N° d'ordre : XXXXX

Université Toulouse III - Paul Sabatier
Institut de Recherche en Informatique de Toulouse (IRIT)

THÈSE

Architectures à composants et agents pour la conception d'applications réparties adaptables

Pour obtenir le grade de :
Docteur de l'Université Toulouse III
Mention Informatique

Sébastien Leriche

Equipe d'accueil : ingénierie des Langages
pour les sYstèmes Répartis et Embarqués (LYRE)

Ecole Doctorale : Informatique et Télécommunications (EDIT)
Filière : Sûreté du Logiciel et Calcul à haute Performance (SLCP)

Directeur de thèse : Jean-Paul Arcangeli

Soutenue le xx/xx/2006 devant la commission d'examen :

...	Président	...
Jean-Paul Arcangeli	Directeur de thèse	IRIT - Univ. Toulouse III
Pr. Guy Bernard	Rapporteur	INT - Evry
Pr. Jacques Ferber	Rapporteur	LIRMM - Univ. Montpellier II
...	Examinateur	...
...	Examinateur	...

Résumé : Les systèmes informatiques modernes sont distribués, pervasifs (embarqués, enfouis), hétérogènes, déployés sur des réseaux à grande échelle et sur des machines administrées indépendamment. Les environnements qui supportent leur exécution sont instables et les applications doivent faire face à la volatilité des ressources et des services. Elles doivent être flexibles et être capables de s'adapter dynamiquement.

Pour concevoir des applications en tenant compte des besoins et des problèmes liés à ce contexte, il est nécessaire de se baser sur des technologies logicielles adéquates. Les principaux besoins à prendre en compte lors du développement sont la localisation des ressources, l'organisation des traitements répartis, la sûreté de fonctionnement et la sécurité. Les objectifs de cette thèse sont de proposer et d'évaluer des technologies logicielles qui contribuent à maîtriser la complexité du développement, du déploiement et de la maintenance d'applications réparties adaptables.

Dans ce mémoire, nous faisons des propositions à base d'une combinaison de technologies : composants, agents logiciels, intergiciels (ou *middlewares*), systèmes adaptables. Après avoir évalué les apports et les limites de ces technologies prises individuellement, nous proposons des architectures qui exploitent leur complémentarité. Nous présentons un modèle d'agent mobile adaptable configurable statiquement, capable de se reconfigurer dynamiquement pour s'adapter aux variations de son contexte d'exécution, réalisé à partir d'un assemblage de micro-composants remplaçables et spécialisables. A partir de ce modèle d'agent mobile adaptable, nous proposons un patron de conception pour la mise en œuvre de systèmes répartis à grande échelle, basé sur le mode pair à pair et le déploiement adaptatif de composants logiciels. Enfin, nous proposons un modèle d'architecture flexible d'agent dans lequel différents assemblages de micro-composants permettent d'engendrer différents modèles d'agents, chacun adapté à un besoin applicatif spécifique. La vérification des assemblages est une préoccupation du modèle. Pour chaque proposition, nous présentons les prototypes que nous avons réalisés (agent mobile adaptable et patron de conception) ainsi qu'un environnement de développement pour la modélisation des styles d'agents, la vérification des assemblages de micro-composants et la génération de leurs squelettes d'architectures.

Mots-clés : agents mobiles, composants logiciels, déploiement, adaptation, répartition à grande échelle, grille, pair à pair, intergiciel

Ces travaux ont été réalisés au sein de l'IRIT :

Institut de Recherche en Informatique de Toulouse - UMR5505.
IRIT, Université Paul Sabatier
118 Route de Narbonne
F-31062 TOULOUSE CEDEX 9

Abstract : Modern information processing systems are distributed, pervasive (embarked, hidden), heterogeneous ; they are deployed on large scale networks and on independently managed computers. Environments which support their execution are unstable and applications must face the volatility of resources and services. They must be flexible and able to adapt themselves dynamically.

To build applications by taking into account the needs and the problems involved in this context, it is necessary to use adequate software technologies. Principal needs to take into account during development are localization of resources, organization of distributed treatments, reliability and safety. The objectives of this thesis are to propose and evaluate software technologies which contribute to control the complexity of the development, deployment and maintenance of adaptive distributed applications.

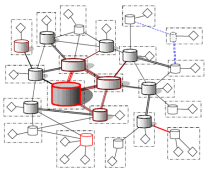
In this memorandum, we make proposals containing a combination of technologies : software components, agents, middlewares, and adaptive systems. After having evaluated the contributions and the limits of these technologies taken individually, we propose architectures which exploit their complementarity. We present a model of configurable adaptive mobile agent, able to be reconfigured dynamically to adapt himself to variations of its context of execution, begining from an assembly of replaceable and specializable microcomponents. From this model of adaptive mobile agent, we propose a design pattern for building distributed systems, based on the peer-to-peer model and the adaptive deployment of software components. Lastly, we propose a model of flexible architecture for agents in which various assemblies of microcomponents make it possible to generate various models of agents, each one adapted to a specific applicative need. For each proposal, we present the prototypes we produced (adaptive mobile agent and design pattern) as well as an environment of development for the modeling of the styles of agents and the generation of their skeletons of architectures.

Keywords : mobile agents, software components, deployment, flexibility, large-scale distribution, grid, peer-to-peer models, middleware

Remerciements

Ce document a été réalisé avec des logiciels libres : LaTeXpour la mise en page, xfig pour les graphiques, Gimp pour les captures d'écran et retouches d'images. Les prototypes ont été développés principalement sous Eclipse et NetBeans. Le fichier pdf a été généré le 6 octobre 2006.

Si vous avez peur de tout lire, essayez cette page[1] !



DANS notre vie quotidienne, les systèmes informatiques sont de plus en plus présents, et il est probable que demain, la plupart des objets qui nous entourent seront informatisés et communiqueront entre eux. Cet ensemble de systèmes, interconnectés à la fois au niveau d'une maison et éventuellement reliés au reste du monde par l'intermédiaire de réseaux comme Internet (on parle de *système réparti à grande échelle*), offre d'énormes possibilités.

On peut ainsi envisager un système de navigation embarqué dans votre voiture qui vous indique les trajets adaptés aux conditions de circulation et de météo en exploitant les informations véhiculées par les autres usagers. Bien sûr, d'autres scénarios existent, y compris dans le monde des systèmes d'informations industriels (administration de logiciels, sûreté de fonctionnement...).

Aujourd'hui, la construction de tels programmes informatiques est possible, mais c'est une tâche complexe et difficile à mettre en œuvre, par manque de technologies appropriées. C'est pour cela que nous essayons de proposer des solutions de nature *génie logiciel*, c'est-à-dire des méthodes et des outils permettant à des programmeurs de gagner du temps, tout en rendant leur production plus sûre et plus fiable. De ce fait, nous n'avons pas pour objectif de proposer des logiciels finis dont le développement revient plutôt à des sociétés d'ingénierie.

Pour simplifier le développement des applications réparties à grande échelle, notre équipe de recherche propose d'utiliser des petits programmes autonomes appelés *agents*. Ceux-ci peuvent se déplacer de machine en machine, et faire des opérations pour le compte de l'utilisateur (humain ou programme) qui les y a envoyés. Dans ma thèse, j'essaie de proposer un modèle de conception générique pour les applications réparties, qui repose entre autre sur l'utilisation de la technologie agent. D'autre part, j'essaie de contribuer à l'amélioration du fonctionnement des agents, en leur fournissant des mécanismes d'adaptation leur permettant de changer de forme en cours d'exécution. En effet, cela leur permet de s'adapter au mieux à leur contexte d'exécution (environnement matériel, logiciel et réseau) pour être plus efficaces.

Je propose également une méthode pour fabriquer des agents spécialisés à partir de petits morceaux de programmes qui réalisent chacun une tâche spécifique, c'est le *modèle d'agent flexible*. Ainsi, construire un agent revient à assembler des bouts de programmes, et il est possible de construire des agents spécialisés pour des domaines d'applications différents (agents réactifs, agents intelligents, agents mobiles...) en réutilisant les composantes existantes. Ce modèle bénéficie toujours des capacités d'adaptation dynamique présentées plus haut.

Maintenant, si vous voulez en savoir plus... il ne vous reste plus qu'à continuer la lecture !

[1] Cette page est destinée avant tout à mes parents et amis qui auraient envie de lire cette thèse mais qui n'ont pas tous fait des études d'informatique (personne n'est parfait ;-)). J'espère leur donner une vision plus claire de ce sur quoi j'ai travaillé pendant trois ans...

Table des matières

Table des figures

Chapitre 1

Introduction

Les systèmes informatiques modernes sont distribués, pervasifs (embarqués, enfouis), hétérogènes, déployés sur des réseaux à grande échelle et sur des machines administrées indépendamment. Les systèmes opératoires qui supportent leur exécution sont souvent instables et les applications doivent faire face à la volatilité des ressources, en étant dynamiquement adaptables. Les principaux besoins à prendre en compte lors de leur développement sont la localisation des ressources, l'organisation des traitements répartis, la sûreté de fonctionnement et la sécurité. Ce chapitre précise le contexte dans lequel se situe cette thèse, puis les travaux menés par mon équipe d'accueil (LYRE) au sein de l'Institut de Recherche en Informatique de Toulouse. A partir de quelques exemples concrets, il présente les objectifs de cette thèse qui sont de proposer des technologies logicielles contribuant à maîtriser la complexité du développement, du déploiement et de la maintenance d'applications réparties de cette nature.

1.1 Contexte

La démocratisation des moyens informatiques permet à la plupart des particuliers de disposer maintenant d'un ordinateur et d'un accès à un réseau. Au sein d'un foyer comme d'une entreprise, de nombreux produits auparavant électroniques deviennent informatisés. Pour le constater il suffit de considérer l'évolution des objets qui nous entourent, du réfrigérateur au modem en passant par l'aspirateur devenu robot, ou tant d'autres éléments de domotique, sans oublier les éléments nomades tels les téléphones portables, les ordinateurs portables (PDA, micro-portable) et peut-être bientôt d'autres formes de micro-périphériques embarqués. De même, la couverture réseau (en particulier sans fil)

ne cesse de croître (campus universitaires, entreprises, lieux publics...) permettant une connectivité importante de ces périphériques. Certains auteurs parlent ainsi d'un univers d'informatique ambiante (*ubiquitous computing*, [Mat01]) dont la taille est gigantesque puisque l'Internet permet l'interconnexion de tous ces éléments.

Le réseau à *grande échelle* ainsi obtenu permet de disposer d'une quantité d'information extrêmement importante et d'offrir un grand nombre de services, dont la plupart sont encore à inventer. Le concept de grille de calcul [FK98] ou de stockage assimile celui-ci à un réseau de distribution semblable à la distribution d'eau ou d'électricité : de nombreux fournisseurs interconnectés offrent des services ou des données de manière plus ou moins standardisée à l'ensemble des utilisateurs accédant au réseau. On peut alors considérer ce réseau comme un système d'information très riche mais peu structuré. Les services les plus utilisés sont d'ailleurs les services de recherche d'information tel Google.

Ce réseau possède une dynamique d'utilisation très fluctuante et imprévisible, impliquant une très forte variation de la qualité de service. Son utilisation n'est pas du tout régulière mais connaît des pics de sur-utilisation et de sous-utilisation qu'il est parfois difficile de prévoir, de compenser ou d'exploiter[1]. Un site peut à tout instant, sans contrainte, décider de quitter le réseau ou de retirer certaines ressources ou certains services. Par ailleurs, le nombre d'éléments intervenant dans son architecture, même avec des taux de pannes extrêmement faibles et une redondance élevée, fait qu'en permanence de nombreux éléments matériels ou logiciels sont en panne.

Ces systèmes sont également très hétérogènes, depuis leurs composants matériels jusqu'au système d'exploitation en passant par leur connectivité. De plus, les systèmes étant administrés individuellement (par leurs propriétaires respectifs), ils peuvent évoluer de manière imprévisible et sans notification : par exemple mise à jour logicielle, passage d'une connexion X10 à une connexion sans fil type WiFi...

Le problème qui nous intéresse ici est celui du développement et de la maintenance des applications dans un tel contexte. Notre travail, de nature *génie logiciel*, a pour objectif de proposer des techniques et des outils (modèles, architectures, *middlewares*, *frameworks*...) qui contribuent à limiter la complexité induite par la prise en compte des problèmes (communs) posés par la grille et l'informatique ambiante : hétérogénéité, volatilité des ressources, volume des données échangées...

Nous présentons dans ce chapitre quelques applications concrètes, puis nous discutons de leurs caractéristiques communes afin d'introduire la problématique et les motivations de cette thèse. En complément, nous situons ce travail dans le contexte des travaux de notre équipe de recherche. Enfin, ce chapitre d'introduction se termine par un plan de ce mémoire.

[1] Voir par exemple l'évolution en temps réel et les graphes d'historiques du trafic internet : http://www.internettrafficreport.com

1.2 Exemples d'applications

Dans cette section, nous décrivons quelques exemples pour donner une vision plus concrète du contexte et des applications que nous envisageons, allant de l'informatique ambiante jusqu'au niveau grille. Les scénarios ci-dessous seront analysés dans la section suivante, et certains seront repris dans d'autres chapitres.

1.2.1 Rendu d'animations 3D

Présentation

Le rendu de films d'animations 3D nécessite une puissance de calcul importante, liée à la technologie employée pour générer chaque image à partir d'une description de scène. Pour obtenir des vidéos de bonnes qualités (haute résolution, et fréquence d'image élevée), les studios d'animation ont généralement recours à des solutions à base de clusters. Pour des particuliers ou des petites entreprises[2] l'achat d'un cluster ou la location de temps de calcul n'est pas financièrement envisageable. Il existe plusieurs projets de calcul distribué utilisant les puissances de calcul d'ordinateurs non exploités[3]. Nous proposons un scénario similaire, mais qui ne nécessite pas de contrôle centralisé ni d'intervention de tiers.

Scénario d'utilisation

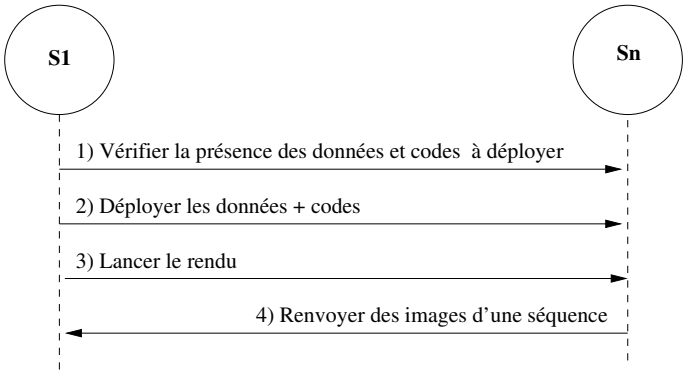


FIG. 1.1 – Pseudo diagramme de séquence du scénario de calcul réparti

Sur la figure 1.1, on a représenté en $S1$ le site client, qui demande et pilote le rendu d'une animation. Il peut contacter en parallèle différents sites S_n dont le fonctionnement est identique, qui mettent à disposition leurs ressources de calcul. Le client commence par vérifier la présence des données (textures, moteurs de rendu...) sur les sites à exploiter, ainsi que la quantité de CPU disponible pour ne pas lancer un calcul sur un site saturé. Si besoin, il déploie les éléments manquants, puis active le calcul sur chaque site. Par exemple,

[2] A l'origine, cet exemple provient d'un besoin exprimé par une association scientifique qui souhaite produire des films d'animations à des fins de vulgarisation.

[3] On pense en premier à SETI, le projet pour la détection de vie extra-terrestre http://setiathome.ssl.berkeley.edu.

une animation peut être divisée de manière temporelle : le client assigne à chaque site une portion à calculer en fonction des ressources disponibles. Pour tenir compte des variations de charges des sites (qui peuvent être exploités en même temps pour d'autres tâches), il est nécessaire de superviser le rendu, et éventuellement de ré-allouer dynamiquement les tranches de calcul. Toutes ces opérations s'effectuent de manière asynchrone, et contrairement aux clusters, les machines des sites S_n doivent pouvoir être différentes (hétérogénéité matérielle et logicielle).

A partir de cet exemple, on peut envisager d'autres scénarios de calcul distribué[4], en remplaçant les parties calculatoires et en déployant les données adéquates.

Analyse

L'application doit pouvoir fonctionner sans tiers, c'est à dire malgré l'existence d'administrateurs multiples (au pire un par machine), en réalisant du déploiement complet de données fournies par le client (de l'installation à l'activation). Comme dans l'exemple précédent, elle doit supporter les aléas liés à la variation de la QoS (réseau, pannes machines...) et permettre dynamiquement un changement de paramètres (reconfiguration dynamique).

1.2.2 Une application de mutualisation de données

Présentation

Considérons une application qui doit permettre à une communauté d'utilisateurs de fournir, de rechercher et d'exploiter des données réparties sur un réseau hétérogène de grande taille, par exemple le réseau mondial (du type application *pair à pair*). Chaque utilisateur connaît d'autres membres de la communauté et dispose de certaines connaissances sur les données qu'ils fournissent. Un utilisateur peut à la fois rendre accessible des données dont il est propriétaire, ainsi que rechercher puis utiliser des données parmi celles mises à disposition par la communauté. La recherche s'appuie sur les connaissances de l'utilisateur, et ces connaissances évoluent dans le temps en fonction des résultats obtenus.

Scénario d'utilisation

Dans ce scénario, on considère les ressources (et pas simplement des données) sous forme de fichiers de type quelconque (son, image, texte...) ou bien de services (*Web service*, base de données...).

Soient cinq sites S_i d'une communauté, proposant chacun des ressources R_j, sauf S_4. Les liens de connaissances initiaux sont représentés sur la figure 1.2. Un utilisateur sur le site S_1 souhaite accéder à une ressource R_c. Celle-ci a été exploitée antérieurement sur

[4]Une présentation plus détaillée et une liste de projets sont disponibles sur http://fgouget.free.fr/distributed

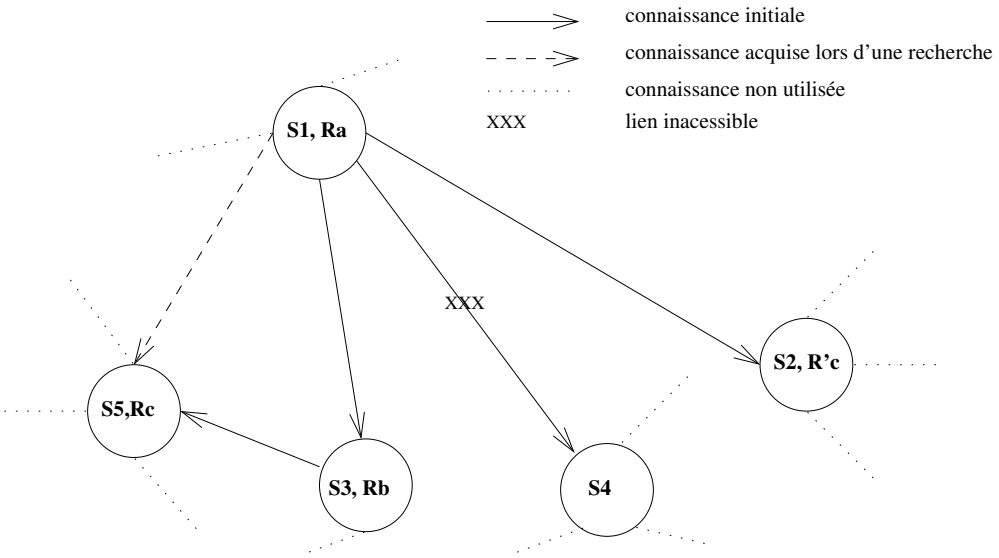


FIG. 1.2 – Scénario d'utilisation du logiciel de mutualisation de ressources

S_2, donc la première tentative de recherche se fait sur ce site. Mais la ressource (notée R'_c) a évolué entre temps (mise à jour, suppression...) et s'avère inexploitable. Le système doit alors poursuivre la recherche sur les autres sites connus. S_4 ne répond pas (panne, déconnexion ou saturation...). Le site S_3 ne contient pas la ressource recherchée, mais connaît deux sites initialement inconnus de S_1. La ressource y est alors recherchée et elle est localisée puis exploitée sur S_5. Sur S_1, le logiciel ajoute le lien de connaissance vers S_5. L'exploitation de Rc peut être par exemple un téléchargement de fichier de S_5 vers S_1, ou bien l'exécution d'une requête sur une base de données du site S_5.

Il est possible d'aller plus loin. Par exemple lorsqu'un utilisateur cherche un composant logiciel (cf. 2.3), il souhaite probablement récupérer en plus de ce composant l'ensemble de ses dépendances (composants requis). Dans ce cas, la recherche doit se poursuivre récursivement de manière automatique, jusqu'à ce que toutes les dépendances soient satisfaites.

Analyse

L'application doit supporter les pannes et les déconnexions de certains sites ainsi que les évolutions des ressources (nouvelles, mises à jour, supprimées). Elle doit être capable de découvrir en cours d'exécution de nouvelles ressources et sites, et suffisamment flexible pour permettre l'utilisation de différents protocoles adaptés à chaque type de recherche, éventuellement fournis par le client.

1.2.3 Services dynamiques pour les véhicules

Présentation

Avec la démocratisation des systèmes de navigation embarqués (GPS et bientôt sa version européenne Galileo) et l'augmentation des capacités des petits périphériques por-

tables (PDA ou même téléphone portable), de nouvelles applications sont en train d'émerger. Dans un futur proche, il est envisageable que ces systèmes soient connectés au réseau Internet (type GPRS) et interconnectés de manière ad-hoc entre des véhicules proches (liaisons sans fil par exemple). Quelques projets s'intéressent à ce genre de scénarios, par exemple *Urban Vehicular Grid*[5], mais souvent en se focalisant sur des problématiques de niveau réseau. Il existe même un projet de norme WiFi, le 802.11p dit WAVE (Wireless Access for the Vehicular Environment), dédié à cette problématique.

Scénario d'utilisation

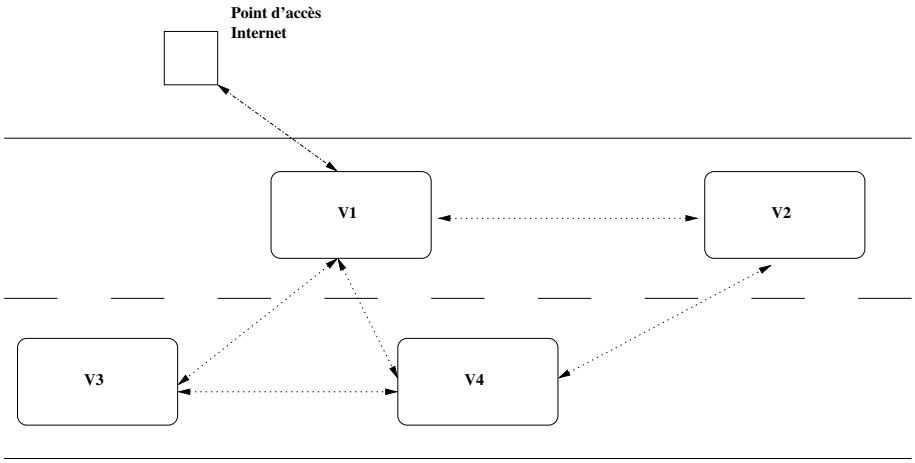


FIG. 1.3 – Services dynamiques pour les véhicules

La figure 1.3 montre un exemple d'interconnexion entre quatre véhicules V_i équipés des périphériques embarqués adéquats. A partir des informations de localisation fournies par son GPS (position, vitesse, direction), V_1 peut utiliser des services web de routage dynamique, ou encore d'autres services de géolocalisation (recherche d'hôtel à proximité...). Etant connecté à des véhicules qui se déplacent en sens inverse (V_3 et V_4), il peut obtenir des informations sur les conditions de circulation à venir. S'il freine brutalement ou dévie brusquement de sa trajectoire, par exemple pour éviter un obstacle sur la voie, les véhicules qui suivent (ici V_2) peuvent être immédiatement prévenus et recevoir des alertes. De plus V_2, qui aimerait bien doubler V_1, peut connaître le nombre de véhicules à venir en face, ainsi que leurs paramètres (distance, vitesse...). Pour plus de sûreté, les codes réalisant ces opérations sont fournis et maintenus à jour (directement ou non) par les constructeurs de chaque véhicule.

Analyse

En supposant l'existence de couches réseau adéquates (ce qui n'est pas, *a priori*, évident aujourd'hui), le fonctionnement du logiciel repose sur la possibilité de déployer du code fourni par des tiers (les constructeurs) et sur l'exploitation conjointe de plusieurs sources d'informations qui évoluent rapidement. La topologie des connexions évolue également

[5] http://www.cs.ucla.edu/ST

rapidement (la portée d'un réseau WiFi[6] en extérieur est d'environ 400m, ce qui laisse un délai de 5 secondes environ sur une autoroute pour des voitures qui se croisent, soit une capacité de transmission d'environ 1Mo) et de manière imprévisible.

1.2.4 Autres exemples

Les possibilités d'applications ne s'arrêtent pas aux trois exemples précédents, nous en avons proposé d'autres (voir par exemple dans [ALPre]), dont voici une description succincte.

Distribution de modules logiciels

Dans une application de distribution (au sens de la livraison) de modules logiciels (applications entières ou parties d'application à des fins de mise à jour) sur un réseau[7], des producteurs de logiciel proposent les codes et leur documentation à des utilisateurs. Producteurs et utilisateurs interagissent par l'intermédiaire de distributeurs suivant deux modes distincts, le flot de données étant orienté des producteurs vers les utilisateurs. En mode *push*, le distributeur (éventuellement, en amont, le producteur) est l'initiateur de l'interaction avec l'utilisateur. Par exemple, il peut s'agir d'une mise à jour générale de tous les postes clients au sein d'une entreprise ou d'un groupe d'utilisateurs. Le distributeur doit d'abord identifier les utilisateurs cibles (éventuellement abonnés au service de distribution), ce qui peut demander une recherche sur le réseau et une analyse de l'existant chez les utilisateurs. En mode *pull*, c'est l'utilisateur (ou son système) qui est à l'origine de l'interaction et qui demande l'installation d'un module ; dans ce cas, c'est l'utilisateur qui doit trouver un distributeur et pouvoir consulter un catalogue qui décrit les modules disponibles.

Dossier médical personnel

Le *dossier médical personnel*[8] (DMP) est un projet du Ministère de la Santé et de la Protection Sociale dont l'objectif est de donner aux personnels de santé l'accès à toutes les informations concernant un patient afin d'augmenter la qualité des soins et d'en maîtriser les coûts. Un dossier médical est un ensemble de données hétérogènes dont le stockage centralisé est coûteux, car il requiert une qualité de service importante (disponibilité, temps d'accès...). On pourrait imaginer une architecture flexible et ouverte qui exploite un stockage réparti et redondant des données chez différents prestataires associés aux différents acteurs : les données seraient alors réparties avec redondance (à l'échelle d'un pays par exemple) sur des supports hétérogènes reliés par différents types de réseaux.

[6] Avec la technologie 802.11b/g.

[7] Cette activité est maintenant disponible dans de nombreux systèmes d'exploitation grand public (distribution Linux Debian, logiciels Microsoft, plugins Adobe Acrobat, Mozilla, Eclipse...).

[8] http://www.sante.gouv.fr/assurance_maladie et http://www.d-m-p.org

Analyse

La mise en œuvre de ces deux derniers exemples recouvre les problèmes de partage, de recherche et d'exploitation de ressources réparties mis en évidence plus haut. De fortes contraintes de qualité de service et de sécurité (confidentialité, intégrité) s'y ajoutent. Ces dernières conduisent, en particulier, à l'utilisation de techniques d'authentification et de chiffrement. En outre, dans le cas du DMP, le volume des dossiers influe également sur leur accessibilité et sur la performance du système.

1.3 Problématique et motivations

Cette section reprend les analyses faites pour les exemples proposés, en étendant les constatations au cadre général des systèmes distribués à grande échelle.

1.3.1 Facteur d'échelle et contexte ouvert

Les applications évoquées s'articulent autour de différentes unités physiques ou logiques (PDA, cluster, PC portable... et serveur d'application, suite logicielle, composant logiciel...) qui fournissent ou exploitent des ressources ou des services. Ces unités sont autonomes, réparties (leur exécution peut donc être parallèle), et **administrées indépendamment** les unes des autres. Le contrôle est **décentralisé** et distribué à travers ces unités.

La principale caractéristique de ces applications est leur **échelle**. Ici, la notion d'échelle recouvre à la fois le nombre d'unités et les dimensions du réseau : le nombre d'unités peut être très grand et leur couplage faible (interconnexions via un réseau filaire de grande taille ou sans fil, à faible qualité de service). Dans ces conditions, le nombre et les volumes des interactions entre les unités influent fortement sur le fonctionnement et la performance.

Une autre caractéristique, conséquence de l'échelle, est l'**ouverture** des systèmes : outre les problèmes de pannes, les unités peuvent intégrer ou quitter le système dynamiquement, voire se déplacer sur le réseau, sans contrôle global et sans qu'il ne soit possible de prévoir l'ensemble des évolutions. De l'ouverture, résultent les problèmes de **disponibilité** (ou volatilité) des ressources et des services. De plus, les utilisateurs peuvent ne disposer que d'une connaissance partielle ou ancienne de sa structure ainsi que des informations sur les services disponibles.

Ces applications se distinguent également par la nature **hétérogène** (matériel, système d'exploitation, connexion réseau) des systèmes à exploiter. De fait, pour les faire interopérer, il est nécessaire d'utiliser des protocoles communs, le plus simple étant d'exploiter des standards pour en faire des systèmes ouverts.

