dit lorsqu'il faut maximiser la satisfaction globale sur des critères variables. Le mot adaptation signifie qu'une partie du système modifie son comportement en fonction du contexte pour atteindre un nouveau point de fonctionnement plus "satisfaisant". Cette satisfaction est mesurée par des critères regroupés sous le terme de QoS¹. L'adaptation a donc pour objectif d'augmenter la QoS globale du système.

Pour mettre en œuvre l'adaptation, il faut dès l'origine prendre une décision majeure d'architecture. Avec une gestion *non intrusive*, les adaptations sont pilotées par l'environnement de l'application, avec deux voies principales :

- Le système d'exploitation, y compris les couches communication, est maître des adaptations. D'une certaine manière, tout système d'exploitation gère déjà la QoS, par exemple en répartissant le temps processeur, donc il faut seulement lui ajouter des fonctions supplémentaires. L'avantage essentiel de cette solution est que toutes les applications s'exécutant dans le même cadre (système d'exploitation, machine) bénéficient des mêmes services puisqu'elles ne sont pas modifiées. Les besoins de QoS sont décrits de façon séparée, typiquement dans un fichier accompagnant toute application.
- Une couche intermédiaire (*middleware*) entre application et système gère la QoS. Dans ce cas il faut une relation privilégiée avec le système d'exploitation pour gérer les ressources de façon prioritaire. L'avantage supplémentaire de cette solution par rapport à la précédente est qu'elle peut être portée sur plusieurs systèmes d'exploitation avec un moindre effort.

Dans le cas d'une gestion *intrusive*, l'application participe elle-même aux adaptations. L'inconvénient évident est que cela ne concerne que les applications

¹ Ensemble de qualités attaché au comportement collectif de un ou plusieurs objets [ISO97]

développées spécifiquement à cet effet. En contrepartie, l'application seule a la connaissance nécessaire pour gérer au mieux ses ressources et se dégrader de façon optimale : une application peut passer de 25 à 12 images par seconde pour diminuer sa consommation de ressources tout en conservant un service acceptable, alors qu'un système d'exploitation qui diviserait par deux les ressources de l'application obtiendrait un résultat insupportable. Dans le domaine considéré, la connaissance que les objets ont d'eux même est primordiale et indispensable pour gérer au mieux les ressources, donc une architecture intrusive a été choisie. De plus, elle répond à la préoccupation de conserver aux objets leur nature autonome et indépendante en les faisant participer aux décisions d'adaptation sans que les demandeurs de service aient à connaître la façon dont s'effectue l'adaptation. Cette solution demande comme la précédente une relation privilégiée de l'application avec le système d'exploitation.

Dans les systèmes à bases de connaissance, la stratégie pour atteindre les buts n'est pas définie a priori [Cap02]. Leur approche est séduisante, mais elle est relativement lourde (moteur d'inférence), et la longueur non prédictible des chaînes d'inférences la rend inadaptée au domaine du temps-réel. La préférence reste donc à une stratégie prédéfinie, de temps borné connu, mise en œuvre dynamiquement.

Concrètement dans ce projet, la stratégie a pour but d'optimiser une fonction qui mesure la QoS. Ce problème classique est NP-Difficile ([Lee98]), donc en pratique il ne peut pas être résolu dynamiquement et il faut utiliser une heuristique. Ce n'est pas aussi pénalisant qu'il paraît car la loi d'arrivée des événements est inconnue, donc tout optimum est éphémère et remis en cause à chaque nouvel événement. La somme de politiques partielles optimales n'est évidemment pas optimale, donc dans ce contexte la politique globale ne peut pas être optimale. Dans les systèmes temps-réel où le respect des contraintes est garanti, tous les événements et leur loi d'arrivée sont connus et une analyse statique permet de fixer une stratégie optimale.

