Les agents mobiles réactifs Mooréa

Une approche réactive pour la transparence à la mobilité et le passage à l'échelle

Bruno Dillenseger[*] — Laurent Hazard[**] — Anne-Marie Tagant[*]
Huan Tran Viet[***]

France Télécom R&D
[*]28, chemin du Vieux Chêne, 38243 Meylan cedex
[**]38-40, rue du Général Leclerc, 92794 Issy-Moulineaux cedex 9
{prénom}.{nom}@rd.francetelecom.com
[***]tvhuan@yahoo.com

RÉSUMÉ. *Les technologies à agents mobiles offrent un modèle de programmation unifié de la mobilité de code, de contrôle d'exécution et de données. La mise en œuvre de ce modèle implique l'implémentation d'un modèle d'agent particulier, associant activité autonome et communications (entre agents et avec l'environnement d'exécution). Or, le choix de ce modèle et des outils d'implémentation (e.g. langage de programmation, intergiciel) détermine deux propriétés non fonctionnelles importantes : (1) la capacité d'assurer une totale transparence à la mobilité pour un coût raisonnable, et (2) l'aptitude au passage à l'échelle en terme de nombre d'agents simultanément exécutables dans un environnement donné. Au travers de la présentation de la plate-forme à agents mobiles Mooréa, nous montrons comment l'utilisation d'un modèle d'agent réactif inspiré de la programmation synchrone apporte une réponse particulièrement pertinente à ces deux problématiques.*

ABSTRACT. *Mobile agent technology comes with a unified programming model combining code, execution and data mobility. Applying this model implies implementing a peculiar agent model, supporting both autonomous activity and communications (between agents and with the execution environment). But choosing such a model and its implementation tools (e.g. programming language, middleware) impacts two critical non-functional features: (1) affordable support for transparency to mobility, and (2) scalability in terms of number of agents simultaneously running on a given execution environment. Through the presentation of mobile agent platform Mooréa, we show how using a synchronous programming-derived reactive agent model brings a relevant solution to both issues.*

MOTS-CLÉS : *agent mobile, objet réactif synchrone, passage à l'échelle, transparence, MASIF.*

KEY WORDS: *mobile agent, synchronous reactive object, scalability, transparency, MASIF.*

1. Introduction

Les technologies à agents mobiles offrent un modèle de programmation permettant de gérer de façon unifiée la mobilité de code, de contrôle d'exécution et de données. L'agent est généralement défini comme une entité logicielle autonome ayant une activité propre et agissant pour le compte d'une personne ou d'une organisation [OMG 97]. Les applications pressenties concernent en particulier le commerce électronique et les télécommunications. Ainsi, à travers la plate-forme à agents mobiles Telescript, [WHI 94] met en évidence le concept *de remote programming* (i.e. déplacement du calcul vers les données, à l'inverse de l'appel de procédure distant), et présente comme application un réseau de places de marché électroniques où des agents se rencontrent afin de proposer, rechercher et négocier différents types de services, pour le compte d'utilisateurs individuels ou de sociétés (e.g. réservations, achat de biens divers...). En ce qui concerne les télécommunications, [INF 99] relate plusieurs projets européens du programme ACTS[1], qui exploitent la technologie des agents mobiles pour fournir des services avancés tels que réseau privé virtuel, réseau actif, prise en compte d'utilisateurs mobiles, qualité de service... Le projet européen ATHOS[2], dans lequel nos travaux s'inscrivent en partie, vise notamment les environnements d'exécution de services de télécommunication avancés, à la convergence du réseau intelligent et du réseau IP.

Dans ces deux types d'application, on voit que la question du **passage à l'échelle** est primordiale, car les systèmes envisagés sont de grande taille (échelle potentiellement planétaire), et les nœuds d'exécution sont susceptibles d'accueillir un nombre très important d'agents. Ainsi, une place de marché électronique internationale doit pouvoir héberger et exécuter des centaines de milliers d'agents représentant des prestataires de service, des marchands ou des clients. De même, un routeur de réseau actif doit exécuter en « temps réel » un nombre équivalent d'agents-paquets contenant leur propre algorithme de routage. Dans le cas du projet ATHOS, un point d'exécution de service dans un central téléphonique doit supporter au moins une dizaine de services actifs pour chacun de la centaine de milliers d'abonnés locaux. Il en résulte que la consommation de mémoire et de temps de calcul par agent doit être faible, ce qui permet de viser également les petits terminaux portables (téléphones mobiles, assistants électroniques personnels) qui doivent eux aussi être impliqués dans ces services, du côté des utilisateurs (voir, par exemple, [VAN 00]).

Outre cette problématique d'échelle, l'utilisation des agents mobiles pose le problème du modèle de programmation de l'agent, et de l'impact de la mobilité sur ce modèle. Dans le domaine du calcul mobile, [FUG 98] définit les concepts de mobilité faible et de **mobilité forte**, pour exprimer le fait qu'un calcul peut être perturbé ou non par un changement d'environnement d'exécution. Ceci est souvent

1. Advanced Communications Technologies & Services (http://www.infowin.org/ACTS/)
2. Advanced platform and Technologies for tHe Offer of communication Services (http://www.itea-athos.com/)

traduit dans les plates-formes à agents mobiles par le fait que l'activité de l'agent reprend soit à un point de redémarrage prédéterminé, soit au point exact de son exécution avant mobilité, l'état de l'agent (i.e. ses données) étant conservé dans les deux cas. De notre point de vue, la mobilité forte doit inclure non seulement les données et le point d'exécution, mais aussi les liens de communication de l'agent, afin d'assurer une **transparence** à la mobilité du modèle de programmation[3].

Face à ces exigences, nous proposons d'utiliser un **modèle d'agent réactif** inspiré des langages de programmation synchrone, se prêtant particulièrement bien à la programmation et à l'exécution des agents de façon économe en ressources, et de l'étendre par une fonctionnalité de mobilité forte parfaitement spécifiée.

Après avoir justifié les caractéristiques du modèle d'agent recherché, nous présentons l'environnement à objets réactifs répartis Rhum, et donnons des résultats expérimentaux concernant son passage à l'échelle. Ensuite, nous présentons l'intégration et l'extension de Rhum au sein de la plate-forme à agents mobiles réactifs Mooréa, en détaillant particulièrement le support de la mobilité des agents et les techniques utilisées pour assurer la transparence des communications à la mobilité. Enfin, nous considérons les travaux existants en rapport avec Mooréa, puis nous concluons en donnant les futurs axes de travail, notamment concernant l'application de Mooréa dans le domaine de l'équilibrage de charge.

2. Quel modèle d'agent mobile ?

2.1. *Le modèle d'activité*

Afin de fournir aux agents une autonomie d'exécution, beaucoup de plates-formes à agents mobiles associent un fil d'exécution (*thread*) à chaque agent en activité (e.g. plates-formes Aglets[4], Grasshopper [IKV 98]). Chaque agent devient donc un programme s'exécutant de manière indépendante. En dépit du confort immédiat qu'elle semble procurer au programmeur, cette approche comporte néanmoins plusieurs inconvénients :

– les fils d'exécution se révèlent très coûteux pour le nœud d'exécution, notamment en terme de consommation de mémoire et de temps processeur, et passe mal à l'échelle (voir mesures dans la section suivante) ;

– la plupart des langages de programmation ne prévoient pas le gel, le transport et le redémarrage d'un fil d'exécution, ce qui complique notoirement, voire interdit, la réalisation de la mobilité forte ;

---

3. Pour être tout à fait complet, il faudrait aussi inclure les liens avec toutes les ressources externes à l'agent, telles que fichiers ouverts, connexions réseau, interface graphique utilisateur... qu'il faudrait maintenir ou mettre à jour de façon paramétrable à chaque mobilité. Mais cet aspect sort du cadre des travaux présentés ici.

4. http://www.trl.ibm.com/aglets/

– la gestion de la concurrence et de la synchronisation à l'aide de verrous devient rapidement complexe, difficile à vérifier de manière formelle, et le moindre défaut peut aboutir à des incohérences ou à des interblocages entre fils d'exécution. En outre, cette gestion est diluée dans l'ensemble du code et n'apparaît pas clairement dans la structure du programme.

Une autre approche consiste à gérer une sorte de boucle d'événements pour l'ensemble des agents, au cours de laquelle les agents sont activés par la réception d'un message asynchrone (ils « réagissent »). Cette approche **réactive** est économe en ressource et élimine les problèmes de concurrence, dans la mesure où les agents sont tous activés par un unique fil d'exécution. Néanmoins, elle n'est pas suffisante en soit si l'on souhaite que les agents ait une activité propre en plus de ses réactions.