1.3.2 Besoins de flexibilité

Pour être aussi efficaces que possible, les applications réparties à grande échelle doivent supporter différents types d'évolutions [Boi02]. Elles doivent être *configurables*, c'est à

dire capables d'adaptation statique (réalisée en dehors de leur utilisation) pour faciliter leur maintenance. Elles doivent être *extensibles*, pour permettre l'intégration de nouvelles fonctionnalités correspondant à des besoins identifiés après la livraison. Enfin elles doivent être *dynamiquement adaptables*, pour se reconfigurer en cours d'exécution afin de présenter un mode de fonctionnement optimal. Cela peut aller de la simple variation de paramètre à la reconfiguration complète de l'architecture logicielle en cours de fonctionnement. Nous parlons de *flexibilité* lorsqu'un système ou une application présente ces trois capacités d'évolution[9].

Adaptation statique

Dans tout projet logiciel, les activités de maintenances sont nécessaires pour conserver un système opérationnel et satisfaisant pour les utilisateurs. On distingue plusieurs catégories de maintenance : corrective (correction des défauts résiduels du logiciel), adaptative (suite à une modification d'une spécification), perfective (amélioration du fonctionnement), évolutive (ajout de fonctionnalité) ou préventive (enrichissement du logiciel par des mécanismes de traitement d'erreurs). Les besoins d'adaptation statiques doivent être pris en compte dès la conception, en proposant des architectures logicielles adéquates, permettant leur configuration.

Adaptation dynamique

Lors de la construction ou du déploiement de systèmes ouverts répartis à grande échelle, on ne peut pas faire d'hypothèse forte sur la disponibilité ou la forme d'un service[10], ni sur l'organisation globale du système. La robustesse est cependant une exigence forte. Les applications doivent supporter les pannes et les déconnexions de certains sites ainsi que les évolutions des ressources (ajout, mise à jour, suppression). Elles doivent être suffisamment souples pour répondre à différents besoins en terme de localisation et d'exploitation des ressources et s'adapter (dynamiquement) à la disponibilité et à la qualité des services (y compris celle du réseau de communication).

Les techniques d'adaptation dynamique visent à prendre en compte « instantanément » les variations du contexte d'exécution, de façon non seulement à continuer à faire fonctionner l'application mais aussi à tirer le meilleur parti de la nouvelle configuration. Toutefois, l'intégration de mécanismes d'adaptation dynamique peut se révéler complexe, et il est donc nécessaire de disposer d'outils pertinents pour réduire cette complexité. De manière similaire à l'adaptation statique, la prise en compte de ces besoins doit être faite dès la conception, en proposant des architectures logicielles adéquates, permettant leur reconfiguration dynamique.

Personnalisation

Les services génériques ou les ressources mises à disposition sur le réseau peuvent ne pas être directement exploitables par un client. Soit par un manque au niveau logiciel (applica-

[9] Il ne semble pas y avoir dans la littérature de terminologie formelle ; aussi nous proposons d'employer le terme *flexibilité* pour regrouper toutes les capacités d'adaptation d'un système citées dans ce paragraphe.

[10] On emploie ici assez indifféremment les termes « service » et « ressource » (voire « information »).

tion, codec...), soit par des capacités de traitement (bande passante, périphériques, etc.) inappropriées. Ainsi, certains de ces exemples requièrent des capacités de personnalisation pour adapter un service aux besoins du client.

Pour cela, une première approche consiste à faire varier des paramètres du côté du système qui rend le service. En allant plus loin, on peut permettre au client de transmettre non plus un ensemble de paramètres mais un programme pour adapter le service en fonction de ses besoins. Dans le cas de la recherche d'information, on peut ainsi permettre à l'utilisateur de déployer son propre algorithme de recherche, pour augmenter la pertinence des résultats trouvés.

1.3.3 Implications sur la conception

La conception de programmes concurrents répartis est beaucoup plus complexe et source d'erreurs que celle des programmes séquentiels centralisés. Ces difficultés sont liées à la multiplication d'entités (processus...) ainsi qu'aux problèmes de communication et de synchronisation. Une réponse possible consiste à utiliser des approches de plus haut niveau dans toutes les phases de conception.

Outre les aspects fonctionnels, les principaux besoins à prendre en compte lors de la conception sont les suivants :

Recherche de ressources

Il faut intégrer aux applications la capacité de rechercher dynamiquement des services (au sens de la localisation sur le réseau, et de la découverte de nouvelles ressources), puis de les spécialiser afin de les adapter aux conditions d'exécution et aux besoins. Par exemple, la localisation sur le réseau de ressources et de services peut être plus efficace en utilisant une algorithmique spécifique personnalisée qui dépend des connaissances et des besoins du demandeur.

Organisation des traitements répartis

Ce besoin concerne l'exploitation à distance ou locale, la gestion des volumes et des flux et le déploiement de services : il s'agit de prendre en compte le coût des échanges sur le réseau et d'adapter l'organisation à l'état du système ou aux spécificités de l'application (un compromis lié au coût de transfert des données et de déploiement de service est nécessaire au niveau du choix du lieu de traitement). Par exemple, déporter sur un site serveur un traitement spécifique qui met en œuvre l'expertise d'un client peut permettre de réduire le nombre et le volume des données échangées sur le réseau qui sont nécessaires à l'interaction ou qui en résultent.

Pour déporter et exécuter les traitements répartis, il est nécessaire de les déployer. Le déploiement de logiciel [CFH+98] recouvre différentes activités postérieures au développement dont l'objectif est de rendre un logiciel opérationnel pour un utilisateur : transfert du producteur à l'utilisateur, configuration (adaptation au contexte d'utilisation), installation et activation de composants logiciels, mise à jour et reconfiguration (évolution)

pour les logiciels en production. La complexité et l'importance du déploiement ont augmenté récemment avec l'évolution des réseaux et la construction d'applications à base de composants. Il en résulte un besoin d'automatisation de cette phase du cycle de vie du logiciel.

Sûreté de fonctionnement

Cette préoccupation concerne l'amélioration de la qualité du logiciel en le rendant globalement plus fiable ; les principaux mécanismes sont la tolérance aux pannes, la gestion du mode déconnecté, ou le support de l'hétérogénéité.

Sécurité

Il s'agit de prendre en compte les potentielles attaques du logiciel, et de proposer des parades ou des moyens de prévention ; par exemple, en prévoyant des services d'authentification des entités et de leurs représentants, ou encore des communications sécurisées...

Les objectifs de nos travaux ne sont pas de prendre en compte directement cette problématique. Toutefois nous n'avons pas négligé cet aspect et nous proposons en annexe A un point sur les problèmes de sécurité (et leurs solutions) qui entourent les technologies que nous proposons.

1.3.4 Motivations et objectifs de cette thèse

Les quelques exemples d'applications réparties à grande échelle que nous avons présenté (section 1.2) montrent des systèmes complexes, autant dans la phase de conception que dans la phase d'exploitation. Les difficultés de conception se situent à différentes étapes (analyse, architecture, implémentation...) dans lesquelles il faut faire face à des préoccupations très différentes : sûreté de fonctionnement, efficacité et sécurité. Les concepteurs devront prendre en compte les problèmes d'exploitation concernant les volumes de données, la volatilité, la personnalisation, l'hétérogénéité matérielle et logicielle, et le déploiement.

Pour concevoir des applications en tenant compte des besoins et des problèmes liés à ce contexte, il est nécessaire de se baser sur des technologies logicielles adéquates. Nos objectifs, de nature *génie logiciel*, sont de fournir des méthodes, des techniques et des outils pour maîtriser la complexité de ces différentes étapes.

Les technologies à la base de nos propositions sont :
- les composants logiciels, qui permettent de structurer les applications en leur apportant modularité, extensibilité et réutilisation,
- les agents logiciels (éventuellement mobiles), qui permettent d'organiser et de personnaliser les traitements répartis,
- le modèle d'organisation pair à pair, qui représente un mode d'interconnexion complètement décentralisé, plus adapté à notre contexte d'exécution que le modèle centralisé.

Nous montrerons les limites de ces technologies et proposerons d'exploiter ces technologies de manière conjointe afin de bénéficier de leur complémentarité.

Notre travail vise les aspects non fonctionnels de la flexibilité et leur mise en œuvre au niveau architectural. En revanche, nous ne nous intéressons pas directement aux problèmes liés à l'acquisition du contexte ou à la prise de décision.

1.4 Présentation de l'équipe d'accueil

Cette thèse a été préparée au sein de l'équipe « ingenierie des Langages pour les sYstèmes Répartis et Embarqués » de l'IRIT (IRIT-LYRE) -répartie sur les sites de l'Université Paul Sabatier (UPS) et de l'Ecole Nationale Supérieure d'Electrotechnique, d'Electronique, d'Informatique, d'Hydraulique et des Télécommunications (ENSEEIHT)- dont les travaux ont pour objectif de proposer des méthodes et des outils d'aide au développement et à la maintenance des applications réparties et embarquées : formalismes de conception, modèles de programmation, moyens de validation par analyse statique. En particulier, l'équipe LYRE s'intéresse à la prise en compte des besoins d'adaptation de ces applications[11].

LYRE - Carte d'identité

Thème 7 de l'IRIT

Sûreté de développement du logiciel

Equipe "ingénierie des **L**angages pour les s**Y**stèmes **R**épartis et **E**mbarqués"

Responsable : **Pr. Patrick Sallé**

Co-resp. UPS : **J.-P. Arcangeli**

Co-resp. ENSEEIHT : **M. Pantel**

Equipe : 11 permanents (1 Pr, 10 MdC) et 6 doctorants en juillet 2006

Une partie des travaux s'appuie sur un modèle de programmation à base d'objets actifs : les acteurs. Dotés d'une activité propre, pouvant changer dynamiquement de comportement et communiquant par messages asynchrones, ils constituent un support simple, réaliste et bien adapté à la mise en œuvre d'applications réparties à grande échelle ou mobiles. Par le passé, différents travaux ont porté sur la réflexivité [Mig99] et son utilisation dans le cadre de l'administration des applications réparties [Bra03].

L'équipe développe la bibliothèque JavAct[12] dédiée à la programmation d'applications concurrentes, réparties et mobiles à base d'agents et basée sur le modèle d'acteur [AHL+04]. JavAct est distribué sous forme de logiciel libre sous licence LGPL, avec un environnement de développement complet (plugin pour Eclipse, générateurs de code et environnement d'exécution).

L'équipe s'intéresse également à la modélisation d'applications réparties sur la grille au moyen d'UML, avec pour objectif d'étendre le formalisme pour mieux prendre en compte les changements de comportement dans les applications adaptables [Rou05], ainsi qu'à leur validation par analyse statique [HHP04]. Les outils de validation statique définis pour le calcul d'acteurs primitifs CAP pourront être exploités pour valider les modèles, à partir desquels on pourra générer des squelettes en code JavAct.

[11] Notre équipe fait partie du groupe de travail ADAPT : Action Adaptation dynamique aux environnements d'exécution - http://adapt.asr.cnrs.fr.

[12] http://www.irit.fr/recherches/ISPR/IAM/JavAct_fr.html

Ces travaux donnent lieu, par ailleurs, à des expériences dans le cadre du projet GRID'5000[13] de l'ACI GRID.

1.5 Organisation et plan du mémoire

Pour une lecture efficace de ce mémoire, chaque chapitre débute par un encart résumant son contenu. Les contributions sont reprises de manière synthétique dans chaque conclusion de chapitre, ainsi que dans la conclusion générale de ce document. Quelques réalisations logicielles (annexes B.3 à B.4) sont également annexées pour présenter le travail diffusé et valorisé dans le cadre de cette thèse.

Le plan de ce mémoire est le suivant :

- Le chapitre 2 décrit les différentes technologies qui seront utilisées par la suite, en les positionnant par rapport à notre problématique
- Le chapitre 3 présente un modèle d'architecture d'agent adaptable à base de composants à grain fin dynamiquement remplaçables et une implémentation en Java, pour permettre à un agent de s'adapter par reconfiguration de son architecture à son contexte d'exécution.
- Le chapitre 4 propose un patron de conception à base d'agents mobiles adaptables et de composants pour mettre en œuvre des systèmes répartis à grande échelle structurés selon le mode pair à pair. Nous présentons également un *framework* qui permet sa mise en œuvre en Java.
- Le chapitre 5 introduit un modèle d'architecture permettant d'engendrer des modèles d'agents différents. Un prototype d'environnement de modélisation est également décrit : il montre comment réutiliser et assembler les différents composants à grain fin pour construire des agents, et comment engendrer les squelettes de codes nécessaires à leur exécution.
- Une conclusion et des perspectives à ces travaux sont proposées dans le chapitre 6.

[13] http://www.grid5000.fr

Chapitre 2

Technologies logicielles

Nous présentons dans ce chapitre différentes technologies à la base de la construction d'applications réparties : modèle pair à pair, intergiciel, composant logiciel et agents. Nous montrons leurs apports à la problématique ainsi que leurs limites, au travers de quelques exemples concrets. En conclusion, nous montrons que ces technologies employées individuellement ne sont pas suffisantes pour répondre aux différents besoins exprimés dans le chapitre précédent. Ce panorama technologique est un état de l'art partiel qui sert de pré-requis pour la compréhension des chapitres suivants.

Nous étudions dans ce chapitre les apports et les limites de certaines technologies qui sont à la base de nos propositions, en les positionnant par rapport à notre problématique et en présentant quelques exemples concrets d'utilisation.

2.1 Modèle pair à pair

2.1.1 Présentation

Les systèmes « pair à pair » (*peer-to-peer* ou P2P) [MKL+02] sont des systèmes répartis à l'échelle du réseau Internet. Ils reposent sur le principe de mutualisation (échange et partage) au sein d'un réseau logique, de services et de ressources comme par exemple des données, des programmes, des capacités de stockage ou de calcul. Tous les participants (appelés pairs) sont à égalité de devoirs et de droits : chacun peut à la fois rendre accessibles des ressources dont il dispose (opération de publication) et exploiter des ressources fournies par d'autres. En ce sens, le modèle P2P est une alternative au modèle client-serveur classique dans laquelle les pairs peuvent jouer en même temps le rôle de serveur et le

rôle de client. Les systèmes P2P sont par ailleurs des systèmes ouverts : les pairs peuvent librement intégrer et quitter le réseau et fournir les ressources de manière intermittente et dans une forme susceptible d'évoluer au cours du temps.

Le modèle P2P a été popularisé par différentes applications de partage et d'échange de fichiers de musique ou de vidéo. Parmi les plus connues, on peut citer Napster[1], Freenet[2], eDonkey[3], Gnutella[4], ou encore Kazaa[5]. Aujourd'hui, les échanges au moyen de ces applications représentent une part importante du trafic sur Internet[6]. A titre d'exemple, nous présentons ci-dessous le protocole BitTorrent. Les principales caractéristiques des autres systèmes sont décrites en 2.1.2.

Exemple de protocole P2P : BitTorrent

Le protocole BitTorrent[7] est l'un des plus récents pour le partage de fichiers en réseau : il profite des expériences de ses prédécesseurs autant sur le plan juridique (l'expérience *Napster*) que sur la résistance à la montée en charge. Son succès est probablement dû à son ouverture (il est *Open Source*, d'où l'implémentation de nombreux clients) mais aussi à l'efficacité de son mode de recherche. Il est particulièrement connu pour être le moyen préconisé de diffusion et de téléchargement des distributions Linux.

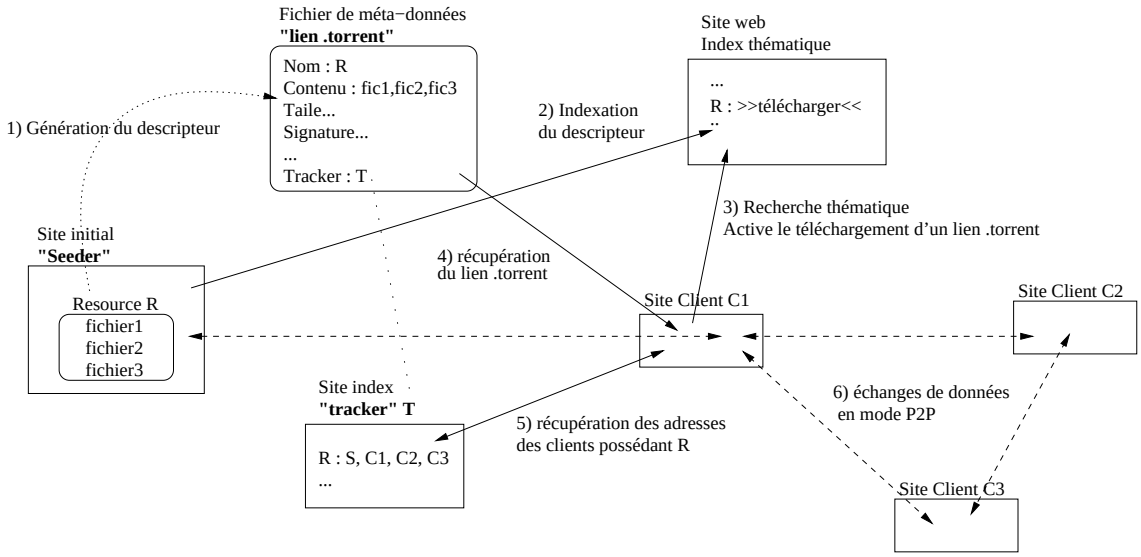


FIG. 2.1 – BitTorrent, un protocle P2P

La phase de publication d'une donnée est originale, puisque le propriétaire (appellé *Seeder*) référence sa ressource sur un site thématique, via un fichier de descripteur (lien

[1] http://www.napster.com
[2] http://freenet.sourceforge.net
[3] http://www.edonkey2000.com
[4] http://www.gnutella.com
[5] http://www.kazaa.com
[6] Selon les études, par exemple sur http://www.sandvine.com/news/pr_detail.asp?ID=88, les chiffres avancés sont entre 70% et 90% d'occupation de l'Internet
[7] http://www.bittorrent.org/protocol.html

.torrent) contenant un ensemble de méta-données (figure 2.1). Entre autres, on y trouve l'adresse ou l'URL d'un *Tracker*, un serveur spécifique qui associe à l'identifiant du fichier une liste de clients qui la possèdent (intégralement ou en morceaux) et la partagent.

Côté client, la recherche se fait initialement par l'intermédiaire d'un navigateur Web, sur des sites référençant des liens *.torrent*. Le téléchargement débute lors d'un simple clic sur un lien donné par le navigateur, provoquant l'activation du logiciel. Celui-ci contacte le *Tracker* associé au lien, afin de récupérer une liste de clients possédant le fichier choisi. Enfin, les transferts de données se font entre clients par l'intermédiaire d'une connexion directe en TCP/IP.

2.1.2 Typologie des systèmes P2P

Par nature, l'instabilité du système et la volatilité des ressources sont fortes et incontrôlées. Un système P2P robuste doit donc être flexible et capable de s'adapter dynamiquement, de manière transparente pour l'utilisateur. Du fait de l'échelle (la dimension du réseau et le nombre de pairs, typiquement des milliers, des dizaines de milliers, voire davantage), la communauté ne peut pas être informée des changements (évolutions, connexions, déconnexions, pannes). Ainsi, lorsqu'un pair veut accéder à une ressource, il n'a *a priori* qu'une connaissance partielle (qui peut être en partie obsolète) de l'état du système P2P (de la disponibilité des serveurs et de la qualité des services). La localisation des ressources est donc une fonctionnalité essentielle.

On peut distinguer trois types d'architecture de systèmes P2P, en fonction du mode de mise en relation entre demandeur et fournisseur, c'est-à-dire en fonction du mécanisme de localisation des ressources. Ensuite, les échanges de données (plus généralement l'exploitation des ressources) se font directement entre pairs sans intermédiaire.

1. Les systèmes les plus simples, à la limite du P2P et du client-serveur, sont dits **centralisés**. Ils se basent sur un serveur unique pour indexer les ressources (i.e. Napster). A chaque connexion d'un pair au réseau, ce dernier annonce au serveur la liste des ressources qu'il met à disposition de la communauté. En phase de recherche, le client effectue une requête auprès du serveur qui retourne la ou les références des pairs qui offrent une ressource correspondant aux critères de la recherche. L'échange se fait ensuite directement entre les pairs concernés. Même si l'expérience « Napster » a démontré la viabilité de cette solution sur le plan technique, la centralisation au niveau du serveur présente des limites en terme d'extensibilité (passage à l'échelle) et constitue un point de faiblesse en cas de surcharge voire de panne. Dans certains scénarios (exemple des services pour les véhicules) une telle organisation semble difficile à imaginer.
2. Les systèmes **semi-décentralisés** ou **hybrides** conservent le principe d'un serveur de localisation mais s'appuient sur un ensemble de serveurs répartis (i.e. eDonkey) afin de minimiser les conséquences des pannes et de réduire les risques de contention. Dans certains cas (i.e. Kazaa), des pairs peuvent être spécialement choisis pour leur capacité de calcul et de bande passante (des *superpeers*) ; leur rôle est d'indexer les ressources d'un sous-ensemble du réseau.

3. Enfin, en mode P2P pur, il n'existe pas de point de centralisation ; au contraire, les systèmes P2P purs sont complètement **décentralisés** et auto-organisés. Dans ce cas, localiser une ressource est une opération critique qui demande *a priori* le parcours d'une partie du réseau. On peut distinguer deux grandes catégories de systèmes P2P décentralisés :

   (a) Les réseaux dits **structurés** (i.e. Freenet, Chord...) conservent un index qui est réparti sur l'ensemble des pairs sur le principe d'une table de hachage distribuée [RFH+01, SMK+01, CSWH00]. La recherche consiste en un routage de la requête à travers le réseau piloté par l'index. La longueur du chemin d'accès à une ressource est bornée (il dépend de la topologie du réseau). Mais l'ajout et le retrait d'un pair sont des opérations complexes et coûteuses. Par ailleurs, si l'un des pairs devient indisponible, les ressources qu'il indexe ne peuvent plus être retrouvées alors qu'elles peuvent être disponibles.

   (b) Les réseaux dits **non structurés** (i.e. Gnutella, PeerCast...) se passent d'organisation globale et d'index. Ainsi, ils éliminent le coût de maintien de l'index réparti dans un état cohérent. Les recherches consistent à explorer le réseau. Elles s'effectuent dynamiquement d'abord auprès de voisins (des pairs connus du client) puis récursivement au sein du réseau. Il résulte de la propagation de la requête un phénomène « d'inondation » (*flooding*) coûteux en bande passante, dont on contrôle les effets en associant aux requêtes une durée de vie limitée (*Time To Live*) ; alors, les recherches ne sont plus exhaustives et la localisation d'une ressource disponible n'est pas assurée, même en l'absence de panne. Néanmoins, cette capacité de découverte dynamique permet de fonctionner dans des réseaux inorganisés, instables et en constante évolution. Aussi, des travaux sont en cours afin de proposer des solutions (à base de profil client et d'apprentissage) pour rendre moins aveugles et donc plus efficaces les procédures de recherche (voir par exemple [HKFM04]).

2.1.3 Apports et limites

Au-delà des problèmes légaux ou moraux (cf. 2.1.5) que posent certaines applications phares permettant le partage de musiques ou de vidéos, le P2P constitue un véritable modèle d'organisation répartie extensible et robuste qui permet la mutualisation et la capitalisation au sein de communautés ou d'entreprises. En particulier, il semble d'un grand intérêt pour la construction de systèmes d'information répartis à grande échelle qui, par nature, sont hétérogènes et instables.

Ainsi, nous pensons que les systèmes P2P purs décentralisés sont les mieux adaptés à la mise à disposition de ressources volatiles ou évolutives et aux systèmes à grande échelle et à topologie instable. De fait, le P2P pur constitue un support élémentaire pour nos travaux.

Toutefois, le P2P n'est pas une technologie, mais un modèle d'organisation des flots de données réseau. Or la construction d'applications sur ce modèle n'est pas simple, il existe donc un besoin pour des plateformes de développement de systèmes P2P.

2.1.4 Quelques environnements de développement pour le P2P

JXTA[8] est une initiative de Sun Microsystems, proposant un ensemble de protocoles ouverts pour interconnecter des dispositifs allant du téléphone cellulaire jusqu'aux serveurs. Le fonctionnement repose sur le découpage de parties du réseau réel en réseaux virtuels, dans lesquels chaque pair peut accéder aux ressources des autres sans se préoccuper de leur localisation ou de leur environnement d'exécution. JXTA propose une base de six protocoles (services de découverte des pairs, de rendez-vous, d'informations sur les pairs, de communication, de routage et de résolution des pairs) dans lesquels la communication se fait par des messages XML. Plusieurs implémentations existent, par exemple en Java avec JXME[9]. Cette approche semble intéressante dans le contexte visé, néanmoins elle nous semble plutôt réservée à l'exécution de services distants, sans possibilité de personnalisation par exemple.

XtremWeb[10] est une plateforme pour la distribution de tâches en mode P2P sur des grilles de calcul, utilisée dans le cadre de l'ACI GRID. Elle a été conçue pour la distribution de calculs sur des pairs qui peuvent être demandeurs ou fournisseurs de ressources de calcul. La distribution est centralisée (par construction) même si une hiérarchie de serveurs permet de réduire la charge sur le serveur central.

La plupart des plateformes de développement de systèmes P2P ne sont pas vraiment adaptées au P2P pur. En particulier, la plupart des environnements de développement pour la grille reposent sur des services centralisés (éventuellement hiérarchisés) d'allocation de ressources à des tâches indépendantes, ce qui pose parfois des problèmes lors de la montée en charge liée au nombre élevé de ressources et de pairs. Au delà de ce problème de performances, s'ajoute la difficulté de la prise en compte des problèmes d'adaptation ou de personnalisation, qui rendent inexistantes à notre connaissance les plateformes réellement adéquates.

2.1.5 Note juridique : DADVSI, P2P et nos travaux...

Lors de la rédaction de ce mémoire, une loi relative au *droit d'auteur et aux droits voisins dans la société de l'information* (DADVSI) a été adoptée. Cette loi, publiée au Journal Officiel le 3 août 2006, vise à interdire bon nombre de logiciels de partage de fichiers tels que présentés plus haut qui contiennent effectivement (mais pas exclusivement !) une grande quantité d'œuvres numériques protégées par des droits d'auteur.

Nouvel article (L335-2-1.), extrait du code de la propriété intellectuelle :

> Est puni de trois ans d'emprisonnement et de 300 000 euros d'amende le fait :
> « 1° D'éditer, de mettre à la disposition du public ou de communiquer au public, sciemment et sous quelque forme que ce soit, un logiciel manifestement destiné à la mise à disposition du public non autorisé d'œuvres ou d'objets protégés ;
> « 2° D'inciter sciemment, y compris à travers une annonce publicitaire, à l'usage d'un logiciel mentionné au 1°.

[8] http://www.jxta.org
[9] http://platform.jxta.org
[10] http://www.lri.fr/~fedak/XtremWeb

> [Les dispositions d'assouplissement ont été déclarées non conformes à la Constitution par la décision du Conseil constitutionnel n° 2006-540 DC du 27 juillet 2006.]

Nous tenons à préciser que nos objectifs sont bien d'exploiter le modèle P2P en tant que support topologique et modèle de communication et non pas de proposer une méthodologie de conception de systèmes P2P purs pour favoriser ou développer le type de logiciel prohibé par cette loi. En effet, il est tout à fait envisageable de proposer l'ajout de *mesures techniques de protection* telles que proposées dans cette même loi, pour limiter les abus au niveau de chaque client du système P2P. Cette préoccupation n'est pas la nôtre (il s'agit d'un problème d'ingéniérie) et par conséquence il n'en sera jamais fait mention ultérieurement.

2.2 Intergiciels

2.2.1 Présentation

Un intergiciel (ou *middleware*, ou logiciel médiateur ou encore bus logiciel) est une couche intermédiaire entre un système d'exploitation et une application, qui offre à cette dernière des services de haut niveau permettant de faire des abstractions sur le système et le matériel sous-jacent (cf figure 2.2).

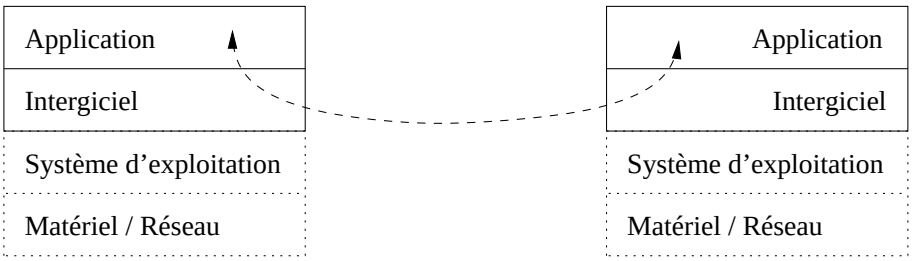


Fig. 2.2 – Abstraction des couches systèmes et matérielles

A l'origine, les premiers intergiciels avaient pour objectif de permettre l'interopérabilité dans le modèle client/serveur[11], pour cacher la répartition et l'hétérogénéité des composants matériels, systèmes et réseaux. On trouve actuellement dans la littérature un large spectre de middlewares. Des middlewares réseau comme CORBA ou RMI permettent de simplifier l'accès à des objets distribués sur des machines distantes. Certains (PVM, Linda, MPS...) cachent uniquement l'aspect répartition, d'autres encore (JDBC/ODBC) permettent l'accès transparent à des bases de données de différentes natures.

[11] Une vision simpliste qui circule sur Internet définit un middleware comme le "/" de client/serveur

2.2.2 Catégories d'intergiciels

D'après Bernstein [Ber96], les intergiciels se déclinent en quatre catégories :

1. Moniteur de transaction (TP, *Transaction Processing monitors*) qui fournit des outils pour le développement et le déploiement d'applications transactionnelles distribuées ;
2. Appel de procédure à distance (RPC, *Remote Procedure Call*) qui permet l'exécution d'une routine sur une application distance comme si elle se trouvait sur la machine locale ;
3. Intergiciel orienté message (MOM, *Message-oriented Middleware*) qui fournit des mécanismes asynchrones d'échange de données entre programmes, ce qui permet la création de véritables applications distribuées ;
4. Bus logiciel (ORB, *Object Request Brokers*), qui permet aux objets qui constituent une application d'être répartis et partagés au travers d'un environnement réseau hétérogène.

Dans nos travaux, nous nous intéressons essentiellement aux catégories 3 et 4.