3. Mécanismes d'adaptation

Pour qu'il y ait adaptation il faut disposer de degrés de liberté. En fonction du contexte et dans les limites du domaine défini par ces degrés de liberté, les objets prennent des décisions. Le principe fondamental des décisions dans le modèle ARTO est qu'il vaut mieux un résultat, même imprécis, que pas de résultat du tout. Les capacités d'adaptation sont obtenues à l'aide de mécanismes généraux très simples et paramétrables. L'un des objectifs était d'arriver, malgré tous les mécanismes introduits, à des performances comparables à celles des environnements du marché dans le même domaine (Rhapsody/I-Logix ou RoseRT/ Rational).

3.1. Structure générale d'une application

Une application est composée d'un ensemble d'objets actifs (Fig. 1) qui communiquent exclusivement via des messages asynchrones avec attente éventuelle de réponse. Les objets sont actifs dans la mesure où ils possèdent chacun au moins deux processus légers (*threads*). L'un d'eux, appelé *contrôleur*, contrôle le

comportement de l'objet et prend toutes les décisions *locales*. Il gère la queue des messages entrants, filtre les messages de façon à éviter les conflits d'accès, élimine les messages non exécutables dans l'état courant, choisit parmi les messages éligibles ceux qu'il faut exécuter etc. Les processus légers supplémentaires exécutent les méthodes de l'objet à la demande du contrôleur. Avoir plusieurs processus légers permet d'exécuter des demandes urgentes sans attendre la fin de l'exécution de la méthode en cours.

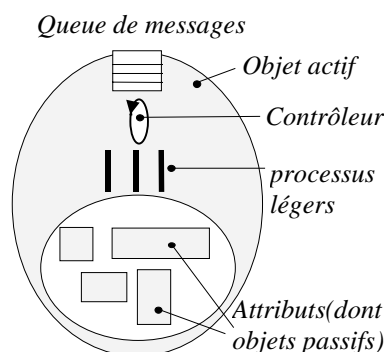


Figure 1. Objet actif

Le contrôle du comportement d'une application ne peut se résumer à une somme de décisions locales au niveau des objets car le partage optimal de ressources comme le temps processeur ou la mémoire ne peut être que centralisé. Au lieu d'introduire un objet centralisateur, ce sont tous les objets qui collectivement gèrent les aspects communs et qui dans ce cadre prennent les décisions globales. A ce niveau global tous les objets appliquent nécessairement la même stratégie, alors qu'au niveau local chacun applique la stratégie qu'il désire. En pratique, chaque application possède un objet racine qui donne accès aux données partagées par tous les objets, par exemple le planning d'exécution des tâches. A tout moment un objet peut réserver l'accès à ces données pour exécuter une procédure globale commune. Cette zone de données est gérée comme si elle était partagée entre plusieurs applications (toute référence est interdite), bien que la maquette actuelle soit mono processus.

En résumé, la gestion est intrusive au niveau de l'application mais aussi au niveau encore plus fin de l'objet. Les objets actifs sont des atomes avec chacun une parcelle d'autonomie et une responsabilité dans le changement de comportement global de l'application pour les besoins d'adaptation. C'est un choix de l'étude, et un projet parallèle est en cours avec des atomes qui sont les applications dans le cadre d'une exécution distribuée. Concernant les rapports avec le système d'exploitation, les objets gèrent complètement les processus et se substituent au gestionnaire standard.

3.2. Demande de service

3.2.1. Message

Un message est une requête pour un service et il porte des spécifications non fonctionnelles de QoS. Le serveur qui reçoit la requête décide de la manière de fournir le service. Par exemple une façon élémentaire est d'exécuter une méthode et, le cas échéant, de renvoyer une réponse. Plus généralement, pour chaque service demandé (message), l'objet dispose d'une gamme de possibilités : il peut refuser immédiatement le message, le mettre en attente, choisir parmi les méthodes dont il dispose celle qui convient le mieux dans le contexte actuel, déléguer la requête à un

autre objet, etc. Au niveau d'un objet, un message est donc associé à un ensemble de méthodes et d'actions (comme *ignorer*, *déléguer*...) :

Message \rightarrow { ..., methode_i, ..., action_j, ... }

Cette correspondance n'est pas figée, c'est à dire qu'elle peut changer en raison du contexte ou de l'historique de l'objet. Chaque méthode a des caractéristiques propres, que ce soit au niveau des besoins en ressource ou de la qualité des résultats. Si le temps manque, on exécutera une méthode rapide mais peu précise alors que si l'on dispose de temps on pourra exécuter un algorithme plus précis. Au pire, en cas d'urgence, rien ne s'exécute pour laisser du temps aux requêtes plus importantes.

