République Algérienne Démocratique et populaire
Ministère de l'Enseignement Supérieur et de la Recherche Scientifique

Université des Sciences et de la Technologie Houari Boumédiène
Faculté de Génie Electrique

Département informatique

Thèse

Pour l'obtention du diplôme de Magister en informatique
Spécialité : I.A. & B.D.D. AV

Conception d'un environnement de développement d'applications réparties, concurrentes et mobiles

Présenté par : Mme BEBBOUCHI Sabrina

Devant le jury composé de :

Président du jury :	Mr. BADACHE	Professeur (USTHB)
Examinatrice :	Mme. ALIMAZIGHI	Maître de conférence (USTHB)
Examinateur :	Mr. CHALLAL	Docteur en informatique (USTHB)
Directeur de thèse :	Mr. AHMED-NACER	Professeur (USTHB)

2006-2007

Table des matières

Introduction générale

L'évolution de l'architecture des réseaux a permis de passer rapidement d'une informatique séquentielle et centralisée à une informatique concurrente, répartie et mobile et ce phénomène ne s'est pas arrêté à l'Internet mais il s'est étendu aux organisations et entreprises. Un autre phénomène aussi important concerne les technologies dites accessibles comme le WWW (World Wide Web) qui a introduit la création de nouveaux domaines d'applications ainsi que de nouveaux marchés. Ce phénomène a changé la nature et le rôle des réseaux, et particulièrement d'Internet.

De nos jours, les réseaux informatiques modernes constituent un média innovateur qui supporte les nouvelles formes de coopération et de communications entre les utilisateurs. Ce phénomène a stimulé les recherches dans les systèmes distribués. Cette nouvelle génération d'applications distribuées exploite la notion de « code mobile ». Ces applications sont appelées « Application code mobile » MCAs.

C'est ainsi, qu'aujourd'hui, la plupart des applications informatiques sont constituées de composants concurrents répartis et de nombreux systèmes sont mobiles. La construction de ces applications dans les contextes actuels (INTERNET, informatique nomade, ...) demande des méthodes et des outils adéquats. En effet, celles-ci doivent faire face à l'hétérogénéité matérielle et logicielle, ainsi qu'à de fortes variations (qualitatives et quantitatives) des conditions d'exécution. Ainsi, ces applications doivent être statiquement, voire dynamiquement, adaptables.

Dans cette thèse nous mettons en valeur, d'une part, l'état de l'art de la technologie des codes mobiles qui représente les langages de programmation et leurs supports, en utilisant différents critères d'appréciation de cette technologie et les problèmes à solutionner, des paradigmes de conception qui sont nécessaires à la construction d'un système logiciel, et de clarifier les relations existantes entre le génie logiciel et le nouveau domaine des applications mobiles. Nous citons par la suite, quelques travaux effectués dans ce domaine à la fois sur des méthodes de programmation et sur des outils d'analyse de programmation.

D'autre part, nous présentons notre approche d'environnement de développement d'applications réparties, concurrentes et mobiles, basée sur le modèle d'acteurs et intégrant les caractéristiques de concurrence et de mobilité du calcul formel CAP. Dans cette partie, nous représentons essentiellement les activités d'un acteur (la migration et la communication) en utilisant le digramme de séquence d'UML. De même, nous utilisons le calcul formel CAP (calcul d'Acteurs Primitifs) qui est basé sur le modèle d'acteurs, dans le but d'introduire ses

caractéristiques de concurrence et de mobilité dans les langage de programmation pour simplifier la conception des applications réparties et mobiles.

Notre mémoire est structuré en quatre principaux chapitres :

- Nous présentons dans le premier chapitre, intitulé ‘les concepts et technologies du code mobile’, les différentes significations de l’expression « code mobile », les technologies qui le supportent, les mécanismes de la mobilité et une présentation puis une comparaison des langages du code mobile.

- Dans le second chapitre, intitulé ‘présentation des paradigmes de conception et le domaine d’applications du code mobile’, nous exposons différents paradigmes de conception dédiés au code mobile ainsi que l’utilisation de ce dernier dans certains domaines d’application.

- Dans le chapitre trois, intitulé ‘le génie logiciel et le code mobile’, nous nous intéressons à l’implication de cette nouvelle technologie dans le domaine du génie logiciel en citant quelques travaux dans ce sens. Nous présentons aussi dans ce chapitre, le calcul formel CAP (Calcul d’Acteurs Primitifs) qui est un calcul dédié aux acteurs.

- Nous présentons dans le dernier chapitre, la problématique et notre approche qui porte sur la conception d’un environnement de développement d’applications distribuées, concurrentes et mobiles

Chapitre I :
Le code mobile : concepts et technologies

I.1) Concepts du code mobile

L'expression 'code mobile' possède plusieurs significations dans la littérature, voici quelques unes [Thor97] :

- Le terme code mobile décrit un programme pouvant se déplacer dans une collection de processeurs hétérogènes et exécuté avec une sémantique identique sur chacun de ces processeurs,

- … code mobile, une approche où les programmes sont considérés comme des documents, et peuvent être donc accessible, transmis et traité (évalué) comme n'importe quel document,

- Agents mobiles sont des objets contenant du code qui sont transmis entre des participants d'une communication dans un système distribué.

Mais le code mobile est généralement défini comme un programme pouvant se déplacer d'un site à un autre sur un réseau. Et informellement comme la capacité au changement dynamique du lien entre les fragments du code et le lieu de leur exécution. Les avantages du code mobile sont :

En terme d'efficacité : quand il y a des d'interactions répétées avec un site éloigné, il est plus efficace d'envoyer le processus (le calcul) sur le site éloigné que de le traiter localement. Cela est le cas quand le trafic du réseau est dense et que les interactions se composent en plusieurs petits messages.

En terme de simplicité et flexibilité : la maintenance d'un réseau peut être plus simple lorsque les applications sont logées sur un serveur et que les clients les installent eux-mêmes sur leur site à la demande. La nouvelle installation ou la mise à jour logicielle devient alors indépendante de la nature et du nombre de clients. Dans la plus part des cas, il est souvent possible de connaître en avance toutes les parties du code nécessaires à un site donné.

En terme d'espace de stockage : charger le code à la demande plutôt que d'avoir à dupliquer tous les programmes sur tous les sites, réduit de façon significative l'espace total de stockage requis.

L'habilité à reloger le code est un concept puissant qui est à l'origine d'un très intéressant domaine de développement. Le principal problème dans ce domaine est de supporter la migration des processus actifs et des objets par le système opérationnel. En particulier, la migration des processus concerne le transfert d'un processus d'un système opérationnel de la machine active à une autre machine. Les mécanismes de migration dirigent les relations entre le processus et l'environnement de son exécution.

Les techniques de migration ont été prises comme un point de départ pour le développement d'une nouvelle famille de systèmes contenant plusieurs formes de code mobile. Ces systèmes, appelés « Systèmes code mobile » (MCSs), exhibent différentes innovations tout en respectant les approches existantes :

Le code mobile est exploité sur l'échelle Internet : Au début, les systèmes distribués munis d'un processus ou objet de migration ont été destinés à être dans des réseaux à petite échelle. Ensuite, les MCSs ont été conçus pour opérer dans des environnements à grande échelle où les réseaux sont composés de hôtes hétérogènes, gérés par des autorités différentes et connectés par des liens avec des bandes passantes différentes.

La mobilité est sous le contrôle du programmeur : Le programmeur est préparé avec des mécanismes et des abstractions qui permettent d'assembler et de récolter les fragments de code à partir de nœuds distants.

La mobilité n'est pas performante seulement pour une charge équilibrée : Les MCSs sont adressés à une large catégorie de besoins et exigences, tel que le service clientèle, l'extension dynamique d'application fonctionnelle, l'autonomie et le support pour les opérations disconnectées (parallèles).

I.2) Les technologies du code mobile

Les technologies du code mobile fournissent tous les concepts et primitives qu'utilise le code, elles représentent les langages de programmation et leurs supports. La figure1 représente les technologies qui supportent le code mobile, dont la partie inférieure, juste sur le hardware, est constituée du Core Operating System COS. COS est la partie fournissant les fonctionnalités basiques du système opérationnel, tel que le fichier système, gestion de la mémoire et le support des processus (i.e un système d'exploitation). Les services de communication sont fournis par le Network Operating System (NOS). Les applications utilisant ces services communiquent, de façon explicite, avec le hôte cible. Le Computational Environment (CE) se trouve au dessus de la partie NOS de chaque hôte du réseau. Le but du CE est de fournir des applications avec la capacité de reloger dynamiquement les composants dans différents hôtes [Fugg98].

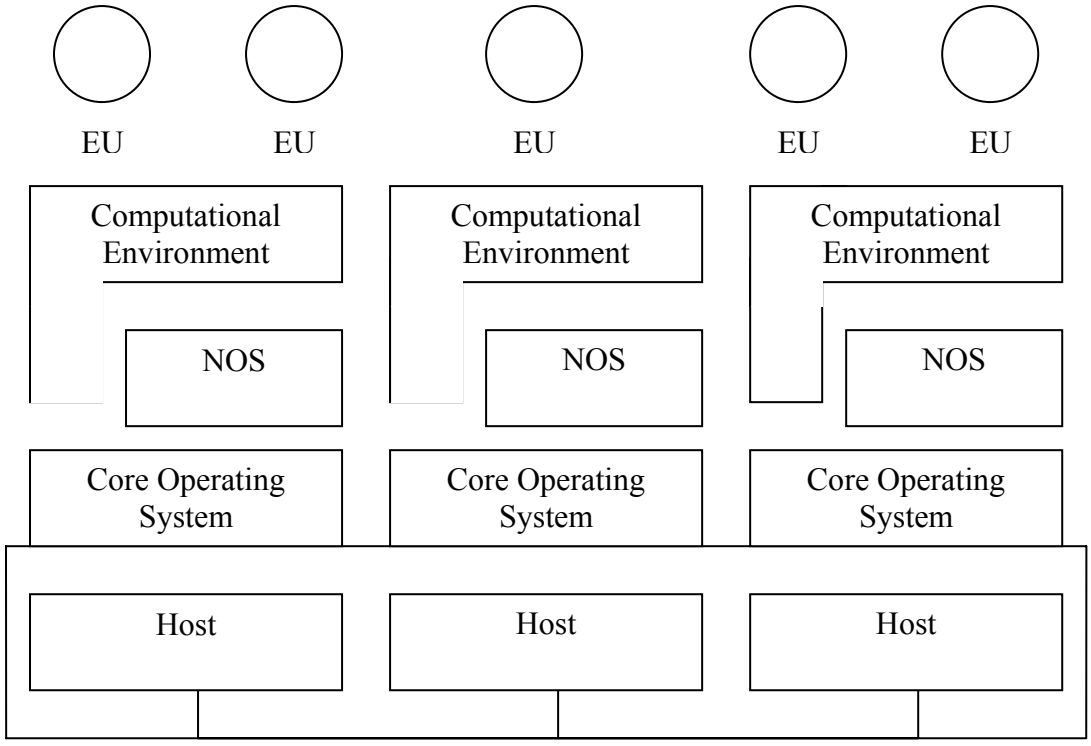


Figure1. Technologies supportant le code mobile

Les composants logés dans le CE se composent en : Unités d'exécution (EUs) et ressources. Les unités d'exécution sont des unités actives (éventuellement mobiles) exécutant un calcul, par exemple : processus, thread, agent mobile. Les ressources représentent les entités qui peuvent être partagées par différents EUs, tel qu'un fichier dans un fichier système. Donc le CE est à la fois le site où se loge l'EU et aussi l'interprète, exemple : site physique + JVM (Java Virtual Machine) [Krak99] [Gui01].

L'EU se compose d'un code segment (code exécutable) décrivant le calcul à faire et d'un contexte composé de l'espace de données et du contexte d'exécution. L'espace de données est un ensemble de références à des ressources qui peuvent être accédées par le EU. Ces ressources ne cohabitent pas nécessairement avec l'EU dans le même CE. Dans la figure2, il est illustré la structure interne des EUs [Fugg98].

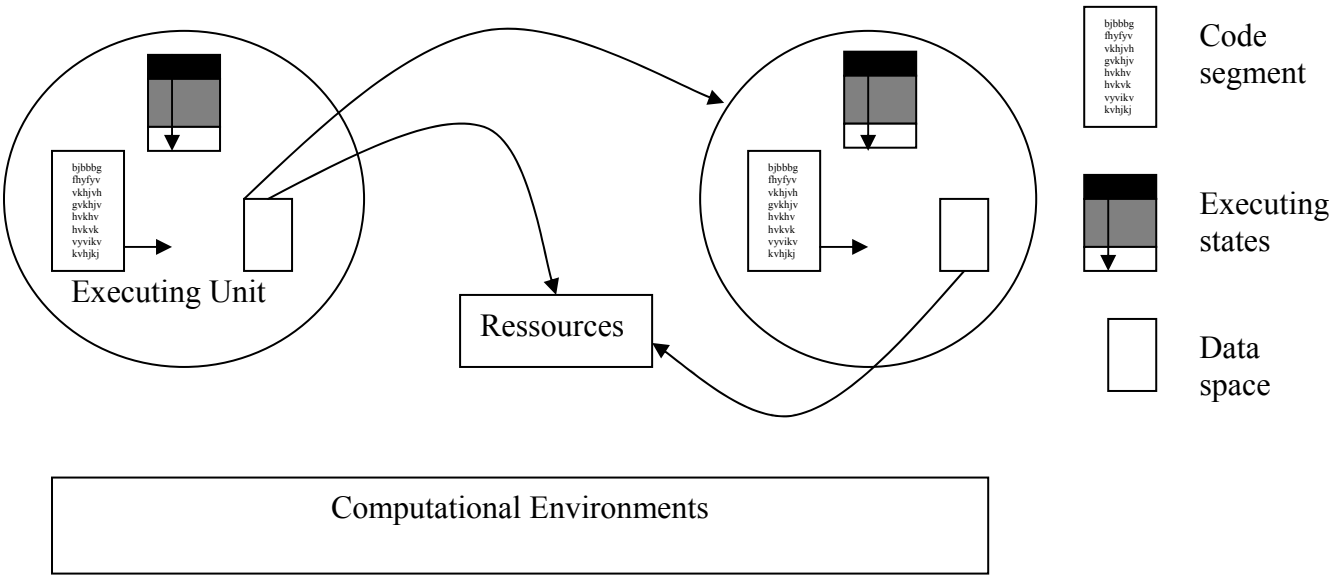


Figure 2. La structure interne d'une unité d'exécution

I.2.1) Les mécanismes de la mobilité

Dans les systèmes conventionnels, chaque EU est lié pendant toute la durée de sa vie à un seul CE. De même dans les environnements qui supportent les liens dynamiques, le code lié à l'EU appartient au CE local. Cela n'est pas vrai pour les MCSs. Dans les MCSs, le code segment, le contexte d'exécution et l'espace de données d'un EU peuvent être relogés dans différents CEs par le principe que chacun des composants d'EU peut se déplacer indépendamment [Oli03].

Le composant d'un EU qui a besoin d'être déplacée est déterminé par la composition orthogonale des mécanismes supportant la mobilité du code et du contexte d'exécution avec les mécanismes dédiés à la gestion d'espace de données [Fugg98].

I.2.1.1) La mobilité du code et de l'état d'exécution

Les MCSs existants offrent principalement deux formes de mobilité, qui sont caractérisées par la migration des constituants de l'EU. Ces formes de mobilité sont :

- La mobilité forte : elle représente l'habilité d'un MCS, appelé MCS fort, à permettre la migration du code et du contexte d'exécution d'un EU vers CE différent,

- La mobilité faible : elle représente l'habilité d'un MCS, appelé MCS faible, à permettre la migration du code à travers différents CEs. Le code peut être accompagné de quelques données d'initialisation, mais aucune migration du contexte d'exécution n'est impliquée.

Il existe une troisième forme de mobilité qui est la mobilité nulle. Elle concerne la communication par messages tel que ces messages contiennent que des données et non du code exécutable, par exemple cette forme est utilisée dans le paradigme client/serveur [Krak99].

La mobilité forte est supportée par deux mécanismes : la migration et le clonage à distance. Le mécanisme de la migration suspend un EU, le transmet au CE destinataire puis le réactive. La migration peut être soit proactive (initiative source) soit réactive (initiative autre). Dans la migration proactive, le temps et la destination de la migration sont déterminés par l'EU migrateur. Dans la migration réactive, le mouvement est dirigé par différents EU qui ont quelques types de parenté avec l'EU migrateur. Le mécanisme du clonage à distance crée une copie de l'EU sur un CE éloigné. Le mécanisme de clonage diffère du mécanisme de migration car l'EU original n'est pas détaché de son CE. Tout comme le mécanisme de la migration, le mécanisme de clonage peut être soit proactive ou réactive [Fugg98].

Les mécanismes supportant la mobilité faible ont la capacité de transférer le code à travers les CEs et le lient de façon dynamique à un EU actif ou l'utilisent comme un code segment pour un nouveau EU. Chaque mécanisme peut être classifié par rapport à la direction du transfert du code, la nature du code à transférer, la synchronisation impliquée et le temps d'exécution du code à destination [Fugg98].

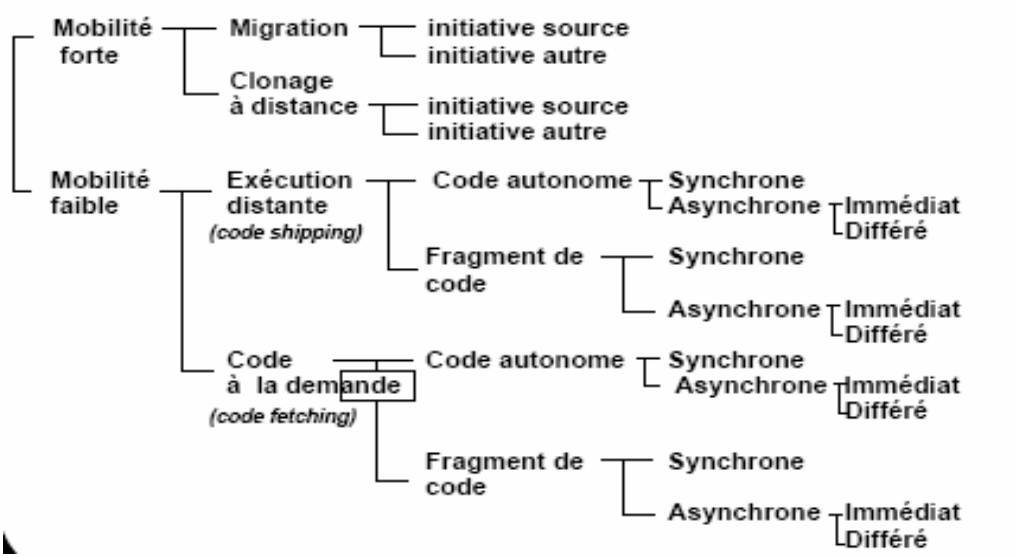


Figure3. La classification des mécanismes de mobilité

I.2.1.2) La gestion de l'espace des données

La ressource est modélisée par le triplet R = < I, V, T >[Fugg98], où I est un identifient unique, V est la valeur de la ressource et T est son type qui détermine la structure de l'information contenue dans la ressource ainsi que son interface. Le type de la ressource détermine si la ressource est transférable ou pas.

Les ressources peuvent être reliées à un EU par trois formes de lien, qui emprisonnent les mécanismes de gestion de l'espace de données qui peuvent être exploités lors de la migration. Le lien le plus fort est 'par identifient', dans ce cas, l'EU déclare qu'à n'importe quel moment, il est obligatoirement relié à une ressource donnée avec un identifient unique.

Un lien établi 'par valeur' déclare qu'à n'importe quel moment, la ressource est obligatoirement conforme avec un type donné, et sa valeur ne peut pas changer telle une séquence de migration. Ce type de lien est souvent exploité quand un EU est intéressé par le contenu d'une ressource et désire d'y accéder localement. Cette ressource doit avoir le même type et la même valeur que celles présentent dans la CE source.

Le lien le plus faible est 'par type'. Dans ce cas, l'EU déclare qu'à n'importe quel moment, la ressource liée est conforme à un type donné, l'identifient et la valeur actuelle ne sont pas pris en compte. Ce type de lien est exploité pour relier des ressources qui sont valables dans tout CE, tel que, les variables systèmes, les librairies ou les périphériques du réseau (imprimante, écran). Il est possible d'avoir différents types de lien pour une même ressource.

Des mécanismes de gestion d'espace de données dans la migration d'un EU, jaillissent deux clases de problèmes : le relogement de la ressource et la configuration du lien.

	Déplaçable	Non déplaçable
Par Identité	Par mouvement (référence globale : site, ref.locale)	Par référence globale (site, ref.locale)
Par Valeur	Par copie / clone (par mouvement, référence globale)	Par référence globale (site, ref.locale)
Par Type	Par équivalence (par déplacement, par copie, par ref.)	Par équivalence (référence globale)

Figure 4. Les mécanismes de gestion d'espace de données, de ressources et de liens

Les MCSs existants exploitent différentes stratégies surtout quand la gestion de l'espace des données est concernée. La nature de la ressource ainsi que le type du lien sont souvent déterminés par la définition du langage ou l'implémentation, que par l'application du programmeur, alors forcement les mécanismes sont exploités.