2.2.3 Apports et limites

Les abstractions offertes par les intergiciels rendent la conception, le développement et la maintenance des applications plus simples. De plus, les intergiciels permettent de résoudre les problèmes d'interopérabilité.

Toutefois, les technologies de type client-serveur, RPC à objets (tels CORBA[12], Java RMI[13]) ou encore *Web Services*, ont été développées dans le cadre de systèmes stables dans lesquels toutes les interactions peuvent être prévues. Elles permettent d'activer à distance des services génériques conçus pour un grand nombre d'utilisateurs. Pour le développeur, elles offrent un modèle de programmation, à base d'interactions synchrones, qui permet de faire abstraction de la répartition. Néanmoins, ces technologies ne permettent pas de répondre de manière satisfaisante aux besoins de flexibilité et d'extensibilité.*A contrario*, les technologies offrant l'adaptation au niveau middleware permettent d'obtenir à moindre frais une qualité de service relativement stable, indépendamment de la complexité du système à mettre en œuvre.

De plus, avec l'augmentation du nombre d'objets interconnectés, les intergiciels doivent d'une part supporter la montée en charge et d'autre part proposer des mécanismes de reconfiguration dynamique aux applications potentiellement mobiles, afin d'optimiser leur fonctionnement. Ces caractéristiques doivent être complétées par des propriétés de sécurité et de sûreté de fonctionnement.

Depuis quelques années, certains auteurs [Agh02] proposent d'introduire des mécanismes d'adaptation au niveau intergiciel pour adapter les applications à leur environnement d'exécution. A ce niveau, l'adaptation reste transparente (et simple) pour le programmeur de l'application en restant suffisamment proche pour être pertinente. En effet, l'adaptation au niveau du système d'exploitation (ou au niveau matériel) ne peut être que très générique et peut ne pas couvrir tous les besoins spécifiques de l'application.

[12] *Common Object Request Broker Architecture*
[13] *Remote Method Invocation*

Les intergiciels adaptables permettent alors d'envisager la conception d'applications efficaces et robustes dans un contexte d'environnement variable. Les évolutions de l'environnement peuvent provenir de la mobilité physique du système, de la mobilité de l'application (éventuellement partielle) autant que des changements dans le contexte d'exécution.

Intergiciels ouverts

L'objectif principal d'un intergiciel est d'aider un concepteur d'application à résoudre ou simplifier des problèmes d'interopérabilité et de distribution. Pourtant, la plupart des intergiciels sont basés sur des implémentations ou des protocoles propriétaires, rendant les applications dépendantes d'un unique distributeur. Cette dépendance peut avoir des impacts négatifs sur la flexibilité et la maintenabilité de l'application. Pour cela, il nous semble judicieux d'utiliser des intergiciels ouverts (au sens de l'ouverture logicielle du terme *Open Source*), c'est à dire dont les spécifications sont publiques ou bien reposant sur des standards ouverts. Par exemple, la communication dans le cadre de *web services* repose sur des protocoles et formats ouverts (HTTP, SOAP, XML...), ce qui offre une interopérabilité maximale.

2.3 Composants logiciels

2.3.1 Présentation

Sur le plan du génie logiciel, développer une application aujourd'hui revient souvent à trouver le bon compromis entre la production de code sur mesure (spécifique mais coûteux et long à développer et valider) et la réutilisation de logiciel existant (qu'il suffit parfois de configurer, mais qui peut être moins adapté aux besoins). L'approche *composant* facilite ce compromis et constitue une technique de conception particulièrement avantageuse dans le cadre des systèmes répartis.

Il n'existe pas de définition canonique d'un **composant logiciel**, une vison consensuelle [Szy02, Ous05] présente un composant comme un morceau de logiciel « assez petit pour que l'on puisse le créer et le maintenir, et assez grand pour qu'on puisse l'installer et le réutiliser ». C'est une unité de structuration et de composition qui spécifie précisément, via des interfaces, les services synchrones et évènements asynchrones rendus et requis. Le composant logiciel est une évolution du concept d'objet qui reprend ses objectifs d'origine tels l'encapsulation et la séparation entre interface et réalisation, la réduction de la complexité et la réutilisation dans une perspective de structuration plus importante que dans le modèle objet. Un composant peut être déployé indépendamment de toute application et notamment de son langage de conception, ce qui simplifie son déploiement.

Le composant peut être vu comme une boîte noire (l'implémentation est cachée), dont seule l'interface permet de connaître les services offerts et requis (dépendances entre composants). Avec un modèle de composant donné, programmer revient à construire des architectures dans lesquelles on assemble différents composants.

2.3.2 Principales caractéristiques

Paradigme composant / conteneur

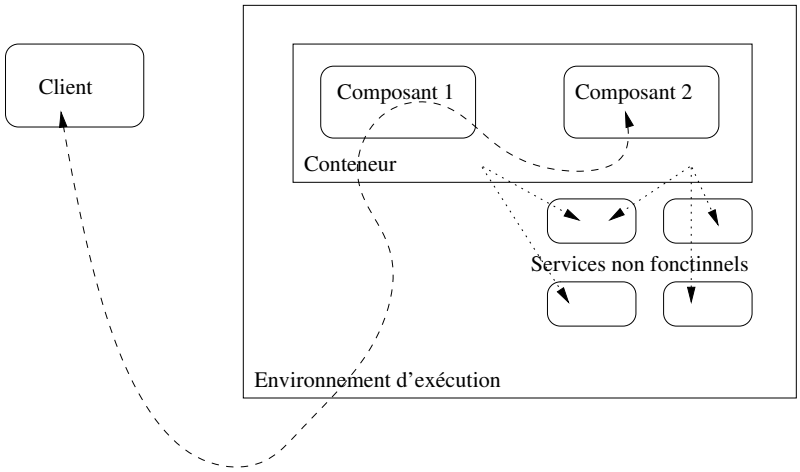


FIG. 2.3 – Modèle composant / conteneur

Pour chaque modèle, un environnement d'exécution (ou plateforme à composants ou serveur d'applications) fournit l'ensemble des services permettant l'instanciation, la configuration et l'exécution des composants, en leur offrant des services non fonctionnels comme par exemple des mécanismes de communication, de transaction, de sécurité, de persistance... Les composants accèdent à ces services au travers d'une entité appelée **conteneur**, sorte de capsule au sein de laquelle s'exécute le composant (figure 2.3). Le paradigme composant / conteneur accentue la séparation entre les aspects non fonctionnels et le code métier du composant : c'est un mécanisme de séparation des préoccupations et des niveaux.

Exemple d'un modèle de composant : EJB

Les EJB ou *Enterprise JavaBeans™*[14] sont une spécification proposée par Sun Microsystems pour décrire un modèle de composants distribués du monde Java, exécutés du côté serveur. L'interopérabilité entre serveurs d'application est basée sur un middleware réseau qui peut être RMI ou CORBA/IIOP. Dans cette norme, un composant correspond à un *Bean*, c'est à dire une classe Java pour laquelle seuls les services fournis sont déclarés (via une interface). Dans la version 2.0 de la norme, la spécification propose trois types de composants : *Entity Bean* représente un objet métier qui existe de façon permanente (une facture, un client...), *Session Bean* associé à un service fourni pour un unique client et *Message-Driven Bean* pour piloter les messages asynchrones.

L'exemple ci-dessous (figures 2.4 et 2.5) montre le code associé à un EJB de session minimal, qui offre un service unique : `String monService()`.

[14] http://java.sun.com/products/ejb

```
package fr.irit.lyre.monEJB;

import javax.ejb.EJBObject;
import java.rmi.RemoteException;

public interface TestEJB extends EJBObject {
        public String monService() throws RemoteException;
}
```

FIG. 2.4 – Interface d'un EJB

```
package fr.irit.lyre.monEJB;

import javax.ejb.SessionBean;
import javax.ejb.SessionContext;
import java.rmi.RemoteException;

public class TestEJBBean implements SessionBean, TestEJB {

        public String monService() {
                return "Et voila le travail !";
        }

        //méthodes déclarées dans SessionBean
        //pas de code particulier nécessaire dans notre exemple
        public void ejbCreate() {}
        public void ejbRemove() {}
        public void ejbActivate() {}
        public void ejbPassivate() {}
        public void setSessionContext(SessionContext sc) {}

}
```

FIG. 2.5 – Code du composant EJB

Modèles d'assemblage

Programmer par composant consiste à prendre des composants existants (on utilise parfois le terme de composant sur étagère, ou COTS *component off-the-shelf*) et à les assembler pour former le logiciel. Pour être réutilisables, les composants ne doivent pas être limités par des problèmes autres que sémantiques. Il est donc nécessaire de fournir des spécifications correspondant à la structure des composants ainsi qu'à la manière de les connecter et les composer. C'est ce qu'on appelle un *modèle de composant* [MP01].

Les premiers modèles de composants se limitent à un assemblage *à plat*, c'est à dire qu'ils ne permettent pas de composer des composants pour former des composants de plus haut niveau (ou composite). C'est le cas de ceux de la famille de Microsoft : COM, COM+ et DCOM (Component Object Model[15]) ainsi que du modèle de l'OMG : CCM (CORBA Component Model[16]).

Un modèle de composant académique, Fractal[17], est développé dans le cadre du consortium ObjectWeb. Il propose une dimension hiérarchique, dans laquelle des composants

[15] http://www.microsoft.com/com
[16] http://www.omg.org/technology/documents/formal/components.htm
[17] http://fractal.objectweb.org

peuvent être assemblés récursivement (d'où le nom du projet) et être ainsi perçus comme de nouveaux composants. L'autre aspect intéressant concerne la capacité d'introspection des composants. Ceux-ci peuvent découvrir dynamiquement l'architecture de l'application via une API dédiée.

Mécanismes de sûreté des assemblages

Lors de l'assemblage de composants, on peut essayer de vérifier si celui-ci est valide en confrontant les spécifications des services offerts et requis par les composants. Une première approche consiste à se baser sur des mécanismes de typage, en exploitant les signatures des services. Mais cela peut se révéler insuffisant. Ainsi, lorsqu'un concepteur développe une application par composants, il peut utiliser des composants écrits par des tiers, dont il ne connaît pas le comportement interne, et qui peuvent ne pas répondre à ses attentes. Pour éviter cela, il est possible d'étendre les spécifications par des *contrats* qui explicitent des propriétés sémantiques (par exemple au moyen d'expressions OCL, *Object Contraint Language*, langage de contraintes d'UML), de qualité de service, ou encore de propriétés environnementales du composant.

Architectures logicielles

Il est possible de monter d'un niveau en abstraction et de considérer l'organisation, les connexions entre composants ou encore les flots de contrôles et de données échangés. C'est ce qui constitue la base d'une *architecture logicielle* (ou *application framework*). Cette branche récente du génie logiciel cherche à faciliter et à organiser l'implémentation, l'exécution, l'exploitation et la maintenance d'un logiciel sans se focaliser sur des problèmes d'implémentation (algorithmique ou structures de données). Le niveau d'abstraction fourni par une architecture permet de réaliser certaines vérifications (propriétés de sûreté, de sécurité, de QoS...) indépendamment de l'implémentation sous-jacente et avant même l'implémentation, ce qui permet de gagner en fiabilité et en temps de conception. La description des assemblages et réassemblages dynamiques d'une architecture peut être réalisée avec des langages plus ou moins formels appelés ADL (*Architecture Description Language*). Certains ADL permettent par la suite de réaliser certaines validations, en particulier au niveau des assemblages obtenus (cohérence de l'architecture).

2.3.3 Apports et limites

L'approche composant permet la séparation des différentes préoccupations (aspects métiers entre eux et aspects métiers *vs.* aspects non-fonctionnels) ainsi que la réutilisation de codes existants. Cela contribue à la simplification du développement et de la maintenance d'applications.

Par contre, dans la plupart des implémentations de serveurs d'applications, les services non fonctionnels sont limités en nombre, non extensibles, et donc peuvent ne pas couvrir tous les besoins.

De plus, l'installation d'un composant dans son environnement d'exécution n'est pas toujours simple. En général, il faut exploiter un descripteur de déploiement (fichier XML)

qui contient toutes les caractéristiques nécessaires au bon déploiement d'un composant (catégorie, interfaces, services...). Le déploiement consiste à partir de ce descripteur à connecter le composant dans l'architecture existante et l'activer (en initiant son cycle de vie), ce qui peut requérir des tâches additionnelles de configuration, à la charge de l'administrateur du serveur d'application. Ce qui rentre en contradiction avec les besoins d'autonomie des applications que nous avons décrites.

Enfin, l'utilisation des modèles de composants académiques ou industriels cités ci-dessus implique l'installation de l'environnement d'exécution adéquat. La lourdeur des serveurs d'application existants ne semble pas adaptée à des petits périphériques tels que présentés dans nos exemples.

Pour ces raisons, nous nous intéressons davantage au concept de composant qu'aux modèles de composants existants. Aussi, nous ne préconiserons pas l'utilisation spécifique de l'un d'entre eux dans le cadre de notre solution. Les prototypes présentés qui sont réalisés en langage Java se limitent à un modèle proche d'EJB, c'est-à-dire qu'un composant est caractérisé par son interface, une ou plusieurs réalisations et un moyen d'exécution (son conteneur).

2.4 Agents

2.4.1 Présentation

Nous appelons **agent logiciel** une entité autonome capable de communiquer, disposant de connaissances et d'un comportement privés ainsi que d'une capacité d'exécution propre. Un agent agit pour le compte d'un tiers (un autre agent, un utilisateur) qu'il représente sans être obligatoirement connecté à lui.

L'agent étend le concept d'objet, en y ajoutant des capacités d'autonomie (indépendance lors de l'exécution), de proactivité (capacité à prendre des décisions de manière autonome) et de communication. Le modèle de programmation par agent est adapté aux environnements décentralisés et évolutifs grâce à ces propriétés.

A partir de cette définition générale, on peut décliner les agents de deux manières distinctes : soit selon leur utilisation fonctionnelle (agents pour la recherche d'informations, la surveillance, le commerce électronique, des agents assistants, des agents conversationnels...) soit selon leurs capacités ou leurs fonctionnalités (capacité de collaboration, capacité d'apprentissage, mobilité, flexibilité...).

Les systèmes multi-agents ou SMA [Fer95] forment une branche de l'Intelligence Artificielle dans laquelle des métaphores sociologiques ou biologiques sont employées pour la conception et la mise en œuvre de systèmes artificiels intelligents. Pour J. Ferber, un agent est une entité physique ou virtuelle :

a. qui est capable d'agir dans un environnement
b. qui peut communiquer directement avec d'autres agents
c. qui est mue par un ensemble de tendances (sous la forme d'objectifs individuels ou d'une fonction de satisfaction, voire de survie, qu'elle cherche à optimiser)

d. qui possède des ressources propres
e. qui est capable de percevoir (mais de manière limitée) son environnement
f. qui ne dispose que d'une représentation partielle de cet environnement (et éventuellement aucune)
g. qui possède des compétences et offres des services
h. qui peut éventuellement se reproduire
i. dont le comportement tend à satisfaire ses objectifs, en tenant compte des ressources et des compétences dont elle dispose, et en fonction de sa perception, de ses représentations et des communications qu'elle reçoit.

Plusieurs modèles d'agent permettent d'implémenter différentes stratégies de fonctionnement. Par exemple dans le modèle BDI [RG95a] (pour Beliefs Desires Intentions), un agent possède trois composantes principales : les croyances, les désirs et les intentions. Un mécanisme d'ordonnancement permet de choisir les tâches à accomplir pour réaliser ses intentions, en tenant compte des autres composantes.

Dans ce mémoire, nous utilisons l'agent comme outil de génie logiciel et par conséquent, nous nous intéressons plutôt à leurs caractéristiques physiques (autonomie, mobilité, communication...).

2.4.2 Agents mobiles

Un **agent mobile** [FPV98, BI02] est un agent logiciel qui peut se déplacer d'un site à un autre en cours d'exécution pour se rapprocher de données ou de ressources. Il se déplace avec son code et ses données propres, mais aussi avec son état d'exécution. L'agent décide lui-même de manière autonome de ses mouvements. En pratique, la mobilité ne se substitue pas aux capacités de communication des agents mais les complète (la communication distante, moins coûteuse dans certains cas, reste possible). Afin de satisfaire aux contraintes des réseaux de grande taille ou sans fil (latence, non permanence des liens de communication), les agents communiquent par messages asynchrones.

Apports à la problématique

L'utilisation d'agents conduit à la décentralisation de la connaissance et du contrôle. Développé aux frontières du génie logiciel et des systèmes répartis, le concept d'agent mobile est *a priori* destiné à la mise en œuvre d'applications dont les performances varient en fonction de la disponibilité et de la qualité des services et des ressources, ainsi que du volume des données déplacées. Le principe de délégation permet à l'agent d'opérer en étant déconnecté du client. La mobilité des agents et leur autonomie améliorent la sûreté (tolérance aux pannes par redéploiement des agents sur des nœuds en service) et permettent le suivi de matériel ou d'utilisateurs mobiles (réseaux actifs, informatique nomade). Les agents mobiles sont donc un outil à part entière pour l'adaptation des systèmes.

Dès les premières publications, la recherche d'information a été présentée comme une application potentielle importante des agents mobiles voire comme la *killer application*.

Dans [CHK94], les auteurs précisent le champ d'application : sources d'information multiples réparties, volumes importants, prise en compte des spécificités du client, interactions avec le client et la source. *A priori*, les avantages des agents mobiles sont nombreux :

- La mobilité d'agent permet à un client d'interagir localement avec un serveur et donc de réduire le trafic sur le réseau qui ne transporte plus que les données utiles (éventuellement pré-traitées). En outre, cela permet des transactions plus robustes que les transactions distantes.
- L'exécution d'agents spécialisés offre davantage de souplesse que l'exécution d'une procédure standard sur les sites serveurs.
- L'asynchronisme, l'autonomie et la réactivité des agents leur permettent de réaliser une tâche tout en étant déconnecté du client, ce qui est particulièrement utile dans le cas de supports physiquement mobiles (clients ou serveurs d'information).

Le principal apport des agents mobiles se situe sur un plan du génie logiciel : les agents mobiles sont des unités de structuration des applications ; ils unifient en un modèle unique différents paradigmes d'interaction entre entités réparties (client-serveur, communication par messages, code mobile).

Limites

Plusieurs expériences significatives ont été menées concernant l'apport des agents mobiles, en particulier dans le domaine de la recherche d'information [BGM+99, PSP00, HF00, KDB01, TCA01]. Cependant, on constate qu'en pratique les technologies à base d'agents mobiles n'ont encore été que peu utilisées. On peut avancer ici quelques éléments d'explication.

Les problèmes de sécurité posés par l'utilisation d'agents mobiles constituent un frein à l'expansion de cette technologie. Les risques concernent la confidentialité, l'intégrité et la disponibilité des machines hôtes et des agents, ainsi que des échanges sur le réseau [Tsc99]. Des éléments de solutions existent à base de chiffrement symétrique ou asymétrique (clé publique/privée) et de confinement (cf. annexe A).

Le problème reste cependant ouvert et dépasse largement la problématique des agents mobiles, comme par exemple le problème des attaques par déni de service (*denial of service*) dans lesquelles un service est saturé de requêtes afin de le rendre indisponible.

Par rapport aux technologies plus classiques, le concept d'agent mobile offre un cadre fédérateur pour traiter des problèmes différents et se substituer à diverses solutions *ad hoc* ; c'est l'argument *génie logiciel*. Mais pour l'instant, cet argument a eu peu de poids face au saut technologique demandé aux développeurs.

Enfin, les expériences primitivement menées avec des applications relativement figées dans des environnements stables et des réseaux locaux mettaient peu en évidence l'intérêt des agents mobiles en terme de flexibilité. On peut penser qu'aujourd'hui le contexte applicatif a suffisamment changé et qu'il est nécessaire d'explorer davantage cette technologie au regard des problèmes posés par le passage à l'échelle et l'ouverture des systèmes.

2.4.3 Intergiciels à agents

Le modèle de programmation par agent offrant des abstractions sur l'exploitation du réseau (localisation, communication...), on peut considérer que le niveau agent est une couche intergicielle.

Par exemple, un système Anthill[18] est un ensemble de machines distribuées sur lesquelles sont déployés des systèmes multi-agents. Leur interaction permet de résoudre des problèmes complexes (grâce à des comportements émergents et des algorithmes génétiques) comme par exemple le routage des messages. Anthill s'inspire des colonies de fourmis, en proposant des agents aux comportements simples, autonomes et pourvus d'un environnement sur lequel ils basent leurs actions [BMM02]. Chaque machine possède une interface appelée *nid*, qui fournit à l'application des services spécifiques aux systèmes répartis : gestion de ressources, communication, gestion de la topologie, planification des actions. Dans leurs exemples applicatifs, les concepteurs montrent que le développement d'une application répartie est simplifié par l'usage de leur système.

Il existe quelques intergiciels permettant de concevoir des applications à bases d'agents mobiles, toutefois ceux-ci sont souvent limités par leur manque de flexibilité, que ce soit au niveau de la gestion de l'hétérogénéité des supports d'exécution ou des capacités d'adaptation leur permettant une exécution dans un environnement réellement ubiquitaire.

Nous présenterons quelques exemples en particulier (Aglets, ProActive...) dans les chapitre 3 et 5, mais de nombreuses plateformes agents peuvent être trouvées à partir de : http://www.cetus-links.org/oo_mobile_agents.html#oo_mobile_agents_software.

2.5 Techniques pour la flexibilité

Dans le chapitre d'introduction, nous avons montré les différents besoins d'adaptation des logiciels répartis à grande échelle et potentiellement mobiles, nous avons également défini la terminologie employée à propos de l'adaptation. Dans cette section, nous présentons des techniques permettant d'implémenter des mécanismes d'adaptation, en insistant sur les principes d'adaptation dynamique. Les technologies classiques d'adaptation statique (principes de modularité, héritage, délégation, code ouvert) sont assez connues pour ne pas être détaillées.

L'adaptation du logiciel est une préoccupation orthogonale aux éléments fonctionnels, au même titre que la sécurité par exemple. Dans le cas général, il n'est pas nécessaire de la prendre en compte ni de l'implémenter pour qu'un système fonctionne, mais cela peut néanmoins lui permettre d'améliorer son exécution. Dans la littérature, on trouve des exemples d'adaptation dynamique à tous les niveaux : à l'échelle de l'instruction (certains virus dits *polymorphes* peuvent modifier dynamiquement leur code -écrit en assembleur-, afin de se cacher des anti-virus), de la méthode (programmation orientée aspect), de l'objet (réflexivité et protocole à méta-objet), du composant (composant adaptable), de l'intergiciel (intergiciel adaptable) et jusqu'à l'application elle-même (IHM adaptée à l'utilisateur, plugins...).

[18] http://www.cs.unibo.it/projects/anthill

2.5.1 Programmation générative

La programmation générative est une approche de l'ingéniérie logicielle qui consiste, lors de l'exécution, à produire des éléments (en général du code) qui vont participer au fonctionnement de l'application. Par exemple, depuis Java 1.5 avec l'API d'invocation de méthodes distantes RMI, la compilation des codes intermédiaires (*stubs/skeletons*) n'est plus nécessaire. A partir de la description des services (interfaces), la machine virtuelle Java génère et charge dynamiquement les classes manquantes, puis en fabrique les instances pour permettre leur utilisation transparente depuis l'application.

Cette technique permet de générer du code spécialisé (la génération peut être pilotée par le contexte). La phase de génération peut produire du code source (adaptation statique) ou directement du code compilé. Celui-ci peut être chargé en cours d'exécution pour permettre l'adaptation dynamique d'un système.

2.5.2 Composants et adaptation

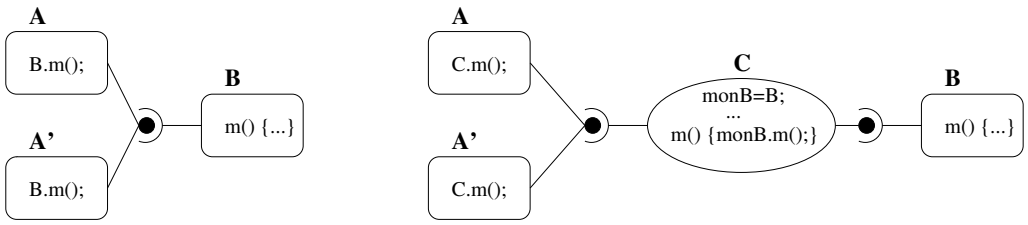


FIG. 2.6 – Intérêt du connecteur dans une architecture à composants

Si l'on compose des objets par un mécanisme de délégation ou des composants directement entre eux, l'adaptation dynamique est difficile. En effet, dans l'exemple de gauche de la figure 2.6, l'appel de méthode de A vers B est généralement compilé. Pour changer B (par un autre composant, donc avec une autre référence) il faudrait mettre à jour donc modifier le code de tous les composants connectés à B, donc de A et A' dans notre exemple. Le modèle *composant/connecteur* permet de s'affranchir de cette limitation, en intercalant un composant délégataire appelé connecteur. Dans l'exemple de droite, les composants A et A' appellent la méthode $m()$ du connecteur C. Celui-ci possède un attribut qui référence l'adresse du composant rendant ce service, ici $monB$, et l'appel de méthode est délégué à cet objet. Ainsi, lors d'une reconfiguration qui change le composant B en B', l'évolution est complètement transparente pour les composants A et A'. Pour réaliser cette adaptation dynamique, il suffit de modifier l'attribut $monB$ du connecteur C.

2.5.3 Programmation par aspects

La programmation par aspects (AOP - *Aspect-Oriented Programming* ou *separation of concerns*, [KLM+97]) est issue du constat des limites de la programmation par objet, lorsqu'il est nécessaire de prendre en compte des propriétés non-fonctionnelles telles la trace, la distribution, l'authentification ou la persistance. La dissémination des portions de code non-fonctionnels dans l'ensemble des applications limite leur réutilisation, et pose

des problèmes de maintenance. L'AOP propose de séparer clairement le code fonctionnel (les services métier de l'application) et les services non-fonctionnels.

Le logiciel est construit par assemblage ou *tissage* (*weaving*) d'aspects non-fonctionnels dans le programme métier, à des endroits appelés *points de jonction*. Le tissage est un mécanisme statique, qui fonctionne comme un compilateur (*aspect weaver*). Les concepteurs peuvent ainsi travailler sur trois parties distinctes : aspects métiers, aspects non-fonctionnels et configuration. De plus, chaque élément peut être réutilisé dans un projet futur.

Par exemple, AspectJ[19] est une extension du langage Java développée au Xerox-PARC[20] et actuellement hébergée par le projet Open Source Eclipse[21]. Les aspects sont codés dans un langage Java-étendu et la configuration repose sur l'utilisation de mots-clés spécifiques. Après tissage, le compilateur dédié retourne du code java pur ou directement du bytecode. AspectJ ne permet pas l'adaptation dynamique, mais d'autres systèmes (JAC ou EAOP[22]) le permettent, en reposant toutefois sur des outils imposants.

Certains auteurs considèrent que les services non-fonctionnels d'un serveur d'application ou plus généralement d'un middleware forment autant d'aspects.

2.5.4 Réflexivité

Un système est dit *réflexif* s'il est capable d'appliquer sur lui-même ses propres capacités d'action. Le terme réflexif est employé dans de nombreux domaines (linguistique, philosophie, mathématiques...). En informatique, un système logiciel est dit réflexif [Smi82] lorsqu'il est capable d'inspecter (propriété d'*introspection*) et de modifier sa structure et sa sémantique (propriété d'*intercession*).

La réflexivité est un moyen puissant de réaliser un découplage entre les aspects applicatifs et les aspects non-fonctionnels d'un système. On parle de méta-calcul pour désigner l'activité du système lorsque celui-ci interprète les parties réflexives (non fonctionnelles) du programme. Les zones qui définissent ce méta-calcul constituent ce que l'on nomme le *méta-niveau* d'un programme réflexif. Par symétrie, on désigne la partie applicative comme étant le *niveau de base*.

L'approche réflexive est une technique générale pour implanter des mécanismes d'adaptation dynamique [BC01]. En effet, pour pouvoir s'adapter et faire les bons choix de configuration, il faut connaître d'abord l'état de l'environnement, mais aussi son propre état (y compris d'architecture logicielle).

Par exemple le langage Java[23] possède certaines capacités réflexives. L'API Java (paquetages `java.lang` et `java.lang.reflect`) met à disposition du programmeur des classes

[19] `http://www.eclipse.org/aspectj`
[20] Palo Alto Research Center
[21] `http://www.eclipse.org`
[22] JAC [PSDF01] (*Java Aspect Components*) est un projet de développement d'un intergiciel orienté aspect, c'est en fait un serveur d'application. EAOP [DS03] (*Event-based Aspect Oriented Programing*) est un modèle qui associe l'exécution des aspects à des évènements émis lors de l'exécution du programme et capturés par un moniteur d'exécution.
[23] `http://java.sun.com`

spécialisées telles que `Class` (les instances de `Class`, des objets donc, représentant les classes connues par la machine virtuelle Java), `Method` ou `Constructor` et un ensemble de méthodes pour les exploiter (figure 2.7). Toutefois, ces capacités ne fournissent que des moyens d'observation de niveau relativement haut du processus d'interprétation du bytecode Java.

```
Constructor[] Class.getConstructors()
Method[] Class.getMethods()
Object Method.invoke(Object obj, Object[] params)
Object Constructor.newInstance(Object[] params)
```

FIG. 2.7 – Exemples de méthodes de l'API réflexive de Java

D'un point de vue extérieur, un système réflexif n'est pas plus puissant qu'un système qui ne l'est pas [Mae87]. Tous les problèmes que peut traiter un système réflexif peuvent aussi l'être par un système non réflexif équivalent. La réflexivité, comme les langages structurés, comme l'orienté objet, est une manière d'organisation interne d'un système pour faciliter son développement, son adaptation et sa réutilisation.