2.2. *Le modèle de communication*

Le modèle de communication a un impact déterminant sur l'autonomie de l'objet, notamment sur son autonomie d'activité et de mobilité. En effet, l'utilisation de communications synchrones, qu'il s'agisse d'appels de procédure distants (*Remote Procedure Call*) ou d'envois de messages avec valeur de retour, sont susceptibles de bloquer l'agent appelant en attente d'une réponse. Du côté de l'agent destinataire de la communication synchrone, cela suppose également de traiter la communication reçue et de retourner un résultat, ce qui le contraindra souvent, suivant la souplesse du modèle d'exécution et du système de communication, à rester sur place. De plus, ce type de communication induit généralement la création d'activités incontrôlées au sein de l'agent (e.g. des fils d'exécution pour les appels de procédure distants).

Par exemple, Grasshopper [IKV 98] propose des communications de type appel de méthode distant, qui peuvent aboutir à des incohérences non détectables, lorsqu'un agent bouge alors qu'il est en train d'être invoqué : le transport de l'agent ayant lieu concurremment avec l'exécution d'une de ses méthodes, l'agent se retrouve dans un état incohérent après mobilité ; l'exécution de la méthode s'achève sur le « fantôme » que l'agent laisse en partant, sans que cela puisse être détecté. Si l'on souhaite se prémunir de ce genre de problème en assurant une exclusion mutuelle entre invocation et mobilité à l'aide de verrous, on peut aboutir très facilement à des situations d'interblocage. Dans la pratique, ce type de communication induit souvent une restriction sur l'utilisation du modèle d'agent : un agent mobile peut invoquer un agent fixe, mais ne doit pas être invoqué lui-même.

Face à ces inconvénients, des systèmes de communication plus élaborés sont proposés. Ainsi, les agents des Aglets communiquent par messages, dont ils contrôlent explicitement la réception et le traitement. Ainsi, l'autonomie des agents est préservée, et la mobilité est gérée de façon transparente et fiable par le système.

Pour résumer, on peut dire que les communications synchrones compliquent notoirement la gestion de la transparence à la mobilité des agents, et qu'il est préférable de les construire, si besoin est, au-dessus d'un système de communication

asynchrone gérant correctement la mobilité. En outre, on note que dans le domaine des agents intelligents, des systèmes multi-agents et de l'intelligence artificielle distribuée, le modèle de communication par message asynchrone est prédominant, comme en témoignent les spécifications du consortium FIPA concernant les communications entre agents [FIP 98].

2.3. *Bilan*

En raison de nos objectifs de mobilité forte, de transparence et de passage à l'échelle, notre préférence va vers un modèle d'agent réactif communicant de façon asynchrone. Notre choix s'est porté sur Rhum, une plate-forme d'objets réactifs répartis que nous avions déjà développée par ailleurs. La section suivante détaille le modèle réactif, la plate-forme d'exécution, et donne des résultats expérimentaux comparant le passage à l'échelle des fils d'exécution et des objets réactifs.

3. Un modèle d'agent réactif basé sur Rhum

3.1. *Rhum, un modèle d'objet réactif en Java*

3.1.1. *Notion de système réactif synchrone et d'instant*

Dans Rhum, une application est un ensemble d'objets constituant un système dit **réactif,** inspiré de la programmation synchrone. Les objets de ce système ont un comportement propre, organisant un certain nombre d’actions en branches parallèles ou séquentielles, et dont l'exécution peut être conditionnée par la présence ou l'absence d'événements. Les comportements de tous les objets du système sont exécutés en parallèle suivant une suite commune d'instants logiques (figure 1). La notion d'instant définit précisément la sémantique de réaction, et borne la durée de vie d'un événement :

– un événement est présent à un instant donné, si et seulement si il a été généré dans ce même instant ;

– la **réaction** à un événement, i.e. le déblocage d'une branche en attente de cet événement, est exécutée dans l'instant de sa génération ;

– les réactions à un événement sont exécutées une et une seule fois au cours de l'instant, quel que soit le nombre d’occurrences de cet événement dans l’instant.

Un instant se termine lorsque toutes les branches de tous les comportements sont soit terminées, soit en attente :

– d'une synchronisation avec une branche parallèle non terminée,

– de la présence (ou de l’absence) d'un événement absent (respectivement, présent) dans l’instant en cours,

– explicitement de l'instant suivant.

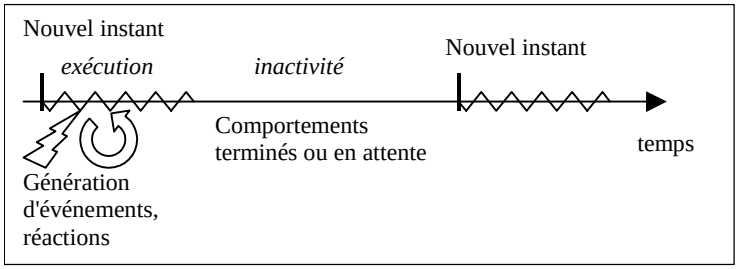

Figure 1. *Principe d'exécution d'un système réactif synchrone*

3.1.2. *L'objet réactif Rhum*

Les objets réactifs de Rhum sont inspirés du modèle synchrone ROM/DROM [BOU 96a]. De tels objets *(i)* offrent une interface pour la communication, locale ou distante, accessible à tout objet qui en possède une référence, et *(ii)* exécutent un comportement propre, éventuellement en réaction à la réception d'invocations sur leur interface ou à l'occurrence d'événements au sein de leur environnement d'exécution.

Dans Rhum, un **objet réactif** est une instance de classe réactive. Une **classe réactive** définit une interface réactive et un comportement pour ses instances. Le **comportement** est une spécification d'actions au fil des instants et de réactions à des événements, basée sur un ensemble d'instructions réactives définies dans un langage propre à Rhum (voir 3.1.3), dont la compilation génère la classe réactive en Java (figure 2). Ce comportement peut faire appel à des méthodes d'un objet associé, dit **objet passif**. Directement programmé en Java, il prend en charge les traitements élémentaires utiles pour l'objet réactif.

L'**interface réactive** déclare un ensemble de **méthodes réactives**, correspondant à des événements particuliers, qui pourront être adressés spécifiquement à l'objet réactif. Contrairement aux événements globaux du système, dits **événements d'environnement**, ces **événements ciblés** ne sont vus que par l'objet désigné lors de leur émission, et peuvent transporter des arguments. Le nom de l'événement ciblé correspond au nom de la méthode réactive, et ses arguments sont définis par la signature de cette méthode. Ainsi, la réception d'un événement ciblé revient à invoquer la méthode réactive associée. Toutefois, ce mécanisme suit le principe réactif, et se distingue clairement de l'appel de méthode Java normal. En effet, une méthode réactive n'est invoquée que si l'état du comportement est tel qu'il attend l'événement correspondant, et ce au plus une fois par instant sur un objet donné, quel que soit le nombre d'occurrences de l'événement adressées à celui-ci au cours de l'instant. En réalité, le mécanisme d'invocation de l'interface réactive est une spécialisation du mécanisme plus général de diffusion/réaction à des événements.

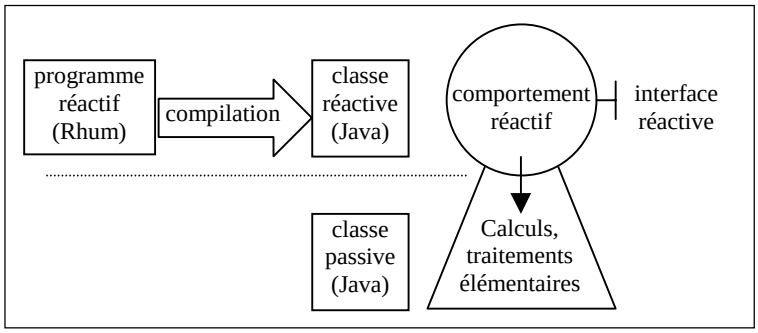

Figure 2. *Structure d'un objet Rhum*

3.1.3. *Le langage réactif synchrone Rhum*

Le comportement réactif d'un objet Rhum est décrit dans un langage dédié - Rhum -, inspiré du langage Esterel [BER 92], mais avec une sémantique légèrement modifiée afin de supprimer les problèmes de causalité, et de permettre une composition dynamique de programmes (i.e. ajouter des objets dans un système réactif en cours d'exécution) [BOU 96a].