Ces décisions sont prises en tenant compte de l'*importance* (QoS) de chaque message dont les valeurs sont : *basse*, *normale*, *haute*, *garantie* et *critique*. Cette importance est utilisée pour décider des méthodes à exécuter lorsqu'il n'est pas possible de toutes les exécuter, donc en cas de surcharge. Les messages critiques sont exécutés en priorité absolue au détriment de tous les autres, même garantis.

3.2.2. Négociation

La politique d'adaptation du serveur conduit à des choix auxquels le client doit être préparé. Par exemple si un serveur décide de ne pas fournir un service, il faut que le client en soit averti et soit d'accord. Cette observation conduit à la notion de contrat, qui en général est l'aboutissement de négociations. Pour limiter le coût des négociations, seuls les messages avec demande explicite de *garantie* d'exécution avant échéance sont négociés. Le serveur donne ou non son accord, et le client décide alors en connaissance de cause. Pour tous les autres messages, le client est supposé être implicitement d'accord avec la politique du serveur, dont d'ailleurs il n'observe que le résultat. Par exemple un service peut avoir été délégué à un autre objet sans qu'il en soit averti.

3.3. Mode de l'application

Le mode de l'application correspond à une situation particulière du système ou de l'application comme *Initialisation*, *Normal*, *Décollage*, *Panique*, *Arrêt*... Ce mode est en fait un état, mais le mot état est réservé aux objets actifs pour éviter toute confusion. A un moment donné, une application et donc tous les objets qu'elle contient sont dans le même mode. Chaque objet possède pour chaque mode une table de correspondance *message* \rightarrow { *méthodes/actions* }. Cette table est automatiquement commutée au changement de mode. Ainsi, un message exécuté dans le mode *Normal* pourra se voir éliminé dans le mode *Panique*. Le changement de mode est effectué par l'un quelconque des objets de l'application en fonction de critères propres à l'application, par exemple une absence de réponse (*Panique*), une anomalie (*Arrêt*) ou tout simplement un changement de phase de fonctionnement (*Décollage*). Ce mécanisme est une adaptation "dure" car sans négociation possible (seul le devenir des messages déjà dans la queue est paramétrable).

3.4. Décisions au niveau global

Au niveau global, commun à tous les objets, les degrés de liberté pour l'adaptation sont le choix des messages à exécuter et leur ordre d'exécution. Le problème posé est l'exécution d'une liste de tâches caractérisées par une date d'échéance, une importance, une précision, etc. Par commodité, on appelle tâche l'association d'un message (et son ensemble de méthodes) avec un processus léger. Les tâches sont définies au niveau local par les objets et soumises à exécution au niveau global, où elles sont toutes traitées de façon identique.

3.4.1. Politique de planification des tâches

Pour optimiser la fonction mesurant la QoS, l'heuristique choisie favorise le critère temporel. Seuls les grands principes de l'ordonnancement choisi sont présentés (voir [Con00][Con01]). L'algorithme EDF, pour *Earliest Deadline First* [Liu73], exécute en premier les tâches dont l'échéance est la plus proche. Cet algorithme très efficace est optimal en sous-charge, c'est à dire trouve un ordonnancement lorsqu'il existe. En cas de surcharge, il n'est pas déterministe et les tâches qui vont dépasser leur échéance sont inconnues. Pour résoudre en partie ce problème, la politique choisie est d'exécuter d'abord les tâches de plus haute importance (ce qui ne minimise pas le nombre de fautes temporelles).

Chaque méthode est caractérisée par une durée au pire cas (WCET : *Worst Case Execution Time*). L'ordonnancement est effectué avec le plus petit WCET des méthodes associées au message, ce qui signifie en pratique avec la méthode de plus basse qualité. Cette politique permet d'exécuter un maximum de méthodes, donc de fournir un résultat, même imprécis, plutôt que pas de résultat du tout. En cas de conflit, les messages les moins importants sont différés et au pire éliminés. Pour un planning de n tâches, la complexité maximale est $O(n^2)$ (le problème général de minimisation des fautes temporelles est NP-complet).