Exemple d'adaptation dynamique : ACEEL

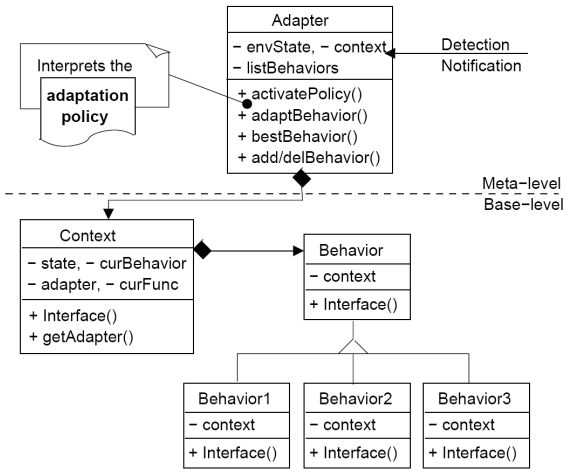


FIG. 2.8 – ACEEL : self-Adaptive ComponEnts modEL

Le modèle de composant ACEEL [CA02] a pour objectif de permettre le changement dynamique du comportement d'un composant (le code métier) réalisant un service particulier. ACEEL utilise le *Strategy design pattern*, un patron de conception [GHJV94] qui définit une même interface pour toutes les variantes d'un algorithme et un accès depuis une classe qui contient le lien de délégation vers l'implantation choisie. Au méta-niveau, un objet *Adapter* prend en charge la politique d'adaptation du composant. Les décisions d'adaptation sont prises à partir des informations fournies par sous-système de détection des changements de l'environnement d'exécution du composant.

ACEEL a été implémenté en Python[24] puis testé avec une application de vidéo à la demande, qui possède des comportements alternatifs correspondant à des algorithmes d'encodages distincts.

Architectures réflexives

La programmation orientée objet permettant de structurer les logiciels de manière particulièrement flexible et modulaire, l'utilisation conjointe des paradigmes orienté objet et réflexif a été étudiée assez tôt [KdRB91]. L'ensemble du système étant orienté objet, le méta-niveau d'un système à la fois orienté objet et réflexif est lui aussi structuré sous forme d'objets particuliers, appelés *méta-objets*, qui encapsulent les aspects réflexifs du système. Ces méta-objets interagissent avec le niveau de base, organisé en *objets de base*, en utilisant les mécanismes de réification, introspection et d'intercession que nous avons mentionnés. Du fait de la structuration en termes d'objets, on parle alors de protocole à méta-objets (MOP en anglais) pour désigner l'ensemble de conventions qui régissent les interactions entre méta-objets et objets de base.

CodA [McA95] est un modèle relativement général d'*architecture réflexive*. L'architecture permet d'étendre le concept objet pour contrôler les opérations de méta-niveau (figure 2.9). Elle contient sept méta-objets sous la forme de composants à grain fin : envoi, réception, stockage et traitement de message, recherche de la méthode à invoquer, exécution de la méthode et accès à l'état de l'objet. Ces composants sont spécialisables et relativement combinables du fait de leur faible granularité.

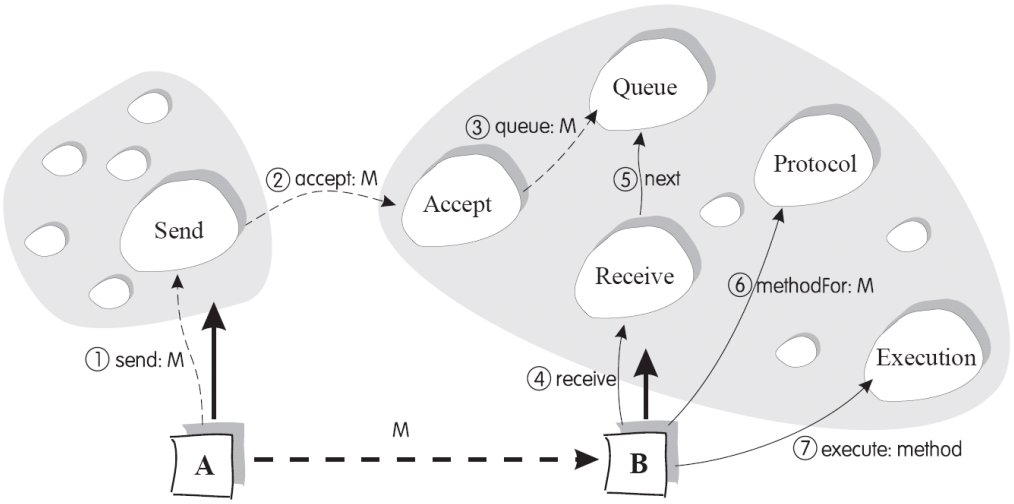


Fig. 2.9 – Architecture réflexive de CodA

[24] Langage de script interprété, orienté objet, et réflexif : http://www.python.org

2.6 Conclusion

Nous avons présenté un ensemble de technologies logicielles qui offrent chacune des apports intéressants à la construction d'applications réparties ouvertes à grande échelle, avec toutefois un ensemble de limitations (tableau 2.1).

Technologie	**Apports**	**Limites**
P2P pur	Modèle d'organisation répartie robuste	Manque d'outils (ou existant inadapté) pour la construction d'applications
Intergiciels	Abstractions du système et du matériel donc simplification de la programmation et maîtrise de l'hétérogénéité	Manque de souplesse dans les solutions existantes, support limité d'une montée en charge importante
Composants	Modularité, structuration et réutilisation, séparation des préoccupations	Déploiement des composants complexe, environnement d'exécution souvent volumineux et peu adapté à des petits périphériques
Systèmes flexibles	Nombreuses techniques qui permettent en particulier l'adaptation dynamique du système au contexte d'exécution	Augmentation de la complexité du développement
Agents logiciels	Décentralisation du contrôle et de la connaissance, adaptation via la mobilité, autonomie et réactivité	Pas d'adaptation au contexte d'exécution, parfois complexité du développement par manque d'outils adéquats (conception et implémentation)

Tab. 2.1 – Résumé des apports et limites des technologies

Sur la figure 2.10, nous avons placé ces technologies selon leurs apports. Le P2P pur est une solution intéressante comme modèle d'organisation des systèmes répartis à grande échelle, mais la difficulté de conception d'applications sur ce modèle impose l'emploi de technologies adéquates, issues du domaine du génie logiciel. Nous étudierons dans les chapitres suivant la possibilité et l'intérêt de les combiner dans des architectures logicielles, à différents niveaux.

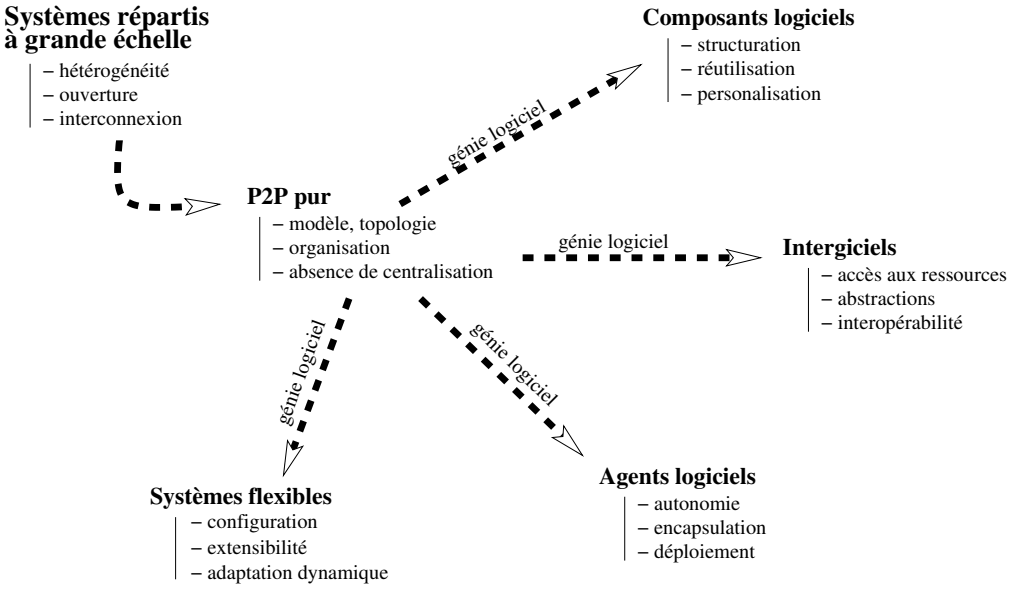


FIG. 2.10 – Positionnement des technologies

Chapitre 3

Un modèle d'agent mobile adaptable - micro architecture

Les agents mobiles peuvent servir de support d'exécution pour les applications réparties, sous la forme d'un intergiciel. Pour être utilisable à grande échelle et rester efficace lors de phases de mobilité, un tel intergiciel doit permettre la prise en compte des variations de l'environnement. Pour cela, nous proposons de mettre en place des mécanismes d'adaptation au niveau des agents. Nous présentons ici un modèle d'agent mobile dynamiquement adaptable, à base de micro-composants remplaçables et spécialisables, puis une implémentation en Java. Nous discutons enfin de quelques travaux similaires.

Les applications proposées dans le chapitre d'introduction, et plus généralement les systèmes déployés dans des environnements de répartition à grande échelle, doivent faire face à l'hétérogénéité des données, des logiciels et du matériel, à la volatilité potentielle des services ainsi qu'à de fortes variations (qualitatives et quantitatives) des conditions d'exécution. Ces différents problèmes doivent être pris en compte lors du développement et les applications doivent être capables de s'adapter dynamiquement.

Les intergiciels classiques (cf. 2.2) permettent aux développeurs de s'abstraire de certains aspects techniques, souvent délicats, liés à la répartition et aux opérations distantes. Les intergiciels adaptables ont pour objectif supplémentaire de fournir des modèles et des services pour la construction et la maintenance d'applications robustes et dynamiquement adaptables aux conditions d'exécution et de contribuer ainsi à la maîtrise de la complexité du développement. Notre démarche s'inscrit dans cette approche : nous avons étudié et développé un intergiciel générique à base d'agents mobiles adaptables par remplacement et spécialisation de composants.

3.1 Besoins d'adaptation individuelle des agents

La mobilité d'agent (cf. chapitre 2) permet de rapprocher client et serveur et en conséquence de réduire le nombre et le volume des interactions distantes (en les remplaçant par des interactions locales), de spécialiser des serveurs distants ou de déporter la charge de calcul d'un site à un autre. Une application construite à base d'agents mobiles peut se redéployer dynamiquement suivant un plan pré-établi ou en réaction à une situation particulière, afin par exemple d'améliorer la performance ou de satisfaire la tolérance aux pannes, de réduire le trafic sur le réseau, ou de suivre un composant matériel mobile.

Ainsi, la mobilité est un mécanisme important d'adaptation (complémentaire) et les agents mobiles sont un outil à part entière pour l'adaptation des systèmes. Toutefois, l'adaptation au niveau de l'application n'est pas suffisante, et nous discutons dans cette section des besoins d'adaptation statique des agents (étapes de modification d'un système en dehors de son exécution), puis des besoins d'adaptation dynamique, lors de l'exécution des agents mobiles et particulièrement lors de leurs déplacements. Ce niveau d'adaptation (intra-agent) vient compléter les différentes possibilités d'adaptation des systèmes.

3.1.1 Adaptation statique

Dans le coût total d'exploitation d'un logiciel, les parts liées à la maintenance sont plus importantes que celles liées au développement. Pour les mainteneurs, il s'agit de pouvoir ajouter de nouvelles fonctionnalités, corriger des fautes de conception ou adapter le système à d'autres plateformes par exemple. Pour les utilisateurs, il s'agit de choisir une configuration adaptée à ses besoins, identifiés *a priori*, par exemple parmi différentes interfaces (IHM) ou protocoles de communication.

Les besoins d'adaptation statique se retrouvent à toutes les échelles du logiciel, et en particulier au niveau des architectures que nous proposons. En fournissant une architecture souple et capable d'évolution, la maintenance est facilitée et par conséquence, le coût total est réduit. Ainsi, les agents doivent pouvoir être configurés à froid, c'est à dire avant leur exécution, et offrir des mécanismes d'évolution de leur architecture.

3.1.2 Adaptation dynamique au contexte d'exécution

Prenons l'exemple (fig. 3.1) d'un agent mobile créé sur un ordinateur relié à un réseau local filaire isolé de l'extérieur par des protections adéquates (pare-feu...) qui se déplace ensuite vers un environnement non fiable du point de vue de la sécurité des communications et avec des pertes potentielles du signal réseau (typiquement un portable dans un environnement public relié au réseau par liaison sans fil de type WiFi). Cet agent peut utiliser à l'origine des protocoles de communications ayant des propriétés non-fonctionnelles de faible sûreté, faible sécurité et forte performance. Après la mobilité, il serait judicieux grâce à des mécanismes d'adaptation dynamique d'utiliser des protocoles dont la sémantique est identique, mais ayant des propriétés non-fonctionnelles de forte sûreté et de forte sécurité, éventuellement au détriment de la performance.

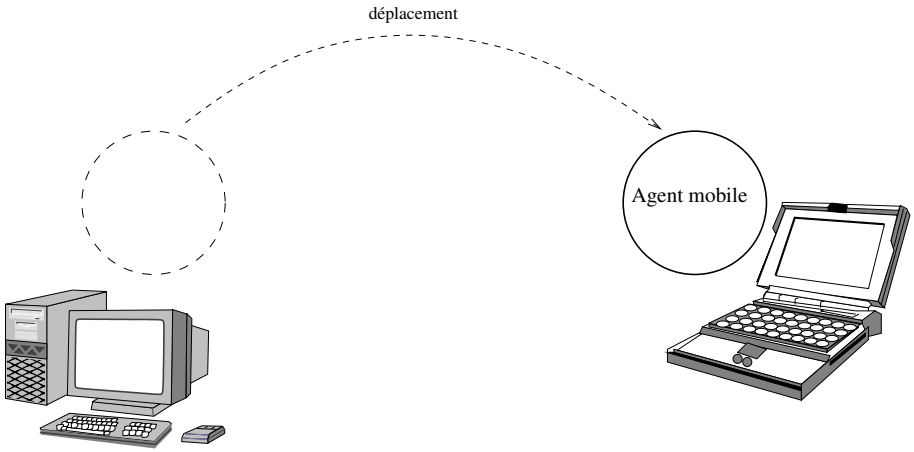


FIG. 3.1 – Besoins d'adaptation dynamique pour les agents mobiles

Dans d'autres cas, il peut être nécessaire d'adapter le protocole de localisation des agents mobiles, les protocoles de communication (suivant le type de connexion), les informations et les services mis à disposition sur les sites visités (découverte et accès à des périphériques, bases de données, informations géographiques...).

La mobilité n'est pas la seule cause des besoins d'adaptation dynamique. En effet, l'environnement d'un agent immobile peut changer, soit dans le cas où le système physique est lui-même mobile (et où l'on retrouve les mêmes besoins que ceux cités plus haut), soit dans les cas où l'environnement peut changer (évolution de la connectivité, des logiciels ou encore des périphériques connectés).

Ainsi, il est nécessaire de permettre l'adaptation dynamique des mécanismes d'exécution non fonctionnels des agents mobiles aux conditions d'exécution qu'ils rencontrent durant leur vie, pour offrir un maximum de services utiles et efficaces aux applications qui les utilisent.

3.1.3 Origine de l'adaptation

Pour rester conforme au principe d'autonomie des agents, l'adaptation dynamique doit être pilotée et réalisée de manière interne exclusivement[1]. Ainsi, dans un intergiciel à base d'agents, les adaptations dynamiques devraient avoir deux origines :

- depuis l'application, par une instruction du niveau du langage de description du code fonctionnel pour répondre à un besoin applicatif particulier (bascule des communications en mode crypté...),
- depuis l'intergiciel, lorsque le système d'accueil détecte un changement dans l'environnement et ce via un organe de décision interne à l'agent. Toutefois, l'adaptation déclenchée par un tel changement n'est qu'une notification qu'il peut ignorer. Ainsi le contrôle de l'adaptation est entièrement à la charge de l'agent.

[1]Nous employons le terme d'*agent adaptable* là où certains utilisent plutôt *auto-adaptatif* pour caractériser ces propriétés d'adaptation autonome, et d'autres proposent le terme *adaptatif*.

3.2 Principes architecturaux

Pour réaliser les solutions répondant aux besoins d'adaptation que nous avons montrés, nous proposons d'aborder le problème en proposant une organisation architecturale adéquate (configurable et reconfigurable) d'un agent mobile. Pour mettre en œuvre une telle architecture, nous utilisons :

- d'une part les principes de séparation des préoccupations et de modularité,
- d'autre part les technologies à base de composants pour réutiliser des éléments existants, permettre leur configuration et apporter les mécanismes d'adaptation dynamique (technologies présentées au chapitre 2).

De plus, le niveau d'adaptation assez fin que nous proposons nécessite de réaliser de la « micro-chirurgie » sur l'architecture des agents, ce qui implique l'emploi d'unités de structuration à grain fin pour les éléments adaptables (mécanismes de communication, de mobilité...).

3.2.1 Principes généraux

Les principes généraux suivants devront être pris en compte :

- *Transparence* : l'adaptation des agents est une préoccupation transversale, elle doit donc s'effectuer de manière transparente du point de vue de l'application agent.
- *Adaptation individuelle* : pour conserver le principe d'autonomie, chaque agent doit posséder ses propres capacités d'évolution, indépendamment des choix de configuration des autres agents du système.
- *Séparation des préoccupations* : nous nous intéressons à l'adaptation dynamique des propriétés non-fonctionnelles des agents, celles-ci doivent être séparées de manière physiques, afin d'être séparément remplaçables.
- *Faible granularité* : l'isolation physique des mécanismes adaptables et la finesse de l'adaptation souhaitée imposent l'emploi d'éléments de faible granularité (échelle d'un service unique).

3.2.2 Architecture à méta-objets

Les protocoles à méta-objets permettent de répondre aux besoins de séparation des préoccupations et de transparence des mécanismes d'adaptation et des services adaptables. Au niveau de base, le code fonctionnel dispose de primitives particulières, réifiées au méta-niveau sous forme de méta-objets. Il devient possible de manipuler ces méta-objets et en particulier de les modifier, pour adapter individuellement les services rendus au niveau de base, sans que celui-ci en soit informé.

L'architecture que nous proposons est inspirée de l'architecture réflexive proposée dans [MMMS98] (elle-même dérivée de CodA [McA95]) et reprise dans JavAct 4 (présenté en section 3.5.1). Le niveau de base contient le code fonctionnel de l'agent (comportement déterminé par le programmeur de l'application agent) et le niveau méta décrit les mécanismes opératoires de l'agent.

3.2.3 Micro-composants remplaçables dynamiquement

Pour spécialiser le méta-niveau d'un agent et le doter de nouveaux mécanismes opératoires, une première approche consisterait à procéder par héritage. Mais, dans ce cas, il serait impossible d'adapter dynamiquement les mécanismes internes des agents. L'approche par délégation, bien plus flexible, s'impose. Le paradigme composant (modularité et mécanismes d'assemblage) nous semble un bon candidat pour supporter le codage des différentes composantes des agents identifiées plus haut. Les agents sont alors construits par assemblage de composants (ce sont donc des composites), les composants fonctionnels étant placés au niveau de base, tandis que les composants non-fonctionnels sont regroupés dans le méta-niveau.

Pour une identification précise des rôles et donc pour simplifier la validation des assemblages, nous proposons l'utilisation de composants de granularité minimale c'est-à-dire n'offrant qu'un service opératoire unique. Ainsi, nous proposons d'employer le terme *micro-composants*, pour représenter les méta-composants à grain fin qui correspondent aux services non-fonctionnels du méta-niveau.

3.2.4 Définition d'un connecteur

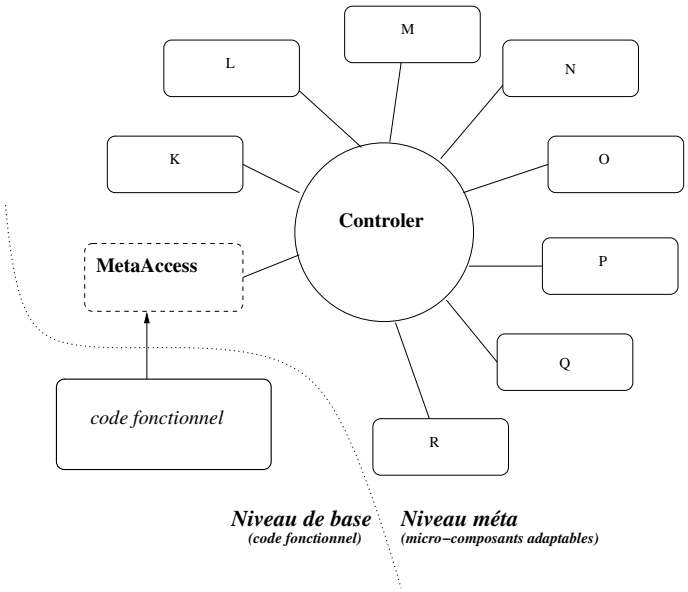


FIG. 3.2 – Architecture en étoile autour d'un connecteur

Pour permettre l'adaptation dynamique des micro-composants, nous proposons d'utiliser une architecture en étoile [LA04a], dans laquelle les micro-composants sont reliées à un connecteur nommé *contrôleur* (`Controler` sur la figure 3.2). Ce connecteur permet d'implémenter le principe de délégation pour faciliter l'évolution de l'architecture. Il est le point d'entrée unique sur les services offerts par les micro-composants et contient également un ensemble de primitives pour permettre le changement des micro-composants par substitution.

L'encapsulation par le contrôleur des appels de méthodes d'un composant autorise l'utilisation d'un composant standard par défaut, autant que d'un composant spécialisé par héritage ou utilisant par délégation les services d'un ou plusieurs autres composants. On peut ainsi envisager l'utilisation de micro-composants composites dans cette architecture.

3.2.5 Interface entre les niveaux

Un objet particulier (`MetaAccess`) permet au code du comportement situé au niveau de base d'accéder au méta-niveau. Cela permet une restriction d'accès aux seules primitives autorisées au niveau de base, à l'exclusion des manipulations du niveau supérieur (changement explicite de configuration, récupération des références des composants, etc.). Cette séparation contribue à la sûreté et à la sécurité de l'architecture d'agent adaptable.

3.2.6 Système d'accueil

Le rôle d'un système d'accueil est d'offrir des mécanismes de création et d'hébergement local d'agents. Afin de pouvoir être informé des variations de l'environnement, il englobe un ensemble de sondes de bas niveau (du type décrit dans [dSeSEK03] par exemple). Lorsqu'une des sondes détecte un changement, elle émet une notification au système d'accueil qui la répercute à l'ensemble de ses agents.

Les agents conservent leur autonomie, car ils peuvent décider d'ignorer les notifications d'évolution de l'environnement d'exécution. De plus, il n'y a pas de dépendance fonctionnelle entre les agents créés sur une place et son système d'accueil, celui-ci n'offrant que des ressources d'exécution.

La gestion de l'hétérogénéité doit être réalisée par le système d'accueil, généralement spécifique à la plateforme physique qui l'héberge, et qui propose aux agents une vision uniforme des évènements et des informations sur le contexte d'exécution.

3.2.7 Analyseur

Une politique d'adaptation est un ensemble de règles définies pour un ensemble de composants d'un agent et un ensemble d'évènements liés aux variations du contexte d'exécution. Un analyseur implémente une politique d'adaptation. Lorsqu'une sonde détecte un changement dans l'environnement, elle prévient le système d'accueil qui émet à son tour une notification individuelle à chaque agent vivant sur son site. Ceux-ci peuvent ignorer cette notification, sinon elle est prise en charge par l'analyseur. En fonction de sa politique d'adaptation, l'analyseur peut décider du remplacement d'un composant. Il utilise les services du contrôleur pour les effectuer.

Par exemple, soit un ensemble de micro-composants : K, K', L, M avec K et K' du même type (ils implémentent la même interface), donc qui sont interchangeables. On dispose de A^{δ}, un analyseur spécifique à cet ensemble de composants, contenant la règle suivante :
$A^{\delta} : p \rightarrow K \Leftrightarrow K'$
qui remplace le composant K par K' lorsqu'une propriété p de l'environnement est vraie.

Les agents sont initialement créés avec les micro-composants K, L, M et l'analyseur A^{δ}. La propriété p n'est pas vérifiée dans le contexte d'exécution.

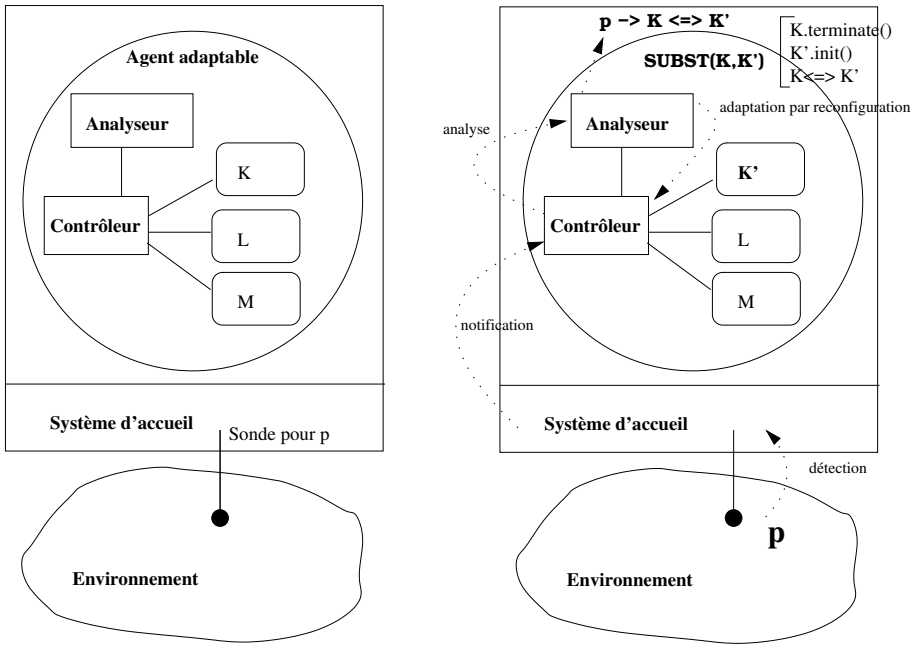


FIG. 3.3 – Mécanisme de changement dynamique des micro-composants

Lorsque la sonde de l'écouteur d'environnement détecte le passage de $\neg p$ à p, le système d'accueil envoie une notification à chacun de ses agents. L'analyseur prend la main lors des phases d'attente bloquante (réception d'un message par exemple).

Celui-ci connaît l'état précédent de l'environnement, notamment $\neg p$. Il découvre que le changement consiste en un passage de $\neg p$ à p. Sa règle d'adaptation est donc vérifiée : $p \rightarrow K \Leftrightarrow K'$. Il utilise alors les services du contrôleur pour appliquer la partie opératoire de la règle, changer K en K' (*SUBST(K,K')* sur le schéma).

Au niveau du contrôleur, le remplacement d'un micro-composant se fait en trois étapes : terminaison du précédant, initialisation du prochain, puis enfin permutation. A l'issue de ces opérations, l'agent a été dynamiquement adapté. La cohérence de la configuration des micro-composants doit être gérée par le programmeur de l'analyseur et les changements peuvent être différés à une période d'inactivité de l'agent pour être sûr que l'adaptation ne génère pas de conflit avec son exécution (cf. section suivante pour les détails d'implémentation).

Conformément au principe de séparation des préoccupations, l'analyseur contient les règles d'adaptation aux variations du contexte d'exécution et le contrôleur sert uniquement d'opérateur mécanique pour réaliser l'échange dynamique des micro-composants.

3.3 Modèle d'agent mobile adaptable

Le concept d'*agent* logiciel est similaire à celui de composant, dans la mesure où il n'en existe pas de définition très précise. De ce fait, il existe de nombreux modèles et implémentations, basés sur des propriétés variables. L'architecture que nous proposons repose sur le modèle d'acteur [Agh86], que nous justifions ci-dessous.

3.3.1 Choix du modèle d'acteur

Les acteurs sont des entités anthropomorphes qui communiquent par messages. Au sein d'une communauté, un acteur est uniquement connu par sa référence. La communication est point à point, asynchrone, unilatérale et supposée sûre. Les messages reçus sont stockés dans une boîte aux lettres et traités en série (en exclusion mutuelle). Le comportement décrit la réaction de l'acteur à un message. Le comportement est privé : il contient les données et les fonctions de l'acteur et constitue son état. L'acteur encapsule données et fonctions mais également une activité (fil d'exécution) unique qui les manipule. Un comportement d'acteur n'est donc, à un instant donné, traversé que par un fil d'exécution au plus (la synchronisation est implicite).

Lors du traitement d'un message, un acteur peut créer (dynamiquement) de nouveaux acteurs, envoyer des messages aux acteurs qu'il connaît et changer de comportement c'est-à-dire définir son comportement pour le traitement du prochain message. Le changement de comportement est un mécanisme puissant d'évolution et d'adaptation individuelle. D'une part il permet à un acteur de modifier dynamiquement la nature des services qu'il fournit (évolution de la sémantique), d'autre part il permet de faire évoluer son interface. Dans ce dernier cas, il n'est pas possible de garantir dans tous les cas qu'un message envoyé pourra effectivement être traité par son destinataire. Cependant, pour le programmeur, le changement de comportement offre une alternative à la gestion de variables d'état et à l'utilisation de gardes, qui accroît l'expressivité. Programmer un acteur revient donc à programmer ses comportements, l'enchaînement des comportements étant décrit dans les comportements eux-mêmes.

L'idée de l'utilisation des acteurs pour la programmation d'applications concurrentes et réparties n'est pas neuve. Néanmoins, elle semble particulièrement pertinente dans le cadre du calcul réparti à grande échelle de par les propriétés qui distinguent les acteurs du modèle d'objet classique (encapsulation de l'activité, changement d'interface, communications asynchrones) [AMM01] :

- d'une part, la communication par messages asynchrones est bien adaptée aux réseaux de grande taille, voire sans fil, où les communications synchrones sont trop difficiles et coûteuses à établir,
- d'autre part, l'autonomie d'exécution et l'autonomie comportementale favorisent l'intégration de mécanismes de mobilité et garantissent un certain niveau d'intégrité.