L'exemple de programme donné ci-dessous (figure 3) définit deux branches parallèles, dont l'exécution provoque l'appel alternatif de la méthode 'say_hello()', puis 'say_world()', sur l'objet passif. Cet ordre est imposé par une génération et une attente ad hoc des événements 'hello' et 'world'. On note que sans l'instruction 'stop', ce programme bouclerait au sein d'un instant unique et sans fin. A l'inverse, on effectue ici une boucle par instant. Le programme se termine à la fin du premier instant rencontré durant lequel l'événement 'quit' est présent (préemption faible).

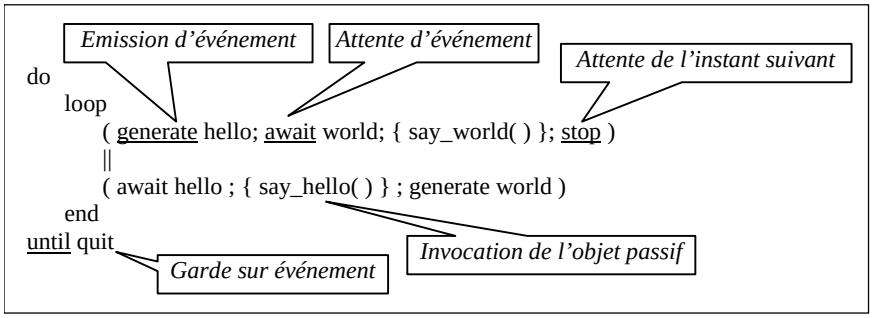

Figure 3. *Exemple de comportement réactif écrit dans le langage Rhum*

Un comportement associe instructions réactives (exécution d'actions en séquence ou en parallèle, attente d'événement, préemption d'action sur occurrence d'événement, etc.), désignation des événements pertinents pour le comportement de l'objet (invocation des méthodes réactives, signaux de l'environnement) et traitements de données (appliqués à l'objet passif associé). La compilation d'une telle spécification permet de générer les classes Java nécessaires à la mise en œuvre des objets réactifs décrits.

L'exécution des comportements est basée sur Junior [HAZ 00], un ensemble de classes qui implémente sous la forme d'objets Java les différents éléments nécessaires à l'exécution réactive (machine d'exécution d'instants, instructions réactives, composition de celles-ci, événements, etc.).

3.2. *Rhum : plate-forme d'exécution répartie*

Rhum étend le principe de programmation réactive en intégrant la possibilité de répartir des objets réactifs dans plusieurs sous-systèmes réactifs. Ces sous-systèmes, qualifiés de **domaines** réactifs, sont indépendants les uns des autres. Les objets Rhum appartiennent à un et un seul domaine, et partagent une même suite d'instants et les mêmes événements d'environnement au sein d'un domaine donné. Chaque domaine crée la suite d'instants d'exécution selon une politique qui lui est propre, déterminée par le programmeur de l'application, et indépendante des objets qui lui sont attachés. Par exemple, on peut démarrer un nouvel instant dès que le précédent est terminé, ou de façon périodique, ou dès qu'un événement extérieur survient.

Les objets réactifs peuvent interagir au travers des domaines par le biais de leur interface réactive, en utilisant des événements ciblés. En effet, la désignation d'un agent utilise un mécanisme de référence assurant une transparence à la localisation. Toutefois, l'émission d'un événement par un objet réactif et la réception de cet événement par l'objet destinataire ne s'exécutent dans un même instant qu'à la condition que les deux objets appartiennent au même domaine. En effet, la notion de « même instant » n'a pas de sens si les deux objets sont dans des domaines distincts. La sémantique suivante s'applique alors : l'événement sera généré une fois dans le domaine de l'objet invoqué à un instant futur non déterminé de ce domaine.

L'implémentation du système de communication de Rhum est basée sur l'ORB flexible Jonathan [OBJ 00]. Rhum exploite cette flexibilité, en ajoutant aux fonctionnalités natives de répartition celles d'une exécution réactive. Dans une application Rhum, les références sur les objets réactifs obtenues par instantiation d'une classe réactive ne pointent pas directement sur un objet d'implémentation, mais simplement sur un objet **souche** adéquat. Cette référence n'est opérationnelle (i.e. elle ne peut être invoquée ou transmise à un tiers) qu'après avoir été « attachée » à un domaine. Le type effectif de la référence rendue est conforme au type de l'interface réactive instanciée, et la fonctionnalité de la souche (i.e. la façon dont elle résout une invocation sous la forme d'une génération d'événement) dépend de la localisation de

la source (l'utilisateur de la référence) et de la cible (l'objet référencé). Rhum assure cette gestion particulière de façon transparente pour le programmeur de l'application.

3.3. *Gel de comportement et migration*

L'une des propriétés du modèle réactif de Rhum est qu'un système est stable et cohérent entre deux instants d'exécution (tous les événements présents à l'instant terminé ont été traités par tous les comportements concernés). En exploitant cette propriété, la construction ***Freeze*** a été ajoutée à Junior afin de permettre la préemption d'un module en cours d'exécution à la fin de l'instant où un événement désigné est présent, et la récupération d'un nouvel objet Junior représentant le reliquat (i.e. la partie non exécutée) de ce module après la fin de cet instant. Le nouvel objet, d'un type particulier, n'est plus exécuté et, comme tout objet de ce type, est d'une part « sérialisable » (au sens de Java, i.e. il peut être transporté sur un site distant), et d'autre part potentiellement exécutable dans un autre domaine réactif. Le service offert par la construction *Freeze* peut ainsi être utilisé pour réaliser la mobilité forte au sein d'une plate-forme à agents mobiles exploitant le modèle d'objet réactif.

3.4. *Evaluation des propriétés de passage à l'échelle*

3.4.1. *Principe de l'expérience*

Afin de vérifier les propriétés pressenties de passage à l'échelle de Rhum, et d'obtenir des ordres de grandeur, nous proposons de mesurer :

– l'évolution de la quantité de mémoire utilisée par un domaine en fonction du nombre d'objets dans ce même domaine,

– l'évolution du temps de création moyen par objet lors d'une création en rafale, en fonction du nombre d'objets créés,

– l'évolution du temps de réponse en fonction du nombre d'objets actifs.

Les agents que nous créons ont tous le même comportement : ils attendent un événement « start », puis ils exécutent une boucle incrémentant un entier. Tous les agents démarrent en même temps et effectuent le même nombre d'itérations, soit une itération par agent et par instant. Le temps moyen pris par une itération pour un agent nous sert de critère d'évaluation du temps de réponse.

A titre de comparaison, nous effectuons les mêmes mesures avec autant de fils d'exécution Java que d'objets réactifs. Ces fils d'exécution sont créés et démarrés en rafale, puis se synchronisent avant d'effectuer chacun le même nombre d'itérations.

3.4.2. *Mode opératoire*

L'expérience a été effectuée sur une station de travail Sun Ultra 60 sous Solaris 8, avec 256 Mo de mémoire et deux processeurs UltraSparc-II à 296 MHz. Nous avons pris soin de nous maintenir dans une zone d'occupation mémoire limitée à la mémoire vive (i.e. pas d'utilisation du *swap*), afin d'obtenir des mesures de temps d'exécution significatifs. Pour cette raison, nous avons arrêté les mesures de temps de réponse à 4500 fils d'exécution simultanés. Par ailleurs, il a été nécessaire de fixer la taille du tas de la machine virtuelle Java (arbitrairement à 160 Mo) pour pouvoir dépasser les 24000 objets réactifs, ce qui nous empêche de connaître la taille mémoire réellement utilisée à partir de cette valeur. La quantité de mémoire totale utilisée est calculée à partir des résultats de la commande Solaris « swap –s ». Les temps sont déterminés en utilisant la méthode System.currentTimeMillis() de Java 2.

3.4.3. *Résultats*

La mémoire utilisée (figure 4) augmente linéairement dans les deux cas (avec toutefois des perturbations dans le cas de Rhum, vraisemblablement imputables au *garbage collector*), mais la pente des deux courbes est très différente : environ 37 Ko/fil d'exécution et 0,37 Ko/objet réactif, soit un rapport de 1 à 100. Certes, ce résultat doit être modulé par le fait que les objets sont très simples, et que la réalisation de comportements plus complexes aurait un impact sensible sur la pente de la courbe représentant les objets Rhum. Mais on voit qu'un fil d'exécution engendre une consommation de mémoire très importante, quel que soit le comportement associé, alors que le surcoût imputable à la gestion des objets réactifs est particulièrement faible.