3.4.2. Politique de choix des tâches à exécuter

Pour lancer une tâche, le temps effectivement libre est d'abord évalué. Puis la méthode la plus longue possible, donc de meilleure qualité, parmi celles associées au message est lancée, ceci indépendamment du fait que c'est la tâche la plus courte qui avait été planifiée. Le temps libre provient soit d'une sous-charge, soit du temps libéré par une méthode qui n'a pas utilisé son quota de temps, ce qui est fréquent car le planning est réalisé avec des temps au pire cas (WCET).

3.5. Décisions au niveau local (objet actif)

La première décision locale est la constitution de l'ensemble associé au message des méthodes et actions { ..., *methode_i*, ..., *action_j*, ... } qui sont chacun susceptible de rendre le service demandé. Actuellement, seuls les ensembles homogènes méthodes ou actions sont autorisés. La deuxième décision locale est le choix des messages à exécuter. Le choix final de la méthode s'effectuera au niveau global.

3.5.1. Ensemble initial des méthodes et actions

Au départ, chaque objet possède une table $Message \rightarrow \{ \text{méthodes} \mid \text{actions} \}$ qui peut ensuite être modifiée selon les besoins. La constitution de l'ensemble des méthodes est approfondie ci-dessous à titre d'exemple, mais les mécanismes sont les mêmes pour les actions. Indépendamment, les actions et méthodes sont modifiées à chaque changement de mode.

3.5.2. Constitution de l'ensemble des méthodes

La constitution de cet ensemble offre toute une gamme de combinaisons dont voici un échantillon :

– Méthodes de caractéristiques de QoS différentes

L'ensemble est constitué de méthodes de durée (polymorphisme en temps [Tak92]), de précision et plus généralement de QoS différentes. Au moment de l'exécution, la méthode qui dans le contexte fournit le meilleur service sera choisie. Evidemment, plus il y a de temps processeur disponible, plus la qualité du résultat sera bonne.

Il se peut que plusieurs méthodes fournissent le même service, par exemple plusieurs algorithmes de localisation d'un objet à partir d'une image caméra, et dans ce cas l'ensemble se construit naturellement. Mais la force de ce mécanisme vient de la possibilité d'ajouter des méthodes artificielles. Supposons une méthode m_i de durée d'exécution d_i qui renvoie une valeur. Une méthode m_j de durée $d_j \ll d_i$ renvoyant la valeur moyenne des n derniers appels (la valeur précédente ou toute autre solution très rapide) est ajoutée. Si le système est en surcharge, la méthode rapide est exécutée et un résultat imprécis est obtenu, sinon la méthode normale est exécutée. Dans l'exemple ci-dessus de localisation d'un objet, il est possible en cas de surcharge de renvoyer la position précédente.

– Ensemble avec règles de décision

Des règles de la forme [Del96] " m_1 et m_2 doivent être exécutées en alternance" sont mises en œuvre facilement en constituant des ensembles artificiels :

- "Exécuter m_1 ou m_2 avec au plus n exécutions consécutives de m_2 " : tant que la contrainte est respectée les deux méthodes sont placées dans l'ensemble, mais dès qu'elle est violée, m_1 seulement est conservée.
- " m_1 doit être exécutée au moins un appel sur deux" : l'ensemble contient toujours m_1 . Si m_1 a été exécutée la fois précédente, une méthode artificielle m_\emptyset de durée nulle est ajoutée. Quand le système est surchargé m_\emptyset est exécutée, mais la fois suivante elle ne figurera pas dans l'ensemble.
- "L'échéance de m_1 peut être dépassée une fois sur deux en général" : l'ensemble contient m_1 , et lorsque la contrainte va être violée une demande de *garantie* est effectuée pour la méthode. La garantie promet une exécution prioritaire en général (un message urgent peut cependant provoquer un dépassement).