I.2.2) les langages du code mobile (MCLs) et leur comparaison

Afin d'exploiter le code mobile, il existe des mécanismes incorporés dans une nouvelle génération de langages de programmation appelés langages du code mobile (MCLs). Ces langages peuvent être regardés comme des langages dédiés aux systèmes distribués, où leur domaine d'application primaire est la création de MCLs sur des systèmes distribués à large échelle comme l'Internet. Ces langages diffèrent des autres langages ou middleware pour la programmation des systèmes distribués (par exemple CORBA) car ils modélisent explicitement le concept des environnements à exécution séparée et comment le code et les évaluations se déplacent dans ces environnements. Voici quelques exemples des MCLs les plus connus [Rom00] :

Agent Tcl

L'Agent Tcl, développé à l'Université de Darthmouth, contient un interpréteur Tcl étendu supportant la mobilité forte (la migration de l'EU entier) [Fugg98] [Cugo98]. Dans l'Agent Tcl, un EU est appelé « Agent ». Un script Tcl exécutable peut se déplacer d'un hôte à un autre avec une seule instruction 'jump'. Un jump gèle le contexte du programme d'exécution et le transmet à un hôte différent qui exécute le script d'exécution par l'instruction qui suit le jump.

Facile

Développé par ECRC à Munich, Facile est un langage fonctionnel qui étend le langage standard ML avec des primitives de distribution, de concurrence et de la communication. Le langage a été étendu afin de supporter la mobilité faible [Fugg98] [Cugo98].
. Les EUs, dans 'Facile', sont appelés « nœuds ». La communication suit le modèle de rendez-vous : l'envoyeur et le récepteur sont tout deux bloqués jusqu'à ce que la communication prend place.

Java

Développé par Sun Microsystems, Java a capturé toute l'attention et les perspectives sur le code mobile. Le but d'origine pour les concepteurs du langage était de fournir un langage orienté objet portable, clair et facile à apprendre. Il a été, par la suite, réorienté par la poussée d'Internet. Le compilateur Java traduit les programmes sources Java en un langage intermédiaire appelé « Java Byte Code ». Le code byte est interprété par la machine virtuelle Java (JVM) [Fugg98].

Un des facteurs clés du succès de Java est son intégration avec la technologie WWW (World Wide Web). Les navigateurs Web ont été étendus pour inclure une JVM. Les classes Java, appelées « applets », peuvent être téléchargées avec des pages HTML pour permettre une présentation active de l'information et un accès interactive à un serveur.

Telescript

Développé par General Magic, Telescript est un langage orienté objet conçu pour le développement d'un large domaine d'applications distribuées [Fugg98]. La sécurité, ajoutée à une concentration de mobilité forte a été un des facteurs clé dans la conception du langage. Telescript emploie un langage portable intermédiaire appelé « Low Telescript », qui est une représentation actuelle transmise entre les machines, 'les Telescript CEs'. Les machines sont en charge d'exécuter les Agents et les Places (les Telescript EUs). Les Agents peuvent se déplacer en utilisant l'opération 'go', qui implémente un mécanisme de migration proactive. L'opération 'send' est aussi utilisée pour implémenter le clonage proactif. Les Places sont des EUs stationnaires qui peuvent contenir d'autres EUs.

Obliq

Développé à DEC par Luca Cardelli, Obliq est un langage d'interprétation non typé, basé objet et lexicalement limité. L'EU Obliq peut demander l'exécution d'une procédure sur une autre machine d'exécution. Le code pour chaque procédure est envoyé à la machine destinataire et exécuté sur celle-ci par une nouvelle création d'EU. L'EU envoyeur est suspendu jusqu'à ce que l'exécution de la procédure se termine. Quand un EU sollicite l'exécution d'une procédure sur un CE étranger, les références aux objets locaux utilisés par la procédure sont automatiquement traduis en des références réseaux (références globaux) [Fugg98].

TACOMA

Dans TACOMA (Tromsø And Cornell Mobile Agents) [Cugo98], le langage Tcl est étendu pour inclure des primitives supportant la mobilité faible. Ces primitives permettent à un EU, exécutant un script Tcl, de solliciter l'exécution d'un autre script Tcl sur un hôte différent. Le code script ainsi que quelques données d'initialisation sont envoyés à un hôte de destination où ils sont évalués.

Safe-Tcl

C'est l'implémentation la plus répandue. TCL a été écrit par John Ousterhout comme un langage de commande simple, propre, interprété et intégrable, équipé de capacités

graphiques (TCL-Tk). Développé par les auteurs de 'the Internet MIME standard', Safe-Tcl est une extension de Tcl conçue pour supporter l'e-mail actif. Dans l'e-mail actif, les messages peuvent inclure le code à être exécuté quand le récepteur reçoit ou lit le message. C'est pour cela que dans Safe-Tcl, il n'y a pas de mécanismes de mobilité ou de communication au niveau du langage. Ces mécanismes doivent être réalisés en utilisant des supports externes tel que l'e-mail. L'originalité du Safe-TCL est de disposer de deux interpréteurs : un pour le code provenant d'hôtes de confiance, et un autre pour les autres hôtes. Ainsi, l'interpréteur ``sûr'' dispose de capacités étendues. A tout moment, une partie authentifiée du code peut être envoyée à cet interpréteur pour étendre de manière simple et temporaire le contexte d'exécution [Fugg98] [Cugo98].

Plus récemment, Safe-TCL a été adapté pour être utilisé sur le Web. Ces ``Tclets'' (prononcer ticlettes) sont téléchargées par le navigateur puis exécutées par un plug-in, disponible pour la plupart des navigateurs. A présent, la plupart des caractéristiques fondamentales de Safe-Tcl ont été incluses dans la dernière réalisation de Tcl/Tk.

I.2.3) Le tableau comparatif

Nous avons comparé ces langages sur différents aspects et le résultat donne le tableau suivant :

	Lien dynamique		**Mobilité**		**Compilation / Interprétation**	
	Code distant	**Ressource locale**	**Forte**	**Faible**	**Compilation**	**Interprétation**
Agent Tcl	**X**		**X**			**X**
Java	**X**			**X**	**X**	**X**
Telescript	**X**	**X**	**X**			**X**
Obliq				**X**		**X**
Tacoma	**X**			**X**		**X**
Safe-Tcl				**X**		**X**
Facile		**X**		**X**	**X**	**X**

I.2.4) Discussion et comparaison

Comme les langages Java, TACOMA et Agent Tcl utilisent le lien dynamique du code distant[1]. Ils sont les mieux adaptés à l'approche « code à la demande ». Quant au langage Telescript, il permet la configuration des deux types de lien dynamique (code distant et ressource locale[2]). La configuration du lien dynamique du code, dans ce langage, se fait par les mécanismes de package et celle du lien dynamique des ressources se fait par les mécanismes d'emplacement.

[1] Le lien dynamique du code distant permet aux programmeurs d'implémenter des MCAs basées sur l'approche 'code à la demande'. Ces MCAs représentent des applications qui téléchargent dynamiquement leur code à partir du réseau suivant différentes stratégies.

[2] Quand un EU se déplace d'un CE à un autre, il doit être capable d'accéder aux ressources se trouvant dans le CE de destination. Les ressources doivent être liées à la représentation interne de leurs EUs. Par exemple ces ressources peuvent être des fichiers.

Du point de vue sécurité, seul Telescript dispose des plus puissants mécanismes qui la supportent. Par contre, le langage Obliq n'en dispose pas de façon explicite. La solution adoptée par les langages Agent Tcl et TACOMA n'est pas adéquate pour une application code mobile complexe. Ces langages fournissent des mécanismes qui acceptent les EUs mobiles qui correspondent uniquement à des hôtes sélectionnés et ne fournissent aucun mécanisme qui gère les différents EUs et les droits d'accès.

La mobilité a introduit des options additionnels pour la compilation /interprétation. Le code peut être interprété ou compilé soit sur la machine expéditive, soit sur la machine réceptrice. Les MCLs utilisent une stratégie commune qui est l'adaptation d'une approche hybride qui consiste en la compilation de programmes sources en un langage intermédiaire qui est utilisé pour la transmission et l'interprétation sur la machine cible. Le langage 'Facile' supporte tous les modèles de compilation et d'interprétation. Le langage Java utilise un langage intermédiaire le 'Java Byte Code' qui est interprété par la machine virtuelle 'Java Virtual Machine'. Ce code peut être compilé s'il est exécuté pour la première fois. Le même scénario est appliqué au langage Telescript tel que le langage intermédiaire utilisé est le 'Low Telescript' sauf qu'il est strictement interprété et non compilé, tout comme les langages TACOMA, Agent Tcl, Safe-Tcl et obliq.

I.3) Conclusion

Le code mobile consiste en la capacité au changement dynamique du lien entre les fragments du code et le lieu de leur exécution. Cette nouvelle technologie a induit de nouveaux langages de programmation et leurs supports. Les langages du code mobile donnent une nouvelle orientation aux langages de programmation destinés aux systèmes distribués. Nous avons définit, dans un premier temps, le concept du code mobile puis nous avons vu ses différentes technologies où nous avons analysé un ensemble de langages dédiés au code mobile, en proposant un ensemble initial de concepts qui ont été utilisés pour les évaluer et les comparer.

Chapitre II:
Présentation des paradigmes de conception et le domaine d'applications du code mobile

II.1) Introduction

Les technologies du code mobile sont les seuls ingrédients nécessaires à la construction d'un système logiciel. Le développement d'un logiciel est un processus complexe où il existe une variété de facteurs devant être pris en compte : technologie, organisation et méthodologie. Le but de la conception est la création d'une architecture logicielle, qui peut être définie comme la décomposition d'un système logiciel en termes de composants logiciels et de leurs interactions. Le paradigme de conception n'est pas nécessairement induit par la technologie (le langage) utilisée pour développer le système logiciel, il représente conceptuellement une séparation d'entité [Ghe98].

Les approches traditionnelles de la conception des logiciels sont insuffisantes lors de la conception des applications distribuées à large échelle qui exploitent le code mobile et la reconfiguration dynamique des composants logiciels. Dans ces cas, les concepts d'allocation, de distribution des composants parmi les allocations et la migration des composants à différentes allocations doivent être pris en compte explicitement durant le stage de la conception.

II.1.1) Les concepts de base

Avant d'introduire les paradigmes de conception, il est nécessaire de présenter quelques concepts de base qui représentent une abstraction des entités qui constituent un système logiciels, tel que les fichiers, les valeurs des variables, le code exécutable ou les processus. En particulier, trois concepts architecturaux qui sont : composants, interactions et emplacements [Fugg98].

- les composants sont des constituants d'une architecture logicielle. Ils peuvent être divisés en :

- des composants code : qui encapsulent la connaissance-du-comment exécuter un calcul particulier,
- des composants ressources : qui représentent les données et les périphériques utilisés durant le calcul,
- des composants de calcul : qui sont des exécuteurs actifs capable d'exécuter le calcul, comme spécifié par la connaissance-du-comment correspondante.

- Les interactions sont des évènements qui impliquent deux ou plusieurs composants, par exemple un message échangé entre deux composants de calcul.

- Les emplacements abritent les composants et supportent l'exécution des composants de calcul. Les interactions entre composants résidents au même endroit (emplacement) sont considérées moins coûteux que ceux entre des composants résidents dans différents emplacements.

II.1.2) Les paradigmes de conception pour le code mobile

Les paradigmes de conception sont décrits en terme de groupes d'interaction qui définissent le relogement et la coordination entre les composants ayant besoin d'exécuter un service. En plus du concept client/serveur, trois principaux paradigmes de conception exploitant le code mobile ont été définis : exécution à distance (remote evaluation), code à la demande (code on demand) et agent mobile (mobile agent). Ces paradigmes sont caractérisés par l'emplacement des composants avant et après l'exécution d'un service par le composant de calcul qui est responsable de l'exécution du code et par l'emplacement où le calcul du service se fait.

Dans le paradigme client/serveur, toutes les données et la connaissance sur l'exécution d'un service ainsi que le composant d'évaluation se trouvent dans le serveur. Donc quand un client demande l'exécution d'un service au serveur. Ce dernier l'exécute et envoie le résultat au client. Par contre, dans les paradigmes d'évaluation distante et du code à la demande c'est la connaissance nécessaire à l'exécution du service qui se déplace. Sauf que dans le paradigme code à la demande, l'exécution du service peut se faire au niveau du site demandeur. Et enfin le paradigme agent mobile est le seul à permettre la migration du composant d'évaluation [Fugg98] [Krak99].

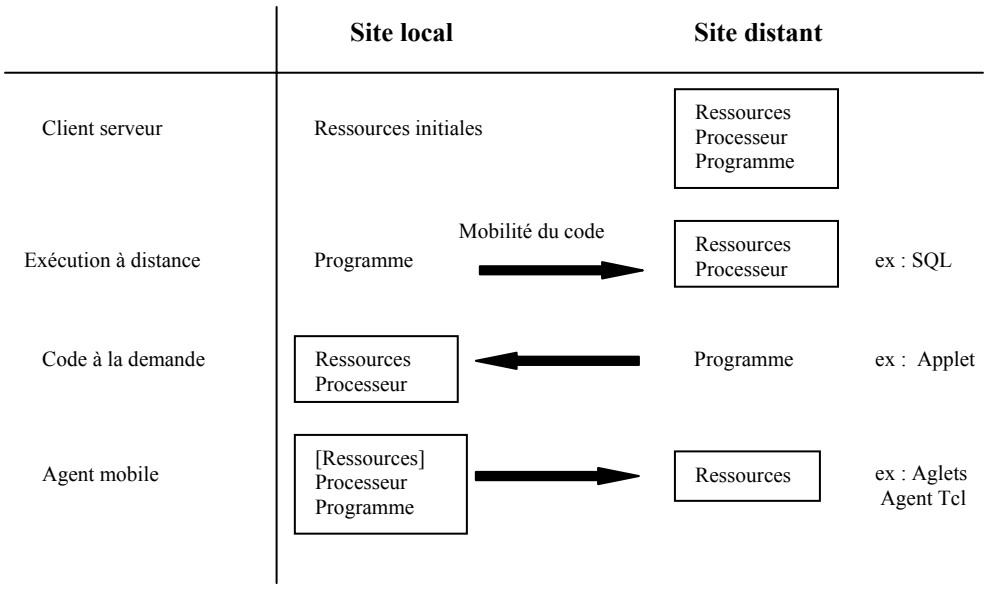


Figure5. Les paradigmes de conception pour le code mobile

La plupart des paradigmes connus sont statiques en respectant le code et l'emplacement. Une fois créés, les composants ne peuvent changer ni leur emplacement ni leur code durant toute leur vie. Donc les types d'interaction et leur qualité (locale ou distante) ne peuvent pas changer. Les paradigmes du code mobile dépassent ces limites en fournissant un composant mobile. Par le changement de leur emplacement, les composants peuvent changer dynamiquement la qualité de l'interaction, en réduisant ses coûts. A cette fin, les paradigmes REV et MA permettent l'exécution du code sur un site distant, entouré d'interactions locales avec des composants logés dans ce site. En plus, le paradigme COD permet aux composants de calcul d'amener le code à partir d'autres composants distants, fournir un moyen flexible pour étendre dynamiquement leur comportement et les types d'interaction qu'ils supportent.

II.2) Les applications du code mobile

L'intérêt dans le code mobile n'est pas motivé par la technologie d'Internet mais par les bénéfices qu'il est supposé fournir, par la capacité d'un nouveau moyen de construction d'applications distribuées et éventuellement par la création de nouvelles applications. Les avantages attendus par l'introduction du code mobile dans les applications distribuées sont invoqués particulièrement dans quelques domaines d'application spécifiques. Afin de comprendre le code mobile, il est important de distinguer clairement entre l'application et le paradigme utilisé pour sa conception ou la technologie utilisée pour son implémentation.

II.2.1) La clé des bénéfices du code mobile

Une activité majeure fournie par le code mobile est le service clientèle. Dans les systèmes distribués conventionnels construis suivant un paradigme CS, les serveurs contiennent un ensemble fixe de services accessible à partir d'une interface statiquement définie. Souvent, cet ensemble de services ou leur interface ne sont pas appropriés aux besoins imprévus du client. Une solution à ce problème est d'équiper le serveur de nouvelles fonctionnalités, alors augmenter sa complexité et son volume sans augmenter sa flexibilité. Par contre, l'habilité à demander l'exécution distante du code, aide à accroître la flexibilité du serveur sans affecter son volume ou sa complexité.

Le code mobile est prouvé utile dans le support des dernières phases du processus de développement d'un logiciel, appelé déploiement et maintenance. Il aide à fournir une automatisation plus sophistiquée pour le processus d'installation. Dans une application distribuée conçue avec des techniques conventionnelles, une nouvelle fonctionnalité a besoin d'être introduite par la réinstallation ou la structuration de l'application dans chaque site. Ce processus peut être long et pire si la fonctionnalité n'est pas fondamentale pour l'activité de l'application. L'habilité à solliciter à la demande le lien dynamique du fragment du code en implémentant la nouvelle fonctionnalité fournit différents bénéfices. Premièrement, tout changement va être centralisé dans le dépôt du code serveur, où la dernière version est toujours présente et consistante. De plus, les changements ne vont pas être exécutés proactivement par un opérateur dans chaque site, cependant ils vont être exécutés réactivement par l'application elle même, qui va solliciter automatiquement la nouvelle version du code dans le dépôt central. En suite, les changements peuvent être propagés dans un chemin paresseux, en concentrant l'effort seulement où il y a réellement un besoin.

La technologie et les concepts du code mobile comprennent une notion d'autonomie des composants de l'application. L'autonomie est une propriété très utilisée pour les applications qui utilisent une communication d'infrastructure hétérogène où les nœuds du réseau peuvent être connectés par une variété de liens physiques avec des performances différentes. Ces différences doivent être prises en compte pendant la phase de conception. Un

composant autonome encapsule tout état impliquant un calcul distribuée, et peut être facilement tracé, pointé et retrouvé localement sans avoir besoin de connaître l'état global.

Un autre avantage issu de l'introduction du code mobile dans une application distribuée est *la flexibilité de la gestion des données* et *le protocole d'encapsulation*. Dans les systèmes conventionnels, quand les données sont échangées entre les composants d'une application distribuée, chaque composant possède le code décrivant le protocole nécessaire à l'interprétation correcte de la donnée. Le code mobile offre des solutions plus efficaces et flexibles. Par exemple, si des protocoles sont rarement modifiés et sont faiblement couplés avec les données, une application peut télécharger le code qui implémente un protocole particulier seulement quand les données impliquées dans le calcul en ont besoin. Si au lieu que les protocoles soient étroitement couplés avec les données, ils les accompagnent, les composants peuvent échanger des messages composés par les données et le code dont il a besoin pour accéder et gérer de telles données [Fugg98] [Src03].

II.2.2) Quelques domaines d'application pour le code mobile

Dans ce qui suit, il est décrit quelques domaines d'application du code mobile où est donné une idée sur l'utilisation des concepts vus plus haut [Fugg98].

II.2.2.1) La reconstitution de l'information distribuée

Les applications de reconstitution de l'information distribuée rassemblent l'information avec des critères spécifiques d'un ensemble d'informations sources dispersées dans le réseau. Les informations sources à être visitées peuvent être définies statiquement ou dynamiquement durant le processus de regroupement. Cela est un vaste domaine d'application, entouré par diverses autres applications. Par exemple, l'information à reconstituer peut concerner la liste de toutes les publications d'un auteur donné. Le code mobile peut améliorer efficacement, par la migration du code qui exécute le processus de recherche, l'information de base à être analysée. Ce type d'application a été souvent considéré comme « l'applications tueuse » en motivant une conception basée sur le paradigme MA. Cependant, les analyses faite pour la détermination du trafic du réseau dans quelque cas typiques prouvent que, d'après les paramètres de l'application, le paradigme CS peut encore être parfois le meilleur choix.

II.2.2.2) Les documents actifs

Dans les applications des documents actifs, les données passives comme e-mail ou les pages Web, sont enrichis par la capacité d'exécution des programmes qui sont reliés avec le contenu du document, ce qui permet l'évolution de la présentation et l'interaction. Le code mobile est fondamental pour ces applications le temps qu'il permet l'encastrement (l'intégration) du code et l'état dans les documents et supporte l'exécution du contenu dynamique durant la réalisation du document. Un exemple paradigmatique qui est une application utilisant les formes graphiques pour composer et soumettre les questions à une base de données distante. L'interaction avec l'utilisateur est modélisée par l'utilisation du paradigme COD, i.e, l'utilisateur demande le composant document actif au serveur puis exécute quelques calculs en utilisant le document comme interface. Un choix typique est la combinaison de la technologie WWW et les Applets Java.

II.2.2.3) Les services de télécommunication avancés

Supports, gestion et estimation des services de communication avancée tel que la vidéoconférence, la vidéo à la demande ou le telemeeting, requièrent un « middleware » spécialisé fournissant des mécanismes pour la reconfiguration dynamique et la customisation de l'utilisation. Une classe particulière des services de télécommunication avancée est celle qui supporte les utilisateurs mobiles.

II.2.2.4) La configuration et le contrôle du périphérique distant

Les applications de contrôle de périphérique distant sont dirigées vers la configuration d'un réseau de périphériques et le signalement de leurs états. Le code mobile peut être utilisé pour la conception et l'implémentation des composants de signalisation qui co-habiteront avec les périphériques concernés par la surveillance. En plus, l'envoi des composants de gestion à des sites distants peut améliorer la performance et la flexibilité.

II.2.2.5) La coopération et la gestion Workflow

Les applications de gestion Workflow supportent la coopération des personnes et des outils impliqués dans un processus d'ingénierie ou de métier. Le Workflow définit quelles activités doivent être entreprises pour accomplir une tâche donnée et aussi comment, où et quand ces activités entraînent chaque partie. Le code mobile peut être utilisé pour fournir un support pour la mobilité des activités qui encapsulent leur définition et leur état. Par exemple, un composant mobile peut encapsuler un document texte qui subit différentes révisions. Le composant maintient l'information concernant l'état du document, les opérations légales sur ses contenus et le prochain pas dans le processus de révision.

II.2.2.6) Les réseaux actifs

L'idée des réseaux actifs a été proposée récemment comme une signification pour introduire la flexibilité dans les réseaux et fournir plus de mécanismes puissant afin de programmer les réseaux d'après les applications voulues. L'approche switch programmable est à la base un exemple du paradigme COD et elle fournit l'extensibilité dynamique des périphériques réseaux à travers le lien dynamique du code. L'approche capsule propose d'attacher à tout paquet découlant du réseau un code décrivant un calcul qui doit être exécuté sur les données du paquet. Les réseaux actifs utilisent les avantages fournis par le code mobile en termes de déploiement et maintenance, de services clientèle et du protocole d'encapsulation.