La mobilité dans le modèle d'acteur

La mobilité d'acteur peut être naturellement définie en se basant sur le traitement des messages en série et le changement de comportement, ainsi que sur la création dynamique

et l'envoi de messages [ABM+00]. Lors du changement de comportement (donc, entre deux messages), l'acteur se réduit d'une part au contenu de sa boîte aux lettres et d'autre part au comportement défini pour le traitement du prochain message. Par nature, le comportement est une entité de première classe qui matérialise l'état et qui est manipulable par programme. Il est alors possible de créer à distance un autre acteur défini à partir du comportement. On obtient ainsi simplement une évolution de l'acteur d'origine à qui on peut faire suivre tous les messages qui étaient en attente. L'état est donc entièrement transporté (via le comportement) et restauré sans que le programmeur n'ait à développer de code spécifique pour cela. Ainsi, après migration, l'acteur peut poursuivre son activité à distance en bénéficiant des acquis résultant des traitements de messages précédents. La mobilité est ainsi différée au moment où l'acteur change de comportement.

Tout acteur est donc potentiellement mobile et, ainsi définie, la mobilité n'altère pas la sémantique des programmes. On peut noter que les problèmes de cohérence de copies et de synchronisation que l'on rencontre quand on introduit la mobilité dans le modèle objet ne se posent pas ici du fait de l'unicité du fil d'exécution et de l'exclusion mutuelle sur le traitement des messages.

3.3.2 Architecture d'agent mobile adaptable

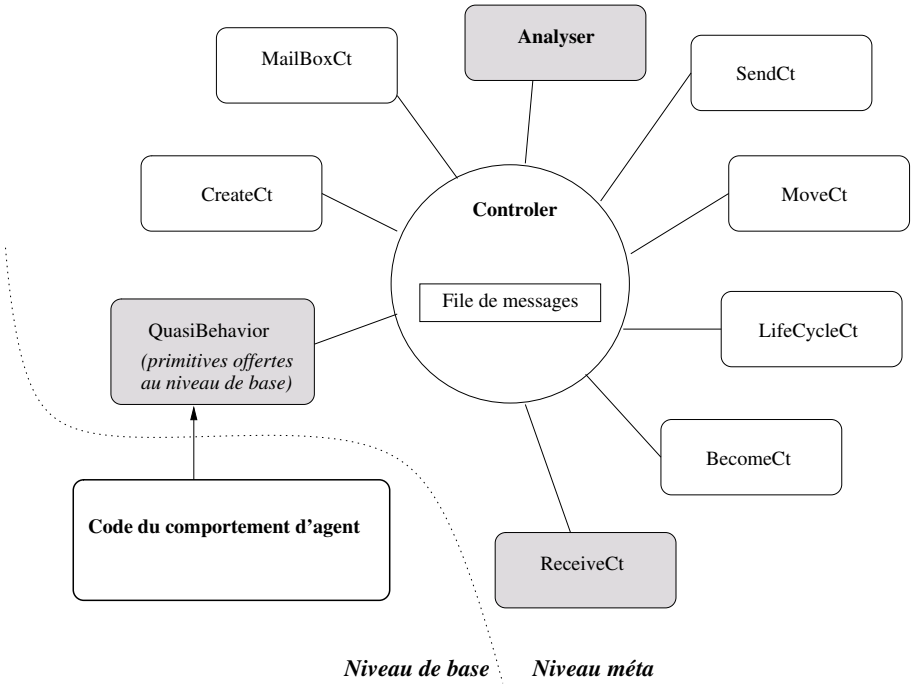


FIG. 3.4 – Architecture d'agent mobile adaptable

A partir du modèle d'acteur et en appliquant les principes proposés dans la section précédente, nous obtenons une architecture de micro-composants en étoile autour d'un connecteur (figure 3.4). Les différentes capacités des acteurs ont été réifiées sous la forme de micro-composants :

– `SendCt` pour l'envoi de message,

- `CreateCt` pour la création locale et distante d'agents,
- `BecomeCt` pour le changement de comportement,
- `MoveCt` pour la mobilité.

D'autres mécanismes opératoires internes à l'agent ont également été réifiés :

- `MailBoxCt` pour la gestion de la boîte aux lettres,
- `LifeCycleCt` pour activer et gérer le cycle de vie de l'agent.

Les micro-composants grisés sur le schéma (figure 3.4) n'ont pas été prévus pour être remplaçables. En effet, ils ne correspondent pas à un mécanisme que nous souhaiterions réifier. Nous reviendrons sur le statut du composant d'analyse dans le chapitre 5. Pour le micro-composant de réception de message, il s'agit d'un choix contraint par la conception. En effet, il constitue le point d'entrée d'activation d'un agent, et modifier ce micro-composant reviendrait à perdre la connaissance de ce point d'entrée, ce qui empêcherait toute communication entrante avec des agents qui possédaient la référence du micro-composant remplacé. Enfin, le `QuasiBehavior`, n'est pas réellement un micro-composant, il implémente le Meta-Access décrit plus haut. Il sert d'interface entre le niveau de base et le méta-niveau, pour renforcer la sûreté et la sécurité de l'architecture. Son adaptation ne nous semble pas intéressante (pas de besoin *a priori*).

Il serait possible d'implémenter des fonctionnalités partagées, via des micro-composants communs à un ensemble d'agents ou à l'ensemble des agents d'un système d'accueil, en particulier pour factoriser les mécanismes de communication. Nous pensons que dans un objectif d'autonomie et de capacité d'adaptation dynamique autonome, les agents doivent être complètement indépendants. De plus, factoriser les micro-composants de communication (`SendCt` et `ReceiveCt`) au niveau du système d'accueil risquerait de provoquer un goulot d'étranglement (en particulier pour la réception) et donc une dégradation des performances globales d'un système d'agents.

3.4 JavAct$^{\delta}$

Nous avons réalisé une implémentation de cette architecture d'agent mobile adaptable, appelée JavAct$^{\delta}$ (le delta représentant la capacité d'adaptation dynamique du modèle). Celle-ci contient tous les éléments présentés ci-dessus ; seule la partie de détection de l'environnement (sondes) a été simulée (en utilisant un fichier de propriétés modifiable en cours d'exécution).

3.4.1 Détails d'implémentation

JavAct$^{\delta}$ offre les mécanismes permettant l'adaptation dynamique individuelle des agents via l'introduction de l'analyseur et du contrôleur. Comme la mobilité, l'adaptation dynamique est calquée sur le principe de reconfiguration entre deux messages. Le contrôleur permet ici l'adaptation de l'agent depuis les 3 niveaux cités au 3.1.3.

Chaque micro-composant est un objet Java qui est à la fois une spécialisation d'un composant générique de l'architecture (il hérite de `JavActComponent`) et qui implémente

un type de micro-composant particulier. Ce type permet de les identifier (les micro-composants d'émission implémentent l'interface `SendCtI`...) et d'en définir le profil (services fournis déclarés dans l'interface).

Dans la figure 3.5, nous avons mis en évidence le chemin et les indirections successives qui permettent d'activer la méthode `send()` du micro-composant `SendCt` depuis le comportement courant.

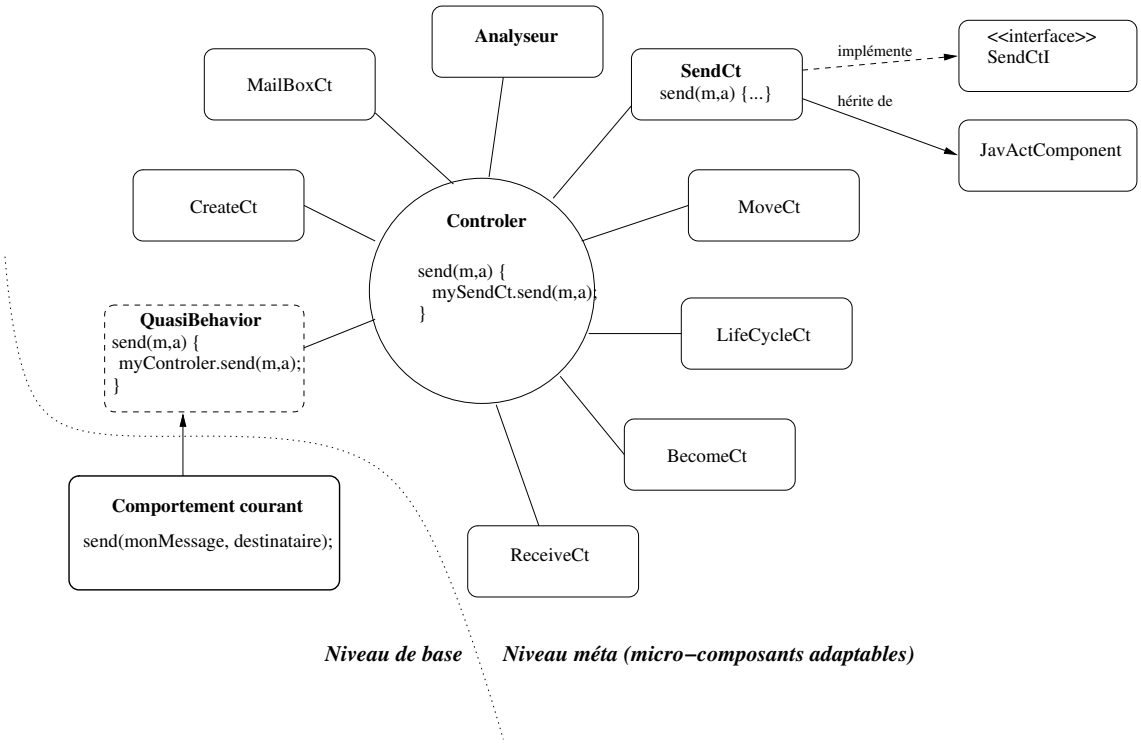


FIG. 3.5 – Micro-composants et envoi de message

Adaptation au niveau de base

Au niveau de base, les primitives `with(SendCtI box), with(MoveCtI bec)...` permettent de redéfinir la configuration de l'agent en indiquant le composant à utiliser[2].

Afin d'éviter des problèmes de cohérence (changement d'un composant en cours d'utilisation), l'adaptation n'est immédiate que si l'acteur est en attente de message, sinon elle est différée après la fin de l'exécution du comportement courant.

Le code applicatif ne peut accéder ni au contrôleur, ni aux composants, afin de supprimer la possibilité d'obtenir des configurations de composants incohérentes. Ceci est rendu possible par l'utilisation d'un lien privé entre la classe `Behavior` (comportement fonctionnel) dont tous les comportements héritent et le contrôleur. L'application ne connaissant pas le contrôleur de son agent, elle est est forcée de se limiter à l'usage des primitives mises à sa disposition dans la classe parente `Behavior`.

[2]Les types des composants (`*CtI`) sont des interfaces Java, spécifiant le profil des méthodes des composants accessibles par délégation. Ils ne contraignent pas à une implémentation particulière.

Certains auteurs [Mig99] poussent plus loin le principe de séparation des préoccupations et suggèrent de ne pas permettre de changements dans le méta-niveau qui seraient pilotés depuis le niveau de base. Nous pensons que le besoin existe (le code du comportement de l'agent peut contenir des mécanismes de décision d'utilisation d'un composant de communication adéquat, communication chiffrée par exemple), mais qu'il faut prendre plus de précautions pour que les évolutions dynamiques demandées au niveau de base se fassent de manière fiable et transparente.

Adaptation au méta niveau

Les composants ont besoin d'avoir accès à la configuration, pour utiliser les services d'un autre composant ou changer un composant afin de conserver une configuration cohérente. Par exemple, lors de la mobilité d'un agent, le composant de mobilité (`moveCt`) remplace directement le composant boîte aux lettres (`mailBoxCt`). Pour cela, le contrôleur offre des primitives d'accès sous la forme `getXXXCt`, ou `setXXXCt` (exemples dans la figure 3.6).

```
public BecomeCtI getMyBecomeCt();
public void setMyBecomeCt(BecomeCtI cpt);
...
```

Fig. 3.6 – Primitives d'adatation du méta-niveau

Cette adaptation étant réalisée intégralement au méta-niveau (donc écrite par un programmeur de composants), elle nécessite moins de précautions. C'est au programmeur de ce niveau de s'interroger sur la cohérence de la configuration de composants lorsqu'il utilise ces primitives d'adaptation.

Il est également possible d'utiliser le mécanisme d'adaptation différé identique à celui du niveau de base.

3.4.2 Exemple de composants et d'analyseur

Pour répondre aux besoins d'adaptation dynamique à l'environnement, à la QoS et à la mobilité, nous avons développé plusieurs micro-composants. Nous détaillons le fonctionnement de certains d'entre eux ci-dessous.

Localisation par *tensioning*

Pour envoyer un message à un agent distant, il est nécessaire de pouvoir l'acheminer au travers du réseau en tenant compte de la mobilité potentielle du destinataire. Dans la littérature, trois protocoles de localisation sont souvent proposés :

- une boîte aux lettres fixe, où l'agent va chercher ses messages,
- une sorte de répéteur laissé sur chaque site qui fait suivre les messages vers leur destination,
- un serveur de localisation qui indique la position d'un agent dont on possède une référence.

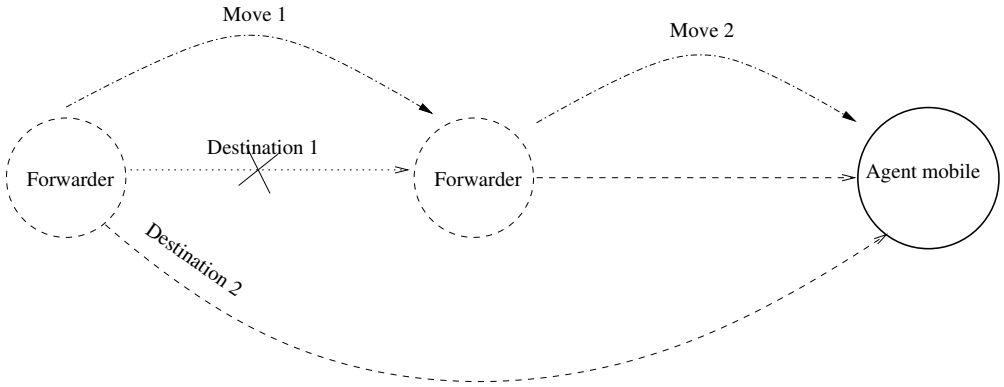


FIG. 3.7 – Localisation par *tensioning*

Dans JAVACT 4, la localisation est mise en œuvre par des répéteurs (classe `Forwarder`) qui transmettent le message de site en site, jusqu'à l'agent destinataire. Nous avons implémenté dans de nouveaux micro-composants une optimisation de ce protocole qui consiste à faire la mise à jour de la référence de l'agent de destination à chaque déplacement de celui-ci (localisation par *tensioning*, fig. 3.7), en envoyant un message de contrôle vers tous les répéteurs de la chaîne.

Localisation par serveur

D'après [BCHV02], la localisation par répéteur réparti est moins efficace avec des applications fortement mobiles, sur des réseaux à grande échelle. Dans ces cas, il est parfois préférable d'utiliser un système de localisation par serveur. Notre architecture permet de basculer dynamiquement d'un protocole à l'autre en remplaçant les micro-composants adéquats (envoi de message et déplacement).

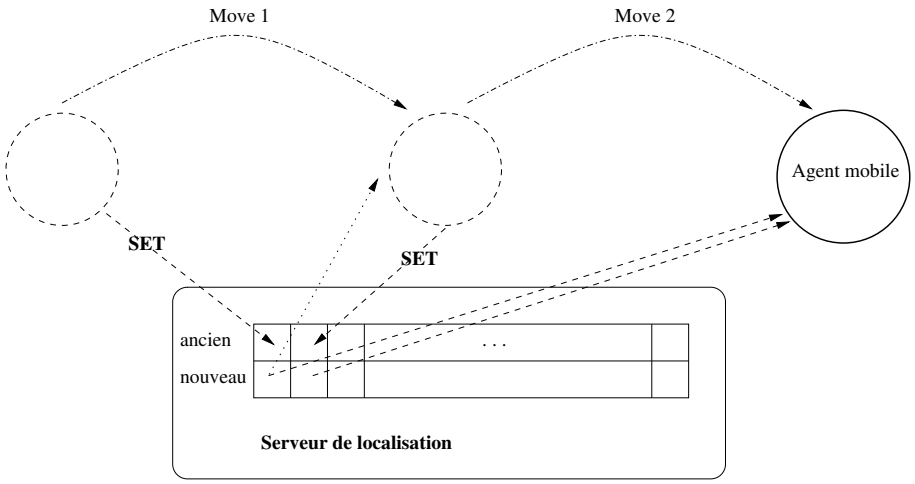


FIG. 3.8 – Localisation par serveur

Nous avons implémenté un serveur de localisation d'agent mobiles ainsi que les micro-composants qui l'exploitent. A chaque déplacement, un agent transmet son ancienne position et la position future à un serveur de localisation qui met à jour une sorte de table de localisation associant ancienne position / nouvelle position. Pour gérer les déplacements successifs, on ajoute à la table chaque nouveau couple de positions et le serveur ne renvoie que la dernière localisation connue. Lors de l'envoi d'un message, si l'agent émetteur récupère une erreur (l'agent destinataire a bougé) il envoie une requête au serveur de localisation demandant la nouvelle adresse de l'agent (fig. 3.8).

Lorsque l'agent destinataire est particulièrement mobile, il est possible que la référence récupérée par l'agent émetteur ne soit plus valide. Pour simplifier la gestion de ces cas, les requêtes de localisation sont ré-émises jusqu'à ce qu'elles aboutissent.

Micro-composant de communication en mode déconnecté

La tolérance aux déconnexions est une des qualités de services particulièrement nécessaires au bon fonctionnement des systèmes d'agents mobiles, lorsque ceux-ci évoluent dans des environnements sans fil : liaisons infra-rouges, radio (WiFi, GSM, etc...). En effet, un éloignement trop important entre l'émetteur et la base (le récepteur) peut interrompre sans préavis la communication, de la même façon que la présence d'un obstacle qui perturberait les signaux (passage dans un tunnel...).

Ce type de déconnexion est généralement de courte durée, les agents mobiles devraient pouvoir fonctionner dans de telles situations.

Pour résoudre cela, nous avons développé un composant d'envoi de messages particulier, appelé `SendCtNC`[3] (figure 3.9). Il hérite du composant original, en réutilisant ses méthodes d'envoi de messages. L'appel de la méthode (send()) de ce composant place le message en attente dans une file, puis retourne à l'appelant[4]. Un thread actif depuis la création du composant s'occupe de vérifier périodiquement si la file est non vide, auquel cas il tente l'envoi (via le composant `Send` standard) des messages. En cas de succès, le message est retiré de la file.

Ainsi, l'ordre d'émission est conservé, tant qu'une panne n'apparaît pas. Lorsque ce composant est remplacé et donc supprimé (appel de la méthode *terminate()*), il reste actif jusqu'à ce que tous ces messages soient envoyés. A moins d'une panne réelle des équipements vers la destination, les messages doivent arriver en un temps fini.

Ce composant ne justifie pas son usage par défaut lors de la configuration du système, car il est assez gourmand en ressources (un thread et une file de messages) et bien moins efficace que le composant standard, qui ne diffère pas l'émission. Son usage n'est donc intéressant que dans un environnement où les déconnexions sont probables.

[3] `SendCt` Non Connecté

[4] On diffère l'envoi des messages pour éviter des situations d'inter-blocage.

```
public class SendCtNC extends SendCt implements SentCtI, Runnable{

  private static final int TIMETORETRY=100; //millisecondes
  private transient Thread myThread=null;   //transient car non sérialisable

  private transient ArrayList msgList=null;
  private transient ArrayList targetList=null;

  public SendCtNC() {
    msgList=new ArrayList();
    targetList=new ArrayList();
    new Thread(this).start();
  }

  public void send(Message m, Actor target) {
    //rem : dans le cadre de la mobilité, il faut tester si le thread est vivant
    //move => perte du thread (car transient) et il faut le relancer
    if (myThread==null) new Thread(this).start();

    msgList.add(m);
    targetList.add(target);
    return;
  }

  public void run() {
    myThread=Thread.currentThread();
    boolean goOn=true;
    boolean terminate=false;
    while ( goOn ) {
      try{
        Thread.sleep(TIMETORETRY);
      } catch (InterruptedException e) { terminate=true; }
      if (! msgList.isEmpty()) massiveSend();  //flush des messages
      if (terminate && msgList.isEmpty()) goOn=false;
    }
  }

  public void terminate() {
    // arrêt du composant (après l'envoi de tous les messages)
    if (myThread != null) myThread.interrupt();
  }

  private void massiveSend() {
    Iterator iterM=msgList.iterator();
    Iterator iterT=targetList.iterator();
    Message auxM=null;
    Actor auxT=null;

    while (iterM.hasNext()) {
      auxM=(Message)iterM.next();
      auxT=(Actor)iterT.next();
      try {
        super.send(auxM, auxT);
        //si envoi réussi (pas d'exception), on enlève le message de la liste
        iterM.remove();
        iterT.remove();
      } catch (Exception e) {//dans ce cas il y a eu une erreur, qu'on absorbe}
    }
  }

}
```

FIG. 3.9 – Extrait du code du micro-composant SendCtNC

Micro-composant de communication chiffrée

Les communications entre agents sont généralement en clair (non chiffrées) par défaut. Lorsque le support de communication est lui-même sécurisé (cryptage des paquets dans une connexion SSL ou WPA par exemple), il n'est pas forcément nécessaire de rajouter une couche. Par contre, le besoin existe dans certains environnements, où le support n'est pas sécurisé et où les risques d'interception sont importants. Par exemple lors de l'utilisation d'un réseau sans fil, il est envisageable que les informations échangées entre la base et le récepteur soient écoutées. Dans ces conditions, le composant `SendCryptCt` trouve sa justification.

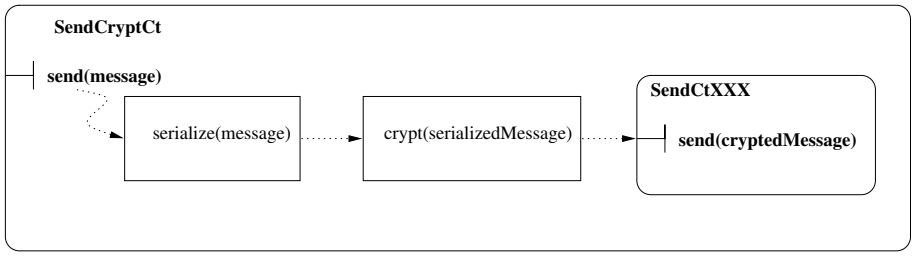


FIG. 3.10 – Composant d'envoi crypté

Ce composant est une couche supplémentaire au dessus d'un autre composant d'envoi de message, son constructeur peut prendre en paramètre la référence d'un composant d'émission. Le travail effectué par ce composant se réduit à la sérialisation puis au chiffrement du message, l'envoi étant délégué au composant d'émission qu'il encapsule. Pour déchiffrer, le composant contient les méthodes permettant d'effectuer les opérations inverses : déchiffrement, puis désérialisation.

La *sérialisation / désérialisation* est réalisée à l'aide des mécanismes d'encapsulation de RMI (`MarshalledObject()`) pour pouvoir télécharger les classes non disponibles localement, puis par les flux d'objets (`ObjectOutputStream()`) pour convertir un message au format binaire et vice-versa.

Dans l'implémentation proposée, le chiffrement est une fonction simple réversible : *ou exclusif* (XOR) entre le message et une clé stockée en dur dans le micro-composant. Toutefois, elle pourrait être générée soit pour une session d'exécution des machines JAVACT (commune à toutes les machines), soit propre à l'application, ou encore générée via un protocole adapté (type Diffie Hellman) après dialogue entre deux agents, pour servir exclusivement entre eux. On peut également implémenter sans difficulté des algorithmes plus réalistes de chiffrement symétrique (DES, Blowfish, IDEA...) sur le même principe.

Ce composant est conçu pour encapsuler n'importe quel composant d'émission. Il offre une méthode qui permet de récupérer la référence du composant sous-jacent, ce qui permet de revenir au mode d'envoi antérieur. Ceci peut se faire statiquement lors de la création d'un agent en passant en paramètre ce composant, ou de façon dynamique avec l'analyseur.

Par analogie à ce composant, on pourrait concevoir un composant de compression de données. A partir de la version sérialisée du message, une fonction de compression renverrait un message plus court, transmis sur le réseau, puis soumis aux opérations inverses du

côté du récepteur. Ce composant serait utile dans le cas de bandes passantes étroites ou saturées, pour limiter la quantité d'informations transférées. Son impact serait par contre négatif pour des petites configurations (PDA par exemple), le temps de compression pouvant dépasser celui gagné lors du transfert du message.

Un analyseur

Un *analyseur* adéquat contient les règles qui permettent aux agents d'adapter dynamiquement l'ensemble de leurs composants à l'environnement d'exécution. Par exemple, lorsque l'environnement est détecté comme passant de sécurisé à non sécurisé, le composant de communications cryptées `SendCryptCt` est activé par l'analyseur en encapsulant le composant d'émission en cours d'utilisation, donné par le contrôleur via la primitive `myControler.getMySendCt()` (pseudo-code, $\odot$ correspond à l'état précédant) :
$if\ (\odot Secured \wedge \neg Secured) \rightarrow$ `// passage en mode communications sécurisées`
$setMySendCt(new\ SendCryptCt(myControler.getMySendCt()))$
Dans le sens inverse, on réinstalle l'ancien composant de communication, ce qui revient à le décapsuler :
$if\ (\odot \neg Secured \wedge Secured) \rightarrow$ `// passage en mode communications non sécurisées`
$setMySendCt(myControler.getMySendCt().getMySendCt())$

Dans l'exemple de scénario ci-dessous (exécution du *crible d'Erathostène* [Hoa78] pour le calcul de nombres premiers, fig. 3.11), l'environnement initial n'est pas sécurisé et les places sont connectées. En cours d'exécution, l'environnement passera à l'état déconnecté. Le comportement attendu de l'analyseur est le suivant : dans une première phase, il doit proposer l'utilisation des micro-composants cryptés lors de la création des agents. Dans une deuxième phase, il doit décider d'employer le micro-composant de tolérance aux déconnexions au moment de la détection du changement de l'environnement. La composition des composants ne doit pas perturber l'exécution.

La place 1 correspond à la fenêtre n^{o}1, idem pour 2 et 3. Ce sont respectivement des places JavAct sur la machine *noman* sur les ports 1099, 1100 et 1101.

La fenêtre n^{o}4 montre le lancement du programme de l'exemple (calcul des nombres premiers <30, récupération des classes manquantes depuis un serveur `http` situé sur *noman.dnsalias.org*. Enfin la fenêtre n^{o}5 montre l'état du fichier d'environnement, commun aux 4 places pour la simulation.

Comme on le voit sur les trois fenêtres, lors de l'initialisation les paramètres *SecuredEnv* et *DisconnectedEnv* sont à *false*. L'analyseur choisit les micro-composants de communication cryptée.

A 21h15'30", la configuration change (pour la validation, on a sauvé le fichier *environment.cfg* avec la valeur *true* pour *DisconnectedEnv*, cf. fenêtre n^{o}5). Aussitôt, les systèmes d'accueil détectent la variation : message de *EnvironmentListener*, puis notification individuelle du système d'accueil *Creator* à ses agents. Dans la fenêtre n^{o}2, aucun d'entre eux n'est actif (ils se sont suicidés auparavant), donc pas de changement. Par contre, dans la fenêtre n^{o}3 l'agent 5 est vivant et se reconfigure, ainsi que l'agent 19 dans la fenêtre n^{o}1.

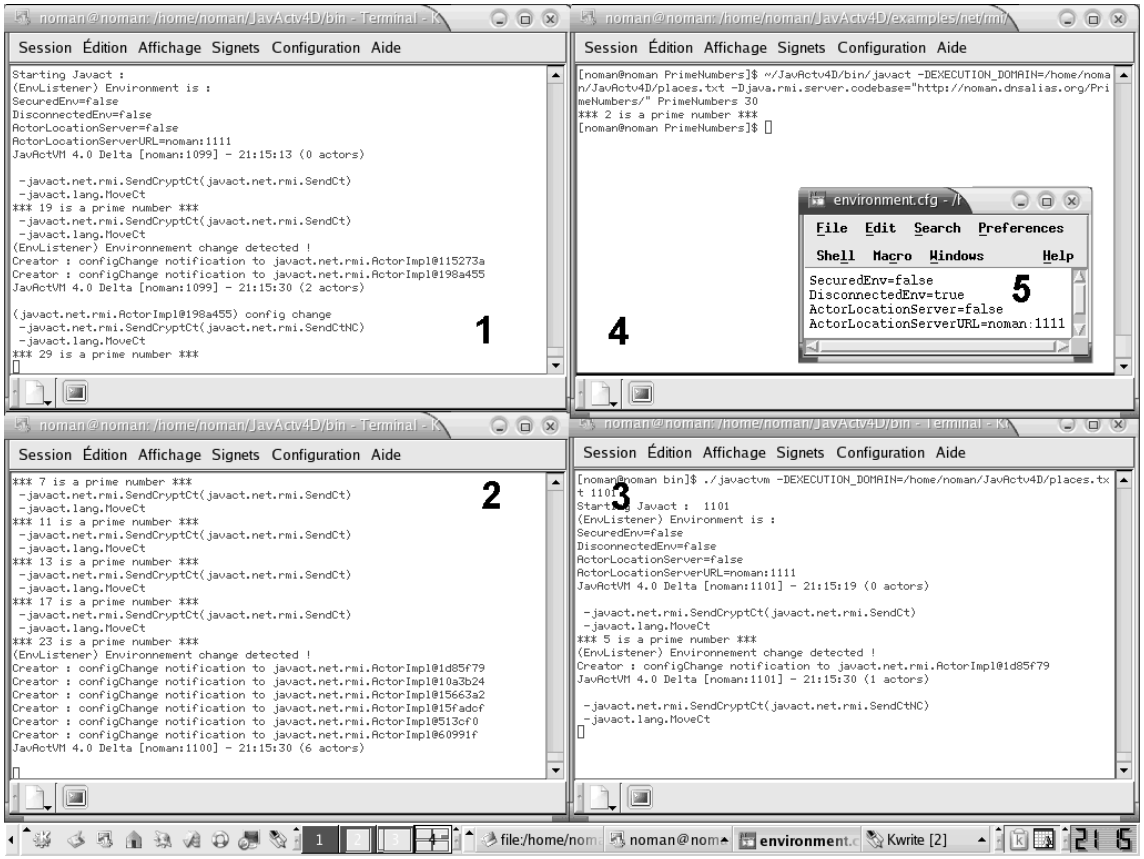


FIG. 3.11 – Capture d'écran, test de JAVACT$^{\delta}$

On voit qu'ils passent correctement au composant de cryptage avec le composant tolérant aux déconnexions (*SendCryptCt* avec *SendCtNC*).