– Task-Pair

La mesure du temps d'exécution des méthodes est parfois difficile, par exemple en cas d'attente de réponse d'un autre objet. Pour résoudre ce problème deux méthodes

sont définies [Str95], une première m_n avec une durée optimiste inférieure à la durée au pire, et une deuxième m_e de durée très faible qui sera exécutée en cas d'exception si la première ne peut se terminer. Ce mécanisme est réalisable en plaçant la méthode normale m_n dans l'ensemble, et en créant un deuxième message artificiel qui contient la méthode d'exception m_e avec une demande de garantie. A la fin de m_n le message contenant m_e est tué. Sinon, comme m_e est garantie donc plus prioritaire, elle sera lancée au dernier moment possible garantissant son exécution (*last chance moment*).

Ce mécanisme demande d'imaginer la combinaison selon le problème à résoudre. A l'usage, le compromis puissance d'expression/ temps d'exécution est très favorable.

3.5.3. Choix des messages

L'ensemble des méthodes passe à travers des filtres qui éliminent toutes les méthodes non compatibles avec les méthodes en cours d'exécution, non exécutables dans l'état courant ou n'offrant pas la QoS requise. Fig. 2, le trajet des messages inclut l'association $Message \rightarrow \{ méthode \}$ due au mode de l'application.

– Méthodes autorisées dans différents contextes

Ce procédé de filtrage augmente encore les capacités d'adaptation car il permet de placer dans un ensemble des méthodes qui s'exécutent dans des contextes différents. En fonction de ce contexte, certaines seront éliminées. Par exemple supposons deux méthodes qui peuvent s'exécuter l'une dans l'état *ouvert*, l'autre dans l'état *fermé* : suivant l'état courant, l'une ou l'autre sera éliminée.

– Choix des messages

A l'issue du filtrage, il ne reste que les messages potentiellement exécutables de la queue (ensemble de méthodes non vide). L'objet peut alors choisir selon sa politique. A titre d'exemple, les politiques de base fournies sont le choix du message dont l'échéance est la plus proche, du premier arrivé, du plus important, du plus important avec réservation d'un processus léger pour les messages critiques.

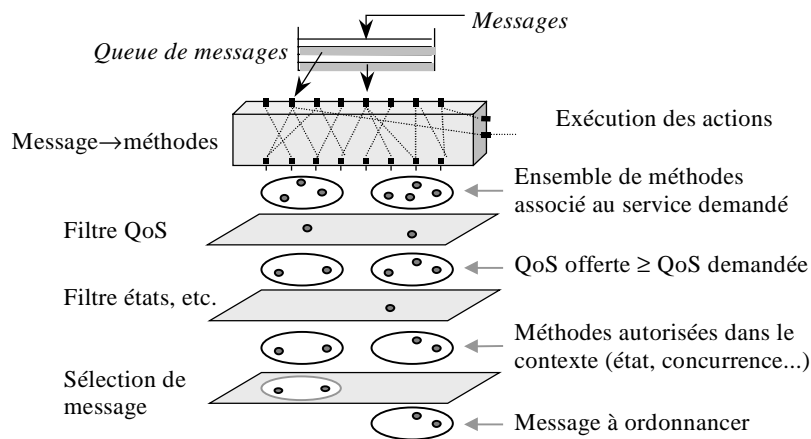


Figure 2. Traitement des messages

4. Description

La description des politiques d'adaptation n'est pas le moindre des problèmes. Dans ce projet le choix s'est porté dès le début sur une description en UML [UML]. Les modèles de l'application sont réalisés normalement avec un outil, mais ils sont décorés avec des informations supplémentaires qui seront utilisées pour engendrer le code. Ces décorations sont des propriétés (*taggedValue*) de forme $\{nom=valeur\}$, qui peuvent être ajoutées à tout élément de modélisation. La *valeur* a la syntaxe spécifique voulue. Le méta-modèle UML étant figé, pour ajouter de nouvelles notions il faut ajouter des pseudo-métaclasses appelées stéréotypes. Les extensions UML comportent les stéréotypes, propriétés et contraintes, et forment des "*profile*". Les fig. 3a-b montrent des décorations (en italiques pour les obligatoires). La description est relative à un contexte d'exécution car certaines valeurs (durée de méthode) dépendent du contexte. Toutes ces descriptions sont montrées dans des notes UML, mais en pratique un tel procédé n'est pas viable car trop lourd, et il faut un outil d'assistance et de contrôle de saisie.