II.2.2.7) Le commerce électronique

Les applications du commerce électronique permettent aux utilisateurs d'exécuter des transactions commerciales à travers le réseau. L'environnement d'application est composé de différentes entités commerciales indépendantes. Une application peut impliquer une négociation avec des entités distantes et peut requérir l'accès à l'information qui est développée continuellement. Dans ce contexte, il y a le besoin de customiser le comportement des parties impliquées afin d'unifier le protocole de négociation particulier. Plus encore, il est désirable de déplacer les composants d'application renfermant l'information nécessaire à la transaction. Ces problèmes font que ce genre d'applications fait appel au code mobile.

Telescript a été conçu explicitement pour supporter le commerce électronique. Pour cette raison, le terme agent mobile est souvent relié avec le commerce électronique.

II.3) Conclusion

Dans cette partie nous avons exposé les différents paradigmes du code mobile ainsi que les applications distribuées utilisant le code mobile. Les paradigmes sont utilisés dans le but de guider la conception des applications distribuées. Ces paradigmes fournissent un composant mobile qui permet de dépasser les limites des anciens paradigmes dont les composants ne pouvaient changer ni leur emplacement ni leur code durant toute leur vie.

Les applications représentent des solutions à des problèmes spécifiques, les avantages attendus par l'introduction du code mobile dans les applications distribuées sont exprimés surtout dans leur phase de déploiement et de maintenance, par exemple, il aide à fournir une automatisation plus sophistiquée pour le processus d'installation.

Chapitre III :
Le génie logiciel et le code mobile

III.1) Introduction

L'application des concepts du code mobile au domaine du génie logiciel n'est pas évidente. En effet, la part croissante de l'informatique dans l'industrie rend crucial le problème de la validité et de la sûreté des logiciels. Le caractère distribué et les contraintes temporelles augmentent la difficulté de ce problème. La mise au point des applications distribuées demande des méthodes et des outils adéquats [Kta03].

Le besoin de flexibilité et d'adaptabilité des réseaux hétérogènes et de la variation des conditions d'exécution augmente, d'une façon significative, la complexité et le coût du développement logiciel et de la maintenance. Afin de faciliter ces activités, des études sont menées à la fois sur des méthodes de programmation et sur des outils d'analyse de programmation. Parmi ces travaux, nous citons dans un premier temps, ceux qui s'appuient sur un modèle de programmation (JavAct) à base d'objets actifs « le modèle d'acteur » qui constitue un support simple, réaliste et bien adapté à la mise en œuvre d'applications réparties à grande échelle ou mobile. L'objectif de ces travaux est de proposer et de valider une infrastructure logicielle permettant le développement et l'exploitation (exécution, maintenance) d'applications réparties à grande échelle et mobile. En second lieu, ceux qui consistent à trouver rapidement des solutions pour supporter l'interopérabilité des environnements répartis hétérogènes. Les préoccupations principales sont de réutiliser et d'assembler des composants logiciels, indépendamment de leurs langages d'implémentation, de leurs systèmes d'exploitation et de leurs sites d'exécution. L'adaptabilité dynamique du bus logiciel rend alors possible l'instauration de nouvelles stratégies d'exécution (e.g. migration des objets). OpenCorba est ce bus logiciel réflexif *adaptable*. Et en enfin, nous présentons quelques travaux sur l'utilisation de XML dans le génie logiciel pour l'intégration des outils de développement.

III.2) Le langage JavAct

Dans une approche génie logiciel, un outil d'aide à la conception et la réalisation d'applications réparties à base d'objets concurrents est en cours de développement, par l'équipe ingénierie des applications mobiles (I.A.M.) de l'I.R.I.T., exploitant le modèle d'acteur, afin de pouvoir spécifier et concevoir des applications, puis générer du code JavAct, de manière automatique ou assistée.

Pour faciliter la mise au point de telles applications, l'équipe ingénierie propose des mécanismes d'analyse statique des programmes pour assurer que chaque message émis sera traité. Ces analyses sont définies pour un modèle formel permettant d'exprimer l'architecture concurrente, répartie et mobile de l'application indépendamment des aspects fonctionnels. Cette architecture est ensuite instanciée dans un langage de programmation sans perdre les propriétés validées. Ce modèle formel de la répartition ainsi que la mobilité et la bibliothèque Java qui en est dérivée seront explicités ci-après.

Il faut également noter que la mise au point des programmes concurrents répartis est nettement plus complexe et source d'erreurs que les programmes séquentiels centralisés.
L'utilisation d'approches plus abstraites de la programmation (telle les agents mobiles) combinées avec des mécanismes d'analyse statique des programmes permet de réduire significativement la surcharge liée à l'introduction de la concurrence, de la répartition et de la mobilité.

L'objet de ces travaux est la définition et la réalisation d'un environnement de programmation en langage JAVA pour cette forme d'applications. Dans un premier temps, nous présenterons le modèle d'acteurs avec ses caractéristiques et les plates-formes utilisées pour leur mise en œuvre, puis l'API JavAct pour les acteurs distribués et mobiles et enfin nous parlerons de l'évolution de JavAct qui est basée sur l'extension du calcul formel CAP.

III.2.1) Les acteurs

Dans une première étape, l'équipe s'intéresse à la phase de description des activités d'un acteur ainsi qu'à celles de sa communauté. Pour cela, il existe des diagrammes d'évènements de Clinger, qui ne prennent pas en compte les changements de comportements. D'autre part, en UML, il existe les diagrammes de séquence ainsi que les diagrammes état-transition, mais ils n'offrent qu'une vue partielle de l'activité de l'acteur. Le travail consiste à étudier l'adéquation d'UML à la description complète d'une application en acteurs, et son enrichissement par de nouvelles constructions à travers un méta-modèle UML [Arc03] [Act00]

Le modèle des acteurs a été proposé par Hewitt et Agha. Un acteur est une entité autonome pourvue de son propre espace mémoire contenant son code d'exécution et ses données. Il est caractérisé par un certain *comportement* qui peut évoluer au cours de son existence [Agh01]. Il peut communiquer avec d'autres acteurs au moyen de messages asynchrones, et possède donc une boîte aux lettres qu'il consulte à chaque changement de comportement. Lorsqu'il accepte un message, il peut :

– envoyer de nouveaux messages à d'autres acteurs ou à lui-même ;
– changer de comportement;
– créer de nouveaux acteurs.

Les plates-formes de mise en œuvre des acteurs

Nous présentons brièvement, dans ce qui suit, quelques plates-formes existantes pour la programmation des acteurs ou agents mobiles [Arc01] [Gnu03] :

ProActive :

ProActive est une librairie Java pour le calcul parallèle, distribué et concurrent. Elle est basée sur la librairie standard RMI Java et elle est faite des classes de Java standard qui ne requièrent aucun changement de la machine virtuelle Java, de la préexécution ou de la modification du compilateur. C'est du code Java standard.

ProActive est basée sur un simple protocole Méta-Objet, ce qui fait que la librérie est elle-même extensible, et fait du système un système ouvert pour des adaptations et des optimisations.

Aglets :

Aglets est une Java API développée par IBM Corporation pour le développement des applications Internet basées sur les agents mobiles. Elle repose sur un protocole spécifique le transfert d'aglet (ATP), et sur un chargeur de classe spécialisé et un modèle de sécurité.

Aglets sont des agents mobiles exécutés à l'intérieur d'un emplacement (un contexte dans la terminologie d'Aglets), qui sont capables de se déplacer d'une manière réactive ou proactive. La mobilité est faible et les méthodes à être exécutées en dispatching ou en entrée et en retour doivent être programmées. Une Aglet peut communiquer avec une autre de manière sychrone ou asynchrone avec l'envoi du prochain message ; la coordination de multiples Aglets est aussi possible grâce aux autorisations multiples.

Voyager :

Voyager est un middleware développé par ObjectSpace. Il n'est pas juste un système d'agents mobiles mais une plate-forme basée-Java développée pour la construction des applications distribuées facile à construire, à développer et à déployer. Il fournit un ORB compatible avec CORBA, RMI et DCOM et aussi s'appuie sur l'interopérabilité.

Voyager permet la mobilité de l'objet et supporte les agents mobiles autonomes (réactifs ou proactifs). Différents protocoles de communication entre les agents sont utilisés. La localisation est basée sur le premier protocole optimisé.

Obliq :

Obliq, développé à Digital SRC est un langage d'interprétation orienté objet et concurrent et sa conception est antérieur au développement de Java. Il est basé sur le champ lexical distribué, la copie distante superficielle et le renommage. La distribution doit être formulée et les calculs sont transparents dans le réseau. N'importe quel objet est un objet réseau qui peut être accessible directement (argument ou résultat d'une méthode distante) ou par un serveur de noms. Son but par rapport aux réseaux à grande échelle, est d'améliorer le contrôle de la complexité et de la sûreté des applications, mais peut aussi affecter la performance.

Les primitives d'objet Obliq sont conçues pour être simples et performants, avec une relation cohérente entre leurs sémantiques locales et distribuées. Les objets Obliq sont des collections de champs nommés, avec quatre opérations de bases : selection/invocation, updating/overriding, cloning, et redirection. Chaque objet est potentiellement et de manière transparente un objet réseau.

Discussion et comparaison :

ProActive est particulièrement adaptée au développement d'applications réparties sur l'Internet grâce notamment à la réutilisation de code initialement non réparti, à une synchronisation automatique et à la possibilité de faire migrer des activités d'une machine à l'autre. Obliq est lui aussi bien adapté aux réseaux à grande échelle, son but est d'améliorer le contrôle de la complexité et de la sûreté des applications, mais cela peut aussi affecter la performance. Il reste néanmoins peu fiable d'un point de vue sécurité car il ne dispose pas de mécanismes explicites pour la sécurité. Voyager et Aglets supportent les agents mobiles autonomes (réactifs ou proactifs) et la mobilité faible. Contrairement à Aglets, Voyager utilise plusieurs protocoles pour la communication entre les agents mobiles ce qui permet une localisation optimale. Ils ont été développés pour la conception d'applications distribuées facile à construire, à développer et à déployer.

III.2.2) JavAct : une API pour les acteurs distribués et mobiles en Java

Grâce à son orientation objet et depuis qu'il offre des outils et abstractions pour traiter avec l'hétérogénéité, la concurrence, la sécurité et les interactions distantes, Java supporte la plus part des plates forme existantes dans le domaine.

Afin de supporter et de valider ces travaux, JavAct est une API développée pour la programmation d'applications concurrentes et réparties en Java, à base d'acteurs adaptables et mobile. JavAct est une librairie Java standard pour la programmation basée sur les acteurs (actor-based) des applications concurrentes, distribuées et mobile. JavAct est un successeur de la plate-forme dédiée à la programmation distribuée d'acteur qui a été développée dans ce groupe à la fin des années 80, basée sur un langage fonctionnel et sur une machine virtuelle distribuée.

JavAct offre un haut niveau d'abstraction par rapport aux technologies plus rudimentaires (processus légers, synchronisation, socket, RMI, Corba,…), et fournit des mécanismes pour la création d'acteurs, leur changement de comportement, leur répartition, leur mobilité et leur communications (locales ou distantes). JavAct a été conçu afin d'être facile à utiliser, minimale dans le code et maintenable à moindre coût. Dans la version actuelle de JavAct, il n'y a pas de préprocesseur et tous les outils standard de l'environnement Java peuvent être utilisés [Iri05].

Un ensemble d'emplacements (qui peuvent changer dynamiquement) constitue un domaine sur lequel des applications tournent. Un emplacement (machine virtuelle) est soit un site physique ou logique ; il peut être vu comme un serveur d'acteurs fournissant un environnement et des ressources pour l'exécution d'un acteur. JavAct est portable, et aujourd'hui s'exécute principalement sur un réseau de Sun workstations avec Solaris. Il peut être utilisé par un programmeur moyen dans Java qui possède quelques connaissances en les acteurs [Arc03]

III.2.2.1) Les abstractions de la programmation JavAct

La programmation des acteurs avec JavAct consiste essentiellement en la définition des comportements. Cela donne par extension la classe comportement (pour le programmeur, c'est la classe principale de la librairie JavAct) avec des méthodes spécifiques pour la programmation de l'acteur : les comportements JavAct sont des objets séquentiels et chaque méthode publique est une réponse possible à un message.

La classe comportement définit des méthodes pour la création et la distribution, la communication, le changement du comportement et la mobilité [Arc01] :

- RefActor createOn(Behavior b, Place p) permet la création d'un acteur sur l'emplacement *p* à partir d'un comportement *b* et retourne la référence de l'acteur ;
- RefActor create(Behavior b) abstrait l'emplacement dans la création de l'acteur ;
- Void send(Message m, String methodName, RefActor act) permet d'envoyer un message *m* d'une façon asynchrone, destiné à être exécuter avec la méthode du comportement *methodName*, à *act* ;
- void become(Behavior b) définit le prochain comportement de l'acteur (si elle n'est pas utilisée, le comportement de l'acteur reste inchangé) ;
- void become(Behavior b, Place p) permet la mobilité proactive.

L'acteur représente des objets réseau qui peuvent être référencés à partir de n'importe quel emplacement du réseau. Les messages peuvent être envoyés localement ou non dans un emplacement d'une façon transparente. Quand les comportement sont des arguments d'interactions distantes (création distante, le comportement de l'envoi, la mobilité), ils sont transmis par valeur, mais les acteurs sont transmis par référence : les acteur ne peuvent pas être déplacés par valeur. Donc aucune crainte n'est nécessaire lors de la programmation distribuée et des acteurs mobiles dans JavAct.

Les programmes JavAct sont des ensembles de classes définissant les comportements et les messages (par l'implémentation de l'interface *Message*) ; un programme principal est utilisé pour lancer l'application.

III.2.2.2) Le degré de mobilité

Dans un sens, le mécanisme de mobilité de JavAct (lié au changement du comportement) fournit une forte mobilité dans Java, puisque les acteurs mobiles reprennent l'exécution au même point où ils se sont arrêtés.

La mobilité peut être entièrement immédiate ou non cela dépend du caractère de la méthode du comportement. Cette méthode ne peut être suspendue jusqu'à ce qu'elle finisse, et le caractère de celle-ci peut être défini par l'utilisateur. Il est possible de diviser la méthode en deux parties et les mettre dans deux comportements qui vont être exécutés par l'acteur séquentiellement, le premier avant la migration et le second après.

En fait, le degré de la mobilité ne dépend pas du modèle d'acteur mais du langage de programmation du comportement (dans notre cas Java). Si c'était un langage fonctionnel avec l'habilité de matérialiser et de manipuler la continuation, le problème serait très différent [Arc01].

III.2.2.3) Les sémantiques et la sécurité

Un comportement distant contient les références des acteurs qui, dans notre système, n'ont pas à être transformés. Comme Obliq, JavAct compte sur un mécanisme de champs lexical de la fenêtre du réseau : le principal avantage est la préservation des sémantiques de l'acteur déplacé et l'indépendance entre le calcul et l'emplacement. Le déplacement est plus efficace car les acteurs référencés n'ont pas à être sérialisés. Réciproquement, il est connu que cette solution peut mener à une augmentation significative du trafic réseau qui peut aller en opposition au bénéfice attendu de la mobilité.

Un acteur mobile ne peut pas accéder à une ressource d'un hôte si la référence n'a pas été communiquée à l'acteur explicitement. En plus, l'intimité des composants de l'acteur est à la base de son intégrité [Arc01].

III.2.3) l'évolution de JavAct

L'équipe a défini le calcul CAP qui formalise le modèle des acteurs à partir du π-calcul asynchrone et des objets primitifs de Cardelli. Ce calcul a permis de définir plusieurs analyses statiques pour assurer que les messages envoyés vers un acteur seront pris en compte par celui-ci. Ces travaux ont servis de support à une réorganisation de la bibliothèque JAVACT. Il est proposé par la suite une extension de ce calcul pour décrire explicitement la répartition et la mobilité et les évolutions associées dans la bibliothèque JAVACT. Cette extension repose sur une étude approfondie des formalismes existants.

III.2.3.1) La bibliothèque JAVACT

JAVACT [Arc02] est une interface de programmation concurrente et répartie en JAVA, à base d'acteurs adaptables et mobiles. Elle offre un haut niveau d'abstraction par rapport aux technologies plus rudimentaires (processus légers, synchronisation, socket, sérialisation, RMI, Corba, SOAP, . . .). Elle fournit des mécanismes pour la création d'acteurs, le changement de comportement, la répartition, la mobilité et la communication (locale ou distante, asynchrone ou synchrone). La mobilité est «forte» mais limitée en pratique au moment du changement de comportement.

JAVACT a été conçue afin d'être minimale et maintenable à moindre frais, mais aussi pour être exploitable par un programmeur JAVA «moyen» initié aux acteurs. Dans sa version actuelle, JAVACT s'appuie sur le *SUN Development Kit* 1.4 standard édition ; il n'y a pas de préprocesseur et tous les outils standards de l'environnement JAVA peuvent être utilisés. La bibliothèque JAVACT est diffusée sous forme de logiciel libre. Elle est utilisée dans le cadre d'enseignements en 2ème et 3ème cycles. La version 4.0 est la dernière version de JAVACT, qui s'appuie sur CAP pour faciliter la validation de programmes et la mise en oeuvre de la sécurité [Arc04].

a) JAVACT 4.0 et la répartition

Comme il a été précisé ci-dessus, la version actuelle de JAVACT contient les notions de répartition et mobilité. Cependant ces notions sont très rudimentaires. En effet, dans la version actuelle les sites doivent être précisés dans un fichier texte, il n'y a pas de notion de domaines administratifs, il n'y a aucun contrôle effectué lors d'une migration et la notion de groupe l'acteur n'est pas présente. JAVACT repose sur une architecture adaptable à base de composants (création d'acteurs, émission de messages, changement de comportement, cycle de vie, boîte aux lettres, . . .). Cette approche sera donc également suivie pour l'ajout des domaines et des groupes (composants pour l'envoie de messages, pour les politiques d'entrée/sortie, pour la migration, . . .) [Hur01] [Hen04].

b) Le transfert du calcul vers JAVACT

b.1) *Les acteurs*

Les acteurs font l'objet de l'interface javact.lang.Actor et de plusieurs réalisations dont javact.net.rmi.ActorImpl, ils ont été peu modifiés par rapport à la version 4.0 de JAVACT. Leurs principaux attributs sont :

– le comportement de l'acteur ;
– le domaine de l'acteur ;
– la liste des groupes de l'acteur ;
– des composants qui gèrent la réception des messages, le changement de comportement, la création, la migration et l'envoi des messages.

Les acteurs communiquent entre eux grâce à des messages. Ils sont créés en indiquant leur comportement initial. C'est lui qui va imposer les messages que l'acteur est capable de traiter et comment il va les traiter. Il sera responsable d'un éventuel changement de comportement. Dans la version de JAVACT avec les domaines, à la création de l'acteur, il faut indiquer le domaine où celui-ci sera créé.

b.2) *Les domaines*

La représentation

Les domaines font l'objet d'une interface javact.lang.domain et de plusieurs réalisations dont javact.net.rmi.DomainImpl. Les principaux attributs sont :

– des informations relatives au domaine (nom, url, hôte, port, liste d'acteurs) ;
– des informations relatives à la hiérarchie (domaine père et domaines fils) ;
– des composants qui gèrent la migration et les entrées/sorties.

Les sites physiques / Domaines administratifs

Pour régler les problèmes liés aux différences entre sites physiques et domaines administratifs et leur coexistence, deux sortes de domaine ont été créées : les domaines logiques et les domaines physiques. Leur différence se matérialise au niveau de leurs composants de mouvement et de politique. En pratique, cette différence s'exprime dans la localisation de leurs acteurs.

– Dans le cas des *domaines physiques* (TiedDomainPolicyCt ; TiedDomainMoveCt), les acteurs se trouvent sur la même place que le domaine.

Lorsque le domaine migre, tous les acteurs migrent avec lui sur la nouvelle place. Lorsqu'un acteur entre dans un domaine physique, il doit migrer vers ce domaine. Lorsqu'il en sort, si le domaine de destination est un domaine physique, il migre vers celui-ci, sinon il n'est pas obligé de se déplacer. Ces domaines permettent de représenter un pare-feu, passage obligé pour l'entrée d'un acteur sur un autre hôte.

– Dans le cas des *domaines logiques*, les acteurs ne sont pas obligatoirement sur la même place et le même hôte que le domaine. Lorsque le domaine migre, les acteurs ne le suivent pas.

Puisque les acteurs ne sont pas nécessairement sur la même place que le domaine logique, il faut décider de leur localisation (si cela n'est pas précisé à la création). Un composant de placement choisit un hôte et une place de façon aléatoire, mais il peut exploiter des politiques sophistiquées d'équilibrage de charge.

b.3) *Les groupes*

Les groupes, comme les domaines, font l'objet d'une interface javact.lang.Group et de plusieurs réalisations dont javact.net.rmi.GroupImpl. Leurs principaux attributs sont :

– des informations relatives au groupe (nom, hôte, liste d'acteurs) ;
– des composants qui gèrent les envois de messages et les entrées/sorties.

b.4) *Les requêtes et messages*

Les requêtes ne sont pas traitées de manière asynchrone comme les messages, elles sont réalisées par un appel distant synchrone transparent pour l'utilisateur. Lorsqu'un acteur veut rejoindre un groupe, il appel la fonction join de celui-ci. S'il veut changer de place, il

appelle sa méthode move, en passant en paramètre la place qu'il veut rejoindre. C'est ensuite le composant de mouvement qui gère le déplacement.