Au final, l'application a correctement renvoyé la liste des nombres premiers <30, sans perdre aucun message, en s'étant adaptée dynamiquement de façon optimale au changement d'environnement du support d'exécution.

3.4.3 Evaluation

Afin d'évaluer sur le plan qualitatif l'apport de notre architecture d'agent mobile adaptable, nous avons développé, en collaboration avec plusieurs groupes d'étudiants, différents prototypes de logiciels répartis. En tant qu'outil de développement, JAVACT$^{\delta}$ s'est révélé encourageant à utiliser, amenant facilité et confort sur toutes les phases de conception.

Les propriétés de QoS issues des composants additionnels et l'adaptation dynamique des agents aux conditions d'exécution permettent d'obtenir des applications réparties à grande échelle globalement plus fiables, plus sûres, aux performances optimisées par rapport à l'environnement. De plus, ces propriétés sont obtenues sans que l'application n'ait été conçue pour cela, la nécessité de l'adaptation (à l'échelle du système entier) étant prise en compte au départ par un choix d'utilisation d'un intergiciel adaptable.

Sur le plan quantitatif, l'adaptation dynamique a un coût qui peut se visualiser dans le code par quelques indirections supplémentaires. Toutefois, le fonctionnement en réseau (transfert des messages, téléchargement de code...), ajoute des temps de latence non prévisibles, auprès duquel l'*overhead* lié aux indirections dans JavAct$^{\delta}$ paraît négligeable. Les tests réalisés n'ont pas permis de mettre en évidence de façon significative des différences de temps d'exécution pour des applications utilisant JavAct ou JavAct$^{\delta}$.

3.4.4 JavAct 0.5.1

La version de JavAct actuellement distribuée par notre équipe[5] reprend de JavAct$^{\delta}$ l'organisation des composants autour du contrôleur et les mécanismes d'adaptation (en laissant de côté les sondes du système d'accueil et sans fournir d'analyseur). Pour remplacer les sondes simulées de notre prototype, il serait possible d'intégrer des services standards de mesure de la qualité de service sur la grille comme *Network Weather Service*[6], *Network Analyser* et *AROMA*[7], ou encore *Fast Agent System Timer*[8].

JavAct offre des mécanismes pour la création d'agent selon la sémantique d'acteur, leur changement d'interface, leur répartition et leur mobilité, leur adaptation statique et dynamique, les communications (locales ou distantes). La mobilité et l'adaptation sont effectives au moment du changement de comportement ; ainsi, on ne diminue pas l'expressivité et le niveau abstraction et on contourne les inconvénients de la mobilité faible de Java.

Pour accélérer le développement, certaines classes sont engendrées par un petit compilateur inclus dans la bibliothèque. En particulier, le programmeur dispose d'un squelette de programme principal compilable, et n'a pas besoin d'écrire les classes de messages. Cette étape de compilation permet également d'insérer dans certaines classes des mécanismes de sûreté (changement de comportement autorisé, activation de service disponible...). Ces mécanismes restent transparents au niveau de la programmation des comportements.

Pour simplifier encore plus le développement, nous proposons un plugin pour l'environnement de développement *open source* Eclipse[9] dont l'utilisation est détaillée en annexe B.3.

3.5 Travaux connexes

3.5.1 JavAct 4

JavAct$^{\delta}$ est basé sur JavAct 4 [AMM01], une bibliothèque Java standard développée dans notre équipe pour la programmation d'applications concurrentes, réparties et mobiles qui s'appuie sur les concepts d'acteur et d'implémentation ouverte. La bibliothèque offre au programmeur un ensemble de primitives comme `send()` pour envoyer des messages,

[5] http://www.irit.fr/recherches/ISPR/IAM/JavAct_fr.html
[6] http://nws.cs.ucsb.edu
[7] http://www.laas.fr/RST/RST.html
[8] http://graal.ens-lyon.fr/~diet
[9] http://www.eclipse.org

`become()` pour changer de comportement ou `go()` pour effectuer un déplacement ; chacune de ces primitives étant réifiée sous la forme d'un micro-composant. La mobilité se fait lors du changement de comportement (entre deux messages), en créant un clone de l'agent à distance et en réduisant l'agent d'origine à une boîte aux lettres qui fait suivre tous les messages.

L'architecture de JavAct 4, grâce à sa structuration en micro-composants, est configurable. On peut interchanger (statiquement) des micro-composants dont la sémantique diffère mais qui offrent la même interface. Toutefois, l'architecture étant figée, il n'est pas possible de rajouter un micro-composant d'un type nouveau (nouvelle fonctionnalité), ni d'en supprimer. Par rapport à JavAct$^{\delta}$, la limite la plus importante de JavAct 4 est l'impossibilité d'adapter dynamiquement les micro-composants des agents.

D'autre part, un effet de bord non désiré de l'architecture de JavAct 4 permet d'accéder aux micro-composants d'un agent distant dont on connaît la référence. En effet les micro-composants sont référencés dans le corps de l'agent, lui-même accessible en tant qu'objet distant (via RMI). C'est un problème de sécurité et de sûreté non négligeable (on ne devrait pas pouvoir agir sur un agent autrement que via l'envoi de message) qui contribue à l'argument de séparation des préoccupations. Il faut séparer, voire isoler, le niveau de programmation standard (code des comportements) et le niveau de programmation des micro-composants, ce qui est fait dans JavAct$^{\delta}$.

Les micro-composants de JavAct 4 sont partagés au sein d'un système d'accueil. Dans JavAct$^{\delta}$ la séparation complète des micro-composants permet une adaptation fine et personnalisée, renforçant l'autonomie des agents.

3.5.2 ProActive

ProActive[10] [BBC+06] est un intergiciel de grille, codé en Java et distribué sous licence libre et *open source*. ProActive est l'implémentation d'un modèle pour la programmation répartie, concurrente et mobile à base d'objets actifs [Car93]. Le projet est hébergé par le consortium ObjectWeb.

Le modèle d'agent de ProActive est une extension du concept d'objet. Les auteurs présentent des exemples d'application pour tous les contextes imaginables du domaine des systèmes répartis (Grid Computing, P2P, Web Services, Client/Serveur, Agent mobiles...). Toutefois, ces avantages peuvent également se transformer en inconvénients. Du fait de la généricité du modèle, le déploiement d'une application ProActive est complexe (utilisation de scripts et de méta-données de déploiement), à rapprocher de celui de composants traditionnels. De même l'expressivité d'un code ProActive est bien moins forte que dans une plateforme spécifiquement dédiée à la programmation agent mobile.

Les possibilité d'adaptation sont avant tout statiques (configuration des protocoles de communication par exemple), mais l'utilisation du modèle de composant hiérarchique Fractal[11] permet d'obtenir des applications dynamiquement adaptables, par reconnexion dynamique des composants. Il est ainsi possible d'emboîter des composants ProActive et de

[10] http://proactive.objectweb.org
[11] http://fractal.objectweb.org

leur appliquer récursivement des opérations de contrôle, comme par exemple la migration de tout un groupe de composants vers un cluster. Cette adaptation au niveau applicatif est assez éloignée de nos préoccupations de micro-adaptation.

3.5.3 Aglets

A l'origine développé par IBM au japon [LM98], la technologie Aglets et son environnement de développement ASDK (Aglets Software Development Kit) sont actuellement maintenus au travers d'un site communautaire[12], selon un modèle libre et *open source*. L'ASDK permet de programmer rapidement et simplement des agents logiciels mobiles en Java. Le modèle d'agent est simple, associant à un objet java un thread et des méthodes d'activation spécifiques. Par exemple, il est possible d'exécuter du code après une opération de mobilité en implémentant la méthode `onArrival(...)`. L'exécution est supportée par un serveur d'Aglets nommé *Tahiti* qui permet de gérer les communications, la sécurité et la mobilité des agents.

Le modèle est intéressant pour la simplicité de la mise en œuvre d'agents mobiles, mais ne présente pas de possibilité d'adaptation. D'autre part, le modèle est plus une extension du modèle d'objet (comparable en ce sens à ProActive) dans lequel le programmeur doit implémenter de nombreuses méthodes pour décrire les activités d'un agent. Notre solution (mise à disposition des méthodes au niveau de base via un objet dont hérite chaque comportement) permet une meilleure expressivité et une meilleure lisibilité.

3.5.4 MadKit

MadKit[13] [GF00] est une plateforme développée par le Laboratoire d'Informatique, de Robotique et de Microélectronique de Montpellier (LIRMM) de l'Université Montpellier II. MadKit, écrit en Java, est fondé sur le modèle organisationnel AGR (Agent/-Group/Role). Il utilise un moteur d'exécution où chaque agent est construit en partant d'un micro-noyau fournissant les mécanismes élémentaires pour le fonctionnement d'un agent : communication locale par messages, gestion des groupes et des rôles, lancement et destruction des agents. Les autres fonctionnalités (communication distante, affichage, supervision...) peuvent être réalisées par des agents du système. Le micro-noyau peut être étendu par un système de *plugins* pour ajouter statiquement des fonctionnalités à un agent. Enfin, un environnement de développement graphique permet de simplifier la construction d'applications multi-agents.

MadKit semble un environnement assez souple pour programmer des agents de différente nature, grâce à l'extensibilité fournie par son système de plugins et les possibilités de programmation dans différents langages, dont un moteur à base de règles. Nos propositions se distinguent par les capacités d'adaptation dynamique de JavAct ainsi qu'une architecture complètement décentralisée, plus adaptée au contexte de répartition et d'instabilité que nous visons.

[12] http://aglets.sourceforge.net
[13] http ://www.madkit.org/

3.6 Conclusion

Pour répondre à la complexité croissante du développement et de la maintenance de systèmes répartis à grande échelle et mobiles ainsi qu'aux besoins d'adaptation des logiciels fonctionnant dans des contextes d'exécution hétérogènes et de qualité de services variable, nous avons proposé une architecture d'agent mobile dynamiquement adaptable qui s'appuie sur le modèle d'acteur (avec changement de comportement) et sur un ensemble de méta-composants à grains fins réifiant les services non fonctionnels organisés autour d'un connecteur. Nous avons également proposé une implémentation de ce modèle sous la forme d'un intergiciel appelé JavAct$^{\delta}$.

Cette architecture d'agent permet de répondre aux besoins de configuration et d'adaptation dynamique au contexte d'exécution, par l'intermédiaire de mécanismes internes. Toutefois, nous avons fait certains choix et imposé des contraintes dans l'architecture, limitant par exemple les composants pouvant être adaptés (par exemple, l'analyseur ne peut être adapté dynamiquement...) et en proposant le modèle d'acteur et la réification de ses mécanismes d'exécution. De plus, les ensembles de micro-composants peuvent être assemblés et modifiés sans plus de garantie que le respect de leurs interfaces de services offerts (typage par l'utilisation d'interfaces dans l'implémentation). Nous proposons dans le chapitre 5 de relâcher ces contraintes pour aller vers plus de flexibilité tout en conservant les principes posés dans ce chapitre et en améliorant les propriétés de sûreté de l'architecture d'agent.

Résumé des contributions :

- Pour aider à la conception d'applications réparties à grande échelle et mobiles, nous avons proposé un modèle d'agent logiciel mobile basé sur le modèle d'acteur, et une architecture dynamiquement reconfigurable constituée d'un ensemble de micro-composants qui implémentent des services non-fonctionnels. L'organisation des micro-composants en étoile autour d'un connecteur central permet d'adapter dynamiquement ces services en fonction du contexte d'exécution [LA04a].
- Nous avons conçu une implémentation complète de ce modèle, sous la forme d'un prototype appelé JavAct$^{\delta}$ [LA04a], ainsi qu'une implémentation simplifiée (sans analyseur et sans sonde dans le système d'accueil), JavAct 0.5 [AHL+04].

Chapitre 4

Un patron de conception des systèmes P2P purs - macro architecture

Dans ce chapitre, nous montrons comment, en associant les concepts de P2P, de composant logiciel et d'agent mobile adaptable, nous obtenons un patron de conception réutilisable pour la construction d'applications réparties à grande échelle. Celui-ci présente deux niveaux d'adaptation. D'une part les propriétés d'autonomie et de proactivité des agents, ainsi que leurs capacités de mobilité et de changement de comportement contribuent à l'adaptation des applications. D'autre part, l'utilisation d'agents adaptables permet une adaptation dynamique au contexte d'exécution (par reconfiguration de l'architecture) pour une meilleure efficacité et plus de sûreté. Nous présentons ensuite le *framework* JavAne qui est une implémentation de ce patron de conception, puis son utilisation ainsi que la façon de le réutiliser sur des exemples concrets. Enfin, nous discutons de travaux approchés.

Nous avons montré dans le chapitre 1 que la conception des systèmes à grande échelle et à topologie instable est complexe et que de nombreuses préoccupations comme la gestion de l'hétérogénéité ou des volumes de données, la volatilité, les performances doivent être considérées. Du point de vue génie logiciel, pour simplifier la conception et la mise en œuvre de tels logiciels et permettre leur réutilisation, il est nécessaire d'employer des technologies adéquates. Nous avons vu que les systèmes P2P purs décentralisés sont les mieux adaptés à la mise à disposition de ressources volatiles ou évolutives et aux systèmes à grande échelle et à topologie instable. Pour cette raison, ce modèle d'organisation est l'élément de base des propositions faites dans ce chapitre.

4.1 Patron de conception

Un patron de conception (*Design Pattern* [GHJV95] ou canevas logiciel) décrit une solution standard pour répondre à un problème d'architecture et de conception logicielle, afin de réduire le temps de conception et d'augmenter la qualité du résultat.

4.1.1 Eléments constitutifs d'un système P2P

Dans cette section, nous proposons d'identifier les différentes fonctionnalités des systèmes P2P décentralisés et d'en déduire l'ensemble des composants d'une architecture générique de ces systèmes. Nous entendons par *ressource* non seulement les éléments physiques (fichier, base de données, périphérique) mais aussi l'ensemble des services offerts sur les différents sites.

Outre les fonctionnalités d'adhésion volontaire au réseau et de retrait, les pairs doivent au minimum pouvoir :

- mettre des ressources à disposition de la communauté (publication),
- rechercher une ressource à partir d'une description,
- exploiter une ressource trouvée.

1. La publication d'une ressource consiste à l'insérer dans le réseau et à permettre à d'autres pairs d'y accéder. Au plus simple, la ressource est hébergée localement chez le propriétaire, mais elle peut aussi être hébergée ailleurs sur le réseau. Un composant de **publication** est donc nécessaire, et du côté de l'hébergeur un mécanisme doit permettre la publication de ressources par un tiers.
2. Au contraire des systèmes structurés qui proposent un ou plusieurs serveurs d'index, la localisation dans les systèmes P2P purs ne passe pas par l'utilisation de serveurs. Pour accéder aux ressources, il y a donc au départ un besoin de découverte et de localisation des pairs serveurs, puis ensuite, sur ces pairs, un besoin de fouille locale pour l'extraction des ressources et services. Pour cela nous distinguons deux composants au sein du mécanisme de recherche : l'un de localisation ou de **recherche globale** pour la découverte et l'accès aux pairs, l'autre pour la **recherche locale** sur le pair serveur.
3. En complément à la localisation, le système doit permettre à un pair de découvrir des méta-informations sur le système (sur les pairs et les ressources) et de maintenir un ensemble de connaissances sur le réseau P2P. Ces connaissances sont exploitées pour optimiser les recherches futures (par exemple par inondation). Pour cela, un élément de **méta-information** répertorie les connaissances des pairs sur eux-mêmes et sur la topologie du réseau P2P, qui peut évoluer en permanence (associations type de ressources/localisation, pairs voisins/ressources, etc.).
4. Après avoir découvert une ressource, il faut pouvoir l'exploiter. Il peut s'agir par exemple de communiquer à l'utilisateur la description (méta-données) de la ressource trouvée, d'exécuter un service puis de transmettre les résultats, de lancer une nouvelle recherche si la ressource dépend d'une autre, ou plus simplement de télécharger un fichier. Ces opérations sont à la charge d'un composant dédié à **l'exploitation des**

ressources. Celui-ci doit également superviser l'exploitation de la ressource (pour adapter par exemple l'exploitation aux conditions d'exécution : variations de bande passante, d'occupation du processeur...).

5. De plus, dans un contexte de ressources volatiles et évolutives, un composant supplémentaire fournissant un moyen de s'interfacer de manière uniforme avec les ressources (accès à une base de données, à un système de fichiers, à un *Web service*...) peut être ajouté. De cette façon, si la forme d'une ressource change (nouvelle version) seul le composant d'**accès aux ressources** doit être modifié. Des systèmes comme Spitfire[1] ou JDBC[2] peuvent être employés.

Il résulte de cette décomposition un ensemble de briques de bases pour la construction de systèmes P2P, que l'on peut implémenter sous forme de composants logiciels : publication, localisation, recherche locale, information, accès aux ressources et exploitation. Il se pose alors le problème de leur déploiement. En effet, pour augmenter les capacités de personnalisation des applications, certains de ces composants logiciels sont fournis par le client. Ils doivent donc être installés sur les sites serveurs. Ainsi, nous proposons dans les sections suivantes un modèle d'architecture distribuée à base d'agents et de composants.

4.1.2 Problématique du déploiement

Dans une première étape, les composants doivent être transportés sur les sites serveurs, pour y être déployés. Leur déploiement est constitué de différentes activités de mise et de maintien en production : installation, configuration, activation mais aussi la maintenance (adaptative et préventive) et mise à jour (voir [CFH+98] pour une définition générale des activités de déploiement). Pour être activés et fonctionnels, ils doivent s'exécuter dans un environnement adéquat qui leur fournit un ensemble de services (communication...) ainsi que des ressources d'exécution (temps processeur, mémoire...).

Le déploiement de composants classiques nécessite généralement l'usage de méta-données spécifiques, appelées *descripteurs de déploiement*, souvent fournies sous forme de fichiers XML ou de scripts spécifiques.

Dans les sections suivantes, nous proposons de réaliser ces différentes étapes, en encaspulant les composants dans des agents mobiles. En effet, ceux-ci, via leur mobilité et leur pro-activité, peuvent localiser des ressources dans des contextes de réseaux instables, et distribuer des composants sur les sites serveurs. De plus, si les composants s'exécutent à l'intérieur de l'agent, ils pourraient bénéficier des services non-fonctionnels des agents (communication asynchrone, mobilité pro-active...). Enfin, le déploiement d'un composant par une entité active et autonome permet de simplifier voire de supprimer les méta-données de déploiement, ce qui simplifie cette étape.

[1] `http://edg-wp2.web.cern.ch/edg-wp2/spitfire`, utilisé dans le projet DATAGRID pour l'accès uniforme aux bases de données

[2] Connecteur d'accès à différentes bases de données depuis le langage Java.

4.1.3 Un modèle pour le déploiement de composants

Notre architecture répartie se distingue des solutions classiques par la décentralisation et l'utilisation d'agents mobiles adaptables (présentés au chapitre précédent) pour supporter le déploiement des composants logiciels.

Déploiement des composants par les agents (macro-architecture)

Dans cette proposition, les composants métiers prennent la forme de comportements d'agents. Cela revient à encapsuler les composants métiers dans des agents mobiles, à raison d'un agent par composant. De cette manière le concepteur bénéficie des avantages du modèle de programmation agent (abstraction, expressivité...). De plus, le composant peut accéder à tous les mécanismes fournis par les agents : autonomie, communication asynchrone, mobilité, adaptation dynamique... En particulier, la mobilité permet de déplacer à volonté un composant métier, afin de le rapprocher des traitements ou d'optimiser son fonctionnement.

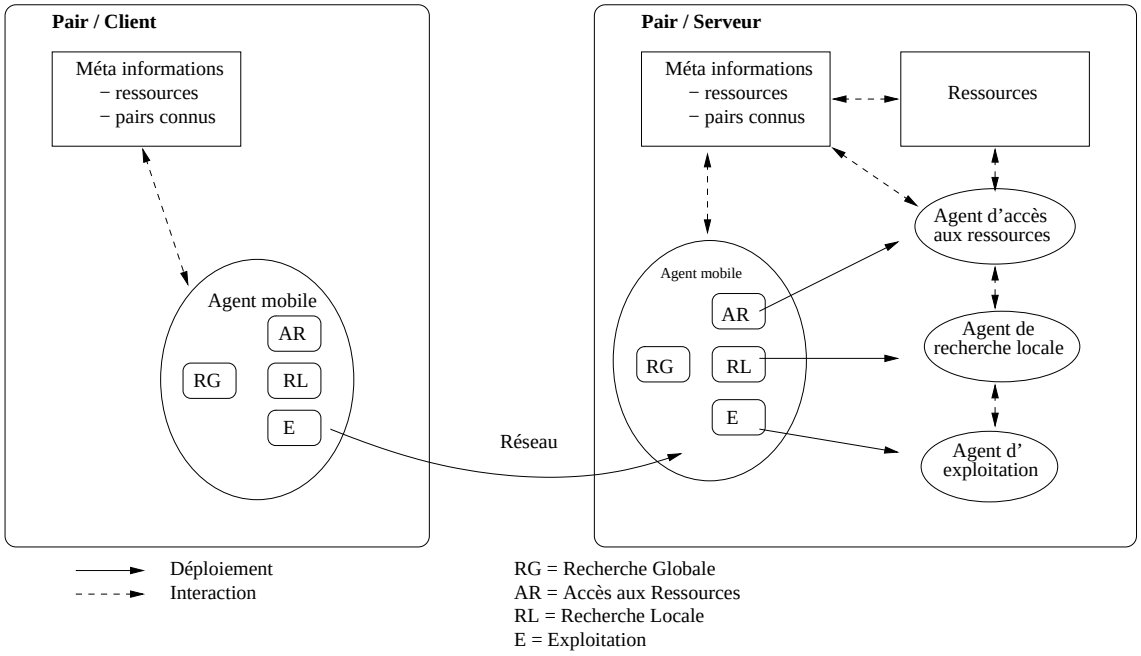


FIG. 4.1 – Déploiement des composants

Le principe du déploiement est schématisé dans la figure 4.1 (pour une meilleure lisibilité, nous avons distingué le pair jouant le rôle de client de celui jouant le rôle de serveur mais ils sont bien sûr complètement symétriques) :

– Les agents mobiles assurent d'abord le **déploiement géographique adaptatif** des composants. Pour chaque requête d'un client, un agent mobile (ou plusieurs pour exploiter le parallélisme) transporte les composants sur le réseau. Le composant de localisation (ou recherche globale, RG) constitue l'essentiel du comportement de l'agent mobile. Celui-ci se déplace de pair en pair en effectuant dynamiquement la

localisation, ce qui lui permet d'adapter la recherche à l'état courant du système P2P.
- Lorsque l'agent mobile arrive à destination, il installe un mini système multi-agent à partir des composants (AR, RL et E), pour traiter la requête localement. Là, chaque composant est encapsulé dans un agent dont il constitue le comportement et à travers lequel il offre ses services. L'agent sert de **conteneur** et d'**activateur** du composant et lui fournit les mécanismes non fonctionnels (communication...) nécessaires à son exécution.
- Les agents ainsi déployés coopèrent pour réaliser la tâche de recherche ou d'exploitation des ressources suivant une algorithmique client, ce qui permet une forte personnalisation de ces activités.

Adaptation pour les composants au sein des agents (micro-architecture)

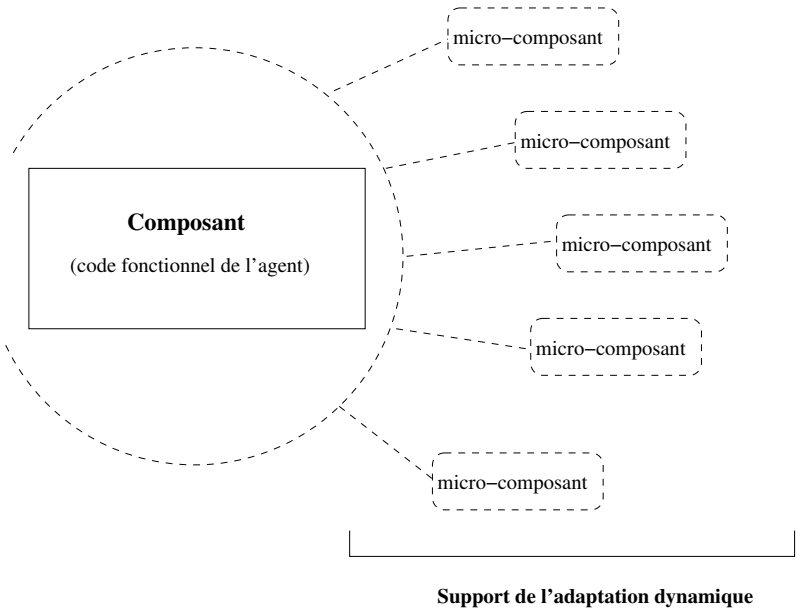


FIG. 4.2 – Environnement d'exécution adaptable

Pour augmenter l'adaptation et la robustesse des applications à base d'agents face aux problèmes d'échelle et en particulier à ceux liés à la sûreté de fonctionnement, nous avons proposé dans le chapitre précédant un modèle d'agent auto-adaptable. Nous nous appuyons sur cette micro-architecture d'agent pour adapter l'environnement d'exécution du composant. La réutilisation du modèle d'agent pour déployer des composants permet de faire bénéficier à ces composants des propriétés d'adaptation des agents (figure 4.2). On peut, par exemple, adapter le mécanisme de communication du composant d'exploitation (chiffrement, compression...) en redéfinissant le micro-composant de communication de l'agent d'exploitation.

Ainsi, l'adaptation est réalisée au plus près de l'application, en restant transparente pour le programmeur du composant métier. L'agent joue le rôle d'un intergiciel adaptable pour chaque composant et l'association d'un agent à un composant permet une finesse

d'adaptation plus importante que dans un environnement d'exécution de composant traditionnel. On peut adapter un seul agent sans que les modifications aient d'impact sur les autres agents s'exécutant dans le même système d'accueil. Cela contribue à la flexibilité globale de l'application.

4.1.4 Apports mutuels Agents-Composants-P2P

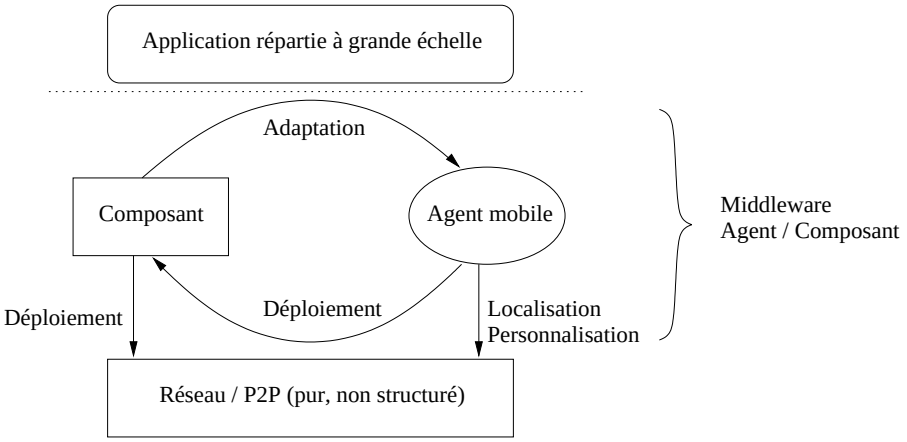


FIG. 4.3 – Complémentarité des technologies

Il est possible de faire une analogie entre le modèle composant-conteneur et notre modèle d'agent adaptable, particulièrement si l'on considère la séparation des préoccupations. D'une part, les composants comme les comportements des agents représentent le code fonctionnel (code métier). D'autre part, le conteneur permet au composant d'instancier, d'activer et d'accéder aux services non fonctionnels fournis par l'environnement d'exécution (par exemple la gestion du cycle de vie, de la sécurité ou des communications). Dans notre modèle, l'agent joue le rôle de conteneur : ses micro-composants enveloppent le comportement fonctionnel et implémentent les mécanismes non-fonctionnels (envoi des messages, réception, cycle de vie...) ou éventuellement les délèguent au système d'accueil. Dans son rôle de conteneur, l'agent supporte donc l'adaptation du composant (configuration et reconfiguration).

Pour déployer les composants, nous bénéficions donc des avantages des agents en termes de mobilité et d'adaptation : on peut déplacer et adapter le composant via l'agent qui le contient. Notre architecture d'agent adaptable permet d'obtenir la flexibilité et l'extensibilité nécessaire à l'adaptation : adapter un agent revient à changer un ou plusieurs de ses micro-composants. L'agent est ainsi le **vecteur de l'adaptation individuelle et du déploiement** des composants. Cette forme d'adaptation est complètement séparée de la programmation des composants, ce qui contribue à simplifier le développement et la réutilisation.

La mobilité et l'adaptabilité des agents jouent donc un rôle essentiel dans la phase de déploiement. Agents et composants[3] constituent une sorte de *middleware* permettant aux applications l'accès transparent au réseau P2P. Nous résumons dans la figure 4.3 la

[3]On trouvera une étude sur les apports mutuels entre agents et composants dans [Bri04].

complémentarité des technologies évoquées précédemment (une flèche de A vers B indique un apport de la technologie A à la technologie B).