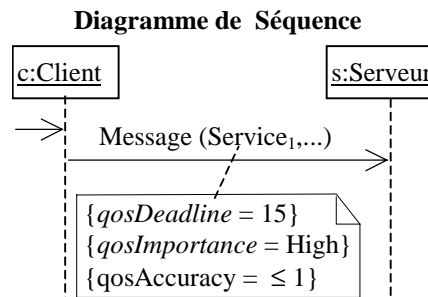


Fig. 3a. Description de la QoS en UML

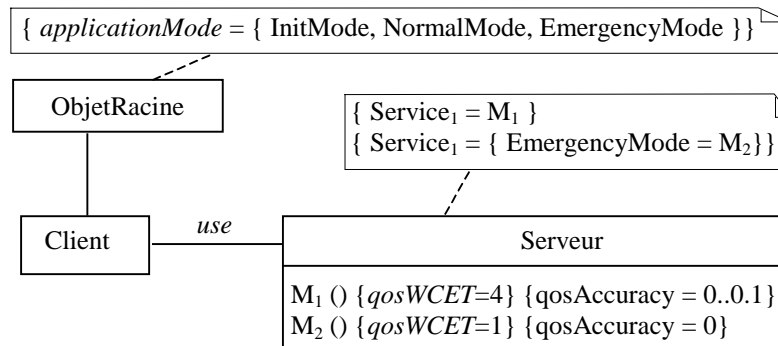


Figure 3b. Diagramme de classe UML décoré de paramètres de QoS

5. Réalisation

5.1. Architecture à méta-objets

Pour le choix de l'architecture, un objectif principal était de conserver la réutilisabilité et la portabilité en ne mélangeant pas ce qui est indépendant et dépendant du contexte. Un objet actif doit avoir des connaissances sur lui-même et doit pouvoir modifier son comportement pour s'adapter. Ces besoins ont conduit

naturellement à une architecture *réfléchie*² [Smi82] avec un niveau de base pour l'objet et un méta-niveau de contrôle du comportement. Le méta-niveau est réalisé par des méta-objets, chaque objet ayant sa propre instance de méta-objet (Fig. 4). Le méta-objet détient des connaissances sur son objet

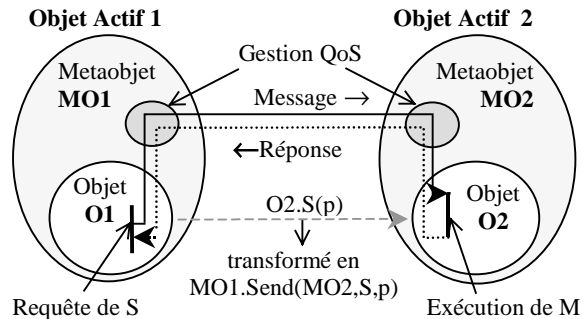


Figure 4. Réification de la communication

(comportement, état courant...) et s'occupe de tout ce qui est relatif au contexte d'exécution et à l'adaptation (ordonnancement, concurrence...). Les seuls mécanismes réifiés du langage sont l'envoi de message (Fig. 4) ainsi que la création/destruction des objets. Cette architecture n'est pas *réfléchie* dans le sens de KRS [Mae87], mais elle a l'avantage d'être facile à réaliser avec les langages à objets courants.

5.2. Prototype

Le prototype réalisé implante les mécanismes de base décrits plus de nombreux autres services (concurrence, états et transitions...). Pour assurer la prédictibilité, il n'y a pas d'allocation dynamique en dehors de la création des objets, et pour les performances des techniques efficaces ont été employées (codage des états [Sou99]).