Les messages entre acteurs n'ont pas été changés par rapport à la version 4.0 de JAVACT. Pour définir un comportement, on crée une interface qui défini les méthodes que sera capable de traiter l'acteur qui possédera ce comportement. A partir de cette interface, pour chaque type de messages, une nouvelle classe est générée, c'est elle qui correspond au message.

Le système de routage des messages consiste à faire parcourir aux messages le même chemin que l'acteur. En effet quand un acteur demande à migrer, il est en fait recréé sur une autre place. Sur sa place de départ, on le remplace par un forwader qui assure le suivi des messages. Pour améliorer les performances, des court-circuits sont introduits entre les forwarder.

b.5) ***La sécurité***

Les contrôleurs sont représentés par les composants qui gèrent les entrées/sorties. Par défaut ces contrôleurs laissent entrer et sortir tout le monde. Pour mettre en place une politique d'entrées/sorties plus complexe, il faut redéfinir les méthodes exitPolicy et joinPolicy du composant myPolicyCt.

Quand un acteur veut entrer ou quitter un domaine ou un groupe, mais qu'il n'y est pas autorisé, une exception est levée. Cette forme est plus proche de la philosophie JAVA du traitement des échecs par rapport au calcul qui propose plutôt une structure de choix (*r n ‹c_1 , c_2›).*

III.3) OpenCorba

Depuis quelques temps, l'engouement pour le réseau Internet souligne la nécessité de trouver rapidement des solutions pour supporter l'*interopérabilité* des environnements *répartis hétérogènes*. Aussi, l'une des préoccupations actuelles des acteurs de l'informatique est de réutiliser et d'assembler des composants logiciels, indépendamment de leurs langages d'implémentation, de leurs systèmes d'exploitation et de leurs sites d'exécution.

Les concepts de la technologie objet offrent de bonnes bases pour répondre à de tels défis. En normalisant des spécifications pour la portabilité et l'interopérabilité des composants objets, le consortium OMG (*Object Management Group*) apporte une réponse intéressante pour aborder le problème de la construction d'applications réparties à objets. Le bus logiciel CORBA (*Common Object Request Broker Architecture*), artère centrale de cette architecture, est responsable de la communication transparente entre objets distants, à travers des environnements répartis hétérogènes. La *modularité* des composantes de l'architecture globale, proposée ainsi par l'OMG, constitue pour nous l'un des avantages majeurs de cette solution.

Cependant, cette volonté de *flexibilité* est en pleine contradiction avec la rigidité des spécifications du bus logiciel lui-même. Ce dernier est une « boîte noire » dont les spécifications sont volumineuses et le modèle objet figé.

Les auteurs Jean-Pierre Briot et Rachid Guerraoui [Bri00] distinguent trois approches pour l'intégration des concepts objets dans la programmation parallèle et répartie :

- L'approche applicative, basée sur le développement des concepts de distribution et de concurrence à travers des bibliothèques de classes ;
- L'approche intégrée, qui correspond à une extension des modèles objets pour la prise en compte de ces concepts ;
- L'approche réflexive, qui consiste en la définition de bibliothèques de métaprotocoles modélisant les concepts de distribution et de concurrence.

En adoptant la classification proposée par Briot et Guerraoui, il apparaît que le modèle de l'OMG correspond à la fois à une *approche intégrée* (optique minimaliste du modèle de l'OMG) et à une *approche applicative* (optique bibliothèques des *CORBAservices*). L'*approche réflexive*, qui permet de combiner les avantages des deux approches, est le meilleur choix conceptuel pour réaliser les futures évolutions du bus. La réflexion permet en effet d'étendre le modèle initial de l'OMG avec des bibliothèques de méta-protocoles spécialisant les mécanismes de la programmation répartie. Il est alors possible d'introduire de

nouvelles sémantiques sur ce modèle initial comme la concurrence, la réplication, la sécurité…

Pour s'adapter à des contextes d'exécution changeants (e.g. équilibrage de charge, tolérance aux fautes), les systèmes répartis doivent supporter la modification dynamique de leurs mécanismes. L'adaptabilité dynamique du bus rend alors possible l'instauration de nouvelles stratégies d'exécution (e.g. migration des objets). Celles-ci, mieux adaptées au comportement de l'application, contribuent ainsi à la construction d'un système ouvert. OpenCorba est ce bus logiciel réflexif *adaptable*.

OpenCorba est basée sur le langage réflexif NeoClasstalk. Ce dernier est le résultat de l'implémentation d'un MOP (*Meta Object Protocol*) dans le langage Smalltalk . Sa principale contribution est d'étendre les aspects dynamiques de Smalltalk en proposant d'une part, une solution efficace pour contrôler l'envoi de messages et, d'autre part, un protocole de changement dynamique de classe.

III.3.1) Un cadre réflexif pour la construction d'architectures ouvertes

Dans cette partie, il est démontré l'intérêt des métaclasses pour la construction d'architectures réutilisables et adaptables, i.e. *ouvertes*. La réflexion peut être un outil pertinent pour le génie logiciel. Dans ce qui suit, nous allons montrer pourquoi les métaclasses peuvent être vues comme des composants logiciels réutilisables et adaptables.

III.3.1.1) Métaclasses et propriétés de classe

La réflexion permet de distinguer *ce que* fait un objet (son niveau de base) de *comment* il le fait (son méta-niveau). Dans un langage réflexif, il existe donc une séparation clairement définie entre les fonctionnalités de l'application, programmées au niveau de base, et leurs représentations et contrôles, programmés au méta-niveau. Dans le contexte des langages à classes réflexifs, la classe d'une classe – une *métaclasse* – va définir des propriétés liées à la création des objets, à leur encapsulation, aux règles d'héritage, à la résolution de l'envoi de messages, … Nous nommons alors par *propriétés de classe*, ces propriétés propres à la classe, qui sont indépendantes du code que la classe décrit pour ses instances.

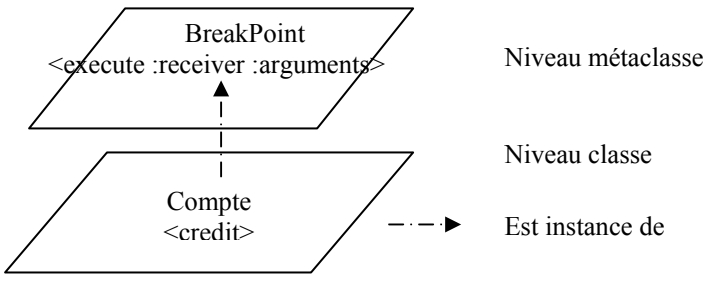


Figure6 : Propriété de classe et (méta)classes

La Figure 6 montre l'exemple d'une classe `Compte`, instance de la métaclasse `BreakPoint`, chargée de fixer des points d'arrêts sur les méthodes de la classe `Compte2`. Le contrôle de l'envoi de messages réifié par le MOP de NeoClasstalk – via la méthode

`execute:receiver:arguments:` – va permettre à la métaclasse `BreakPoint` d'intercepter les messages reçus par une instance de la classe `Compte`.
La méthode `execute:receiver:arguments:` décrite ci-dessous, ouvre un débogueur sur le contexte d'exécution des méthodes de `Compte` (e.g. `credit:`), puis appelle la méthode standard d'invocation de messages grâce à l'héritage.

`BreakPoint>>execute: cm receiver: rec arguments: args` ***Code métaclasse***

```
"Fixer un point d'arrêt sur les méthodes"
self halt: 'BreakPoint for ', cm selector.
^super execute: cm receiver: rec arguments: args
```

`Compte>>credit: aFloat` ***Code classe***

```
"Réaliser un crédit sur le solde courant"
self solde: self solde + aFloat
```

En considérant les classes comme objet de plein droit, nous avons pu éviter l'entrelacement entre le code métier (compte bancaire) et le code décrivant une propriété spécifique sur la classe métier (point d'arrêt). Implémentées par les métaclasses, les *propriétés de classe* encouragent la lisibilité et la réutilisabilité du code dans le développement des classes.

III.3.1.2) Changement dynamique de métaclasse

Le *changement dynamique de classe* de NeoClasstalk est un protocole permettant aux objets de changer de classe à l'exécution. Ce protocole a pour but de tenir compte de l'évolution du comportement des objets dans le temps, et permet d'améliorer ainsi l'implémentation de leurs classes respectives. L'association des classes comme objet de plein droit avec le protocole de changement dynamique de classe permet le *changement dynamique de métaclasse* à l'exécution. Puisque les propriétés de classe sont implémentées par des métaclasses, notre mise en œuvre autorise ainsi *l'adaptabilité dynamique des propriétés de classe*. Le protocole de changement dynamique de classe rend en effet possible l'ajout et le retrait des propriétés de classe *à l'exécution* sans re-génération de code au niveau des classes. En reprenant l'exemple précédent, on peut associer temporairement la propriété `BreakPoint` à une classe le temps de sa mise au point. La classe `Compte` « bascule » de sa métaclasse d'origine3 vers la métaclasse `BreakPoint` pour déboguer les messages, puis revient vers son état antérieur par un nouveau basculement une fois le programme réalisé.

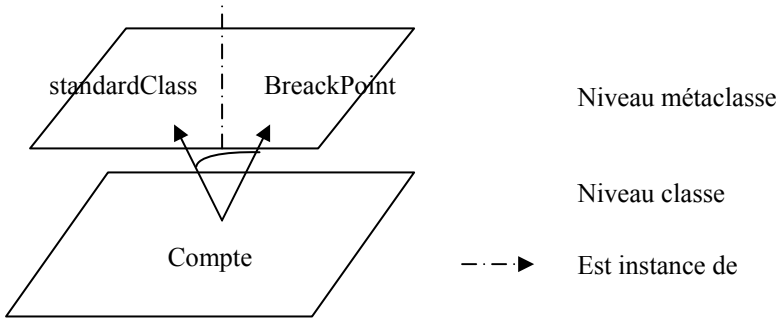


Figure7 : Adaptabilité dynamique des propriétés de classe

Le protocole de changement dynamique de métaclasse permet à un système de remplacer une propriété de classe par une autre, pendant son exécution, sans affecter le reste

du système. Pour nos travaux sur les bus logiciels *adaptables*, ce protocole est extrêmement important car il autorise la modification des mécanismes des architectures réparties *à l'exécution*. Ainsi, une invocation synchrone peut devenir asynchrone, un objet volatil peut devenir persistant, un objet passé en valeur dans une requête peut être passé par référence, un objet *proxy* peut gérer un cache, etc.

III.3.2) OpenCorba : un ORB ouvert

OpenCorba est une application implémentant les API CORBA dans NeoClasstalk, i.e. un *ORB (Object Request Broker) réflexif*. Il réifie différentes propriétés du bus CORBA par le biais des métaclasses afin de favoriser la séparation des caractéristiques internes de l'ORB. L'utilisation du protocole de changement dynamique de métaclasse permet alors d'adapter le comportement du bus à l'exécution en modifiant les mécanismes de l'ORB représentés par des métaclasses. Dans les paragraphes suivants, nous allons présenter trois aspects du bus parmi ceux que nous avons réifiés : le mécanisme d'invocation à distance via un *proxy*, la gestion d'erreurs lors de la création du dépôt d'interfaces4, et le contrôle de type IDL sur la classe serveur. Auparavant, nous présentons la projection IDL OpenCorba et les classes Smalltalk DirectToOpenCorba qui permettent la mise en oeuvre de ces trois aspects réflexifs.

Création des classes proxies et serveurs dans OpenCorba

a) Projection IDL OpenCorba

Le pré-compilateur IDL OpenCorba génère une classe *proxy* sur le client et une classe patron sur le serveur en respectant le *mapping* Smalltalk de la norme CORBA. La classe *proxy* est alors associée à la métaclasse `ProxyRemote` implémentant l'invocation à distance ; la classe patron, à la métaclasse `TypeChecking` implémentant le contrôle de type sur le serveur. La partie gauche de la Figure 8 présente les résultats de la projection IDL de l'interface `Compte` dans OpenCorba : la classe *proxy* `CompteProxy` est instance de `ProxyRemote` et la classe patron `Compte` est instance de `TypeChecking`.

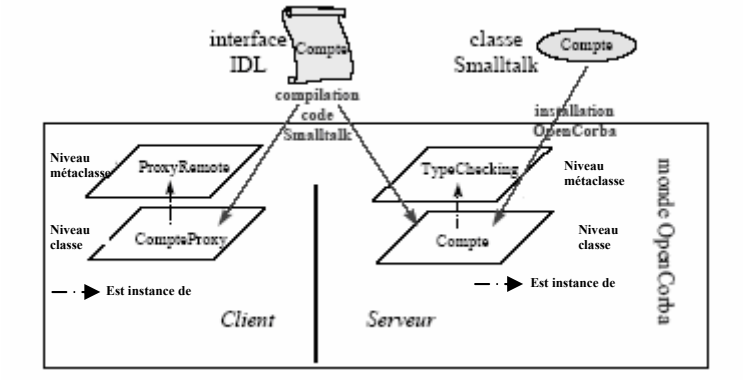


Figure8 : Création des classes proxies et serveurs dans OpenCorba

b) Classe Smalltalk DirectToOpenCorba

OpenCorba permet aux développeurs Smalltalk de transformer toute classe standard Smalltalk en une classe serveur Smalltalk dans l'ORB, nommée classe Smalltalk DirectToOpenCorba5. Cette fonctionnalité est introduite d'une part pour réutiliser du code Smalltalk existant, d'autre part pour décharger le programmeur de l'écriture du fichier IDL. Pour posséder le statut de classe serveur dans OpenCorba, une classe DirectToOpenCorba est instance de `TypeChecking`.

III.4) XML dans le Génie Logiciel

Dans le génie logiciel, XML jusqu'ici a été la plupart du temps employé pour soutenir trois sous activités : la gestion de la documentation, les échanges de données et le stockage léger de données. Deveaux et Le Traon [Dev01] proposent d'employer la technologie de XML comme infrastructure pour la gestion intégrée de toute l'information de développement du noyau logiciel. Ils ont développé le concept de la conception pour la testabilité basée sur un modèle de classe auto-documenté et auto-testable.

La majorité des outils logiciels d'avant et après l'ingénierie sont construits autour d'une représentation interne, qui reste toujours une syntaxe abstraite plus au moins sophistiquée. La nature incompatible de toutes les propriétés de ces représentations représente un obstacle majeur pour l'interopérabilité des outils. La première conséquence est une perte de continuité dans le développement, l'évolution et la maintenance.

XML apparaît alors comme une solution bien adaptée au problème d'intégration de toute information au cours du développement du logiciel afin de la gérer. La divergence existante entre les outils d'analyse et de conception (basés sur UML) et les outils code engineering (code source) peut être défaite grâce aux capacités de XML pour la manipulation de syntaxe à haut niveau. Le principal intérêt de cette approche est une réelle continuité en amont et en aval dans le flux d'information.

Le but de cette approche est de proposer un type de document principal qui capture toute l'information appropriée pour une classe, c.-à-d. documentation, contrats, tests, …etc. Ce document est défini par une DTD de XML et aussi d'essayer de nommer le langage résultant OOPML (Object-Oriented Programming Markup Language).

Le but du projet OOPML est de proposer des modèles qui capturent toute l'information nécessaire pour la construction source avec des outils et exigences de testabilité et de confiance qui facilitent de manière directe ou non la construction source pour les langages orientés objet [Dev00]

La tendance dans la construction de logiciel orienté objet est d'augmenter les activités de développement dans plus de niveaux d'abstraction. Plusieurs programmeurs travaillent maintenant avec UML et utilisent les outils de génération pour construire le code source et le matériel exécutable. Le moyen de générer directement le code à partir d'UML n'est pas simple à implémenter en pratique et pose de réels problèmes au processus de continuité.

Donc il serait plus utile d'introduire un niveau de construction de source intermédiaire entre le niveau d'analyse générale et la conception (qui est géré par UML et son langage d'interchange XMI) et le niveau source. Ce niveau intermédiaire capture, en plus de la

fonctionnalité et des sémantiques du modèle, les exigences de testabilité et de confiance, le contrôle de versionnement et les constructions de langage spécifique qui sont souvent des items transversaux à plusieurs applications.

III.5) Le calcul formel CAP

L'évolution des applications réparties a suscité un grand intérêt pour les méthodes formelles qui sont devenues une exigence dans le développement de logiciels critiques. En effet, la répartition et la mobilité ont introduit une complexité supplémentaire dans la mise au point des applications à objets concurrents. Ces applications doivent faire face à des erreurs qui peuvent apparaître au sein des composants concurrents ou bien issu de problème de communication ou de synchronisation ainsi qu'à l'hétérogénéité matérielle et logicielle. L'utilisation d'approches plus abstraites de la programmation combinée avec des systèmes de type des programmes permet de faciliter l'introduction de la concurrence, de la répartition et de la mobilité.

Pour faciliter la mise au point de ces applications concurrentes, l'équipe ingénierie des applications mobiles (I.A.M.) de l'I.R.I.T a définit des mécanismes d'analyse statique qui assurent que chaque message émis sera traité. Ces analyses ont été définies pour le modèle formel CAP (Calcul d'Acteurs Primitifs) permettant d'exprimer l'architecture concurrente de l'application. CAP est un modèle de programmation à base d'objets actifs, les acteurs, qui constitue un support simple, réaliste, bien adapté à la mise en œuvre d'applications réparties à grande échelle ou mobile.

III.5.1) CAP

Un des intérêts des calculs réside dans leur simplicité, leur concision et leur élégance. Les processus sont décrits par les termes d'une algèbre, c'est à dire qu'ils sont construits à partir d'autres processus et d'opérateurs de composition de ces processus (*composition parallèle*, *choix indéterministe*, *séquence*, etc...). Les processus de base sont des actions élémentaires que l'on assimile généralement à de la communication (envoi ou réception d'une information). Parmi les premiers calculs de processus, on peut citer le *Calculus of Communicating Systems* (CCS) de Milner et le *Communicating Sequential Processes* (CSP) de Hoare. Le calcul le plus connu est le π-calcul de Milner [Hur04].

Le noyau des langages d'acteur est basé sur la communication asynchrone entre les acteurs. Les langages existants contiennent des structures de données prédéfinies et des structures de contrôle séquentielles. Mais la communication est suffisante pour exprimer tous les calculs possibles et les acteurs représentent alors les structures de données [Dag97].

CAP (Calcul d'Acteurs Primitifs) a été développé par J.-L. Colaço au sein de cette équipe. Il est primitif car il utilise seulement le contenu de la communication pour exprimer un calcul. L'équipe a défini le calcul CAP qui formalise le modèle des acteurs à partir du π-

calcul asynchrone et des objets primitifs de Cardelli. Ce calcul a permis de définir plusieurs analyses statiques pour assurer que les messages envoyés vers un acteur seront pris en compte par celui-ci [Col96].

CAP intègre de manière primitive les noms (ou adresse), les comportements (ensemble des messages que l'acteur sait traiter, et des champs privés contenant des connaissances dont l'acteur seul peut y accéder), les acteurs et les messages.

La syntaxe

Les programmes sont construit à partir de noms et comportements utilisant les constructeurs suivants [Col96] : messages, acteurs, composition concurrente, création de noms, inaction, modification et sélection de champs.

Les messages sont adressés à un seul acteur qui est identifié par son nom (ou boite mail) (T_1), ils ont un libellé message (m) et un tuple ($\check{T}_2$) qui représente le contenu du message $T_1 \lhd m(\bar{T}_2)$**.** Un acteur possède un seul identifiant (boite mail) (T_1) associé à un comportement (T_2) : $T_1 \rhd T_2$.

Les comportements encapsulent un ensemble fini des libellés message avec leurs arguments formels, et un ensemble de champs privés sans arguments formels (aucune substitution n'est possible durant la sélection du champ) :

$$[m_1(\bar{x_1}) = \zeta(e_1, s_1)C_1 \ \cdots \ m_n(\bar{x_n}) = \zeta(e_n, s_n)C_n,\ p_1 = T_1 \ \cdots \ p_l = T_l]$$

La définition complète de la syntaxe est donnée dans la définition suivante :

Définition 1 :

Soient N un ensemble infini de noms, V un ensemble fini de variables et L un ensemble fini de libellés. N* (respectivement V*) un ensemble de séquences finies dans N (respectivement dans V).

Convention : dans ce qui suit : a,b,c Є N – v,x,e_i,s_i Є V - $\bar{x}, \bar{x_i}$ Є V* - C,D,C_i sont des configurations (composition parallèle d'acteurs et de messages).

Les règles de syntaxe sont [Col99] :

Config	::=	$\emptyset$	*Configuration vide*
	|	*(Config)*	*Parenthésage*
	|	*Config* $\|\|$ *Config*	*Composition parallèle*
	|	νa *Config*	*Restriction*
	|	*Terme* $\triangleright$ *Terme*	*Acteur*
	|	*Terme* $\triangleleft m(\widetilde{Terme})$ [1]	*Message*
	|	$\mathfrak{C}\_\mathfrak{Err}$	*Erreur dynamique*
Terme	::=	$[m_i(\widetilde{Id_i} = \zeta(e_i, s_i)C_i)^{i \in I}, p_j = T_j{}^{j \in J}]$	*Comportement*
	|	*Terme*.$p_j \Leftarrow$ *Terme*	*Mise à jour*
	|	*Terme*.p_j	*Sélection*
	|	*Id*	*Identificateur*
	|	*(Terme)*	*Parenthésage*
	|	$\mathfrak{T}\_\mathfrak{Err}$	*Erreur dynamique*
Id	::=	a	*Nom*
	|	s	*Variable*

1. Le ~ est utilisé pour représenter la séquence.

La sémantique

Elle est donnée sous la forme de règles de déduction comme celles montrées dans la définition 5. Afin de minimiser le nombre de règles, une relation de congruence est définie pour caractériser les expressions équivalentes. Il faut alors également définir les notions de noms libres et variables libres, ainsi que les règles de substitution de noms/variables. La sémantique peut ensuite être donnée, et contient une partie traitement des erreurs : l'ensemble des expressions syntaxiquement correctes mais sémantiquement fausses. Par exemple, un comportement ne peut pas adopter de nouveau comportement, seul un (nom d')acteur le peut :

$$\text{ACT-ERR}_1 : [m_i = \cdots] \triangleright T \longrightarrow \mathfrak{C}\_\mathfrak{Err}$$

Les variables représentent des identifiants qui peuvent être substitués par des noms ou des comportements, mais aucune substitution n'est possible sur les noms [Col96].