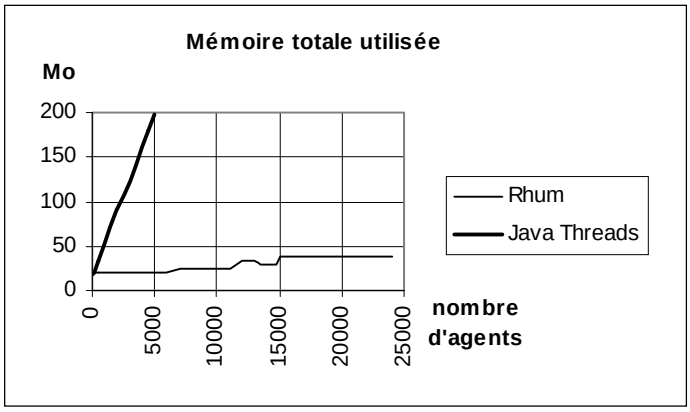

Figure 4. *Comparaison de la consommation de mémoire*

Le temps de création en rafale (figure 5) révèle clairement les limites des fils d'exécution en terme de passage à l'échelle. En effet, il apparaît que le temps de

moyen de création d'un fil d'activité augmente considérablement avec le nombre total de fils d'exécution. A l'inverse, on observe que ce temps moyen est à peu près constant à partir de 1000 objets réactifs.

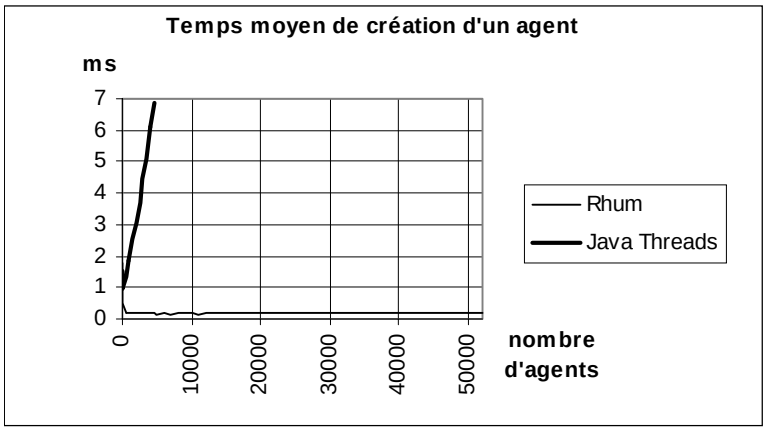

Figure 5. *Comparaison de l'évolution du temps de création*

Enfin, le temps d'exécution (figure 6) évolue d'une manière équivalente au temps de création : plus le nombre de fils d'exécution est important, plus les performances se dégradent, alors qu'elles restent stables pour Rhum à partir de 8000 objets réactifs environ, après une zone de dégradation de l'ordre de 50 %.

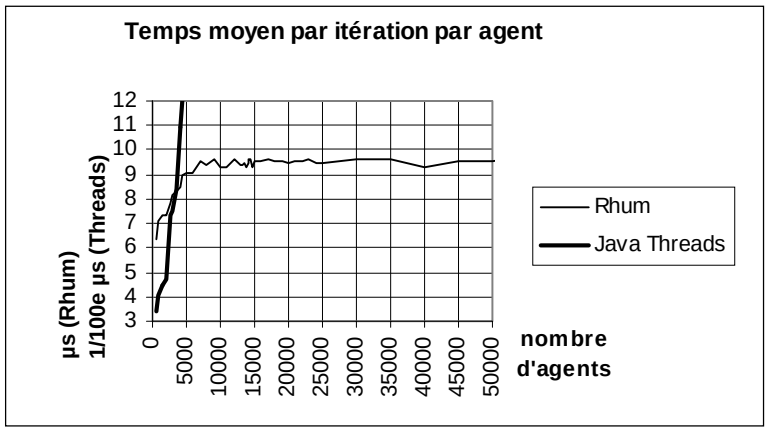

Figure 6. *Comparaison de l'évolution du temps d'exécution*

NOTE. — Les mesures effectuées visent à montrer des tendances et des ordres de grandeur pour le passage à l'échelle. Les valeurs des temps d'exécution ne peuvent être comparées directement, car la boucle des objets réactifs a été implémentée dans le langage Rhum, et non en Java comme c'est le cas pour les fils d'exécution, ce qui est évidemment beaucoup plus coûteux. Cette différence de valeur est, avant tout, un argument en faveur d'une implémentation des mécanismes d'exécution réactive au sein même des infrastructures (machine virtuelle, système d'exploitation). On pourrait alors comparer les performances en valeur brute, et apprécier pleinement les qualités de ces mécanismes réactifs, y compris pour un faible nombre d'activités, face à ceux implémentés, souvent au niveau du noyau de l'infrastructure, pour la synchronisation et l'ordonnancement des activités. Par ailleurs, l'étude du passage à l'échelle s'intéresse moins aux performances pures qu'à l'évolution de ces performances en fonction de l'échelle du système.

En plus de ces résultats, nous avons effectué des calculs statistiques sur les temps d'exécution. Il s'avère que la distribution des temps d'exécution est homogène pour les objets réactifs (écart-type de l'ordre de 2 % de la moyenne), alors qu'elle est particulièrement hétérogène pour les fils d'exécution (écart-type de l'ordre de 10 % de la moyenne sur Solaris 8, ou même 50 % observé sur Windows 98[5]).

3.4.4. *Analyse des résultats*

Les objets Rhum sont économes en mémoire et en temps de création, car ils représentent uniquement quelques objets Java de taille réduite, alors que les fils d'activité sont gros consommateurs de ressources (allocation d'une pile d'exécution pour chaque fil, gestion par l'ordonnanceur du système hôte). Au niveau du temps d'exécution, on illustre ici la différence entre un ordonnancement dynamique coopératif pour les objets réactifs, et un ordonnancement dynamique préemptif pour les fils d'exécution, dont les performances s'écroulent en raison du coût des changements de contexte et de la gestion de la mémoire.

En ce qui concerne le temps de création des fils d'exécution, une connaissance détaillée des mécanismes de leur gestion par le système d'exploitation serait nécessaire pour expliquer avec précision la dégradation de performance observée. On peut néanmoins penser à un problème de gestion de la mémoire, très sollicitée par la création de fils d'exécution, phase au cours de laquelle les différents caches (mémoire cache, cache de la table des pages de la MMU) sont probablement peu opérants car rafraîchis continuellement. On pense également au coût des structures et à la complexité algorithmique de l'ordonnancement des fils d'exécution, ainsi qu'à la gestion de la synchronisation (chaque fil d'exécution se met en attente une fois créé), tant au niveau système que Java.

La conclusion essentielle est que la consommation de ressources par Rhum (et Junior) reste linéaire en fonction du nombre d'objets réactifs, et qu'on n'observe pas

5. Mesures complémentaires réalisées sur un ordinateur portable équipé d'un processeur Pentium III à 600MHz, de 128Mo de mémoire vive, et fonctionnant sous Windows 98.

de dégradation de performances. Ainsi, les expériences ont permis d'aller jusqu'à plus de 50000 objets réactifs, à comparer aux 5000 fils d'exécution maximum. Enfin, on note que le temps d'exécution d'un système réactif est stable (reproductible, prévisible), alors qu'il est très variable pour un système à base de fils d'exécution.

3.5. *Bilan : adéquation de Rhum à nos besoins*

Le choix de Rhum comme modèle et support aux agents mobiles paraît judicieux pour les applications envisagées, et ce à plusieurs titres :

– aptitude au passage à l'échelle,

– aptitude au support de mobilité forte (gel et reprise du comportement),

– système de communication bien adapté aux agents mobiles (communications asynchrones au sens utilisé en 2.2), et gérant la répartition,

– représentation du comportement par un langage explicitant le parallélisme et la synchronisation, et permettant de définir une activité autonome sans recourir aux fils d'exécution.

Pour construire une plate-forme à agents mobiles, passant à l'échelle et proposant la mobilité forte, il reste à ajouter à Rhum un support pour la mobilité des objets, et à adapter le système de communication pour prendre en compte la mobilité. C'est ce que nous décrivons dans la section suivante.

4. Construction de la plate-forme Mooréa

4.1. *L'architecture de Mooréa*

Mooréa - *MObile Objects, REactive Agents* – est une plate-forme à agents mobiles réactifs intégrant l'environnement de programmation et d'exécution fourni par Rhum et Junior (ainsi que Jonathan, intergiciel exploité par Rhum). Pour construire une plate-forme à agents mobiles sur cette base, il faut :

– donner aux objets réactifs une « saveur d'agent », et définir une infrastructure pour assurer la mobilité de ses agents ;

– assurer la transparence des communications à la mobilité.