Vu de l'utilisateur, l'environnement est constitué d'un ensemble de classes de méta-objets de base dont héritent les méta-objets utilisateur. En plus des décorations mentionnées, des informations supplémentaires sont nécessaires comme la compatibilité d'exécution entre méthodes, le nombre de processus légers d'un objet ou la taille de sa queue. Un outil est donc devenu indispensable. Toutes les saisies UML standard y compris le diagramme d'états sont effectuées via *Rational Rose* enrichi d'un stéréotype pour les objets actifs. Un *Ajout*³ prend en charge la saisie de toutes les décorations et modifie directement les modèles, avec l'avantage que l'utilisateur ne voit qu'un seul outil dont les menus sont modifiés. Cet ajout engendre automatiquement le code pour les méta-objets, en particulier les filtres Fig. 2, tandis que le générateur de code d'origine (C++) prend les objets normaux en charge.

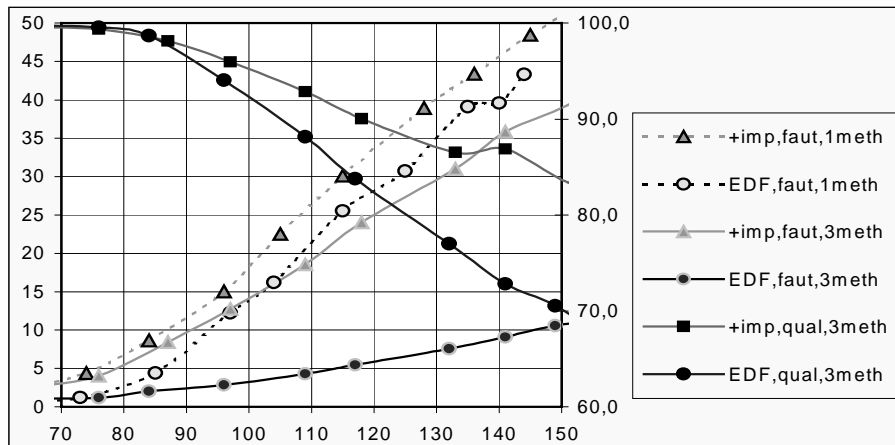
5.3. Exemple de résultats

Les courbes ci-dessous montrent, en fonction de la charge processeur demandée, le pourcentage de *fautes* moyen à gauche, et la *qualité* des résultats (proportion de méthodes de meilleure qualité exécutées) à droite, pour des messages ayant 1 et 3 méthodes de durées d et $d/8$. Le message choisi localement est soit l'échéance la plus

² *Reflective* en anglais, aussi appelée *réflexive* en français

³ Add-in, première version réalisée par Suneet Suri pendant son stage

proche (EDF), soit le plus important (+imp). Comme attendu, l'ajout de méthodes diminue le nombre de fautes mais aussi la qualité des résultats. Cette diminution dépend de la politique locale : EDF provoque en moyenne moins de fautes, mais en regardant dans le détail plus de fautes pour les messages importants.



6. Travaux voisins

Ce travail se situe dans la lignée des travaux centrés sur la tolérance aux fautes, la dégradation "gracieuse", la gestion de la QoS, qui tous ont pour but d'augmenter la QoS globale fournie par un système. L'idée de ce projet était de construire une architecture autour de l'adaptation [Cla99]. Le résultat est un cadre supportant un ensemble cohérent de mécanismes et non une technique particulière (task-pair [Str95], règles de décision [Del96], polymorphisme [Tak92]). De nombreux travaux (dont [Abd99], [Cha97]...) autour de "exécution distribuée + QoS + adaptation + *middleware*" utilisent les mêmes techniques. Les architectures réfléchies sont communes dans le domaine du temps réel ([Hon92][Mat91]...). Certaines approches utilisent plusieurs méta-objets affectés chacun à une fonction particulière, ce qui n'est pas conciliable avec la gestion collective adoptée. Finalement, ARTO est le seul qui offre des capacités de décision au niveau de l'objet, ce qui conduit à une architecture décentralisée et des objets réellement autonomes.