4.1.5 Utilisation du patron de conception

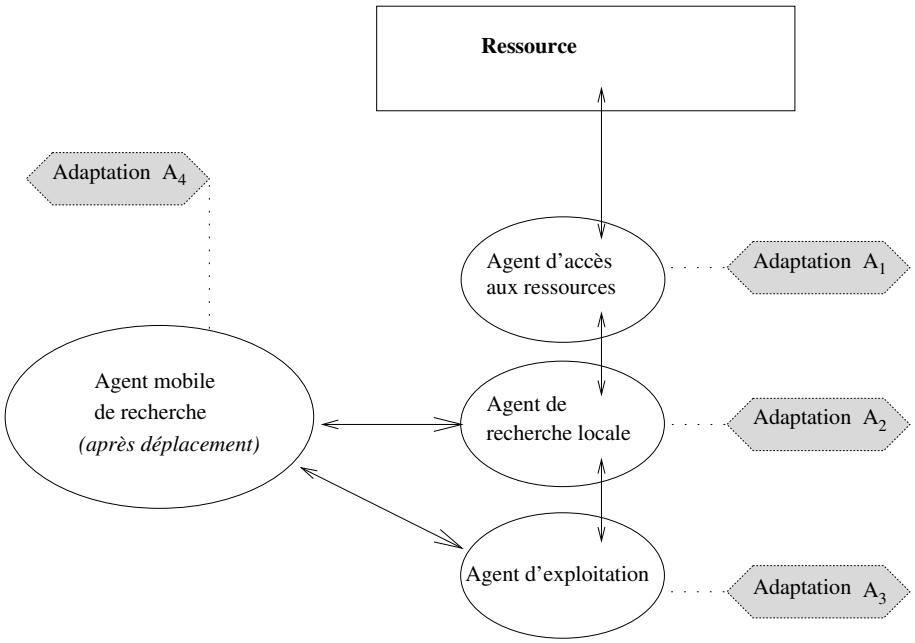


FIG. 4.4 – Les différents points de spécialisation

Adaptation de la recherche globale - A_4

Par nature, la mobilité d'agent est *proactive* : c'est l'agent mobile lui-même qui décide de ses déplacements de manière autonome. Dans notre cas, l'agent de recherche s'appuie sur une politique personnalisée qui peut faire intervenir une évaluation de la qualité des résultats trouvés, l'état du réseau, voire les connaissances du site serveur sur le reste du domaine. Il peut arrêter la recherche (si les résultats trouvés sont satisfaisants), la poursuivre sur un autre site, retourner sur le site client, éventuellement relancer une nouvelle recherche locale, etc. Ainsi, le chemin de l'agent mobile peut être construit dynamiquement et c'est au plus près des données que la décision de déplacement est prise.

Dans nos prototypes (et en particulier dans l'implémentation du composant de recherche globale proposé à la page 87), un mécanisme complémentaire d'acquisition dynamique d'informations sur le domaine a été ajouté. En parcourant le réseau et en explorant des zones initialement inconnues du client, il est possible pour ce dernier de glaner des connaissances sur les sites visités lors de la phase de recherche. Ces connaissances permettent de mettre à jour les bases de données qui, du côté du client, répertorient les propriétés des serveurs distants connus et alimentent les (futures) sélections de serveur.

Plus le client dispose d'informations sur d'autres sites, plus les chances de trouver l'information recherchée augmentent. Le protocole de recherche présenté ci-dessous permet l'acquisition dynamique d'informations par le client, *via* un simple envoi de message de l'agent au client.

Adaptation de la recherche locale - A_2

Pour effectuer une recherche non standard spécifique au client, il est habituel de déplacer les données sur le site du client. En cas de gros volumes de données, la solution alternative consiste à déplacer le code de la recherche sur le site serveur. Dans notre proposition, l'agent mobile de recherche se déplace avec un composant de recherche spécialisé. Ce composant, qui implémente l'expertise du client, est un comportement à partir duquel un agent est créé sur le site serveur. Son autonomie permet de réagir dynamiquement aux résultats obtenus et d'orienter le processus de recherche en conséquence (en soumettant une requête affinée par exemple).

Ainsi, on personnalise le processus de recherche tout en limitant les volumes déplacés sur le réseau (hors l'agent lui-même) aux résultats (et non plus à toutes les données nécessaires au calcul du résultat).

Outre la mise à jour des bases de données du client sur les serveurs répartis (qui alimentent la sélection de serveur), on peut noter l'apport indirect de la mobilité d'agent à la pertinence des résultats : le volume de données accessibles n'est plus limité par les contraintes liées au réseau et par conséquence, la recherche peut donc s'effectuer sur des données plus complètes.

Spécialisation de l'exploitation des résultats - A_3

De la même manière que précédemment, à partir d'un composant spécialisé pour l'exploitation des résultats et *embarqué* au sein de l'agent mobile de recherche, on peut créer un agent d'exploitation sur le site serveur. En fonction des résultats obtenus et du réseau (de ses propriétés et de son état), l'agent d'exploitation peut décider de transmettre les informations obtenues (ou une partie d'entre elles) à l'état brut ou compressées, de transmettre seulement leurs descripteurs, de crypter les informations transmises, etc. Les résultats peuvent également ne pas être transmis, mais conservés par l'agent mobile de recherche lui-même.

Ici, les avantages résident dans l'autonomie des agents et leur capacité à traiter l'information sur le site où elle se trouve.

Adaptation de l'agent à l'hétérogénéité des serveurs - A_1

On fait l'hypothèse que si un serveur est connu d'un client, alors le client sait également comment interagir avec lui. Pour cela, sur le site client, l'application dispose d'un composant d'accès aux ressources avec chaque serveur connu. Ce composant est *embarqué* au sein du comportement de l'agent mobile afin de servir d'interface avec le serveur

sur le site distant. Lorsqu'une nouvelle place est découverte dynamiquement, le site client doit acquérir non seulement l'adresse de la place mais aussi le composant d'accès aux ressources avec le serveur. Ceci implique que les places doivent offrir une fonctionnalité supplémentaire pour fournir ce composant.

En pratique, le composant d'accès aux ressources est un comportement d'agent : il contient tous les éléments utiles (script d'interaction et données non fonctionnelles) connus statiquement sur le site client. Sur le site distant, l'agent mobile de recherche crée dynamiquement un agent de dialogue en lui donnant ce comportement (installation) et lui transmet par message la requête et ses paramètres.

Avec un tel protocole, lorsqu'un serveur évolue, les informations connues sur celui-ci par d'autres sites deviennent obsolètes. Ainsi, il se peut qu'un agent de recherche se déplace sur une place avec un composant de communication périmé. Dans ce cas, l'agent peut acquérir *in situ* et à la volée le nouveau composant d'accès aux ressources[4] (et le donner au client pour mémorisation).

De manière générale, les composants peuvent être fournis par le pair client, par le pair serveur ou par un tiers. Ainsi, le composant d'accès aux ressources peut ne pas être systématiquement fourni par le serveur. Mais en cas de mise à jour côté serveur (évolution du mécanisme d'accès aux ressources), il faut prévoir un protocole permettant l'acquisition *in situ* et à la volée du nouveau composant d'accès. Ceci contribue à la robustesse de la recherche et permet l'adaptation dynamique aux évolutions des serveurs.

L'utilisation d'un agent de recherche mobile permet donc l'adaptation dynamique de l'agent aux évolutions du serveur, sans que les opérations de maintenance du serveur ne dépassent le cadre local.

4.1.6 Quelques exemples concrets

Nous présentons ici quelques exemples d'utilisation du *framework*, en explicitant comment réutiliser ou spécialiser les différents composants du patron de conception.

Appel de procédure à distance

Il s'agit non pas de retrouver un document quelconque mais un service logiciel rendu par un serveur, de transmettre des paramètres à ce service, d'exécuter celui-ci et de rapatrier ensuite le résultat de l'exécution.

L'adaptation requise se situe au niveau de A_3 : ici, l'exploitation consiste à exécuter le service logiciel trouvé sur le serveur puis à rapatrier le résultat de l'exécution. Notons que le composant A_3 devra être déployé chez le client et initialisé avec les valeurs des paramètres du service recherché. Celles-ci seront ensuite exploitées lors de l'exécution de ce service.

[4]Cette stratégie élémentaire peut ne pas satisfaire les besoins de sûreté ou de sécurité. Au besoin, elle peut être affinée. Dans tous les cas, la stratégie choisie résulte d'un compromis entre les contraintes de performance, d'efficacité, de sûreté et de sécurité.

Recherche de composants logiciels

Ici, il s'agit de rechercher, pour le compte d'un client, un composant logiciel à des fins de mise à jour ; l'installation du nouveau composant peut demander la recherche et l'installation d'autres composants complémentaires.

Les adaptations requises se situent au niveau de A_3 et de A_4. Au niveau de A_3, le composant doit transmettre (ou mémoriser) le nouveau composant (et les documents associés). Au niveau de A_4, la recherche globale doit prendre en compte les besoins découverts dynamiquement et rechercher les composants complémentaires nécessaires à la mise à jour.

Mise à jour de composants logiciels

Il s'agit maintenant de rechercher un service logiciel rendu par différents serveurs et de remplacer le composant logiciel qui rend ce service par un autre composant (ici, la mise à jour est initiée par le fournisseur du composant).

Les adaptations requises se situent au niveau de A_3 et de A_4. En effet, l'exploitation des résultats consiste maintenant à remplacer le composant logiciel trouvé par le composant fourni par le client lors du démarrage de la mise à jour (et éventuellement à installer des composants complémentaires dont la localisation est initialement connue ou issue d'une nouvelle recherche). Par ailleurs, que la recherche locale ait réussi ou pas, il faut continuer à parcourir l'ensemble des serveurs accessibles pour faire la mise à jour là où cela est nécessaire.

Mise à jour hiérarchique

L'objectif est similaire au précédent, mais la mise à jour du composant sur un serveur peut imposer la mise à jour d'autres composants sur le même serveur.

Les adaptations requises se situent au niveau de A_3 et de A_4 comme précédemment. En effet, lorsqu'un composant logiciel est mis à jour sur le serveur, celui-ci peut demander la mise à jour d'autres composants. Il est donc nécessaire au niveau de la recherche globale de prendre en compte ces demandes pour rechercher les composants nécessaires à cette opération, l'effectuer sur ce serveur et conserver ces composants de manière à prévoir les éventuelles demandes de mise à jour des autres serveurs lorsque la recherche globale reprendra celle du composant initial. Il est également souhaitable d'introduire une forme de parallélisme pour réduire la durée de la mise à jour globale.

4.2 Le *framework* JavAne

Afin de vérifier la validité de nos propositions, nous avons conçu, à partir du patron de conception proposé dans ce chapitre, un premier prototype de logiciel de mutualisation de ressources baptisé JavAne[5] [LAP04]. Cette appellation a été reprise pour le *framework* lui-même dont nous décrivons quelques éléments ici.

En pratique, le *framework* JavAne est une implémentation du patron de conception présenté en 4.1 : c'est un ensemble de classes Java auquel est associée une méthodologie de conception (identification des composants à adapter ou à spécialiser puis des micro-composants d'agent nécessaires pour l'adaptation dynamique). Les classes Java définissent des composants sous forme de comportements d'agents ou des micro-composants qui se greffent sur le *middleware* JavAct[6], auquel on ajoute un ensemble de micro-composants non-fonctionnels permettant l'adaptation dynamique des agents à leur contexte d'exécution (communication cryptée, mode déconnecté, protocoles de localisation...).

Un composant destiné à être embarqué dans un agent est construit à partir d'une classe Java dont le diagramme de classe simplifié est présenté en figure 4.5. Par héritage et implémentation d'interfaces prédéfinies, chaque composant est un comportement d'agent JavAct transportable sur le réseau. Pour chaque composant `XXX` (par ex. localisation...), le travail du programmeur consiste à écrire uniquement le code (`XXXImpl`) en spécialisant par redéfinition le composant générique existant.

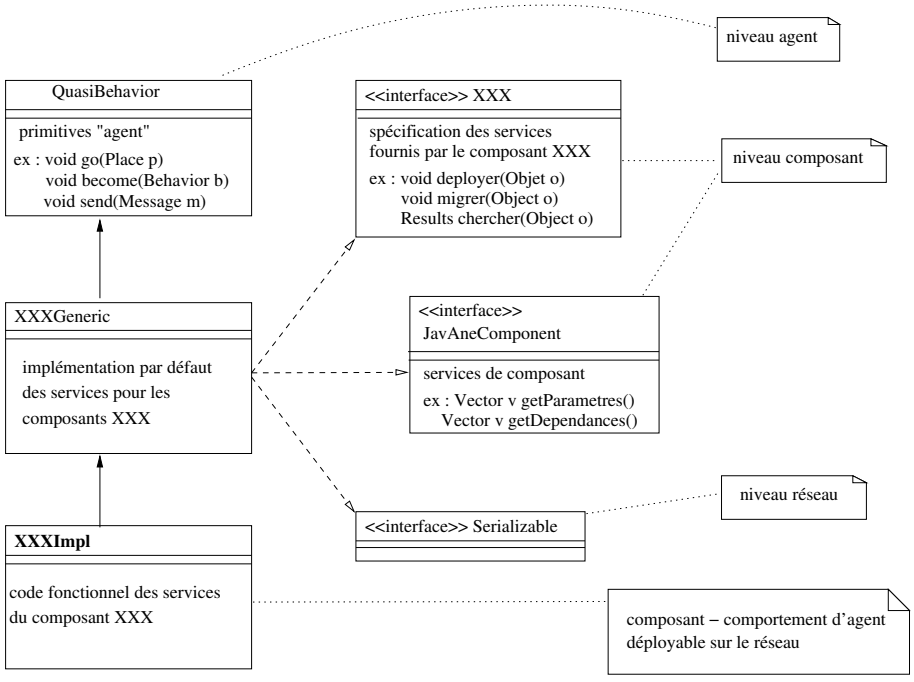


Fig. 4.5 – Détail de l'implémentation d'un composant

[5]Ce nom a été proposé par une équipe d'étudiants qui a réalisé un prototype d'échanges de fichiers P2P s'inspirant du logiciel E-Donkey, cf. `http://noman.flabelline.com/ter-2003.html`. La macro-architecture présentée ici est une abstraction de l'architecture de ce prototype.

[6]Implémentation de notre modèle d'agent adaptable, cf. 3.4

Ce *framework*, qui a été entièrement implémenté, testé et validé expérimentalement dans des configurations variées (réseaux internet et intranet, wifi, PC, stations Sun...), permet de réaliser des prototypes pour les exemples proposés au chapitre 1. Différents composants ont été implémentés sous forme de comportement d'agents JavAct, mettant en œuvre différents protocoles, dont certains seront détaillés plus bas :

- composant de localisation : recherche de pairs par inondation (type Gnutella), par cheminement d'agent mobile, hybride
- composant d'exploitation : téléchargement de fichier
- composant de recherche locale : recherche de fichier par filtre multi-critères (taille, type, nom...)
- composant d'accès aux ressources : accès à une base de données au format texte (fichiers + descriptions)

4.2.1 Un composant de localisation

Dans le cadre de cette étude, plusieurs composants de recherche globale (localisation) ont été effectivement conçus, implémentant différents protocoles de recherche. Nous décrivons ici l'un d'entre eux, afin d'illustrer la mise en œuvre au moyen des agents mobiles. Celui-ci s'inspire des systèmes de recherche par inondations (type Gnutella). Pour éviter de générer trop de charge de calcul, nous l'avons paramétré par un nombre maximal d'agents de recherche déployés en parallèle, et limité par une variable strictement décroissante dite d'énergie.

La recherche se déroule en deux phases. Dans la première, un agent représentant le client crée n agents de recherche globale sur n sites initialement connus du client, sur lesquels ils effectuent leur recherche. Lors de la seconde, chaque agent de recherche globale peut se déplacer sur un site connu de son hôte (et pas nécessairement du client originel) pour y poursuivre ses recherches.

La figure 4.6 présente le code fonctionnel du comportement de l'agent de recherche globale (avec le mécanisme d'acquisition dynamique d'informations sur le domaine), débarrassé des déclarations de variables et des traitements d'erreurs. La méthode *run*() est automatiquement exécutée lors de l'arrivée de l'agent sur une place (déclarée dans l'interface *StandAlone*).

L'expansion du système de recherche est limitée par l'utilisation d'une variable d'énergie interne à chaque agent qui représente son potentiel d'activité et qui décroît strictement à chaque mouvement et en fonction des résultats (trouvé, non trouvé, erreur). Un agent qui n'a plus d'énergie ou qui ne peut plus se déplacer (panne, déconnexion...) s'arrête. La couverture du réseau est donc partielle (sur un réseau de grande taille l'exhaustivité est trop coûteuse voire impossible), mais en précisant les paramètres n et *energie*, l'utilisateur peut influencer la quantité de documents trouvés ou la rapidité de réponse. Par ailleurs pour des raisons d'efficacité, chaque requête est identifiée (par une estampille temporelle) afin que deux agents n'effectuent pas la même recherche sur la même place.

Ici, l'état de l'agent mobile de recherche évolue en cours d'exécution par réduction d'énergie. Dans d'autres protocoles, l'état peut traduire des situations plus complexes ; dans le cas général, il évolue par changement de comportement.

```
public class InondationChercheurImpl extends RGGeneric implements StandAlone {

    public InondationChercheurImpl(Actor client, Vector parms, ComposantAR cAR,
                                   ComposantRL cRL, ComposantE cE) {
        energie=((Integer) parms.get(0)).intValue();
        signatureRech=(String) parms.get(1);
        ...
    }

    public void run() {
        // Si plus d'énergie, on stoppe la recherche globale
        if (energie <= 0) {
            suicide();
            return;
        }

        // A-t-on déjà visité la place courante ?
        if (! marquerPlace(myPlace(), signatureRech, client))
            energie -= E_VISITEE;
        else {
            energie -=E_NOUVEAU;

            // Création du système d'agents
            AgentAR=create(cAR);
            AgentRL=create(cRL);
            AgentE=create(cE);

            // Activation de l'agent de recherche locale,
            // puis poursuite de la recherche globale.
            // Les activités sont complètement parallèles
            send(new JAMstart(client,AgentAR,AgentE),AgentRL);
        }

        // Collecte d'informations sur le domaine
        String[] infosPlaces = getInfosPlaces(myPlace());
        send(new JAMaddInfosPlaces(infosPlaces), client);

        // Déplacement aléatoire s'il reste de l'énergie, sinon suicide
        if (energie > 0) {
            String[] places = getPlaces(myPlace());
            String prochainePlace = places[random.nextInt(places.length)];
            go(prochainePlace);
        } else suicide();
    }
}
```

FIG. 4.6 – Comportement de l'agent de recherche globale

4.2.2 Un composant d'exploitation

Nous avons développé un mécanisme de téléchargement qui exploite le potentiel de parallélisme inhérent à la répartition des données. Il permet le téléchargement à partir de sources multiples, dans le cas où plusieurs fichiers identiques sont disponibles sur des serveurs distincts. L'identité est définie à partir de la taille du fichier et de sa signature (implémentée sous forme de signature SHA-1[7]).

Les agents d'exploitation créés sur les sites serveurs sélectionnés envoient des parties distinctes du fichier à télécharger à un agent récepteur situé sur le site client. Celui-ci les récupère en parallèle. Afin de s'adapter aux bandes passantes disponibles, il peut stopper ou réviser dynamiquement les tailles des tranches allouées à chaque agent d'exploitation en fonction du débit de chaque canal. Côté client, un contrôle d'intégrité est effectué lorsque le fichier est réputé complet, en recalculant sa signature et en la comparant avec celle fournie par la recherche.

Dans la figure 4.7, le code a été débarrassé des déclarations de variables et des traitements d'erreurs. La méthode *executer*() (respectivement *adapter*() et *stop*()) est exécutée lorsque l'agent traite un message de la classe $JAMexecuter$ (respectivement $JAMadapter$ et $JAMstop$).

L'autonomie et l'adaptation dynamique de ces agents aux débits et aux pannes de communications permettent d'obtenir un protocole de téléchargement particulièrement efficace et robuste, adapté à la réalité des réseaux hétérogènes, et pourtant implémenté de façon très simple.

[7] http://www.itl.nist.gov/fipspubs/fip180-1.htm

```
public class ExploitationTelechargementFichier extends ComposantE{

    public void executer(Object gestFichiers) {
        nomComplet=(String) gestFichiers;

        // Lors de la première activation, on ouvre le fichier et on positionne
        // le pointeur au début de la tranche allouée
        if (premiereActivation) {
          fichier=new FileInputStream(nomComplet);
          fichier.skip(maTranche.getDebut());
        }
        premiereActivation=false;
        if (maTranche.Finie()) {
          stop();
          return;
        }

        // Lecture de la tranche suivante
        maTranche.progression(nbLus);
        if (maTranche.getAvancement()+TAILLE_MORCEAU > maTranche.getFin()+1)
                              nbALire=maTranche.getFin() -
          maTranche.getAvancement() + 1;
        else nbALire=TAILLE_MORCEAU;
        tab=new byte[(int)nbALire];
        nbLus=fichier.read(tab);

        // Envoi du morceau au client et réactivation
        morceau=new MorceauDeTranche(maTranche.getNumero(), tab,
                                     numTelechargeur);
        send(new JAMtraiterResultat(morceau), client);
        send(new JAMexecuter(gestFichiers), ego());
    }

    public void stop() {
        fichier.close();
        suicide();
    }

    public void adapter(int nouvelleTaille) {
        TAILLE_MORCEAU = nouvelleTaille;
    }
}
```

FIG. 4.7 – Composant de téléchargement de fichier

4.2.3 Evaluation qualitative

Nous avons confié le développement de JavAne à une équipe de quatre étudiants en maîtrise d'informatique (dans le cadre d'un TER -Travaux d'Étude et de Recherche-) après quelques heures de formation aux concepts d'agents mobiles, au modèle d'acteur et à la bibliothèque JavAct.

L'expérience montre bien les avantages des agents mobiles dans ce contexte où, par nature, il n'y a pas de centralisation possible de l'information et des méta-données qui sont fortement instables et volatiles. Ici, la recherche s'adapte dynamiquement à l'état du système (liens entre les places, contenu) et de la recherche (résultats trouvés) sans interaction avec le client et par le biais d'agents aux droits limités (donc *a priori* inoffensifs). En complément, la sélection personnalisée sur le site serveur limite le volume du trafic sur le réseau aux seules données sélectionnées. Sur le plan du développement, deux avantages majeurs se sont dégagés :

- Le *framework* est relativement simple à appréhender (les étudiants l'ont confirmé). La conception des protocoles de recherche d'information et de téléchargement à base d'agents mobiles est naturelle.
- En cachant les problèmes de répartition, de synchronisation et de communication, JavAct simplifie le développement. Le passage au codage n'apporte pas de difficulté, ce qui augmente la fiabilité du logiciel et le volume de code écrit a été relativement faible (voir les codes sources proposés plus haut).

Bien entendu, nos objectifs ne sont pas de proposer des algorithmes de recherche ou d'exploitation sophistiqués ou nouveaux, mais de proposer des outils permettant de les implémenter. Nos exemples nous semblent montrer la faisabilité de l'implémentation de composants et d'algorithmes très variés.

4.2.4 Eléments de solution des exemples du chapitre 1

Nous proposons de montrer dans cette section les principaux éléments architecturaux qui permettraient leur conception à partir de nos propositions. En particulier, nous identifions les rôles des principaux composants spécialisés et nous essayons de montrer comment réutiliser certains d'entre eux.

Les micro-composants présentés au chapitre 3 qui permettent l'adaptation des agents à leur contexte d'exécution peuvent être réutilisés dans tous ces exemples : adaptation des protocoles de localisation, gestion du mode déconnecté, sécurité des communication...

Mutualisation de ressources

Les principaux composants ont été détaillés dans la section 4.2. Un prototype a été commencé par un groupe d'étudiants de niveau maîtrise (master 1^{ere} année) dans le cadre d'un travail d'étude et de recherche (TER) (`http://noman.flabelline.com/ter-2003.html`), puis a été amélioré par des étudiants de 3^{eme} année de l'école d'ingénieur ENSEEIHT. Le fonctionnement du logiciel est repris en annexe B.2.

Dans ce prototype (figure B.1), nous avons expérimenté divers protocoles de localisation (inondation d'agents mobiles, parcours préconçus, parcours dynamiques...), qui constituent autant de composants de *Recherche globale*. Pour la *Recherche locale*, seul un composant de fouille d'une base de données de fichiers a été mis en œuvre. Nous aurions pu développer des composants de recherche dans un système de fichier (similaire à la commande GNU `locate` par exemple), etc.

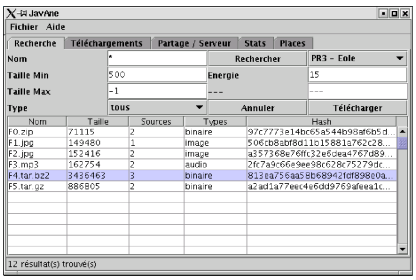


FIG. 4.8 – Prototype de mutualisation

Pour l'*Accès aux ressources*, dans le cas de notre base de données nous avons mis en place un service spécifique (recherche d'une entrée, lecture...) ; dans le cas d'un composant destiné à l'accès au système de fichier, nous pouvons utiliser l'API de Java(accès fichiers) qui fournit le niveau d'abstraction nécessaire.
Enfin, pour l'*Exploitation*, nous avons réalisé un composant de téléchargement de fichier présenté plus haut (section 4.2.2), ainsi qu'un composant qui retransmet au client les données brutes fournies par le composant de recherche locale (utilisable pour retourner un résultat issu d'une simple recherche de données ou d'une invocation de service par exemple).

Distribution automatique de logiciel

La distribution automatique de logiciels s'appuie sur des mécanismes de recherche sur le réseau et des opérations effectuées sur des sites distants du site initiateur. Le tableau 4.1 identifie le rôle des différents composants.

	Scénario *push*	**Scénario *pull***
Recherche Globale	Localisation des utilisateurs connus	Localisation de distributeurs ou de producteurs pour un composant donné
Recherche Locale	Recherche et vérification de la version d'un module	Recherche d'un composant logiciel
Accès aux ressources	Directement via le système de fichiers	Base de données, Web Service...
Exploitation	Déploiement : copie physique et configuration du composant à déployer	Téléchargement d'un composant, et notification à l'agent de RG des dépendances (composants requis)

TAB. 4.1 – Spécialisation des composants du *framework*

Nous avons effectivement implémenté un scénario de type *push* dans le cadre d'un exemple de distribution d'une mise à jour de sécurité : pour cela, nous avons simplement réutilisé tous les composants réalisés dans le prototype de mutualisation de ressources à

l'exception du composant d'exploitation, dont la nouvelle instance permet d'appliquer un correctif à des fichiers (opération du type `patch` sous Linux/Unix etc.).

Rendu d'animations 3D

La conception d'un prototype a débuté dans le cadre d'un projet étudiant récent. Le sujet initial est disponible sur http://noman.flabelline.com/ter-2006.html, l'objectif étant de produire une version de démonstration illustrant certaines de nos propositions (déploiement, supervision, et adaptation dynamique du système P2P réalisé par des agents mobiles), tout en restant ludique et intéressant pour les étudiants. Nous avons proposé de déployer un moteur de rendu de scènes numérique et les données nécessaires à ce rendu selon le mode P2P, puis de superviser le rendu par les agents en place.

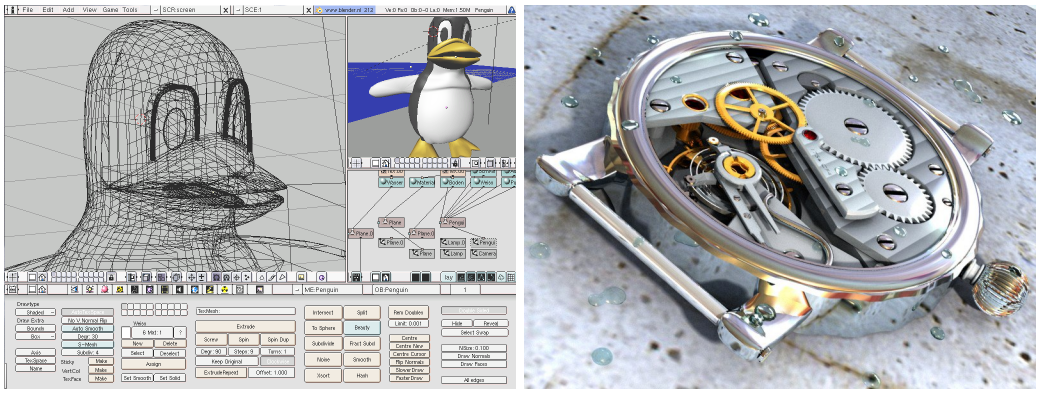


FIG. 4.9 – Blender + YafRay

Les composants de *Recherche globale* peuvent être réutilisés pour explorer le réseau à la découverte de pairs acceptant de participer au rendu réparti. Côté *Recherche locale*, il s'agit cette fois de vérifier les capacités de calcul (occupation de la machine, bande passante, puissance...) ainsi que de tester l'existence du moteur de rendu (version correcte, bon fonctionnement...). Le composant d'*Accès aux ressources* le plus simple (API Java pour l'accès fichier) peut être réutilisé. Enfin, côté *Exploitation*, il faut mettre en place un composant de supervision du rendu, sur le modèle du composant d'*Exploitation* du prototype de mutualisation de ressources, capable d'adapter les tranches de calcul aux variations de capacité de la machine.

Services dynamiques pour les véhicules

Dans l'exemple proposé, les véhicules proches sont capables de communiquer via des réseaux *ad-hoc* (en supposant l'existence de technologies réseau adéquates), ainsi que de manière intermittente avec des bornes d'accès localisées en bordure des voies. On peut distinguer les services exploitables en deux catégories (tableau 4.2) : ceux qui concernent le pilotage du véhicule et ceux qui concernent l'utilisateur.