Pour ces deux aspects, nous réutilisons respectivement l'infrastructure générique pour agents mobiles SMI [DIL 00], et le cadre offert par l'ORB flexible Jonathan [OBJ 00] dans lequel nous ajoutons la transparence à la mobilité (figure 7).

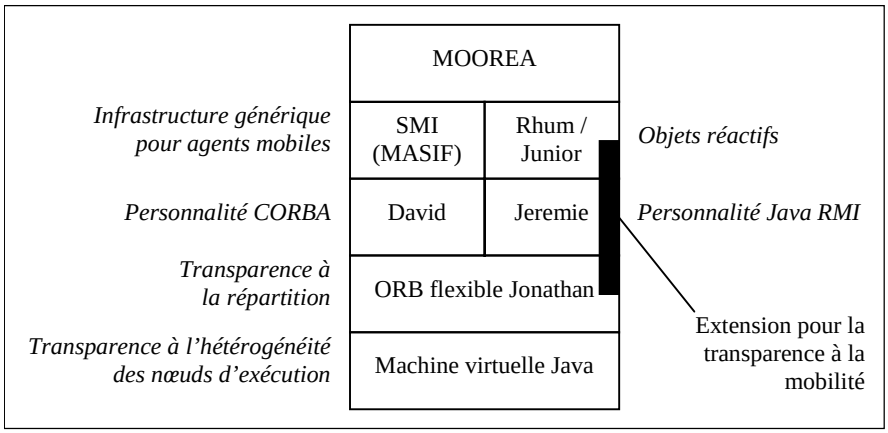

Figure 7. *L'architecture de Mooréa*

4.2. *L'infrastructure générique pour la mobilité SMI*

4.2.1. *Présentation générale*

SMI fait partie de la boîte à outils MobiliTools [DIL 00], dont l'objectif est de fournir des modules indépendants mais combinables, gérant de façon bien délimitée les communications, la mobilité, et l'exécution de divers modèles d'agents, afin de permettre la construction de plates-formes à agents mobiles sur mesure, mais toujours interopérables. L'autre particularité de MobiliTools se trouve dans l'utilisation de standards de l'OMG[6], notamment CORBA, dans le but de favoriser l'interopérabilité. SMI – *Simple MASIF Implementation* – est un module de gestion d'objets mobiles en Java, implémentant les spécifications MASIF[7] [OMG 97].

4.2.2. *Le cadre conceptuel et les services spécifiés par MASIF*

Conformément aux spécifications MASIF, SMI (et donc Mooréa) réalise l'architecture suivante (figure 8) : les **agents** agissent de manière autonome pour le compte d'une **autorité** (personne, organisation). Ils sont accueillis et exécutés dans des **places** au sein de **systèmes agents** (ou **agence**) caractérisés par leur **type** (identifiant une implémentation particulière de MASIF). Les **agents mobiles** ont la capacité de bouger de place en place, sous réserve que leur type soit reconnu par l'agence de destination. Les agences sont également liées à une autorité, et des agences de même autorité peuvent être groupées en **région**. Les agents et les agences possèdent un **nom** globalement unique résultant du triplet {autorité, identité, type}.

6. *Object Management Group*, http://www.omg.org/.
7. *Mobile Agent System Interoperability Facilities*, aussi connues sous le nom de *MAF - Mobile Agent Facilities.*

Les interactions au sein de ce système réparti sont basées sur la définition de deux interfaces CORBA :

– l'interface `MAFAgentSystem` doit être implémentée par les agences pour gérer les agents (création, activation, gel d'activité, destruction), recevoir les agents entrants, et transférer le code des agents sortants ;

– l'interface `MAFFinder` est dédiée à l'enregistrement et à la recherche des agents, agences et places.

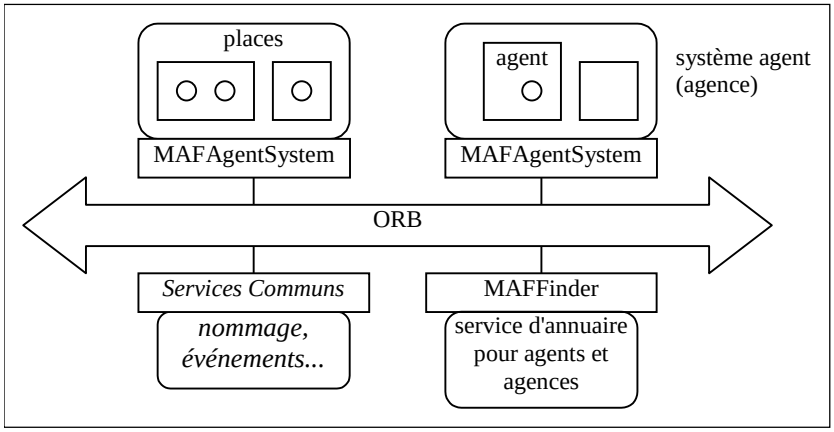

Figure 8. *L'architecture MAF/MASIF*

4.2.3. *Les agents/objets mobiles*

SMI permet de créer des objets mobiles Java par instantiation de n'importe quelle classe implémentant l'interface `MobileObject`. Cette interface stipule deux obligations pour l'agent : d'une part être « sérialisable », car la sérialisation Java est utilisée pour obtenir une représentation de son état lors de la migration, et d'autre part fournir un ensemble de méthodes (***call-backs***) qui sont invoquées par l'agence hôte lorsqu'il est créé, bougé, suspendu, réactivé, détruit... Ainsi, l'interface `MobileObject` décrit le cycle de vie de l'agent, mais l'agent définit lui-même sa réaction aux diverses sollicitations, qu'il peut même rejeter par le biais d'exceptions. La libre implémentation de l'interface `MobileObject` laisse toute latitude sur le modèle effectif d'agent. Cette caractéristique nous permet d'intégrer l'environnement réactif de Rhum, et d'obtenir les agents réactifs mobiles Mooréa.

4.2.4. *Les agences*

Pour créer et gérer des objets mobiles, il faut tout d'abord créer des agences. Outre son interface CORBA, une agence SMI est un objet Java offrant un certain nombre de méthodes pour créer, suspendre, réactiver, bouger ou détruire un agent.

Ces méthodes ont notamment pour effet d'appeler des méthodes de l'interface `MobileObject` sur l'agent concerné. De façon optionnelle, une agence peut aussi être personnalisée, par association avec un objet « **personnalité »** implémentant l'interface `AgencyPersonality`. Cette interface définit un ensemble de méthodes (*call-backs*) qui sont invoquées par l'agence avant et après l'appel aux méthodes de `MobileObject`. Ce mécanisme facilite la réalisation de modèles d'exécution d'agent particuliers. Dans Mooréa, l'agence devient également un domaine réactif.

4.2.5. *La gestion de la mobilité de code*

Lorsqu'un agent est créé ou lorsqu'il bouge, les classes dont il a besoin ne sont pas nécessairement toutes définies dans l'agence hôte. Par ailleurs, ses classes ne doivent pas être confondues avec d'autres classes ayant éventuellement le même nom. Pour cela, tout agent est associé à une information de provenance des classes (*code-base*). Donnée sous forme d'URL, cette information désigne un fichier ou un répertoire accessible par le système de fichier local ou via un serveur HTTP.

La particularité de SMI réside dans le fait que cet URL est interprété une seule fois, par l'agence où l'agent est créé. Lors des mobilités successives, les classes manquantes sont demandées à l'agence source et mises dans une mémoire cache de l'agence destination, l'information de *code-base* participant uniquement à une clé utilisée pour stocker et retrouver ces classes. Ainsi, les agents de SMI peuvent continuer à bouger d'agence en agence librement, même si l'agence initiale ou le serveur HTTP désigné par le *code-base* sont arrêtés ou inaccessibles. Cette propriété est importante, car une des justifications de la mobilité est la possibilité de transférer un agent sur une autre machine, afin d'éteindre la machine de départ, ou d'explorer un sous-réseau avec lequel la connexion n'est pas permanente.

4.3. *Transparence des communications entre objets réactifs mobiles*

4.3.1. *Notion de référence et problématique de la mobilité*

Une référence désigne sans ambiguïté un objet et donne à son client un accès à cet objet. Dans notre exemple (figure 9), l'objet *X* détient une référence *refY* qui lui permet d'établir un canal de communication avec l'objet *Y*. La migration de l'objet Y pose non seulement le problème de l'invalidité de *refY* mais aussi de la nature de la communication entre les deux objets (locale $refY_2$ ou distante $refY_1$). Dans le cas où une référence est un objet intermédiaire entre deux objets, la migration de l'objet client *X* demande aussi la mise à jour de *refY*, notamment dans le cas où (a) *X* se arrive dans la plate-forme de *Y* ou (b), inversement, la quitte.