Définition 2 : (Les variables libres et les variables liées)

Cette notion est définie dans un ordre général. 'v' est le seul lien possible pour les noms, 'ζ(e,s)' et '$m(\tilde{x}_i)$' sont les deux formes de lien pour les variables. L'ensemble des noms libres (respectivement variables libres) d'une expression est donné par la fonction $\mathcal{FN}()$ (respectivement $\mathcal{FV}()$).

Définition 3 : (Substitution)

Soit σ = {T/X} une substitution (où T est un terme et X une variable). Soit E une expression dans laquelle les variables ont été renommées de telle sorte que toutes les variables liées de E sont toutes différentes de X et que la capture des noms et variables libres de T se fait lors de la substitution.

$$
\begin{array}{rcl}
(\nu a\ C)_\sigma & = & \nu a\ C_\sigma \\
(C \parallel D)_\sigma & = & C_\sigma \parallel D_\sigma \\
(T \lhd m(\tilde{U}))_\sigma & = & T_\sigma \lhd m(\tilde{U_\sigma}) \\
(T \rhd U)_\sigma & = & T_\sigma \rhd U_\sigma \\
(T.p_j \Leftarrow U)_\sigma & = & T_\sigma.p_j \Leftarrow U_\sigma \\
(T.p_j)_\sigma & = & T_\sigma.p_j \\
[m_i(\tilde{x_i}) = \zeta(e_i, s_i)C_i{}^{i=1\cdots n}, & = & [m_i(\tilde{x_i}) = \zeta(e_i, s_i)C_{i\sigma}{}^{i=1\cdots n}, \\
p_j = T_j{}^{j=1\cdots l}]_\sigma & & p_j = T_{j\sigma}{}^{j=1\cdots l}] \\
a_\sigma & = & a \\
x_\sigma & = & \begin{cases} T & if\ x = X \\ x & otherwise \end{cases}
\end{array}
$$

Afin de définir les règles de réduction, il faut d'abord introduire la relation de congruence entre les expressions du calcul.

Définition 4 : (Congruence)

'≡' est la plus petite congruence dans les expressions de CAP définie par les règles suivantes :

1. C≡ D if C is α-convertible to D (α-conversion of names and variables).
2. C || Ø≡ C
3. C || D≡ D || C
4. (C || D) || E≡ C || (D || E)
5. T ▷ T_1≡T ▷ T_2 if T_1≡ T_2
6. $[m_i(\tilde{x_i}) = \zeta(e_i, s_i)C_i{}^{i=1\cdots n}, p_j = T_j{}^{j=1\cdots l}]$
   $\equiv [m_{\pi_1(i)}(\widetilde{x_{\pi_1(i)}}) = \zeta(e_{\pi_1(i)}, s_{\pi_1(i)})C_i{}^{i=1\cdots n}, p_{\pi_2(j)} = T_{\pi_2(j)}{}^{j=1\cdots l}]$
   where π_1 and π_2 are permutations.
7. υa C || D ≡ υa(C || D) if a $\notin$ *FN*(D)

Définition 5 : (Les règles de réduction)

La réduction, dénotée par '→ ', est la plus petite relation générée par les règles :

$$STRUCT : \frac{D \equiv C \quad C \longrightarrow C' \quad C' \equiv D'}{D \longrightarrow D'}$$

$$PAR : \frac{C \longrightarrow C'}{C \parallel D \longrightarrow C' \parallel D} \qquad RES : \frac{C \longrightarrow C'}{\nu x C \longrightarrow \nu x C'}$$

$$MESS : \frac{T \longrightarrow T'}{T \lhd m(\tilde{t}) \longrightarrow T' \lhd m(\tilde{t})} \qquad ACT : \frac{T_1 \longrightarrow T_1' \quad T_2 \longrightarrow T_2'}{T_1 \rhd T_2 \longrightarrow T_1' \rhd T_2'}$$

$$COMM : \textit{if } k \in [1 \cdots n] \textit{ and } len(\tilde{v}) = len(\tilde{x_k})$$

$$a \lhd m_k(\tilde{v}) \parallel a \rhd [m_i(\tilde{x_i}) = \zeta(e_i, s_i) {C_i}^{i=1\cdots n}, p_j = {T_j}^{j=1\cdots l}] \longrightarrow C_k\{a/e_k\}\{[m_i = \cdots, p_j = \cdots]/s_k\}\{\tilde{v}/\tilde{x_k}\}$$

$$SELECT : \textit{if } k \in [1 \cdots l]$$

$$[m_i(\tilde{x_i}) = \zeta(e_i, s_i) {C_i}^{i=1\cdots n}, p_j = {T_j}^{j=1\cdots l}].p_k \longrightarrow T_k$$

$$REDEF : \textit{if } k \in [1 \cdots l]$$

$$[m_i(\tilde{x_i}) = \zeta(e_i, s_i) {C_i}^{i=1\cdots n}, p_j = {T_j}^{j=1\cdots l}].p_k \Leftarrow T \longrightarrow [m_i(\tilde{x_i}) = \zeta(e_i, s_i) {C_i}^{i=1\cdots n}, p_j = {T_j}^{j \in [1\cdots l] - \{k\}} \quad p_k = T]$$

Utilisant ces règles de réduction, il est possible que l'envoi d'un message ne puisse être compris par l'acteur de destination. Dans ce cas, le message demeure dans l'expression. Un futur comportement de l'acteur de destination pourrait peut être le traiter, sinon ce message est appelé orphelin.

La réflexivité

La réflexivité exprime la possibilité pour un langage de s'auto-redéfinir, en modifiant sa sémantique. De manière plus restreinte, cela correspond à la capacité d'un programme de modifier sa fonction. Cela se ferait de manière transparente pour les autres programmes qui pourraient continuer à l'utiliser comme si rien n'avait changé. On se place ici à un méta-niveau, où le programme produit lui-même son futur code. Le langage Java par exemple offre une certaine forme de réflexivité dans la mesure où il permet d'extraire d'une classe la plupart des informations la définissant : classe dont elle hérite, interface(s) qu'elle réalise, champs, méthodes, etc. Il est ensuite possible de redéfinir certaines méthodes [Col96].

En CAP la réflexivité s'exprime par la possibilité pour un acteur d'accéder à son comportement courant (appelé self) et à son nom (ego) lors de la définition d'un nouveau comportement. Il s'agit de la construction ζ(ei; si). Pour illustrer cela J.-L. Colaço donne l'exemple de la duplication de serveurs : le serveur est capable de traiter un message spécial (reify) qui lui demande d'envoyer ses nom et comportement à un duplicateur d'acteurs ; le duplicateur s'occupe alors de créer des clones et redéfinit le comportement de l'ancien serveur, qui s'occupe désormais de transmettre les requêtes aux nouveaux serveurs [Che00].

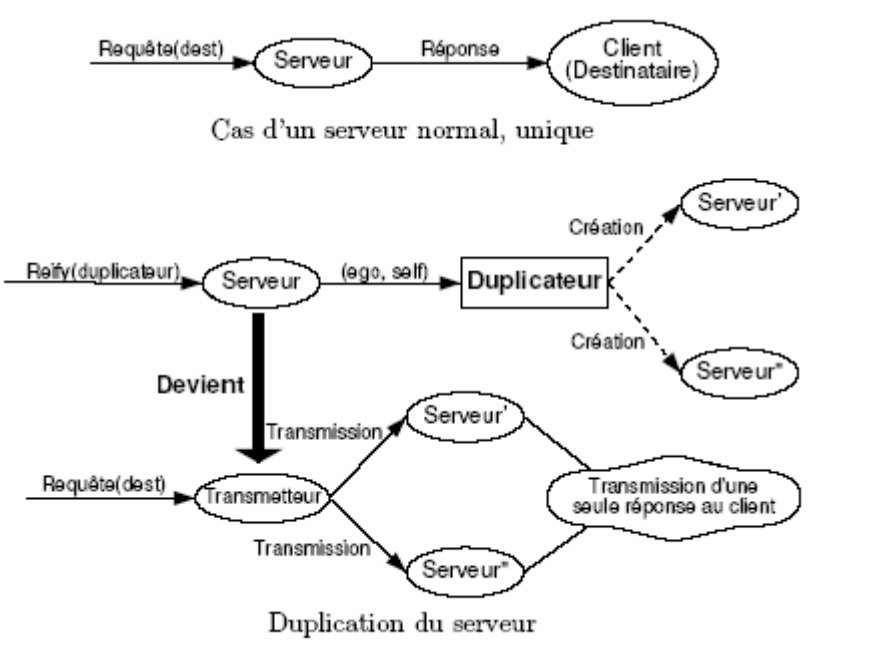


Figure13. Expression de la réflexivité en CAP

III.5.2) Typage de CAP

À chaque acteur est associé un type calculé par inférence ; il correspond à l'ensemble des messages que l'acteur peut traiter, avec pour chaque message le type de ses paramètres : $\langle m_i(\widetilde{\alpha_i'})^{i\in I}\rangle$ (ici le ~ exprime la séquence).

La notion de sous-typage est naturelle puisqu'elle est liée au fait qu'un acteur puisse toujours être remplacé par un autre acteur offrant plus de méthodes et imposant moins de contraintes sur les arguments. Cela est exprimée par une relation d'inclusion : $\alpha \supseteq \alpha'$ signifie que tout nom de type α peut remplacer tout nom de type ά ; on dit qu'un acteur en subsume un autre s'il peut le remplacer.

Un type de nom correspond à l'ensemble des messages qu'un acteur peut traiter, avec pour chaque message le type de ses paramètres $\langle m_i(\widetilde{\alpha_i})^{i\in I}\rangle$ dans ce cas, la notion de contravariance apparaît : Un acteur peut en remplacer un autre s'il peut traiter au moins autant de messages que l'autre ; de plus les paramètres des premiers doivent pouvoir remplacer les paramètres des seconds, ils doivent être des sous-types des seconds. Cela nous amène à la définition suivante [Hen03] :

Définition 6 (Inclusion de types de noms) :

Un nom de type α peut remplacer un nom de type α' $(\alpha \supseteq \alpha')$ si et seulement si :

$$\langle m_i(\widetilde{\alpha_i'})^{i\in I}\rangle \subseteq \langle m_j(\widetilde{\alpha_j})^{j\in J}\rangle \iff I \subseteq J \wedge ((\forall k \in I)\ \widetilde{\alpha_k'} \widetilde{\supseteq} \widetilde{\alpha_k})$$

Où $\widetilde{\supseteq}$ représente l'inclusion terme à terme entre séquences de même longueur.

Le problème est : « Comment représenter fidèlement dans le type d'un acteur l'ensemble des comportements qu'il est susceptible d'adopter au cours de son exécution ». Il s'agit de « fusionner » les ensembles de messages associés à chaque comportement. Une première idée peut être de prendre l'intersection de ces ensembles, pour aboutir à un sous-ensemble commun dont on est sûr qu'il sera accepté dans toutes les branches d'exécution possibles. Ce sous-ensemble est trop restrictif et un typage par cette méthode rejettera trop de programmes corrects. La solution opposée, consistant à prendre l'union des types de comportement, est trop permissive et ne permet pas non plus de faire un typage pertinent.

Une solution intermédiaire a donc été développée : l'aplatissement.

Définition 7 (Aplatissement) : L'aplatissement, noté $\natural$, est défini par :

$$\langle m_i(\widetilde{\alpha_i})^{i\in I}\rangle \natural \langle m_j(\widetilde{\alpha'_j})^{j\in J}\rangle = \langle m_k(\widetilde{\alpha_k}\widetilde{\cup}\widetilde{\alpha'_k})^{k\in I\cap J}\, m_i(\widetilde{\alpha_i})^{i\in I\setminus J}\, m_j(\widetilde{\alpha'_j})^{j\in J\setminus I}\rangle$$

De même ici $\widetilde{\cup}$ correspond à l'union terme à terme entre séquences de même longueur.

Contrairement à l'union, l'aplatissement est sûr : il n'accepte pas de programmes incorrects. Il est également moins restrictif que l'intersection et rejettera donc moins de programmes corrects. La figure suivante permet d'illustrer l'aplatissement ; un acteur est représenté sous la forme d'un arbre éventuellement infini mais régulier (un automate) ; les noeuds représentent les divers comportements qu'il peut adopter, et les arcs correspondent aux changements de comportement liés à la réception d'un message. Le comportement de départ est α_e (la racine de l'arbre ou l'état initial de l'automate), et selon les messages qu'il reçoit il peut adopter ensuite les comportements α_{e1}, α_{e2}, …, α_{en}. Son type est donc :

$$\alpha_e = \langle m_1(\alpha_1), \ldots, m_n(\alpha_n)\rangle \natural \alpha_{e_1} \natural \cdots \natural \alpha_{e_n}$$

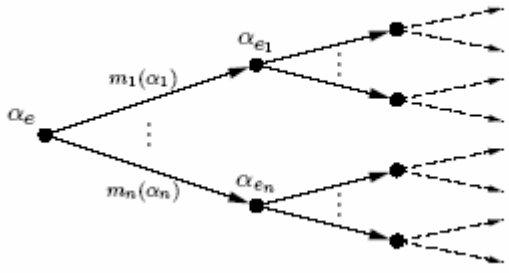


Figure14. Les comportements d'un acteur vus comme les noeuds d'un arbre

Les messages orphelins :

Le typage permet d'assurer que toute communication potentielle se fera sans erreur sémantique (nombre et type corrects des arguments de messages). Selon la branche d'exécution que l'acteur adoptera certains messages ne pourront plus être pris en compte, et de ce fait deviendront orphelins. On peut donner la définition informelle suivante : « Un message est orphelin s'il est destiné à un acteur dont ni le comportement actuel, ni les comportements

futurs (quelle que soit l'exécution) ne l'accepteront ». Un orphelin trivial étant un orphelin avant même toute exécution du programme [Col97].

III.5.3) L'introduction de la répartition et la mobilité dans CAP

CAP est un calcul concurrent, mais il ne permet pas de modéliser de manière claire la mobilité et la répartition. Pour ajouter ces deux notions au calcul, L'équipe a réalisé une étude bibliographique, afin de dégager, des calculs existants, les différentes approches possibles. Dans les dérivés du modèle «*acteur*», l'approche usuelle, simple et pragmatique, repose sur la notion de site (ou place) qui correspond à un support physique d'exécution d'acteurs (machine réelle ou virtuelle) caractérisé par une adresse et un numéro de port Internet. Les sites d'une application forme un graphe connexe. Les acteurs sont répartis sur les différents sites et peuvent se déplacer de site en site. Leur déploiement peut être décrit explicitement dans le programme ou dans un descripteur indépendant. La description explicite est utile quand il s'agit de découvrir dynamiquement la structure d'un réseau «*ouvert*» par l'échange d'informations sur les sites existants avec les autres acteurs et les sites connus [Hura04].

Cette approche est satisfaisante pour des applications de taille modérée dans un environnement ouvert et bienveillant. Par contre, elle n'est plus adaptée si le nombre de site est important, (en particulier si ceux-ci appartiennent à plusieurs entités administratives distinctes) et si des contraintes de contrôle d'accès ou de sécurité doivent être prises en compte. Dans le cadre des «*ambients*», Cardelli propose l'introduction d'un modèle similaire aux répertoires utilisés dans les systèmes de fichiers. La proposition de l'équipe s'inspire de ses travaux et de leurs évolutions.

Un domaine est une organisation logique (souvent appelée administrative) qui contient des acteurs et des sous domaines (comme les répertoires contiennent des fichiers et des sous répertoires). Dans le cadre de la mobilité, il faut envisager le déplacement d'un acteur ou d'un domaine entier de son domaine père vers un autre domaine. Pour maîtriser la mobilité, il est souhaitable de pouvoir interdire l'accès d'un domaine. Ce contrôle peut être statique ou dynamique, un contrôleur peut être associé à chaque domaine pour en administrer l'entrée et la sortie. Une application comporte ainsi une organisation hiérarchique. Celle-ci peut également être utilisée pour représenter l'organisation physique. Les différents sites sont alors associés à des domaines spécifiques. Cette association peut être explicite dans le programme ou décrite dans un descripteur de déploiement. Un acteur n'appartient qu'à un seul domaine.

La notion de groupe (duale des domaines) est souvent utile. Un groupe est une structure non hiérarchique. Un acteur peut appartenir à plusieurs groupes. Un contrôleur est également associé à chaque groupe pour en administrer l'entrée et la sortie. Les domaines, groupes et contrôleurs permettent alors de structurer et de sécuriser, logiquement et physiquement, une application concurrente, répartie et mobile.

L'étude bibliographique a porté sur cinq calculs : Le π_{II}-calcul de Roberto Amadio et al., Le Join-Calcul Distribué de J.J Lévy et al., Le Nomadic π-calcul de Peter Sewell et al., Les Ambients de Luca Cardelli et al. et Le M-calcul de Alan Schmitt et al..

a) Le π_{II}-calcul (Roberto Amadio et al.) :

Dans le π_{II}-calcul, version avec site du π-calcul asynchrone, les processus sont positionnés explicitement sur un site. Il s'agit, dans ce calcul de site physique. Un système de

typage des sites est proposé, il permet de savoir s'ils tournent ou s'ils sont arrêtés. Il propose également des primitives explicites pour la mobilité. Il accepte des communications locales ou distantes, grâce à un système de routage des messages basé sur des processus démons immobiles qui tournent sur les sites et font suivre les messages aux acteurs. Cependant, il ne propose aucune structure hiérarchique pour les sites. Il n'y a également aucune notion de sécurité.

b) Le Join-Calcul Distribué (J.J Lévy et al.) :

Le Join-Calcul est enrichi pour permettre d'exprimer explicitement les *emplacements* et les primitives de mobilité. Ce calcul propose une structure hiérarchique des emplacements (structure que l'on retrouvera dans le M-calcul et qu'elle sera adoptée pour le calcul CAP). Il propose aussi un système de détection et de gestion des erreurs (que nous retrouverons également dans le calcul CAP). Cependant aucun contrôle n'est effectué lors d'une migration, tous les acteurs peuvent donc aller dans tous les emplacements.

c) Le Nomadic π-calcul (Peter Sewell et al.) :

Le Nomadic π-calcul, propose deux systèmes de routage, l'un centralisé et l'autre distribué. Le système de routage distribué est très proche de celui de JAVACT. Les communications, peuvent être locales ou distantes. Une fois encore, les problèmes de ce calcul sont l'absence de notion de sécurité et de structure hiérarchique pour les domaines.

d) Les Ambients (Luca Cardelli et al.) :

Les Ambients ont été créés pour pouvoir modéliser l'existence de domaines séparés, les différences entre ces domaines, le déplacement d'un domaine à un autre, les barrières à la mobilité et la possibilité ou l'impossibilité de traverser ces barrières. Les ambients sont très intéressants, car ils permettent de gérer les problèmes de sécurité, de domaine administratif. Cependant, ils ne proposent pas de primitive globale de déplacement (l'utilisateur doit indiquer le chemin d'accès) et les communications sont uniquement locales.

e) Le M-calcul (Alan Schmitt et al.) :

Le M-calcul fourni un modèle de programmation répartie qui autorise la programmation explicite des comportements des emplacements. Il a été créé car ses auteurs reprochent aux autres calculs l'uniformité des sites (ils ont tous le même comportement), le manque de contrôle lors des accès aux ressources et lors des migrations. Le M-calcul est le calcul qui se rapproche le plus de ce que souhaite faire l'équipe, il gère les migrations, les communications distantes, il y a des contraintes d'entrées et sorties des domaines pour les acteurs et les messages. Celles-ci sont imposées par un contrôleur. L'inconvénient de ce calcul est sa complexité, liée en particulier à la programmation explicite dans le calcul des contrôleurs.

Bilan

Cette étude bibliographique des calculs de processus a permis de dégager leurs aspects intéressants pour la gestion de la répartition et de la mobilité, et de les adapter au calcul CAP selon les besoins. De cette étude est ressorti cinq grands domaines particulièrement intéressant : la représentation des sites, le routage des messages, la sécurité, les groupes d'acteurs et le

traitement des pannes. Chaque calcul traite plus particulièrement un (ou plusieurs) de ces aspects, ce qui les rend, certes très complet pour cet aspect, mais parfois aussi très complexe. Dans CAP, ils ont essayé de faire une synthèse des aspects intéressants de ces calculs tout en essayant de ne pas trop le compliquer pour pouvoir lui appliquer des analyses statiques.

π_{1l}-calcul	
Représentation des sites	+typage de site +primitives explicites de mobilité - aucune hiérarchie de site / pas de domaines administratifs
Routage des messages	+communications locales ou distantes +proposition de routage basée sur les démons
Sécurité	- non traité
Groupe d'acteur	- non traité
Panne	+détection et traitement des pannes
Join-Calcul Distribué	
Représentation des sites	+hiérarchie de site +primitives explicites de mobilité - pas de domaines administratifs
Routage des messages	Pas d'émission / réception «classique»
Sécurité	- non traité
Groupe d'acteur	- non traité
Panne	+détection et traitement des pannes
Nomadic π-calcul	
Représentation des sites	+primitives explicites de mobilité - aucune hiérarchie de site / pas de domaines administratifs
Routage des messages	+communications locales ou distantes +système de routage centralisé ou distribué
Sécurité	- non traité
Groupe d'acteur	- non traité
Panne	- non traité
Ambients	
Représentation des sites	+domaines administratifs
Routage des messages	- communications uniquement locales
Sécurité	+possibilité d'introduire de la sécurité
Groupe d'acteur	+enrichissement prenant en compte les groupes
Panne	- non traité
M-calcul	
Représentation des sites	+domaines administratifs - pas de primitives explicites de mobilité
Routage des messages	+communications locales ou distantes +système de routage «pas à pas»
Sécurité	+contrôles en entrée et sortie des domaines
Groupe d'acteur	- non traité
Panne	+détection et traitement des pannes (Join-Calcul Distribué)

Figure15. Le bilan

Le résultat : DCAP

La syntaxe

Les principaux éléments de la syntaxe sont :

– *Les acteurs :* $(a : \tilde{g}) \triangleright c$ Identiques à ceux de CAP, à la différence près qu'ils sont étiquetés par l'ensemble des groupes, ğ, auxquels ils appartiennent.