7. Conclusion

Ce projet avait pour but d'améliorer le comportement d'applications dans le domaine du temps réel mou en utilisant des techniques d'adaptation. Le point de contact entre ces aspects est la QoS. L'optimisation de la QoS déclenche les mécanismes d'adaptation qui ont pour effet la diminution des fautes temporelles et le contrôle de la dégradation des applications (*graceful degradation*). Ce résultat a été

obtenu sans remettre en cause les principes de l'approche objet : au niveau local les décisions sont spécifiques et chaque objet garde son autonomie, tandis qu'au niveau global les décisions sont communes pour optimiser l'utilisation des ressources. L'architecture à méta-niveaux sépare ce qui est dépendant du contexte de ce qui ne l'est pas, ce qui assure portabilité, réutilisabilité et évolutivité. Sur le plan des performances, les résultats se situent dans la même zone que les outils du marché qui eux ne traitent aucune contrainte de temps : ni ordonnancement et a fortiori ni adaptation. Le processus de développement s'inscrit dans une démarche génie logiciel : description en UML assistée avec un outil de contrôle, génération de code automatique libérant des tâches de bas niveau. Dans le futur, l'objectif est d'assister le développement et d'étendre le modèle à une exécution distribuée.

Références

- [Abd99] Tarek Abdelzaher, "QoS Adaptation in Real-Time Systems", PhD, Michigan, 1999
- [Cap02] Guy Caplat, *Modélisation Cognitive* PPUR ed. ISBN 2-88074-495-4
- [Cha97] S. Chatterjee, & Al "Modeling Applications for Adaptive QoS-based Resource Management", *Proc. IEEE Workshop HASE97*, 1997
- [Cla99] Clark R., Jensen E. D. & Al, "An adaptive, Distributed Airborne Tracking System", *Workshop on Parallel and Distributed RT Systems* (www.real-time.org), 1999
- [Con00] J.L. Contreras, J.L. Sourrouille, "A Scheduling Strategy to Preserve Object Autonomy", *WRTP'00*, 2000, pp.153-159
- [Con01] José Lino Contreras, Jean Louis Sourrouille, "A Framework for QoS Management", *TOOLS'39*, USA, IEEE press, 2001, pp.183-193
- [Del96] Delacroix J., "Towards a stable Earliest Deadline scheduling algorithm", *Real-Time Systems Journal*, Vol.10(3), 1996, pp.263-291
- [Hon92] Honda Y, Tokoro M., "Soft Real-Time Programming through Reflection", *IMSA Workshop Reflection and Meta-level Architecture*, 1992,
- [ISO97] Information Technology – Quality of Service – Guide to Methods and Mechanisms – ISO/IEC 13243 Draft 1.0 | Project JTC1 21.57, 15/10/1997
- [Lee98] Lee C, Siewiorek D, "An Approach for QoS Management", CMU-CS-98-165, 1998
- [Liu73] Liu C.L. , J.W. Layland, "Scheduling algorithms for multiprogramming in a hard real time environment". *Journal of the ACM*, Vol 20(1), 1973, pp.46-61
- [Mae87] Maes P., "Concepts and experiments in Computational Reflection", *OOPSLA'87*, 1987, pp.147-155
- [May95] Vicky de May, "Visual Composition of Software Application", In Nierstrasz and Tschritzis Eds: *Object-Oriented Software Composition*, 1995, pp-276--303.
- [Mat91] Matsuoka S., Watanabe B., Yonezawa A., "Hybrid Group Reflective Architecture for O.O. Concurrent Reflective Programming", *ECOOP'91*, LNCS 512, pp.231-250
- [Smi82] Smith B., "Reflection and Semantics in a Procedural Language", MIT, TR272, 1982
- [Sou99] J.L. Sourrouille, "UML Behavior: Inheritance and Implementation in Current Object-Oriented Languages", *UML'99*, LNCS 1723, 1999, 457-472
- [Str95] Streinch H., "Task Pair Scheduling: an Approach for Dynamic Real-Time Sytems", *Journal of Mini & Microcomputers*, Vol.17(2), 1995, pp.77-83
- [Tak92] Takashio K., M. Tokoro, "DROL: an object-oriented programming language for distributed real-time systems", *OOPSLA, ACM Sigplan*, 1992, pp.276-294
- [UML] *OMG Unified Modeling Language Specification (Action Semantics)*, V1.4, 2002