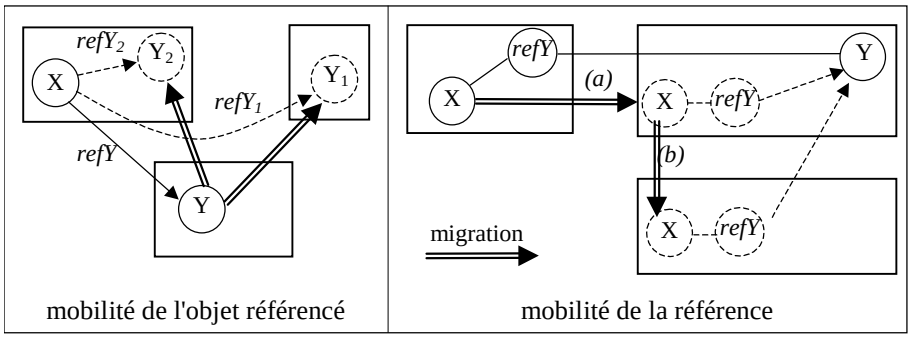

Figure 9. *Impact de la mobilité sur les références d'objets répartis*

La mise à jour des références doit satisfaire des contraintes de transparence (le fait que l'objet se soit déplacé doit être masqué au client) et d'efficacité (le temps de mise à jour doit être du même ordre que celui de migration de l'objet).

4.3.2. *Support de la transparence dans Rhum/Mooréa*

Pour Mooréa, il est critique que les événements ciblés parviennent à l'agent destinataire, même si celui-ci est en train de bouger, et que la validité des références vers les interfaces réactives soit maintenue. Cette transparence est réalisée par une combinaison originale de deux techniques courantes, qui sont la retransmission et le service de nommage :

La **retransmission** consiste à remplacer l'agent ayant bougé par un objet retransmetteur qui fait suivre les invocations vers la nouvelle localisation de l'agent. La plate-forme Voyager [OBJ 97], par exemple, utilise ce mécanisme. Si on applique cette technique simple sans précaution particulière, on risque d'aboutir à une chaîne de références longue, inefficace et fragile, nécessitant le maintien en activité de toutes les agences visitées, ce qui n'est pas réaliste et va à l'encontre d'une des motivations majeures de la mobilité.

Le service de nommage - ou **service de relocalisation** - permet d'associer un nom déterministe et invariable à une référence d'objet réparti contenant les informations de localité. En mettant à jour la référence d'objet après chaque mobilité, les agents mobiles peuvent toujours être localisés, sauf pendant leur période de transit, durant laquelle l'ancienne référence n'est plus valide et la nouvelle n'existe pas encore. Là encore, une utilisation immédiate de cette technique montre ses limites, en terme d'autorité centralisée et de goulet d'étranglement.

Mooréa combine ces deux techniques tout en maîtrisant leurs inconvénients. Tout d'abord, la chaîne de références est limitée à **une seule indirection**, en faisant en sorte que le retransmetteur obtienne directement la nouvelle référence auprès du service de relocalisation. De plus, le retransmetteur met à jour la référence détenue

par le client au moment de l'invocation, ce qui diminue de façon notoire l'effet de goulet d'étranglement au niveau du service de relocalisation, dans la mesure où les clients n'ont pas besoin d'utiliser ce service. Enfin, il est possible d'utiliser plusieurs serveurs de relocalisation, ce qui permet, d'une part, de répartir la charge, et, d'autre part, d'avoir plusieurs contextes de nommage et des noms qui ne soient pas globalement uniques. Ceci est associé à une structure de noms particulière, combinant un identificateur d'objet, une référence vers l'objet, et une référence vers le serveur de relocalisation correspondant. Cette technique est utilisée par la plate-forme FlexiNet [HAY 98] et adoptée dans les spécifications FIPA [FIP 98].

Supposons, par exemple, qu'un agent Mooréa soit en train de bouger d'une agence A à une agence B, et que d'autres agents lui envoient des événements ciblés :

– pendant le déplacement, et aussi longtemps que l'agent n'a pas été réinstallé dans l'agence B, les événements sont stockés dans l'agence A ;

– l'agence A obtient la nouvelle référence de l'agent auprès du service de relocalisation, puis retransmet les événements stockés vers l'agent dans l'agence B, et envoie la nouvelle référence de l'agent aux émetteurs de ces événements ;

– si l'agence A devient inaccessible (e.g. elle a été arrêtée), les détenteurs de référence vers l'agent s'adressent directement au service de relocalisation pour obtenir la nouvelle référence (au moment de l'utilisation de cette référence).

4.3.3. *Implémentation*

Ces fonctionnalités de transparence sont implémentées au cœur de la couche d'intergiciel (Jonathan et sa personnalité RMI Jeremie, et Rhum), notamment au niveau des objets souches (*stubs*) et des adaptateurs d'objet. Ces travaux ne sont pas spécifiques à Mooréa, et sont réutilisés dans d'autres travaux en cours, pour supporter la mobilité d'objets dans David, la personnalité CORBA de Jonathan.

5. Présentation et mise en œuvre de la plate-forme Mooréa

5.1. *Présentation générale de la plate-forme*

5.1.1. *Gestion des agents et des agences*

Une application construite sur Mooréa est formée par un ou plusieurs agents communicants, qui peuvent être sur des agences différentes, et peuvent être mobiles. Une agence Mooréa est un domaine Rhum, qui intègre un moteur d'exécution synchrone (Junior). Les agents sont des objets réactifs, dont le comportement est décrit en langage Rhum, et le traitement de données en Java (objet passif). Les communications inter-agents se font de manière asynchrone, par envoi d'événements Rhum. Un agent peut générer un événement ciblé, portant si besoin des arguments de types de base ou objets, pour un agent particulier local ou distant, ou pour un groupe

d'objets locaux ou distants. Il peut aussi diffuser un événement d'environnement (sans argument) dans le domaine réactif local (i.e. son agence hôte).

L'ambition de Mooréa est de fournir une agence d'agents fortement mobiles. Elle se doit donc d'offrir aussi les fonctions classiques de gestion répartie des agents et des agences. Mooréa offre aux agents des services permettant :

– la manipulation des agents (soi même ou un autre agent), de façon répartie :
  - création d'un agent, dans l'agence locale ou dans une agence distante ;
  - migration d'un agent local ou distant, vers une agence locale ou distante ;
  - suspension, relance, terminaison d'un agent local ou distant ;
  - accès aux propriétés d'un agent local ou distant ;
  - recherche d'agents par nom, propriétés ou localisation ;
– la manipulation des agences :
  - recherche d'agences par nom ou localisation ;
  - fermeture d'une agence locale ou distante.

NOTE. — La migration d'un agent pouvant être initiée par lui-même ou par l'environnement, Mooréa ne force pas de modèle particulier de mobilité (mobilité demandée ou subie par l'agent). Quelle que soit la provenance de la demande de migration, elle exige un consensus entre le système, qui parvient ou non à migrer l'agent, et l'agent lui-même, qui accepte ou refuse de bouger (via l'interface `MobileObject`). A partir de ce cadre général, toute politique de migration plus restrictive peut être déclinée.

5.1.2. *Le cycle de vie des agents*

Le cycle de vie d'un agent Mooréa (figure 10) découle de celui défini par SMI (qui découle lui-même des spécifications MASIF), dont il reprend les primitives (*call-backs*) associées aux manipulations sur un agent, et le protocole d'invocation. Les agents sont libres d'implémenter ces primitives (un comportement vide étant réalisé par défaut) :

– *afterBirth* après création,

– *beforeMove* avant migration,

– *afterMove* et *afterMoveFailed* après migration (respectivement en cas de succès ou d'échec),

– *beforeSuspend* lors d'un gel d'activité,

– *beforeResume* lors d'une reprise d'activité,

– *beforeDeath* avant destruction,

– *beforeShutdown*,avant la fermeture de l'agence hôte.