– *Les messages :* $a \triangleleft m(\tilde{t})$ Identiques à ceux de CAP.

– *Les domaines :* d(C)[P] Un domaine possède un nom d, un contrôleur C et un contenu P. Dans le cadre de DCAP, les contrôleurs sont représentés par une relation qui indique si un acteur est autorisé à entrer. Celle-ci est une abstraction d'une fonction complexe qui peut contenir des conditions indécidables statiquement (l'accès pour un même acteur peut être à la fois autorisé et interdit). Le contenu comporte les acteurs, les messages, les sous-domaines et les groupes. Le domaine racine (le réseau) est noté Є et ne possède pas de contrôleur ;

– *Les groupes :* g(C) Proche des domaines, on y retrouve le nom g et le contrôleur C qui détermine les droits d'entrée et de sortie dans le groupe ;

– *Les requêtes : r n ‹c₁; c₂›* Fondées sur le modèle des messages synchrones, elles permettent aux acteurs de communiquer avec les domaines et les groupes. Elles sont au nombre de trois : go (pour rejoindre un domaine), join (pour rejoindre un groupe) et unjoin (pour quitter un groupe). En cas de succès de la requête r à l'entité n, l'acteur poursuivra l'exécution de c_1, en cas d'échec le domaine et les groupes de l'acteur ne seront pas modifiés et l'acteur poursuivra l'exécution de c_2. Les requêtes de migration sont traitées par tous les contrôleurs des domaines traversés. Les requêtes pour les groupes sont traitées uniquement par le contrôleur du groupe auquel elles sont destinées. Les messages, quant à eux, circulent librement, sans intervention des contrôleurs (ce choix est différent des autres calculs. Il correspond bien au modèle d'équité usuel des acteurs : un message émis doit toujours atteindre son destinataire).

Un exemple

Pour décrire cet exemple nous utiliserons une notation graphique inspirée de celle utilisée pour les «*ambients*» et le M-calcul.

Le point de départ est :

$$(\nu a, b, c, d, e, g)\ \epsilon[\ d(C_d)[(c : g) \triangleright B_c \mid g(C_g) \mid c \triangleleft run()] \quad | \quad e(C_e)[(a : g) \triangleright B_a \mid (b : \varnothing) \triangleright B_b]\ ]$$

Soient B_a le comportement de l'acteur a, B_b celui de b et B_c celui de c :

$$B_c = [run() = \zeta(e,s)(a \lhd migre(d) | b \lhd migre(d) | e \rhd s)]$$
$$B_a = [put(d) = \zeta(e, s_{empty})(e \rhd [get(c) = \zeta(e', s_{full})(v \lhd value(v) \| e' \rhd s_{empty})]),$$
$$migre(d) = \zeta(e,s)(e \rhd (go\ d\ \langle s, s\rangle))]$$
$$B_b = [value(v) = \zeta(e, s_{print})(print\ 'received\ v' \| e \rhd s_{print}),$$
$$migre(d) = \zeta(e,s)(e \rhd (go\ d\ \langle s, s\rangle))]$$

Cet exemple est composé de deux domaines : d et e.

- Sur d se trouve un acteur c qui appartient au groupe g. g se trouve également sur d, son contrôleur est noté C_g. Le domaine d ne laisse entrer que les acteurs qui appartiennent au groupe g. Le contrôleur de d est noté C_d et correspond à la fonction :

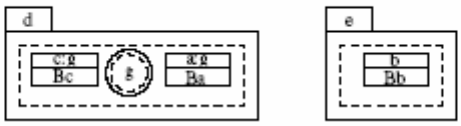
Acteur	$\mapsto$	boolean
$(a : \tilde{g}) \rhd c$	$\rightarrow$	vrai si $g \in \tilde{g}$
		faux sinon

- Sur e se trouve un acteur a qui appartient au groupe g et un acteur b qui n'appartient à aucun groupe. L'acteur a est une cellule et l'acteur b est un client. Lors de la réception du message migre, a (si elle est vide) et b souhaitent quitter e pour entrer dans d. Le domaine e laisse entrer et sortir tous les acteurs, son contrôleur est noté C_e avec $\forall$a, C_e(a) = vrai.

- L'acteur c connaît les acteurs a et b et à la réception d'un message run, il leur envoie un message migre(d). Lorsque les acteurs a et b reçoivent ce message, ils tentent de rejoindre le domaine -d.

L'application des règles de réduction produit le terme :

$$(\nu a, b, c, d, e, g)\ \epsilon[\ d(C_d)[(c : g) \rhd B_c \mid g(C_g) \mid (a : g) \rhd B_a] \quad | \quad e(C_e)[(b : \varnothing) \rhd B_b]\ ]$$



La cellule a rejoint le domaine d, mais b, n'ayant pas le droit d'accéder à d, est revenu sur son domaine de départ.

III.6) Conclusion

Suite à l'apparition du code mobile, des études ont été menées à la fois sur les méthodes de programmation et les outils d'analyse de programmation, sur l'intégration des outils de développement et sur des solutions rapides pour supporter l'interopérabilité des environnements répartis hétérogènes. Nous avons présenté dans ce chapitre quelques travaux sur l'implication du code mobile dans le Génie logiciel.

Chapitre IV :
Notre approche de conception d'un environnement de développement d'applications distribuées, concurrentes et mobiles

IV.1) Problématique

Les réseaux à grande échelle donnent un grand nombre de sources d'informations distribuées et hétérogènes. La répartition et la mobilité utilisées dans ces sources d'information introduisent une complexité et une difficulté supplémentaire dans le développement et la maintenance des applications à base d'objets concurrents et mobiles. Cela se répercute sur la validité et la sûreté de ces applications.

Pour faciliter d'une part, la mise au point de telles applications et assurer d'autre part, leur validité et leur sûreté, nous proposons une approche de conception d'un environnement de développement intégrant des mécanismes d'analyse statique des programmes. Ces analyses sont définies pour un modèle formel permettant d'exprimer l'architecture concurrente, répartie et mobile de l'application indépendamment des aspects fonctionnels. Cette architecture est ensuite instanciée dans un langage de programmation sans perdre les propriétés validées. Ce modèle formel est CAP (Calcul d'Acteurs Primitifs).

L'utilisation d'approches plus abstraites de la programmation (telle les agents mobiles) combinées avec des mécanismes d'analyse statique des programmes permet de réduire significativement la surcharge liée à l'introduction de la concurrence, de la répartition et de la mobilité. L'objet de ces travaux est donc la définition et la conception d'un environnement de développement pour cette forme d'applications.

IV.2) Notre approche

L'approche que nous proposons porte sur la conception d'un environnement de développement d'applications réparties à base d'objets concurrents et mobiles, basée sur le modèle d'acteurs. Cet environnement intègre les concepts de distribution et mobilité du calcul formel CAP [Bebb06]

L'idée forte de notre approche réside premièrement dans la représentation du comportement de l'acteur et sa communication avec ses collègues et deuxièmement dans l'extension du langage de programmation par l'intégration des aspects de concurrence et mobilité du calcul formel CAP, comme explicité dans la figure14.

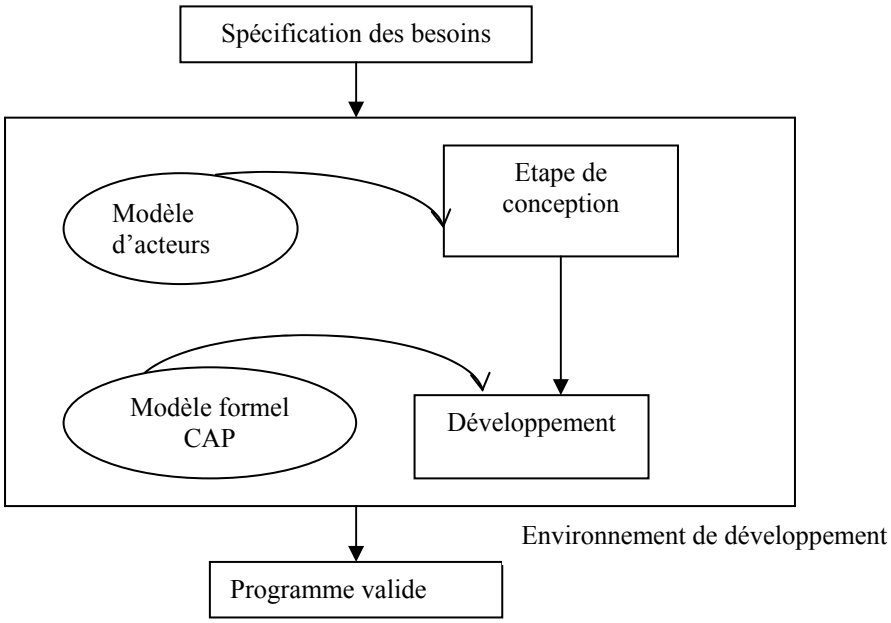


Figure14. Environnement de développement d'applications réparties, concurrentes et mobiles

IV.2.1) Les acteurs

Le modèle d'acteur permet de décrire une application répartie comme une communauté d'agents concurrents (appelés acteurs) coopérant par envoi asynchrone de messages [Col97].

Les acteurs [All98] sont des entités anthropomorphes autonomes qui réagissent aux messages et les exécutent un à un. Lors de l'exécution d'un message, un acteur peut créer d'autres acteurs (de manière dynamique), communiquer par l'envoi de messages asynchrones point à point avec les autres acteurs qu'il connaît, et peut aussi changer son propre comportement (définir le comportement pour l'exécution du prochain message). Le changement du comportement peut être utile pour les acteurs qui parcourent le Web car il permet l'évaluation et la compréhension [Ler04a] [Arc02].

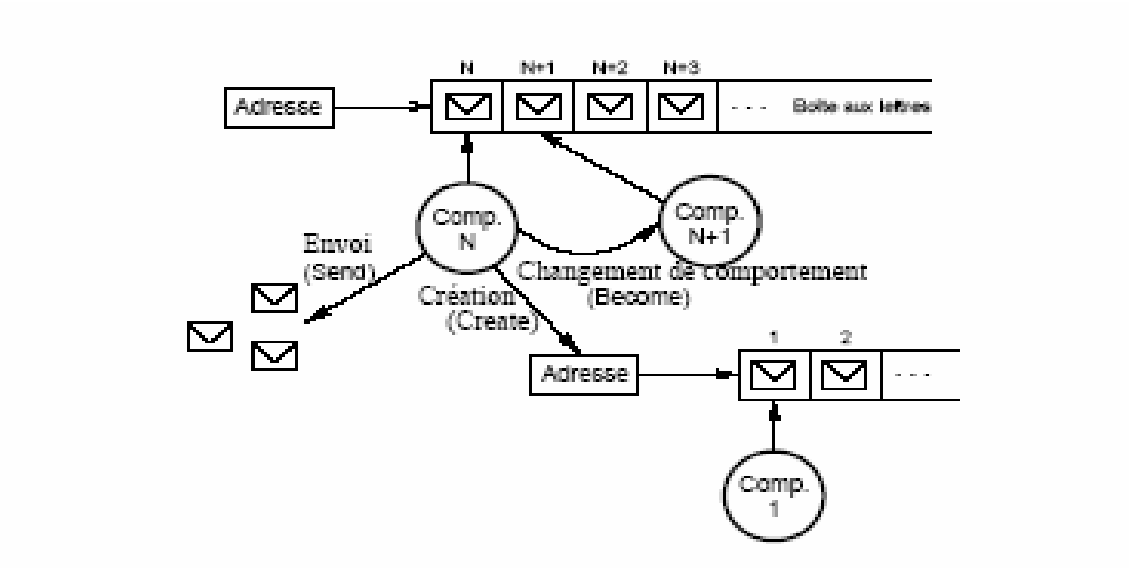


Figure9. Les actions possibles d'un acteur lors du traitement d'un message

Donc, les acteurs peuvent être vus comme des objets actifs avec l'habilité de changer leur interface. Dans les applications, les acteurs comme les agents peuvent être à la fois client et serveur. Grâce à l'autonomie, la transmission asynchrone du message et le changement du comportement, les acteurs sont, naturellement, des unités mobiles [Arc01] [Ler04b] :

- *Autonomie* est une propriété importante que les acteurs doivent avoir pour leur propre décision de la mobilité. L'encapsulation des données et des méthodes dans le comportement privé de l'acteur garantie conceptuellement l'intimité et l'intégrité. Les acteurs n'encapsule pas seulement les programmes et les données mais aussi l'activité : une seule activité par acteur [Arc00] ;

- *Transmission asynchrone de message* est une autre caractéristique importante pour les acteurs mobiles, car les communications synchrones sont coûteuses et difficiles à maintenir dans le contexte des réseaux à grande échelle ou sans fil ;

- *Le comportement de l'acteur* contient toutes les données de l'acteur ainsi que son code. Lors d'un changement de comportement, l'état de l'acteur est entièrement contenu dans le comportement (et dans la boite aux lettres), donc facilement capturable et transférable. En cet instant de transition, il n'y a rien de plus dans l'état d'exécution. Le mouvement est retardé jusqu'à la fin de l'exécution du message. La mobilité des acteurs est basée sur la création distante d'un acteur à partir d'un comportement destiné à une prochaine exécution de message. Par conséquent, l'acteur se déplace en transportant la connaissance et l'expérience acquises. Donc les acteurs se déplacent mais les comportements, durant leur exécution, ne bougent pas [Arc00].

La localisation du déplacement des acteurs est possible en utilisant soit un serveur de noms, soit un système avancé. Dans les systèmes d'acteur, chaque acteur possède une référence unique (une adresse postale) et la localisation est naturelle au moyen d'une chaîne de concepts avancés. Comme les acteurs mobiles se déplacent tout le temps, l'acteur local devient alors un proxy pour cet acteur mobile : il reçoit ses messages et les lui envoie. Une telle méthode autorise les messages, entreposés dans la boîte aux lettres avant de se déplacer, d'atteindre l'acteur après qu'il se soit déplacé. La référence de l'acteur mobile demeure valide même si le déplacement est en cours ou si l'acteur est distant. Aussi, la mobilité est transparente pour les communications (le code de l'agent émetteur n'a pas à être modifié si l'agent en question est mobile ou non). Cependant, ce protocole de base est connu pour être peu efficace et sensible aux erreurs à cause des relais multiples : différent types d'optimisation peuvent être fournis [Arc01].

L'activité des acteurs

Nous avons choisi UML pour la modélisation des activités d'un acteur car UML est le langage standard de modélisation orientée objet [Thi05] [Uml00]. Il propose des notations pour la conception de différents modèles de spécification des aspects comportementaux d'un système en utilisant les digrammes de séquences [Dou06], de collaboration et d'activités qui permettent à la fois, une représentation fonctionnelle et dynamique du logiciel voulu [Car01a][Car01b].

Dans ce qui suit nous définissons les activités d'un acteur en utilisant le diagramme de séquence d'UML [Peg05] [Sig03]. Ces activités sont :

- Envoi de messages
- Migration

Deux modélisations sont possibles :

a) La modélisation centralisée où un acteur est désigné comme acteur intermédiaire afin de coordonner les activités des acteurs. Cet acteur possède les coordonnées de tous les acteurs et route les messages.

b) La modélisation décentralisée où chaque acteur possède les coordonnées de tous les acteurs.

a) La modélisation centralisée :

Lorsqu'un acteur veut migrer vers un autre site, il envoie une demande de migration à l'acteur intermédiaire. La migration se fait par la création d'un nouvel acteur sur le site de destination possédant le même comportement que l'acteur initial. Le nouvel acteur transmet ses coordonnées à l'acteur intermédiaire. Une fois la migration faite, l'acteur initial change son comportement et devient un forwarder, il redirige les messages qui lui sont destinées au nouvel acteur.

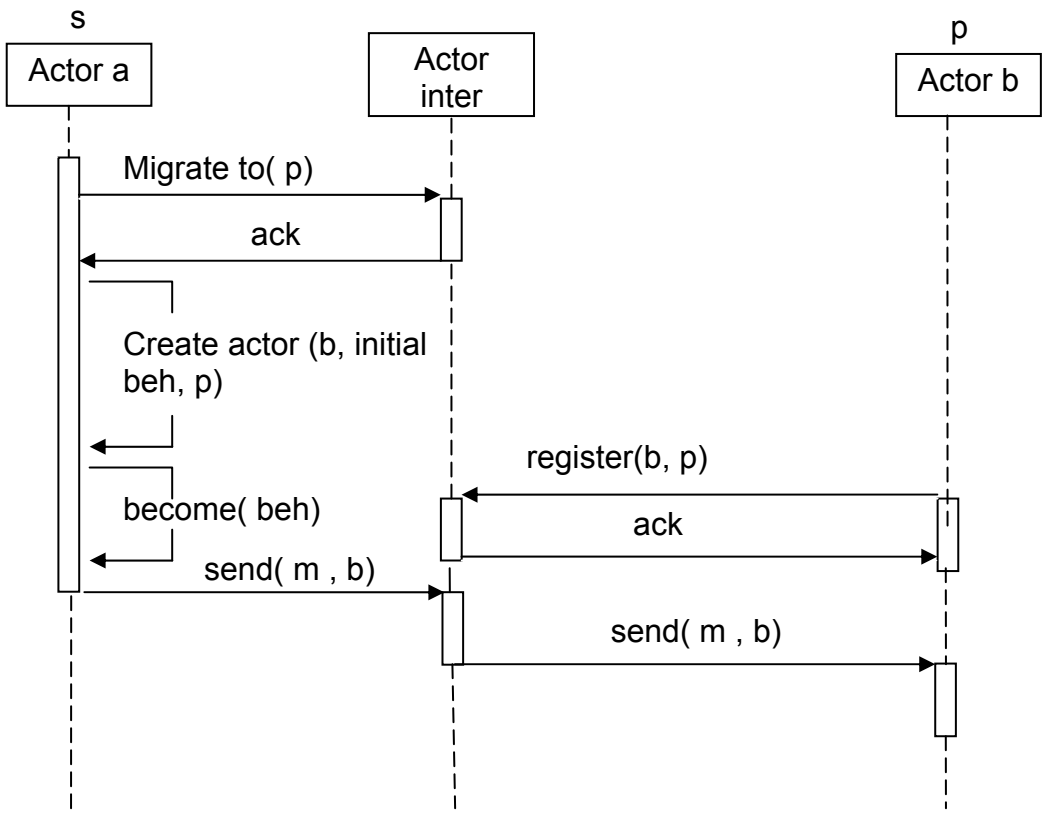


Figure11. Diagramme de séquence pour la migration d'un acteur

Lorsqu'un acteur désire envoyer un message à un autre, il le transmet à l'acteur intermédiaire qui le transmet à son tour à l'acteur de destination. Tous les messages envoyés transitent par l'acteur intermédiaire car celui-ci connaît l'emplacement de chaque acteur.

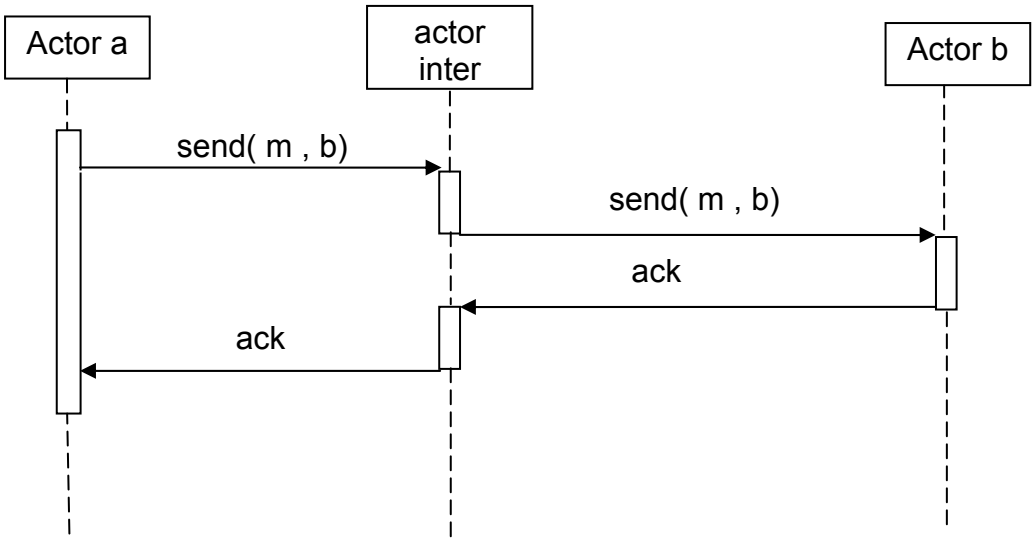


Figure12. Diagramme de séquence pour l'envoi de message entre deux acteurs

Avantage :

- Cette modélisation est plus adaptée à des applications de petite taille qui se trouvent sur un seul site, elle allège les acteurs et leur permet de se consacrer uniquement à leur fonction.