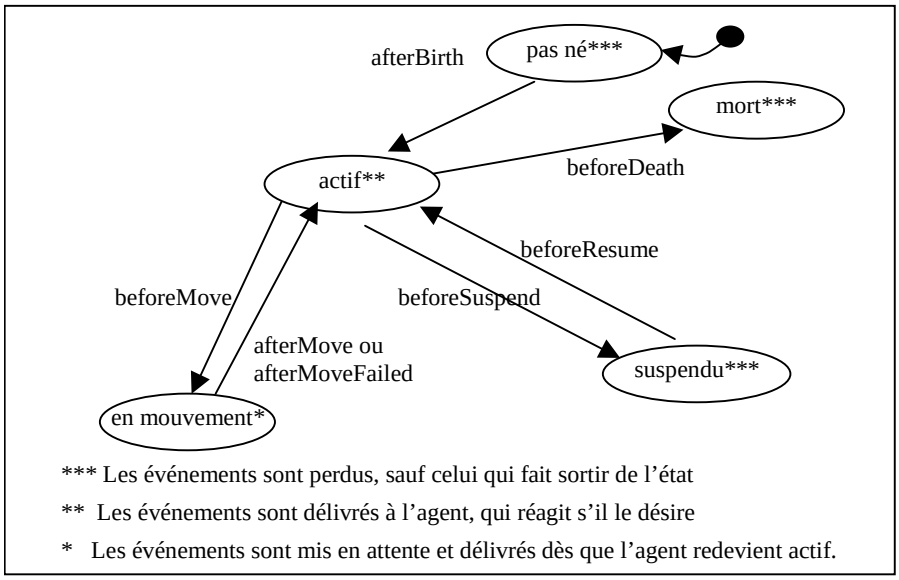

Figure 10. *Le cycle de vie d'un agent Mooréa*

Ce cycle de vie définit un certain nombre d'états, qui déterminent notamment la façon dont les événements à destination d'un agent sont pris en compte. En résumé, tous les événements envoyés à un agent sont délivrés dans un instant futur, sauf quand l'agent est mort, suspendu, ou pas encore né. Si l'agent destinataire est actif et dans la même agence que l'émetteur, l'événement est reçu dans le même instant synchrone[8].

5.1.3. *La désignation des agents*

L'élément principal de désignation des agents Mooréa est la **référence**, instance de la classe `MooreaObject`, qui n'est autre que la référence Rhum étendue pour être transparente à la mobilité. Cette référence est donc utilisée pour envoyer un événement à un agent local ou distant, par une simple invocation de méthode locale Java. De plus, elle sert à désigner les agents Mooréa dans toutes les méthodes de manipulation et de recherche d'agents, de façon répartie. Ce double usage fait de la référence Mooréa un élément fondamental et particulièrement commode, non seulement en raison de l'uniformité de la désignation des agents pour la gestion et la communication, mais aussi et surtout par son caractère totalement transparent à la mobilité des agents et des références.

8. Exception faite d'un événement ciblé vers un agent venant d'arriver suite à une migration, tant que la référence vers cet agent n'a pas été utilisée au moins une fois et ainsi mise à jour.

5.1.4. *Mise en œuvre de la mobilité*

L'exécution synchrone est telle qu'à la fin d'un instant synchrone, les objets réactifs sont au repos, dans un état cohérent. Le moteur d'exécution (Junior) sait geler l'agent réactif à ce moment-là. Par conséquent, lorsqu'une demande de mobilité survient au cours d'un instant, la migration effective (i.e. sérialisation de l'objet réactif et de son objet passif, transport de cet état, puis désérialisation dans l'agence d'arrivée) intervient **à la fin de cet instant**.

5.2. *Comment écrire un agent Mooréa ?*

5.2.1. *Ecriture du comportement réactif en Rhum*

Le comportement d'un agent Mooréa est défini par une spécification en langage Rhum (figure 11 – exemple d'un agent Timer). Le comportement d'un agent Mooréa possède d'emblée un comportement générique, permettant notamment de gérer le cycle de vie de manière transparente, qu'il enrichit à sa guise. Ce comportement de base est défini dans un fichier Rhum à inclure (cf. instruction « include »). Ecrire le comportement particulier d'un agent consiste à écrire le corps du comportement « moncomportement() » en Rhum standard, avec des appels éventuels aux méthodes d'un objet passif associé (méthodes `tick()` et `findinstant()` dans l'exemple donné), dont la classe est déclarée au début de la spécification du comportement (classe « Ptimer » dans l'exemple).

```
package moorea.utilsTimer;                                   // code Java standard
rclass RTimer (MobileName name) {                    // Définition de la classe réactive
  implements moorea.RIAgent;                      // qui implémente l'interface RIAgent
  uses moorea.utilsTimer.PTimer(MobileName name);          // d'objet passif PTimer
  environment {tick;}                              // sensible à l'événement local tick
  include "../RAgent.rhuminc";                  // qui a le comportement d'un agent SMI.
  behavior moncomportement() {          // Définition du comportement propre de Timer
    loop await tick; {tick();}; stop; end;                   // attend l'événement tick
    ||                                                       // et en parallèle
    loop stop; {findinstant();};end; }                       // attend un autre instant
  behavior {                                      // Comportement lancé au démarrage,
    run cœur }                                      // lance le comportement par défaut
}                                                    // fin de la classe réactive RTimer
```

Figure 11. *Comportement réactif d'un agent Mooréa « Timer »*

5.2.2. *Ecriture de l'objet passif associé, en Java*

Dans Mooréa, l'objet passif a deux fonctions : (1) il doit implémenter les méthodes appelées par le comportement réactif, comme pour tout objet Rhum ; (2) il peut implémenter des méthodes de l'interface MobileObject (cf. SMI) afin de

prendre en compte de manière spécifique le cycle de vie de l'agent Mooréa auquel il est lié. Ainsi, l'exemple ci-dessous (figure 12) implémente les méthodes `tick()` et `findinstant()`, liées au comportement réactif, ainsi que la méthode `afterBirth()`, correspondant à l'étape de création dans le cycle de vie.

```
package moorea.utilsTimer;
class PTimer extends PAgent implements Serializable {
long delta;             // la fréquense des tick en ms
long h0, h;             //le temps precedent, le temps présent
Event  tick_event=…;
// constructeur : toujours un argument de type MobileName
public PTimer (org.objectweb.mobile.apis.relocator.MobileName n) { super (n); }
// cycle de vie (cf. interface MobileObject)
public void afterBirth(AgentSystem agency, AgentInfo entry, Object data)
            throws BadOperation
            { delta= (TimerD)data.delta;  h0=System.currentTimeMillis(); }
// méthodes appelées par le comportement réactif
public void endofinstant() {
    h=System.currentTimeMillis();
   if ((h-h0)>=delta) { h0=h; tick_event.generate (new Object[] {});}
}
public void tick(String strevent) { System.out.println("*"); }
}
```

Figure 12. *Programmation de l'objet passif d'un agent Mooréa « Timer »*

6. Autres travaux en rapport

6.1. *Programmation synchrone, agents mobiles réactifs*

Esterel [BER 92] est sans doute le langage fondateur de l'approche synchrone. SL [BOU 96b] est une variante qui supprime les problèmes de causalité découlant de la réaction instantanée à l'absence d'événement. Rhum supprime également ce problème, et permet d'ajouter dynamiquement des comportements, ce qui n'est pas possible en Esterel. Cependant, seul Esterel offre aujourd'hui des possibilités de preuve formelle et de validation de contraintes temporelles [BER 00].

Rejo [ACO 01] est une plate-forme d'agents réactifs mobiles pouvant fonctionner sur Junior. Ce travail est donc très proche de Mooréa, avec toutefois quelques différences. Rejo propose une intégration poussée entre Java et le langage réactif, qui affranchit le programmeur de l'écriture séparée des classes passives. Autre différence, les événements de Rejo sont purement locaux, et la communication distante entre agents n'est donc pas directement supportée.

6.2. *La mobilité forte*

Plusieurs approches existent pour assurer la mobilité forte. On peut créer un nouveau langage ad hoc (e.g. Telescript [WHI 94]), ou étendre un langage existant et modifier son interpréteur (e.g. AgentTCL/D'Agents [GRA 97]) ou sa machine virtuelle (e.g. machine virtuelle Java Aroma [SUR 00]). Certaines approches consistent à instrumenter les programmes afin de positionner des points d'arrêt/mobilité/reprise dans le code. Dans le cas du langage Java, la technique la plus prisée est celle de post-compilation du *byte-code*, car elle permet de rendre mobile une classe sans disposer de son code source, et elle se révèle en général plus efficace. Néanmoins, ces approches ont un coût sensible en taille de code et en temps d'exécution. A titre d'exemple, [TRU 00] montre un accroissement de 35 % de la taille du code, et de 1,5 % à 27 % du temps d'exécution. Enfin, on peut aussi implémenter un langage existant permettant de manipuler l'état d'exécution (e.g. la « continuation » dans le langage Scheme), au-dessus d'un langage tel que Java, qui simplifie considérablement les problèmes d'hétérogénéité, ainsi que de transport d'état d'objet et de code (e.g. interpréteur Scheme en Java [HAL 97]).