Inconvénient :

- L'inconvénient de cette modélisation est quand l'acteur intermédiaire ne joue plus son rôle de coordinateur pour une raison ou une autre (matériel ou logiciel), il n'y aurait donc plus de communication entre les différents acteurs suite à la perte de leur localisation. Dans ce cas, un second agent intermédiaire est nécessaire afin d'assurer le back up.

b) La modélisation décentralisée :

La migration se fait par la création d'un nouvel acteur sur le site de destination possédant le même comportement que l'acteur initial. Le nouvel acteur transmet ses coordonnées à tous les acteurs qui se trouvent dans le répertoire de l'acteur initial. Une fois la migration faite, l'acteur initial change son comportement et devient un forwarder, il redirige les messages qui lui sont destinées au nouvel acteur.

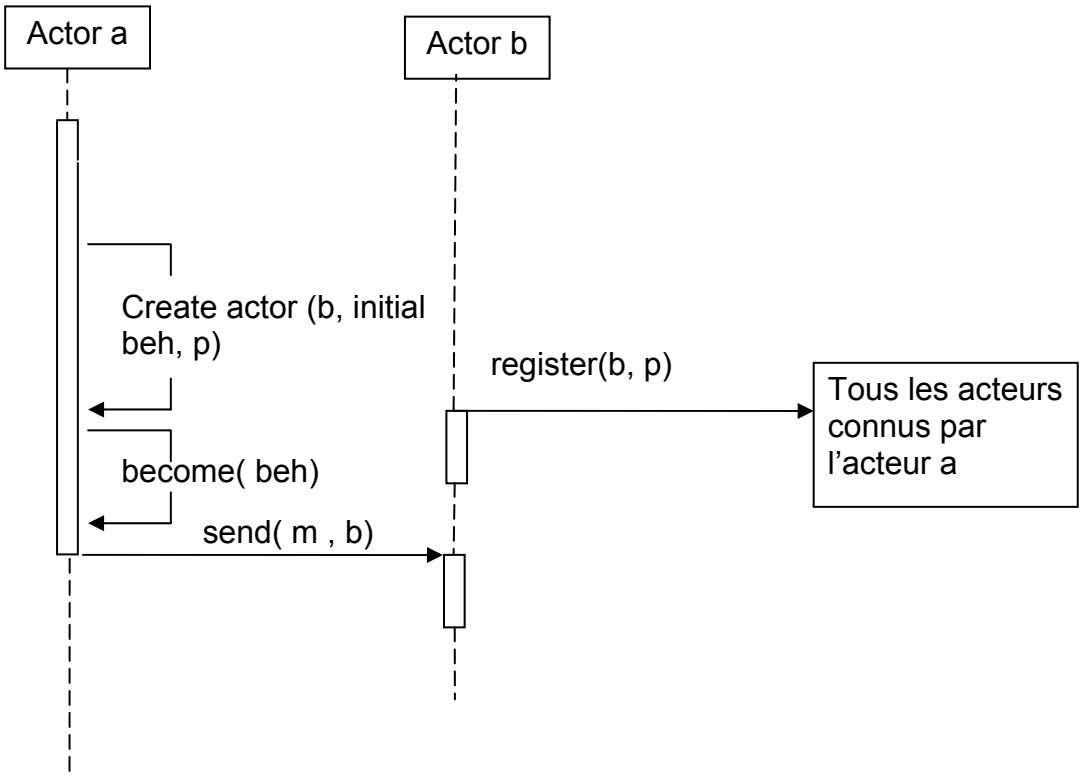


Figure13. Diagramme de séquence pour la migration d'un acteur

Lorsqu'un acteur désire envoyer un message à un autre, il le transmet à l'acteur de destination car celui-ci connaît l'emplacement de chaque acteur.

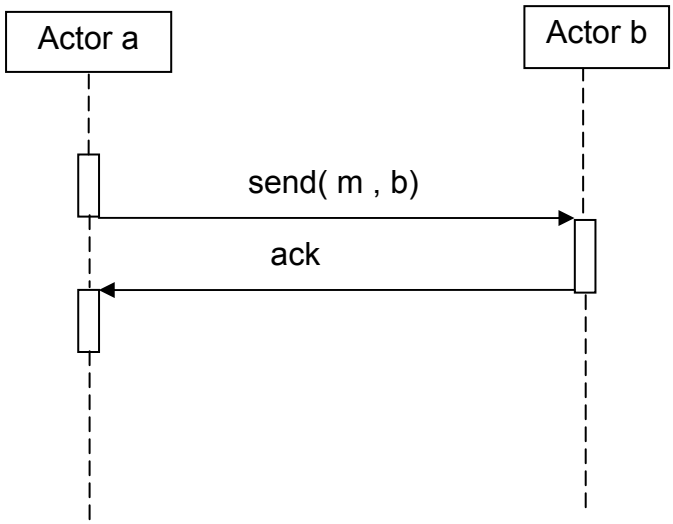


Figure14. Diagramme de séquence pour l'envoi d'un message entre deux acteurs

Avantage :

- Fluidité de la communication : la communication ne transit plus par un acteur intermédiaire ce qui permet donc le back up n'est plus nécessaire.

Inconvénient :

- L'inconvénient de cette méthode réside dans la surcharge des acteurs avec les coordonnées de tous leurs collègues.

c) La nouvelle modélisation :

Nous avons opté pour une modélisation combinée (centralisée et décentralisée à la fois) qui nous semble pour le moment la mieux adaptée à notre environnement. Cette modélisation comprend un acteur intermédiaire dans chaque site physique différent. Cet acteur possède la localisation des acteurs du site physique où il se trouve ainsi que celle de tous les acteurs intermédiaires.

Lorsqu'un acteur veut migrer vers un autre site, il envoie une demande de migration à l'acteur intermédiaire de son site qui la transmet à son tour à l'acteur intermédiaire du site de destination. La migration se fait par la création d'un nouvel acteur sur le site de destination possédant le même comportement que l'acteur initial. Le nouvel acteur transmet ses coordonnées à l'acteur intermédiaire de son site. Une fois la migration faite, l'acteur initial change son comportement et devient un forwarder, il redirige les messages qui lui sont destinées au nouvel acteur.

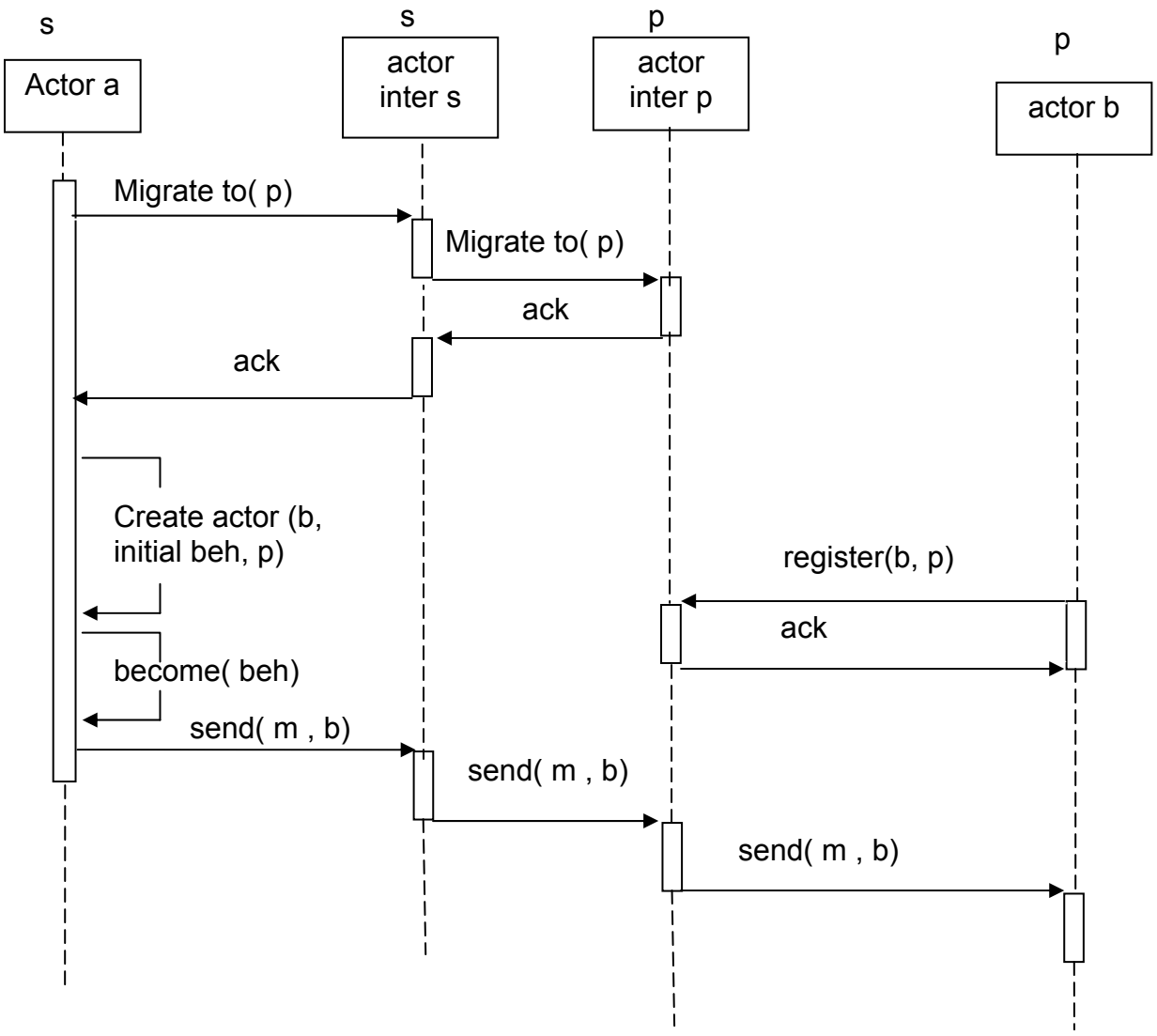


Figure15. Diagramme de séquence pour la migration d'un acteur

Lorsqu'un acteur désire envoyer un message à un autre, il le transmet à l'acteur intermédiaire de son site qui l'envoie à l'acteur intermédiaire du site où se trouve l'acteur de destination. Ce dernier le transmet à son tour à l'acteur de destination.

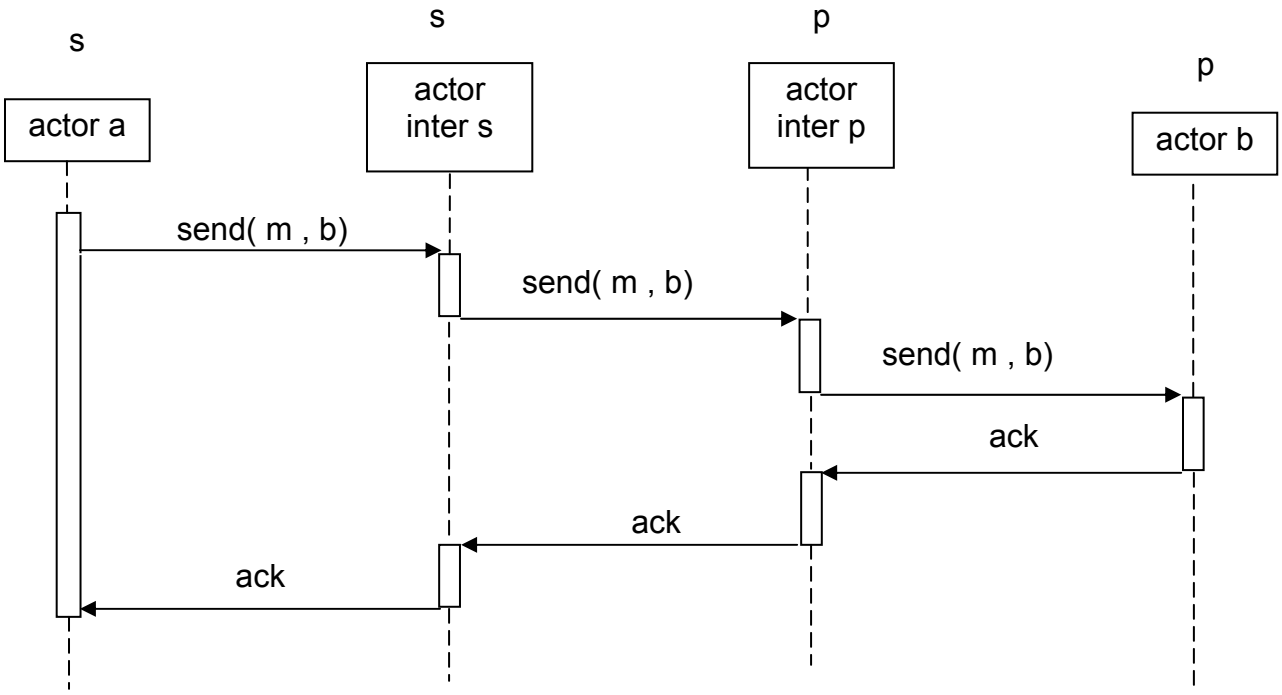


Figure16. Diagramme de séquence pour l'envoi d'un message entre deux acteurs

Avantages :

- Décharger les acteurs des coordonnées des autres acteurs : L'acteur ne perdra pas de temps à chercher dans sa liste les coordonnées des acteurs, il décharge cette tâche à l'acteur intermédiaire;

- Alléger les acteurs intermédiaires : l'acteur intermédiaire disposera de la liste des acteurs se trouvant sur son site ainsi que celle de tous les acteurs intermédiaires;

- Cette méthode convient à la fois aux petites applications qu'aux grandes. pour les petites applications qui se trouvent sur un seul site, cette méthode se transforme en une méthode centralisée. Dans le cas contraire, cette dernière se transforme en une méthode décentralisée;

- La localisation de l'acteur migrant est sauvegardée dans deux endroits différents : en cas de problème au niveau d'un acteur intermédiaire, il est toujours possible de localiser l'acteur migrant.

Inconvénient :

- Prévoir des back up pour tous les acteurs intermédiaires ce qui alourdira le système : chaque acteur intermédiaire devra disposer d'un back up pour sauvegarder la localisation des acteur du site où il se trouve ;

- Lourdeur dans la communication : la communication entre deux acteurs de deux sites différents est lourde car elle doit passer par plusieurs acteurs ce qui peut provoquer la perte de quelques paquets.

IV.2.2) Extension en mobilité et concurrence d'applications à base d'acteurs

Afin d'introduire la mobilité et la concurrence au niveau des applications basées sur des modèles d'acteurs, nous étendons ces applications par des caractéristiques propres à la mobilité qu'offre le calcul formel CAP. Ces caractéristiques sont principalement représentées par les notions de domaine et de groupe intégré au concept d'acteur.

Les acteurs :

Les acteurs font l'objet d'une classe abstraite Actor qui implémente plusieurs interfaces dont celle du comportement et leurs principaux attributs sont :

Acteur = {comportement, domaine, groupe}

Tel que :

- le comportement de l'acteur ;
- le domaine de l'acteur ;
- la liste des groupes de l'acteur ;

Il y'a aussi des composants qui gèrent la réception des messages, le changement de comportement, la création, la migration et l'envoi des messages.

Un acteur est créé en indiquant son comportement initial et à quel domaine il appartient.

a) Les domaines :

Les domaines font l'objet d'une classe abstraite AbstratDomain qui implémente principalement l'interface Actor. Un domaine est formalisé comme suit :

Domaine = {name, url, racine, host, actorList, father, subDomainList}

Tel que:

- name : c'est un nom symbolique ;
- url : l'url du domaine (elle indique le chemin logique pour atteindre le domaine depuis la racine, c'est-à-dire Internet dans son ensemble) ;

- host : la machine qui l'héberge ;
- actorList : la liste des acteurs du domaine ;
- father : le domaine père ;
- subDomainList : la liste des domaines fils ;

En plus de ces attributs, nous rajoutons le composant qui gère la migration et le composant qui gère les politiques d'entrées/sorties.

Les principales méthodes du domaine sont join et exit qui gèrent les entrées/sorties.

Nous avons deux sortes de domaines : le domaine physique et le domaine logique. Leur différence s'exprime dans la position de leurs acteurs :

a1) Domaine physique (ou site physique) :

Les acteurs se trouvent dans le même hôte que le domaine. Quand le domaine migre, tous les acteurs migrent avec lui sur le nouvel hôte. Lorsqu'un acteur entre dans un domaine physique, il doit migrer vers celui-ci. Lorsqu'il en sort, il doit migrer vers le domaine de destination si celui-ci est un domaine physique.

a2) Domaine logique :

Les acteurs ne sont pas sur le même hôte que le domaine. Lorsque le domaine migre, les acteurs ne le suivent pas.

b) Les groupes :

Les groupes, comme les domaines, font l'objet d'une classe abstraite AbstractGroup. Un groupe est formalisé comme suit :

Groupe = {name, host, actorList}

Tel que :

- name : le nom symbolique ;
- host : le site physique où se trouve le groupe ;
- actorsList : la liste de tous les acteurs du groupe ;

Plus le composant qui gère les envois de messages aux éléments d'un groupe et le composant qui gère et les politiques d'entrées/sorties.

Comme pour les domaines, les principales méthodes sont celles qui contrôlent l'entrées/sorties du groupe (join et exit).

La figure ci-dessous montre la structure d'une application dans notre approche, chaque application est divisée en profils acteur, en comportements et en domaines. Tout profil acteur appartient à un domaine et fait appel à une ou plusieurs interfaces comportement.

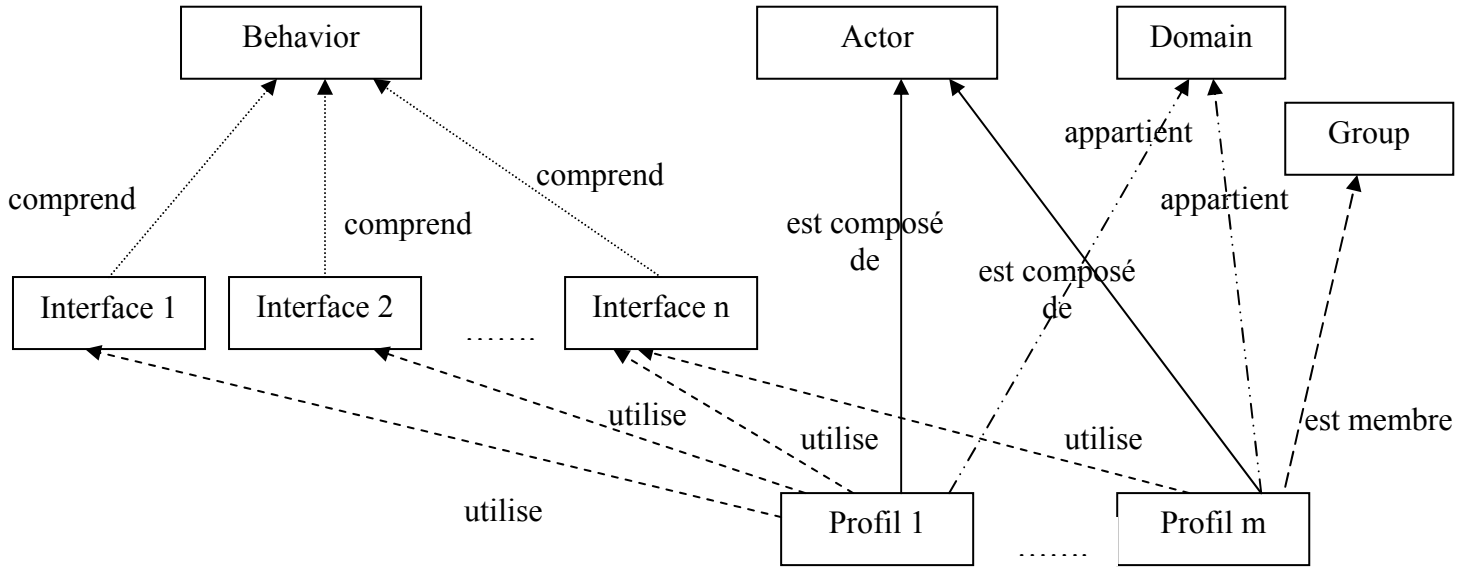


Figure16. Structure d'une application dans l'environnement de programmation

Un profil comprend donc :

- un ou plusieurs comportements

- un sous domaine

- un ou plusieurs groupes

IV.2.3) Exemple d'application

Prenons comme exemple la partie logistique d'une société qui comprend cinq services : la gestion des stocks, les achats, le transit, la réception et la valorisation. Ces services sont situés dans des lieus physiquement différents et contenant chacun un ou plusieurs postes de travail.

Le service gestion des stocks est composé de 2 postes de travail :

- ***gestion des stocks :*** nous représentons ce poste par trois acteurs effectuant des taches différentes. Le premier enregistre les sorties du magasin et met à jour le stock et le second s'occupe du réapprovisionnement (lorsque le stock arrive à un seuil limite une commande de réapprovisionnement est automatiquement lancée). Quant au dernier, il traite les limites de stockage, une fois un matériel arrivé à sa limite de stockage, l'acteur le met en quarantaine puis crée un acteur qui se charge de son suivi puis se tue à la fin de sa tache.

- ***traitement des requêtes :*** ce poste comprend un acteur qui vérifie la requête et si elle est conforme (le stock en magasin est insuffisant ou nul, référence correcte, quantité demandée ne dépasse pas le seuil maximal d'approvisionnement)

Le service achats est composé de 2 postes de travail :

- ***prospection :*** dans cette tache un acteur s'occupe de contacter les fournisseurs concernés par le matériel à acheter se trouvant sur sa liste, il reçoit par la suite leurs offres qu'il traite.

- **suivi des commandes :** une fois le fournisseur choisi, un acteur établit la commande et fait son suivi jusqu'à ce qu'elle soit soldée.

Le service transit est composé aussi de 3 postes de travail :

- ***domiciliation des factures :*** un seul acteur se charge de cette tache qui consiste à transmettre la valeur du matériel commandé à la banque puis récupérer le numéro de la domiciliation attribué par cette banque.

- ***positionnement des commandes :*** l'acteur représentant cette partie effectue une recherche dans la base de données et affecte une position au matériel déclaré.

- ***Dédouanement :*** cet acteur récolte l'information nécessaire pour le dossier dédouanement.

Le service réception est composé d'un poste de travail :

- ***réception des commandes :*** deux acteurs s'occupes de la réception des commande, le premier enregistre le matériel reçu ainsi que la facture définitive puis effectue une vérification avec le matériel commandé, si cela correspond il valide la réception, sinon il envoie l'information au second acteur qui ouvre un dossier litige et d'après sa liste des litiges, il effectue la transaction correspondante au type de litige.

Le service valorisation est composé de 2 postes de travail :

- ***valorisation des stocks :*** ce poste comprend deux acteurs, le premier s'occupe de la valorisation du matériel reçu (il calcule la valeur total qu'à coûter ce matériel en introduisant au prix de celui-ci le prix de transport et de dédouanement). Le second acteur fait une recherche régulière sur les sites des fournisseurs pour mettre à jour la valorisation du stock.

- ***apurement :*** un seul acteur intervient dans cette tache, il traite le mode paiement (lettre de crédit ou virement) et récupère la valeur du matériel reçu afin d'apurer cette valeur au niveau de la lettre de crédit.

Dans chaque site physique nous avons un acteur intermédiaire qui coordonne entre les différents acteurs. Chacun de ces acteurs dispose des coordonnées des acteurs se trouvant sur son site ainsi que celles des autres acteurs intermédiaires.

Chaque service représente un domaine et chaque poste un groupe, donc nous disposons de 5 domaines :

- Stock ;
- Achats ;
- Transit ;
- Réception ;
- Valorisation.

Et 10 groupes :

- Gestion des stocks ;
- Traitement des requêtes ;
- Prospection ;
- Suivi des commandes ;
- Domiciliation ;
- Positionnement ;
- Dédouanement ;
- Réception ;
- Valorisation ;
- Apurement.

Les 14 acteurs peuvent avoir les comportements suivants :

- Recherche d'information ;
- Création d'un acteur ;
- Contrôle des données ;
- Calculs ;
- Transfert de données ;
- Suivi ;
- Mise à jour des données.

Nous schématisons, dans ce qui suit, les acteurs du groupe gestion des stocks et ceux du groupe valorisation :

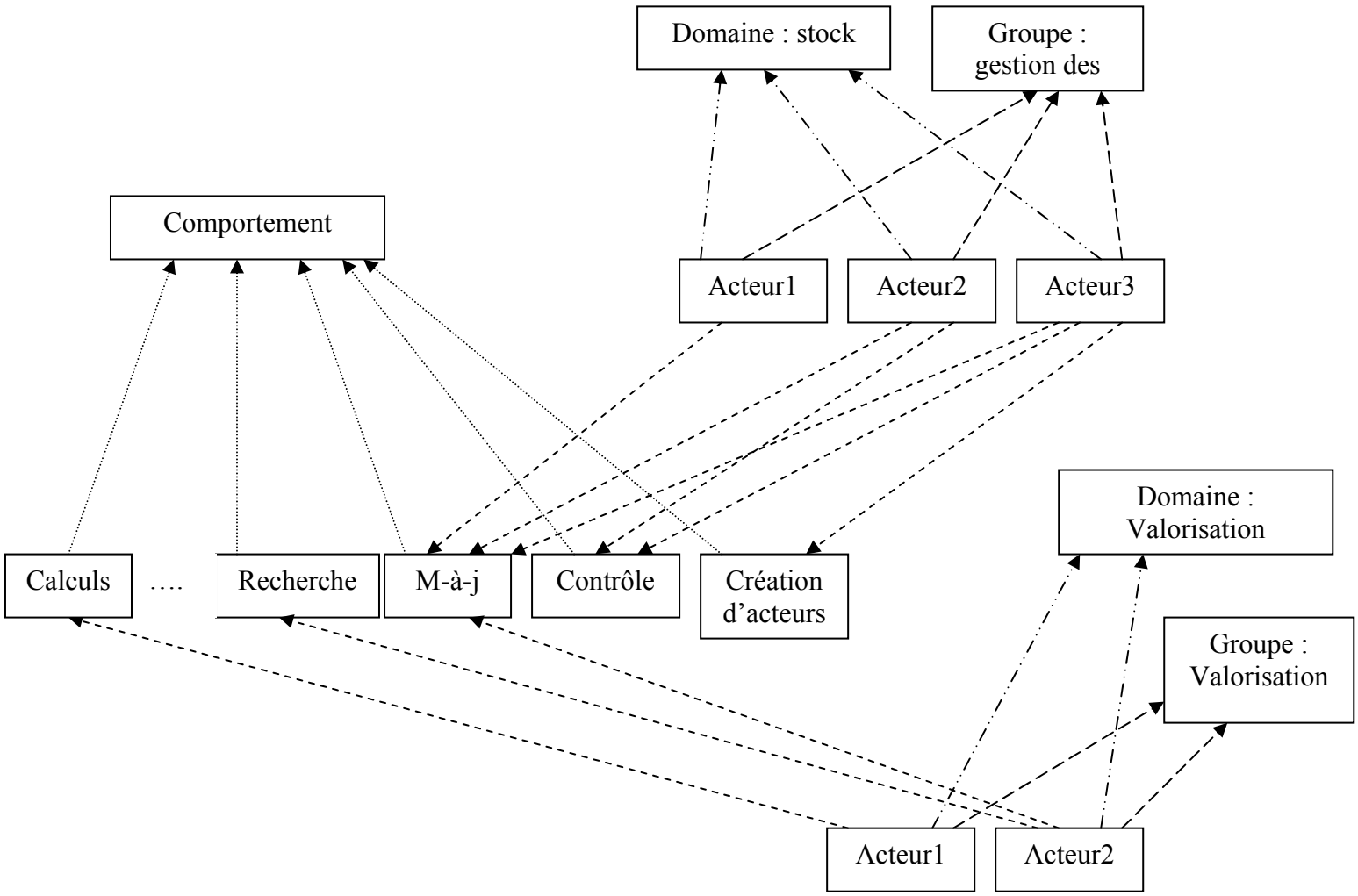


Figure17. Structure d'une partie de l'application dans l'environnement de programmation

IV.3) Conclusion

Dans ce chapitre nous avons présenté notre approche d'environnement de développement d'applications distribuées, concurrentes et mobiles basé sur le modèle d'acteurs et intégrant les caractéristiques du calcul formel CAP. Nous avons commencé par représenter les activités de l'acteur en utilisant le diagramme de séquence d'UML. Ensuite, nous avons proposé une extension du langage de programmation par l'intégration des aspects de concurrence et mobilité du calcul formel CAP. Cette extension est basée essentiellement sur l'introduction des notions de domaines et de groupes. Nous avons terminé ce chapitre par un exemple d'application sur la logistique commerciale d'une société.

Conclusion et perspectives

Le code mobile est une solution prometteuse pour la conception et l'implémentation des applications distribuées et mobiles à grande échelle mais cela accroît la complexité et la difficulté dans le développement et la maintenance de ces applications à base d'objets concurrents et mobile.

Le point de départ de ce travail consistait à voir l'implication du code mobile dans le domaine du génie logiciel et quels étaient les problèmes et difficultés engendrés par cette mobilité. De ce fait, nous nous sommes intéressés directement aux problèmes liés à la conception d'environnement de développement d'application distribuées, concurrentes et mobiles. L'environnement sur lequel est basé notre approche, intègre les concepts de distribution et mobilité du calcul formel CAP qui est un calcul dédié aux acteurs. Ce calcul permet de définir plusieurs analyses statiques pour assurer que les messages envoyés vers un acteur seront pris en compte par celui-ci.

Nous avons donc établi, dans une première partie, un état de l'art sur la modélisation conceptuelle que nous avons structuré en trois classes de concepts : les applications, les paradigmes de conception et les technologies liés au code mobile.

- Les applications représentent des solutions à des problèmes spécifiques, dans une application distribuée conçue avec des techniques conventionnelles, une nouvelle fonctionnalité a besoin d'être introduite par la réinstallation ou la structuration de l'application dans chaque site. L'habilité à solliciter à la demande le lien dynamique du fragment du code en implémentant la nouvelle fonctionnalité permet la centralisation des changements sur le serveur d'où la disponibilité à tout moment de la dernière version. De plus, les changements ne vont pas être exécutés proactivement par un opérateur dans chaque site, ils vont être exécutés réactivement par l'application elle même, qui va solliciter automatiquement la nouvelle version du code qui se trouve sur le serveur.

- Les paradigmes guident la conception des applications dont la plupart connus sont statiques en respectant le code et l'emplacement. Une fois créés, les composants ne peuvent changer ni leur emplacement ni leur code durant toute leur vie. Les paradigmes du code mobile dépassent ces limites en fournissant un composant mobile. Par le changement de leur emplacement, les composants peuvent changer dynamiquement la qualité de l'interaction, en réduisant ses coûts.

- Les technologies du code mobile fournissent tous les concepts et primitives qu'utilise le code, elles représentent les langages de programmation et leurs supports. Les langages du code mobile donnent une nouvelle orientation aux langages de programmation destinés aux systèmes distribués. Nous avons analysé un ensemble de langages du code mobile utilisés

concurremment, en proposant un ensemble initial de concepts qui ont été utilisés pour évaluer et comparer les différents langages.

Dans la même partie nous avons montré l'implication du code mobile dans le domaine du génie logiciel à partir de travaux de recherche portant sur le développement et l'exploitation (exécution, maintenance) d'applications réparties à grande échelle et mobile.

Cependant, la technologie du code mobile a augmenté la complexité, le coût du développement logiciel, de sa maintenance et a affecté l'intégrité et la sécurité des applications réparties, concurrentes et mobiles. Afin de faciliter la conception de telles applications et assurer leur validité et leur sûreté, nous avons proposé une approche de conception d'un environnement de développement intégrant des mécanismes d'analyse statique des programmes. Cette approche est basée sur le modèle d'acteurs et intègre les caractéristiques du calcul formel CAP. Dans notre approche nos travaux reposent en premier sur la représentation des activités de l'acteur en utilisant le diagramme de séquence d'UML. Nous proposons une modélisation mixte (centralisée et décentralisée à la fois) des activités de l'acteur en utilisant un acteur intermédiaire dans chaque site physique. En second lieu, nous proposons une extension du langage de programmation par l'intégration des aspects de concurrence et mobilité du calcul formel CAP. Cette extension est basée essentiellement sur l'introduction des notions de domaines et de groupes.

Ce travail ouvre plusieurs perspectives :

- Enrichissement de la définition des activités des acteurs, en exploitant la notion de méta-modèle dans UML [Cia03] [Fow04];
- Validation du transfert des caractéristiques de CAP vers les langages de programmation;
- Renforcement de la sécurité dans CAP ;
- Conception d'un compilateur pour la portabilité de l'environnement ;
- Permettre au développeur de choisir la modélisation d'acteurs à utiliser dans la partie conception.

Références

[Bebb06] : S. Bebbouchi, M. Ahmed-Nacer,
« Un environnement de programmation pour des applications distribuées, concurrentes et mobiles » Poster au colloque COSI06, Mai 2006

[Thor97] : Tommy Thorn
« Programming languages for mobile code » Rapport de recherche n_3134| 37 pages, Mars 1997

[Fugg98] : A. Fuggetta, G-P. Picco, et G. Vigna
« Understanding Code Mobility » IEEE Transactions on software engineering, Vol. 24, N° 5, Mai 1998

[Cug98] : G. Cugola, G-P. Picco, et G. Vigna
« Analyzing Mobile Code Languages » Dip. Elettronica e Informazione, Politecnico di Milano, Italie, 1998

[Krak99] : Sacha Krakowiak
« Code mobile Principes et mise en œuvre » Université Joseph Fourier Labo. Sirac SK, 1999

[Bri00] : Thomas Ledoux
« Le bus logiciel OpenCorba » conférence *LMO'99 - Langages et modèles à objets*, Villefranche-sur-Mer, France, Janvier 1999

[Dev01] : D. Deveaux et Y. Le Traon
« XML to Manage Source Engineering in Object-Oriented Development: an Example » In C. Mascolo, W. Emmerich and A. Finkelstein, editors, *Xml technologies and software engineering*, pages 28-31.XSE01 workshop at ICSE'2001, Toronto, Canada, Mai 2001.

[Arc01] : J-P. Arcangeli, C. Maurel, et F. Migeon
« An API for high-level software engineering of distributed and mobile applications » IEEE-CS Press, pp.155-161, 2001

[Arc02]: J-P. Arcangeli, A. Hameurlain, F. Migeon, et F. Morvan
« An Adaptive Hash Join Algorithm using Mobile Agents » Proceeding of JaDa'02, Erfurt (Ge.), 2002

[Arc03] : J-P. Arcangeli, C. Maurel, et F. Migeon
« Ingénierie des Applications Mobiles » IRIT-ENSEEIHT Activités de recherche de l'équipe I.A.M. 10 janvier 2003

[Hur04] : A. Hurault et M. Pantel
« Répartition et mobilité en JavAct ; Une approche dérivée d'un modèle formel » Actes de la conférence LMO'2004 Langages et Modèles à Objets, 15-17 mars 2004 à Lille

[Hen03] : V. Hennebert et M. Pantel
« Typage de comportements non uniformes » Actes de la conférence LMO'2004 Langages et Modèles à Objets, 2004

[Hura04] : Haurault Aurélie et Marc Pantel
« Introduction de la répartition et de la mobilité dans le calcul concurrent CAP: Étude bibliographique » présenté à la conférence LMO'2004 Langages et Modèles à Objets, 15-17 mars 2004 à Lille

[Col97] : J-L. Colaço, M. Pantel et P. Sallé
« Elimination des messages orphelins par typage dans un calcul d'Acteurs Primitifs » Rapport Technique Interne IRIT/LIMA/INPT, Avril 1997

[Ler04a] : S. Leriche, J-P Arcangeli, M. Pantel,
« Agents mobiles adaptables pour les systèmes d'information pair à pair hétérogènes et répartis » *Actes des NOuvelles TEchnologies de la REpartition NOTERE*, p. 29-43, 2004b
Maroc, 2004

[Ler04b] : S. Leriche, J-P Arcangeli,
« Déploiement et adaptation de systèmes P2P : une approche à base d'agents mobiles et de composants » JMAC 2004 (Journée Multi-agents et Composants), Paris, 2004

[Kaf 93]: D. Kafura, M. Mukherji et G. Lavender
« ACT++ 2.0: A Class Library for Concurrent Programming in C++ Using Actors » Journal of Object-Oriented Programming pp. 47-55, October 1993

[Arc00]: J-P. Arcangeli, L. Bray, A. Marcoux, C. Maurel, F. Migeon,
« Reflective actors for mobile agents programming » In. ECOOP'2000 Workshop on Reflection and Meta-level Architecture, Cannes, France, 2000

[Rom00]: G-C. Roman, G.P. Picco, A.L. Murphy,
« Software Engineering for Mobility: A Roadmap » In The Future of Software Engineering, A. Finkelstein (Ed.), ACM Press, pp. 241-258, 2000

[Gui01]: F. Guidec
« Prise en compte des ressources dans les composants logiciels parallèles » Rencontres grappes FG/UBS/VALORIA/05-01, 2001

[Src03]: Fabrice KORDON
« Systèmes répartis et coopératifs » bilan synthétique des recherches pages 273-284 Mai 2003

[Kta03] : B. Ktari,
« Certification de logiciels » Séminaire du département d'informatique et de génie logiciel, Université Laval, 26 Février 2003

[Dev00]: D. Deveaux, G. St.Denis, R.K. Keller,
« XML Support to Design for Testability » In Proc. of XOT'2000 workshop at ECOOP'2000, Cannes (France), Juin 2000

[Ghe98] : C. Ghezzi, G. Vigna
« Mobile Code Paradigms and Technologies: A Case Study» Proceedings of the First International Workshop on Mobile Agents, 1998

[Oli03] : D. Olivier
« Code mobile: Un exemple », DEA- Modélisation et implémentation des systèmes complexes, Année 2002/2003

[Arc04] : J-P. Arcangeli, V. Hennebert, S. Leriche, F. Migeon, M. Pantel.
« JavAct 0.5.0 : principes, installation, utilisation et développement d'applications » Rapport IRIT/ 2004-5-R, 2004

[Car01a] : J. Cardoso, C. Sibertin-Blanc, C. Soulé-Dupuy
« Une sémantique formelle des diagrammes d'interaction d'UML via les réseaux de Petri » In modélisation de systèmes réactifs (Actes MSR 2001, Toulouse, France), pg 497-512, ISBN 2-7462-0329-4, Hermès, Toulouse, France, 2001

[Car01b] : J. Cardoso, C. Sibertin-Blanc
« An operational semantics for UML interaction : sequencing of actions and local control » JESA. Volume X – n° X/2001, 2001

[Peg05] : F. Peguiron, O. Thiery
« Modélisation des acteurs et des ressources : application au contexte d'un SIS universitaire » Conférence 'ISKO-France', 28-29 Avril 2005

[Hen04] : V. Hennebert, M. Pantel
« Typage de JavAct : Approche dérivée du typage de CAP » *Journées FAC'2004* Formalisation des Activités Concurrentes, Toulouse, 9-10 mars 2004

[Che00] : D. Chemouil
« Architecture répartie pour un langage fonctionnel d'acteurs » DEA Programmation et systèmes, INP Toulouse, Année 1999/2000

[Dag97] : F. Dagnat
« Conception d'un langage fonctionnel d'acteurs et réalisation de son compilateur », DEA Informatique Fondamentale et parallélisme, INP, Année 1996/1997

[Arc03]: J.-P. Arcangeli, A. Hameurlain, F. Migeon, and F. Morvan
« Mobile Agent Based Self-Adaptive Join For Wide-Area Distributed Query Processing » *Journal of Database Management;* 15, 4; ProQuest Computing Pg25.
Oct-Dec 2004

[Sig03]: O. Sigaud
« Introduction à la modélisation orientée objets avec UML » support du cours in204 « Génie logiciel et programmation orientée objets » de l'ENSTA, 2003

[Col99]: J-L. Colaço, M. Pantel, F. Dagnat et P. Sallé
« Static safty analysis for non-uniform service availability in Actors » Formal Methods for Open Object-based Distributed Systems, February 1999

[Col96] : J-L Colaço, M. Pantel et P. Sallé
« Cap_ An Actor dedicated process calculus » Article publié dans les Journées Francophones des Langages Applicatifs, Janvier 1996

[Agh01]: G. Agha, P. Thati, R. Ziaei
« Actors: A Model for Reasoning about Open Distributed Systems » In H. Bowman and J. Derrick (editors), Formal Methods for Distributed Processing - An Object Oriented Approach, Chap. 8, Cambridge University Press, 2001

[All98]: Arthur Allen
« Actor-baser computing: Vision forestalled, vision fulfilled » 2nd international conference on autonomous agents, 1998

[Fow04] : Martin Fowler
« UML 2.0 », Edition CampusPress
Année de la publication 2004.

[Thi05] : T. Cros
« Concepts et Formalismes UML » www. Thierrycros.net, 2005

[Dou06] : J-M. Doudoux
« Java et UML » perso.wanadoo.fr/jm.doudoux/java/tutorial/chap048.htm, 2006

[Cia03] : Cian
« Nouveautés UML 2.0 » cian.developpez.com/uml2/tutoriel/,
3 Décembre 2003

[Gnu03] : GNU Project and the Free Software Foundation www.gnu.org , 2003

[Uml00] : L. Piechocki
« Cours UML » http://uml.free.fr/index-cours.html, 2000

[Act00]: GNU Free Documentation License
« Actor Model » http://actor-model.brainsip.com, 2000

[Iri05]: I.R.I.T
« JAVACT : un intergiciel Java pour les agents mobiles adaptables »
http://www.irit.fr/recherches/ISPR/IAM/JavAct_fr.html, 2005

UNIVERSITE DES SCIENCES ET DE LA TECHNOLOGIE HOUARI BOUMEDIENE

FACULTE DE GENIE ELECTRIQUE
Laboratoire des systèmes d'information
Thèse de magister Option Intelligence Artificielle et Bases de données avancées
Intitulé

Environnement de développement d'applications réparties, concurrentes et mobiles

Présentée par : Mme BEBBOUCHI Sabrina

Résumé. Les applications réparties à l'échelle du réseau mondial, et en particulier les systèmes d'information, sont complexes par leur nature décentralisée et ouverte. De part la décentralisation, la technologie du code mobile a donné une nouvelle orientation aux systèmes distribués ; elle a permis plus de flexibilité dans leur conception et implémentation. Cependant, cette nouvelle technologie a augmenté la complexité, le coût du développement logiciel, de sa maintenance et a affecté l'intégrité et la sécurité des systèmes répartis. Afin de pallier à ces problèmes des travaux ont été menés dans le domaine du génie logiciel. Dans ce contexte notre thèse présente une approche d'un environnement de développement d'applications concurrentes et mobiles exploitant le modèle d'acteur. Nous exposons dans un premier temps les technologies du code mobile. Nous proposons, ensuite, notre approche d'environnement de programmation d'applications mobiles, basée sur le modèle d'acteurs et intégrant les caractéristiques du calcul formel CAP.

Mots clés : code mobile, génie logiciel, acteur, calcul formel

Abstract. The distributed applications on the world network scale, and in particular the information systems, are complex by their decentralized and opened nature. Of share decentralization, the technology of the mobile code gave a new orientation to the distributed systems; it allowed more flexibility in their design and implementation. However, this new technology increased the complexity, the cost of the software development, its maintenance and has affected the integrity and the safety of the systems distributed. In order to resolve these problems, many works were carried out in the field of the software engineering. In this context our these presents an approach of environment of development of concurrent and mobile applications exploiting the actor model. We initially expose the technologies of the mobile code then we propose our approach of programming environment for mobile applications, based on the actor model and it integrate the characteristics of formal calculus CAP.

Key words: mobile code, software engineering, actor, formal calculus

---

Directeur de thèse: Professeur M. AHMED-NACER