L'approche la plus comparable à nos travaux (en dehors de Rejo) est le système Bond [BOL 00], qui propose également un langage dédié à la description formelle du comportement des agents (BluePrint). Cette description est compilée sous la forme de machines à états, qui font appel à des procédures (les « stratégies ») écrites en Java. La mobilité est forte dans le sens où l'état d'exécution des machines à états - et donc du comportement - est conservé par la mobilité, mais, comme dans le cas de Mooréa, la mobilité ne peut intervenir qu'à des moments bien précis (i.e. hors de l'exécution des stratégies pour Bond, et à la fin des instants pour Mooréa).

7. Conclusion

Cet article présente la plate-forme à agents réactifs mobiles Mooréa. D'une part, nous avons mis en évidence les qualités du modèle réactif synchrone comme modèle d'exécution et de communication pour les agents mobiles. Il passe bien à l'échelle (en terme de nombre d'agents) et permet d'offrir une transparence de l'exécution et des communications vis-à-vis de la mobilité (mobilité forte). De plus, l'approche réactive basée sur Rhum simplifie la tâche du programmeur, par élimination *de facto* les problèmes courants liés à la concurrence entre agents, et par utilisation d'un langage dédié, intégrant des constructions de haut niveau, pour gérer le parallélisme et la synchronisation.

D'autre part, nous avons montré comment une plate-forme d'agents mobiles peut être construite de façon modulaire (support de répartition Jonathan, modèle d'activité réactif Rhum, gestion de la mobilité d'objets par SMI), en analysant l'impact de la mobilité sur l'activité et les communications, et en justifiant nos choix par rapport au modèle agent en général, et au domaine applicatif visé. Cette architecture particulière offre un avantage en terme d'interopérabilité : au-delà de la

conformité MASIF offerte, il est envisageable de faire cohabiter - et interopérer - Mooréa avec d'autres modèles d'agents sur la même base SMI.

Outre les améliorations (e.g. évolution du langage Rhum vers une intégration avec Java, cf. Rejo) et évaluations de performances que nous allons réaliser sur Mooréa, les travaux futurs incluent l'application de la plate-forme à l'équilibrage de charge dynamique préemptif, notamment pour le support d'environnement d'exécution de services de télécommunication (projet ATHOS). Concernant l'approche réactive elle-même, nous envisageons son implémentation dans les couches inférieures (machine virtuelle, ou noyau de système d'exploitation).

Remerciements

SMI et Mooréa sont développés dans le cadre du projet ATHOS du programme européen ITEA. Le modèle réactif est développé en collaboration avec l'INRIA et l'Ecole des Mines de Paris (Centre de Mathématiques Appliquées).

8. Bibliographie

[ACO 01] ACOSTA BERMEJO, R., « Programming in REJO », à paraître dans *Réseaux et systèmes répartis - Calculateurs Parallèles*, numéro spécial *Evolutions dans le domaine des intergiciels*, Hermès éditeur, 2001.

[BER 92] G. BERRY, G. GONTHIER, « The Esterel Synchronous Language: Design, Semantics, Implementation », *Science of Computer Programming*, 19(2), 1992.

[BER 00] BERTIN V., POIZE M., PULOU J., SIFAKIS J., « Towards Validated Real-Rime Software », *12th Euromicro Conference on Real-Time Systems*, Stockholm, 2000.

[BOL 00] BÖLÖNI L., JUN K., PALACZ K., SION R., MARINESCU D., « The Bond Agent System and Applications », *actes de ASA/MA 2000*, Lecture Notes in Computer Science 1882, Springer, p. 99-112.

[BOU 96a] BOUSSINOT F., DOUMENC G., STEFANI J.-B., « Reactive Objects », *Annales des Télécommunications No 51*, 1996, p. 9-18 (aussi Rapport de recherche Inria 2664, 1995).

[BOU 96b] BOUSSINOT F., DE SIMONE R., « The SL synchronous language », *IEEE Transactions on Software Engineering*, Vol. 22 No 4, 1996, p. 256-266.

[DIL 00] DILLENSEGER B., « MobiliTools: An OMG standards-based toolbox for agent mobility and interoperability », *actes de 6th IFIP Conference on Intelligence in Networks (SmartNet 2000)*, Vienne, septembre 2000, Kluwer Academic Publishers, p. 353-366.

[FIP 98] FOUNDATION FOR INTELLIGENT PHYSICAL AGENTS, « FIPA 98 Specification », 1998, http://www.fipa.org/.

[FUG 98] FUGETTA A., PICCO G.-P., VIGNA G., « Understanding Code Mobility », *IEEE Transactions on Software Engineering*, vol. 24, No 5, 1998, p.342-361.

[GRA 97] GRAY D., KOTZ D., NOG S., RUS D., CYBENKO G., « Mobile Agents : the next generation in distributed computing », *actes de Second Aizu International Symposium on Parallel Algorithms/Archirtectures Synthesis (pAs '97)*, Fukushima, Japon, IEEE Computer Society Press, 1997, p. 8-24.

[HAL 97] HALLS D., « Applying Mobile Code to Distributed Systems », Doctoral dissertation, Computer Laboratory, University of Cambridge, 1997.

[HAY 98] HAYTON R., HERBERT A., DONALDSON D., « FlexiNet - A flexible component oriented middleware system », *actes de 8th ACM SIGOPS European Workshop: Support for Composing Distributed Applications*, Sintra, Portugal, 7-10 septembre 1998.

[HAZ 00] HAZARD L., SUSINI J.-F., BOUSSINOT F., « Programming with Junior », Rapport de recherche Inria 4027, 2000.

[INF 99] INFOWIN PROJECT, « Agents Technology in Europe, ACTS activities », ISBN 3-00-005267-4, 1999.

[IKV 98] IKV++, « Grasshopper Development System User's Guide », 1998.

[OBJ 97] OBJECTSPACE INC., « Voyager core package version 1.0 technical overview », 1997.

[OBJ 00] OBJECTWEB COMMUNITY, « Jonathan : a white paper », 6 octobre 2000, http://www.objectweb.org/.

[OMG 97] OBJECT MANAGEMENT GROUP, « Mobile Agent System Interoperability Facilities », TC document orbos/97-10-05, 1997.

[SUR 00] SURI N., BRADSHAW J., BREEDY M., GROTH P., HILL G., JEFFERS R., « Strong mobility and fine-grained resource control in NOMADS », *actes de ASA/MA 2000*, Lecture Notes in Computer Science 1882, Springer, p. 16-28.

[TRU 00] TRUYEN E., ROBBEN B., VANHAUTE B., CONINX T., JOOSEN W., VERBAETEN P., « Portable support for transparent thread migration in Java », *actes de ASA/MA 2000*, Lecture Notes in Computer Science 1882, Springer, p. 29-43.

[VAN 00] VAN THANH D., « A multi-agent system for user mobility support », *actes de IC-AI 2000*, H.R. Arabnia éditeur, CSREA Press, ISBN 1-892512-56-4, p. 143-149.

[WHI 94] WHITE J., « Telescript technology: the foundation for the electronic market place » *General Magic White Paper*, General Magic, 1994.

Article reçu le 5 février 2001
Version révisée le 4 octobre 2001
Rédacteur responsable : Jean-Paul Arcangeli

Bruno Dillenseger *est ingénieur et docteur de l'Université de Caen en informatique. Au sein du CNET puis de France Télécom R&D, il travaille depuis une dizaine d'années sur la synergie entre intergiciel et technologies agent. Auteur de plusieurs plates-formes, il*

s'intéresse à leur application dans le domaine des systèmes d'information, et des architectures et services de télécommunication.

Laurent Hazard *est ingénieur expert en systèmes distribués et travaille actuellement dans l'équipe "Architecture de systèmes répartis" à France Télécom R&D. Partisan de l'utilisation des formalismes dits réactifs ou synchrones pour la programmation de ces systèmes, il élabore en particulier des outils et des infrastructures d'exécution adaptés à ces langages.*

Anne-Marie Tagant *est ingénieur R&D depuis plus de trente ans à France Télécom R&D, et s'est consacrée ces dernières années à l'étude de la mobilité logicielle d'éléments d'applications réparties, en particulier aux techniques assurant la mobilité forte, et une grande transparence à la mobilité pour les utilisateurs.*

Huan Tran Viet *est diplômé de l'Institut de la Francophonie pour l'Informatique à Hanoï. Après des travaux sur la communication entre agents (cf. FIPA), il se consacre à l'introduction de mécanismes de support de la mobilité au sein des intergiciels dans le cadre d'une thèse à France Télécom R&D.*