	Services *pilotage*	**Services *utilisateur***
Recherche Globale	Recherche et découverte de véhicules à proximité	Recherche et découverte de services accessibles (météo, trafic, infos...)
Recherche Locale	Vérifications (sécurité, informations accessibles...)	Recherche d'une information demandée par l'utilisateur parmi les services découverts (restaurant à proximité...)
Accès aux ressources	Abstraction des données du véhicule (vitesse, direction, position...)	Idem exemples précédents (Base de données, Web Service...)
Exploitation	Transmission au client des données, partage des informations découvertes...	Transmission / affichage d'informations à l'utilisateur

TAB. 4.2 – Spécialisation des composants du *framework*

Dossier médical personnalisé

L'accès à travers différentes formes de réseaux, la gestion du mode déconnecté, les étapes de recherche d'information (dossier médical du patient, archives des personnels, références externes dans les centres universitaires, chez les fournisseurs d'équipements ou de médicaments...) pourraient être traitées par les solutions évoquées précédemment.

L'exploitation a posteriori de résultats d'examens nécessite la migration d'un agent vers l'appareil d'examen qui contient les données et ses spécifications techniques. Là encore, les composants développés pour le prototype de mutualisation de ressources pourraient être réutilisés.

4.3 Travaux connexes

L'un des premiers travaux utilisant les agents mobiles (Aglets) pour la découverte de ressources (matérielles ou logicielles) sur des réseaux P2P est décrit dans [Dun01]. L'implémentation met en évidence le besoin de méta-données sur chaque pair, ainsi que d'un agent pour accéder à chaque ressource. Dans [MMKA04], les auteurs utilisent des agents mobiles (Kodama) pour rechercher de l'information répartie dans des réseaux P2P. Leur *framework*, ACP2P, permet le maintien d'historiques au sein de chaque agent et l'échange d'historiques entre agents. Ainsi, les agents peuvent acquérir la connaissance de nouvelles sources d'information. Par contre, seuls les documents reliés de manière connexe peuvent être retrouvés.

M3 -MultiMedia Database Mobile agents- [KDB01] est un système de recherche de données multimédia par le contenu qui repose sur les agents mobiles, Java (et JDBC) et CORBA. Les données sont des images ou des vidéos. Un agent mobile peut se déplacer de site en site et y extraire l'information au moyen d'un code spécifique. L'agent mobile peut mémoriser les informations recueillies sur un site, les utiliser sur les sites visités ensuite, les

faire évoluer pendant le parcours. Il y a donc une véritable exploitation de la notion d'agent mobile ici (déplacement du solveur sur le site où se trouvent le données, avec évolution de l'état). Les problèmes de sécurité sont pris en compte via des mécanismes de sessions indépendantes, les mécanismes de sécurité de CORBA et des restrictions de droits. Une évaluation comparative des performances montre, d'une part, l'intérêt de la spécialisation pour retrouver l'information recherchée (et éviter l'intervention de l'utilisateur), d'autre part, une réduction significative des temps de réponses.

SoftwareDock [HHvdHW97] est un prototype universitaire pour le déploiement et la configuration de logiciels dans des environnements répartis. Il permet de gérer intégralement le cycle de vie du déploiement d'un logiciel (installation, activation, mise à jour...), de manière complètement décentralisée. L'architecture de SoftwareDock contient deux parties principales : le *Release Dock* est un serveur placé du côté du producteur de logiciel, contenant les composants applicatifs et pilotant leur déploiement. Chaque site client héberge un *Field Dock*, un serveur qui permet au client de connaître les différentes évolutions possibles (mises à jour, nouveaux logiciels...) et de fournir au *Release Dock* les informations de configuration du système. Les interactions entre ces éléments ainsi que la mise en œuvre de toutes les phases du cycle de vie du déploiement est réalisée au moyen d'agents mobiles. Ceux-ci transportent les composants logiciels sur le réseau, et interprètent leurs descripteurs de déploiement pour réaliser les opérations nécessaires.

I. Satoh [Sat04] propose de migrer des composants Java de type *JavaBean*™ au moyen d'agents mobiles, en leur permettant de conserver leurs liens de connaissance conformément à certaines politiques de déplacement. La mobilité d'agent ne sert que pour la migration physique et les communications inter-composants sont réalisées au niveau de la plateforme. Il ne semble pas possible de découvrir ou d'utiliser dynamiquement des composants qui n'auraient pas été initialement connus.

A notre connaissance, l'association des trois technologies agent, composant et P2P n'a pas encore été proposée ni étudiée.

4.4 Conclusion

Les agents mobiles peuvent contribuer avantageusement au développement de systèmes distribués à grande échelle. Nous avons montré que les capacités d'adaptation individuelle des agents permettent de faire évoluer le système afin de répondre aux besoins d'adaptation rencontrés lors de la recherche qu'ils soient :

- opérationnels : décentralisation des serveurs de données, hétérogénéité matérielle (architectures et réseaux), hétérogénéité logicielle (systèmes d'information), sécurité et qualité de service des communications, volatilité et volume des données...
- fonctionnels : spécialisation des algorithmes à la fois au niveau distribué (parcours de la grille et découverte des différents serveurs) et au niveau local (mécanismes de communication avec chaque serveur, exploitation des ressources...).

L'utilisation de notre patron de conception à base de composants déployés par des agents mobiles adaptables peut contribuer à la maîtrise des coûts de développement et de maintenance, tout en réduisant les temps de réponse aux requêtes par la limitation des transferts de données sur le réseau. Au delà des exemples proposés, on peut envisager son utilisation pour exploiter effectivement le stockage distribué à grande échelle (données multimédia, données astronomiques, données d'observation de la terre, données financières ...) ou dans le cadre d'autres systèmes de recherche d'information comme la recherche d'information coopérative, la fouille de données réparties [PK02], les systèmes d'information pilotés par les fournisseurs de l'information[8]... En outre, on peut noter que les composants de recherche globale peuvent se déployer et se redéployer dynamiquement de manière autonome sans contrôle explicite du client ou des sites serveurs. Par ailleurs, l'architecture du système supporte simplement la maintenance des serveurs répartis : l'adoption *in situ* de composants fournis par le serveur permet d'adapter automatiquement le système (client) aux évolutions imprévues et incontrôlées des serveurs hébergeant des ressources.

La simplicité apportée par le modèle réside essentiellement dans la séparation des aspects opérationnels et fonctionnels et dans le changement de comportement. Les apports majeurs concernent la facilité offerte pour mettre en œuvre l'adaptation dynamique de la recherche aux conditions d'exécution et aux résultats, ainsi que l'extensibilité.

D'une certaine manière, on peut considérer que notre modèle d'agent mobile adaptable avec ses mécanismes de déploiement constitue une sorte de *modèle de composant* adapté aux environnements d'exécution dynamiques instables et répartis.

Toutefois, un inconvénient de ce modèle réside dans la rigidité de l'architecture d'agent mobile adaptable que nous proposons, qui contient un ensemble figé de micro-composants, qui ne sont pas systématiquement nécessaires lors de l'exécution de certains agents. Nous proposons dans le chapitre suivant une réponse à ce problème.

[8] en mode *push*, au contraire du mode *pull* où les opérations sont initiées par le client

Résumé des contributions :

- Nous proposons d'utiliser des agents mobiles pour simplifier la phase de localisation des ressources dans les systèmes P2P, en particulier dans les systèmes P2P purs. Nous proposons également d'utiliser les agents mobiles pour personnaliser la phase d'exploitation des ressources, par déploiement (transport, mise en production) d'un comportement adéquat fourni par le client [LAP04].
- Nous proposons d'employer des agents mobiles adaptables comme support de déploiement pour les comportements (composants fonctionnels) : d'une part la mobilité d'agent permet un transport adaptatif des composants fonctionnels sur le réseau, d'autre part le méta-niveau joue le rôle de conteneur pour le composant fonctionnel, en lui fournissant les services nécessaires à son exécution et permettant sa configuration (choix des micro-composants) ainsi que sa reconfiguration dynamique [ALPre].
- Nous proposons un patron de conception pour aider à la conception de systèmes répartis à grande échelle en mode P2P, à base de composants et d'agents mobiles adaptables [LA05].
- Nous avons réalisé un *framework* pour mettre en œuvre ce patron de conception [ALP04], ainsi qu'un prototype de système P2P qui implémente ces concepts [ALP03].

Chapitre 5

Un modèle d'agent flexible - micro architecture

Dans ce chapitre, nous identifions un besoin de flexibilité supplémentaire dans l'architecture d'agent adaptable présentée au chapitre 3 : l'adaptation de l'architecture elle-même. Notre objectif est alors de permettre la conception d'un modèle d'agent adapté au besoin applicatif. Notre approche consiste à définir une architecture d'agent par assemblage de types ou d'instances de micro-composants non-fonctionnels. Nous présentons le prototype d'environnement de modélisation qui en résulte, ainsi que quelques architectures d'agent produites. Nous comparons cette approche à certains travaux similaires.

Les agents mobiles sont par nature, des outils d'adaptation en contexte réparti. Afin d'accroître les possibilités d'adaptation, nous avons proposé un niveau de flexibilité intra-agent contrôlé par l'agent lui-même et nous avons défini un modèle d'agent mobile configurable et auto-adaptable dynamiquement. Dans ce chapitre, nous proposons un niveau supplémentaire d'adaptation et nous introduisons un modèle d'agent flexible.

5.1 Besoins d'adaptation supplémentaires

Le modèle d'architecture d'agent mobile que nous avons présenté au chapitre 3 permet la configuration des agents par un choix de micro-composants lors de leut création, ainsi que leur adaptation dynamique au contexte d'exécution par le remplacement de certains de ces micro-composants. Toutefois, la structure et le squelette de l'architecture de l'agent sont figés et les adaptations sont limités à un ensemble de micro-composants dépendant de ce choix d'architecture. Dans notre implémentation, il s'agit d'un ensemble de micro-composants réifiant les services non-fonctionnels nécessaires à l'exécution d'un acteur mobile.

5.1.1 Vers davantage de flexibilité

Prenons l'exemple de notre prototype de logiciel pair à pair cité précédemment. Il est implanté par un système d'agents répartis. Certains agents sont des agents d'interface (avec les pairs clients ou serveurs). Certains agents sont mobiles sur le réseau et peuvent être amenés à percevoir leur environnement physique (charge du réseau, niveau de sécurité...) ; pour s'y adapter, ils peuvent redéfinir leurs mécanismes de communication ou de déplacement. Certains agents peuvent développer une algorithmique complexe (recherche de ressource, exploitation d'une ressource...). En termes d'évolution, de décision, de perception, de réactivité et de mobilité, les différents agents qui constituent le système ont des caractéristiques et des besoins variables. Or, dans notre implantation, tous ont du être développés à partir du modèle d'agent unique offert par le *middleware*.

Plus précisément, dans la proposition décrite au chapitre 3 et mise en œuvre dans le *middleware* JavAct, l'adaptation se réduit au choix initial des micro-composants et au remplacement dynamique d'un micro-composant par un autre qui offre le même service (spécifié par une interface Java). L'évolution des services opératoires est limitée par une interface fixe (limitation de l'évolution au changement d'implémentation) et il n'est pas possible de décider quels micro-composants sont remplaçables et lesquels ne le sont pas. L'architecture de l'agent est également figée : les agents sont des *acteurs* [Hew77, Agh86] répartis et mobiles et il n'est possible ni d'introduire un nouveau micro-composant ni de retirer un micro-composant inutile (par exemple, le gestionnaire de déplacement pour un agent immobile).

D'autre part, on peut aussi noter que notre expérience de la maintenance du *middleware* JavAct nous a conduit à réviser plusieurs fois l'architecture des agents et à modifier le *middleware*. Par exemple :

- lors de l'introduction de la mobilité, il a fallu ajouter au méta-niveau un micro-composant dédié (il interagit avec d'autres composants de méta-niveau) et donc modifier l'architecture des agents ;
- pour mettre en œuvre un mécanisme de communication synchrone, nous avons dû modifier les interfaces existantes, en l'occurrence celles des micro-composants de communication pour éviter d'intégrer un micro-composant supplémentaire ;
- nos agents (modèle d'acteur) se sont révélés mal adaptés pour implanter des agents AMAS [CGGG03] (lors d'une collaboration avec une équipe du domaine des sytèmes multi-agent) et une nouvelle architecture d'agent a dû être définie [Déj03].

Si les expérimentations effectuées ont permis de vérifier l'intérêt de l'architecture proposée, celle-ci reste donc limitée en terme de flexibilité : elle n'est pas minimale (si un agent n'effectue pas certaines opérations, par exemple changement de comportement ou déplacement, tous les micro-composants lui sont quand même associés) et ne permet pas de modeler et de personnaliser la structure des agents. Par exemple, il n'est pas possible de définir un agent immobile ou un type quelconque d'agent doté de mécanismes particuliers. Pourtant, il semble que si le modèle et l'architecture d'agent sont ajustés au besoin, l'adaptation dynamique est simplifiée et l'implémentation est plus efficace (performances d'exécution en temps et en occupation mémoire).

5.1.2 Différents modèles d'agents

Afin de complèter ce besoin de flexibilité de l'architecture, nous avons étudié quelques modèles d'agent, en s'intéressant à leurs implémentations et plus précisément à leurs architectures. Dans une démarche de classification, on peut être amené à distinguer les agents suivant différentes capacités (dans la littérature des systèmes multi-agents, on trouve fréquemment le terme de *composantes*) indépendantes et orthogonales, à caractère individuel ou social :

- l'autonomie, déclinée diversement dans différents modèles d'agent et leur implantation, qui induit les notions de cycle de vie et d'activité ;
- le savoir et le savoir-faire (éventuellement offert sous forme de services) ;
- une capacité de décision ;
- l'évolutivité (apprentissage...) ;
- la possibilité d'engendrer de nouveaux agents ;
- la communication ;
- l'interaction avec l'environnement (perception...) dans lequel l'agent est situé (par exemple, un agent mobile est situé géographiquement sur un réseau de machines et interagit avec lui *via* un ensemble des services) ;
- la réactivité[1] ;
- la mobilité ;
- ...

	Com. asynchrone	Création	Com. synchrone	Mobilité	Perception	Décision	Cycle de vie	...
Acteur	*	*					*	
Agent logiciel	*	*	*				*	
Agent mobile	*	*		*			*	
Agent déploiement	*			*			*	
Agent conteneur	*	*	*	*			*	
Agent réactif		*			*	*	*	
Agent BDI	*	*	*		*	*	*	
Agent AMAS	*	*	*		*	*	*	
...								

FIG. 5.1 – Quelques modèles d'agents et leurs capacités

La notion de cycle de vie se retrouve dans tous les modèles sous des formes différentes (cf. figure 5.1) et la plupart des agents ont besoin de moyens de communication respectant certains standards (invocations de services distribués, KQML, FIPA-ACL...). Les différentes propriétés d'un agent définissent le modèle : par exemple, les agents BDI [RG95a],

[1] Un agent est réactif s'il répond de manière opportune aux changements de son environnement issus de stimuli externes. Il n'a pas besoin d'une représentation symbolique élevée de la perception de son environnement.

les agents AMAS, les agents de déploiement [HHvdHW97], les agents conteneurs[2], etc.

La mise en œuvre de ces différents modèles est en général liée à une plateforme spécifique qui permet de simplifier la conception et le développement d'une application agent. On peut citer par exemple AgentBuilder[3] pour le modèle BDI ou encore Jade[4] pour concevoir des agents suivant le standard FIPA[5]. De nombreuses autres plateformes de développement de systèmes multi-agents peuvent être trouvées sur le site suivant : http://www.agentlink.org.

Au sein d'une application, on peut trouver des modèles d'agents différents, y compris des modèles hybrides possédant des composantes issues de différents modèles. On peut s'interroger sur l'intérêt qu'il y aurait (pour faciliter le développement) à pouvoir définir des modèles d'agents « à la carte ». Nous pensons que certains agents pourraient bénéficier de composantes issues d'autres modèles, pour obtenir par exemple des agents BDI mobiles ou des acteurs sans changement de comportement.

L'ambition de notre démarche n'est pas de proposer un modèle et une architecture tellement générique qu'elle permettrait de réaliser ces différents modèles d'agents, cela serait probablement utopique et sûrement moins pertinent que d'employer les plateformes adéquates pour chaque modèle. Nous expérimentons simplement des architectures suffisamment flexibles pour réutiliser des capacités spécifiques à certains modèles d'agents et permettre la conception de modèles adéquats.

5.1.3 Du modèle d'agent au méta-modèle

Nous proposons d'étendre les idées et principes architecturaux énoncés au chapitre 3, en relâchant les contraintes (modèle d'acteur, modèle dynamiquement adaptable, mobilité) et en améliorant les points faibles, en particulier sur la sûreté de l'assemblage des micro-composant.

Au lieu de proposer un unique modèle d'agent, nous essayons d'en abstraire les caractéristiques principales et de travailler au niveau supérieur. Ainsi, en s'inspirant de la vision *ingéniérie des modèles*[6], on peut considérer que le modèle d'agent mobile adaptable est au niveau 1 et permet d'engendrer des agents de niveau 0. Pour générer des modèles d'agents différents, nous proposons une sorte de méta-modèle d'agent (niveau 2), dans lequel un ensemble de règles défini ce que nous nommons un *style* d'agent (figure 5.2). L'interface graphique que nous proposons peut être vue comme un langage de modélisation.

Ainsi, l'un de nos objectifs est de vérifier si les principes proposés précédemment que nous reprenons sous le terme de *style d'architecture* (organisation en étoile autour d'un connecteur, séparation réflexive des codes fonctionnels au niveau de base et des codes non-fonctionnels au méta-niveau, utilisation de composants à grain fin pour réifier les services non-fonctionnels au méta-niveau) sont toujours applicables, en allant vers plus de flexibilité. L'idée est de permettre la définition de modèles d'agents par le biais de la construction d'une architecture conforme au style d'architecture du chapitre 3.

[2] Cf. chapitre précédent
[3] http://www.agentbuilder.com
[4] http://jade.cselt.it
[5] http://www.fipa.org
[6] http://www.omg.org/mof

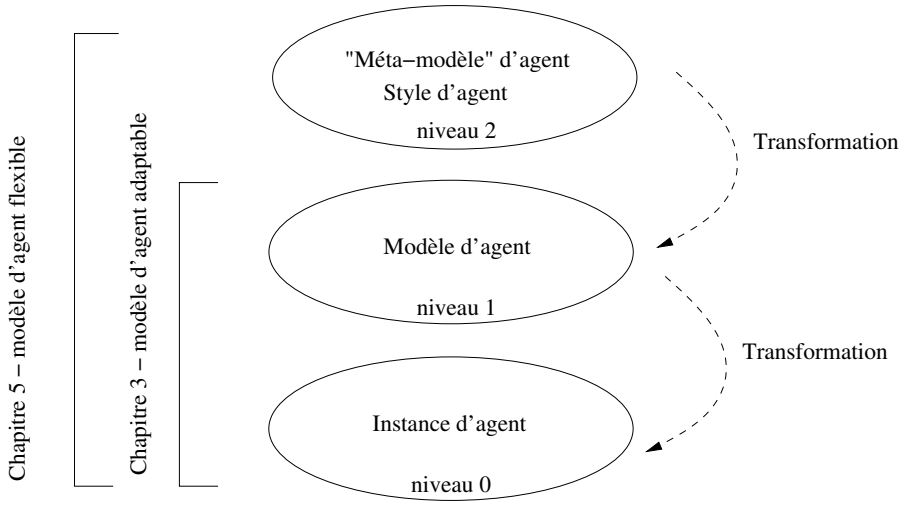


FIG. 5.2 – Style d'agent flexible

5.1.4 Besoin de validation

Dans l'approche de l'architecture figée précédente, il était possible de valider différentes propriétés soit par typage vérifié lors de la compilation soit lors de l'exécution par des tests. Ces vérifications concernent à la fois le niveau structurel (fonctionnalités nécessaires, micro-composants adéquats) ainsi que le niveau sémantique (cohérence de l'assemblage de micro-composants). Dans la nouvelle approche, les possibilités de modèles d'agent sont infinies. La seule abstraction disponible pour raisonner est le style d'architecture dont les caractéristiques ne sont pas suffisantes (trop abstraites) pour obtenir des propriétés fortes.

Il serait pourtant intéressant de pouvoir valider les assemblages de composants (dépendances...), ainsi que certaines propriétés comportementales (terminaison, sûreté, vivacité...). Par exemple, après avoir défini un nouveau modèle d'agent il faudra s'assurer qu'un agent réagira à une perception issue d'un composant d'interaction. Ainsi, il semble nécessaire de proposer des outils de validation qui pourraient être utilisés de manière systématique lors de la définition d'un nouveau modèle d'agent.

5.1.5 Vers un environnement de développement dédié

La flexibilité de l'architecture doit permettre non seulement le remplacement dynamique de composants, mais aussi la définition de structures d'agent personnalisées (éventuellement l'ajout de nouveaux composants à une structure existante et symétriquement le retrait). Ainsi, le concepteur doit pouvoir :

- définir un modèle d'agent adapté au besoin applicatif,
- réaliser le modèle d'agent, par assemblage de composants logiciels de grain fin (réutilisés, spécialisés ou nouveaux) qui constituent l'ensemble de ses mécanismes non-fonctionnels,
- valider l'assemblage de micro-composants (cohérence des types, résolution des dépendances...) ainsi que des propriétés comportementales,

– générer un squelette de code de l'architecture choisie pour simplifier et rendre plus fiable l'étape d'implémentation dans laquelle il n'aura plus qu'à intégrer les codes fonctionnels,
– et enfin, programmer la reconfiguration des agents en cours d'exécution en faisant évoluer leurs capacités pour leur permettre de s'adapter dynamiquement à leur environnement d'exécution.

La dernière étape n'est pas du même niveau, l'adaptation recherchée se déroulant dans une phase d'exécution de l'agent. C'est simplement une préoccupation qui ne doit pas être oubliée, car nous cherchons à produire des agents dynamiquement adaptables.

5.2 Architecture flexible d'agent logiciel

Pour mettre en œuvre les propositions précédentes, nous devons pouvoir :
– décrire des architectures d'agents, donc utiliser un langage de description d'architecture (ADL) ;
– valider l'architecture, donc utiliser un formalisme de vérification basé sur sa description ;
– générer du code et l'optimiser, donc utiliser un mécanisme de génération de code.

Il existe un certain nombre d'outils répondant à ces besoins [Med96]. Toutefois, le temps d'investissement nécessaire à l'acquisition de ces outils nous a semblé supérieur au développement d'une solution *ad hoc*, sachant que notre objectif est d'obtenir un simple prototype permettant de valider nos propositions. Malgré tout, l'intégration de ces outils reste envisageable pour former une solution plus robuste et plus complète.

La figure 5.3 donne une vision globale du processus présenté dans ce mémoire et constituant une première étape, destinée à valider par prototypage les concepts proposés.

5.2.1 Conception d'une architecture d'agent

Le concepteur d'une application agent peut soit réutiliser un modèle et une architecture existante, soit réaliser un nouvel assemblage. La réutilisation d'une architecture offre un gain de productivité intéressant puisque cela permet de se dispenser d'une phase de validation (réalisée antérieurement). L'architecture choisie peut ensuite être spécialisée pour former un modèle d'agent adéquat. Cette spécialisation se fait par ajout ou retrait (sous réserve d'une nouvelle validation dans ce cas) :
– de micro-composants qui seront intégrés par défaut dans l'architecture de chaque nouvel agent ;
– ou bien de types de micro-composants. Ce qui permet de retarder le choix au moment de la création d'une instance d'agent (pour une configuration adaptée au contexte d'exécution par exemple).

Pour faciliter la réutilisation et le choix des micro-composants, ceux-ci doivent disposer de méta-données décrivant au minimum leur usage en langage naturel, ainsi que de manière plus formelle leur type, les services offerts, les services requis et éventuellement d'autres propriétés utiles. L'assemblage des micro-composants peut se faire de manière graphique si l'on souhaite simplifier et accélérer cette phase de conception.

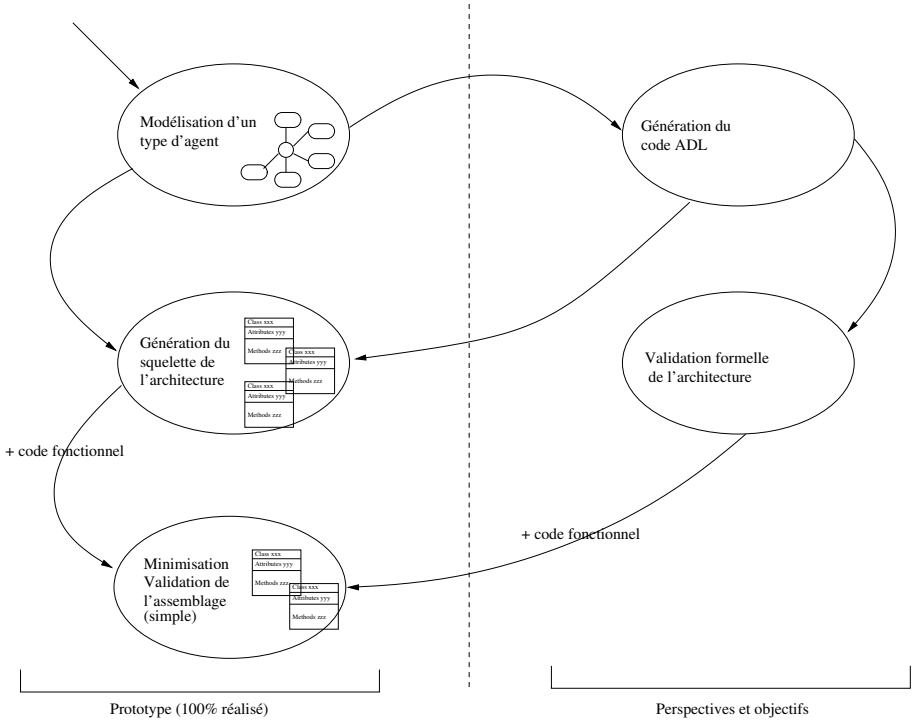


FIG. 5.3 – Processus de développement

5.2.2 Principes architecturaux

Différents niveaux de micro-composants

Dans l'architecture d'agent mobile adaptable présentée précédemment, tous les micro-composants n'ont pas le même statut dans le méta-niveau. Ceux qui offrent leurs services au niveau de base exportent leurs primitives d'accès dans un composant qui fait le lien entre les deux niveaux (classe `QuasiBehavior`, cf. l'exemple de l'envoi de message page 63). Mais certains micro-composants n'offrent des services qu'au niveau méta (par exemple les primitives d'accès au cycle de vie d'un acteur ne doivent pas être accessible depuis le code fonctionnel), et l'accès depuis le niveau de base ne doit pas être possible. D'autres servent de point d'entrée (micro-composant de réception de message et composant de perception en général) dans l'agent, et dans le cas général, cela n'a pas de sens d'invoquer leurs services depuis l'intérieur de l'agent. Ainsi nous distinguons 3 catégories de micro-composants et des propriétés d'accessibilité pour ces différents cas :

- *Micro-composants de service pour le niveau de base* : accès depuis le niveau de base (envoi de messages, mobilité...),
- *Micro-composant de service pour le méta-niveau* : accès limité au méta-niveau (cycle de vie, analyseur...),
- *Micro-composant d'interaction* : accès limité à l'extérieur de l'agent (réception de messages, perception de l'environnement...).

Ces distinctions permettent de renforcer la sûreté et la sécurité de l'architecture produite. En effet, cela permet de définir une politique de contrôle d'accès garantie par la

construction : le code fonctionnel, au niveau de base, n'a pas de visibilité sur le méta-niveau. Par exemple, on ne peut pas perturber l'exécution du thread de l'agent, géré par un micro-composant de service pour le méta-niveau. De même, l'invocation des services d'interaction n'est pas possible pour aucun des micro-composants de l'agent. Ces distinctions n'étaient pas nécessaires dans la version précédente de l'architecture d'agent adaptable, car les mécanismes de protection avaient été intégrés de manière *ad-hoc*.

Style d'architecture

Un agent est composé d'un ensemble de micro-composants distincts, connectés à un connecteur unique.

Il n'est pas possible d'intégrer deux micro-composants de même type, car ils offriraient les mêmes services via des méthodes identiques (par exemple deux composants d'envoi de messages). Au delà du conflit créé à la compilation, il serait anormal d'offrir des services identiques dupliqués sans moyen de les différencier. Ainsi, un concepteur qui aurait besoin de re-définir une architecture d'agent nécessitant plusieurs micro-composants identiques peut toutefois passer par l'utilisation d'un micro-composant composite, agrégeant un ensemble de micro-composants identiques et offrant à l'agent un service spécialisé. Par exemple, un agent possédant deux boîtes aux lettres de messages pourra utiliser un composant composite encapsulant deux micro-composants de boîte aux lettres standards, en offrant des méthodes d'accès différenciées (par ex. `put(Messsage m, int identBAL)` et `Message get(int identBAL)`). On peut toujours définir des méthodes par défaut pour cacher cette duplicité au niveau de base par exemple.

Il n'y a pas de nombre minimal de micro-composants. Toutefois l'absence d'un composant de cycle de vie risque de limiter l'intérêt d'un tel agent. Ce niveau de vérification (plutôt sémantique) pourra être effectué lors des étapes de validations formelles de l'architecture, par la vérification de propriétés adéquates (vivacité, etc.). Concrètement, un agent doit disposer d'un composant de perception et d'un cycle de vie pour permettre un fonctionnement minimal. Il n'y a pas de limite en nombre de micro-composants.

L'interface entre le niveau fonctionnel et les services offerts par les micro-composants est réalisée par un objet spécifique (`QuasiBehavior` dans la figure 3.4), afin de réaliser la séparation des niveaux de services proposée plus haut et de renforcer l'expressivité du code fonctionnel. Tous les services offerts au niveau de base sont mis à disposition du code fonctionnel par cette classe intermédiaire.

Adaptation dynamique de l'architecture

Dans la proposition du chapitre 3, la capacité d'adaptation dynamique de l'architecture est une caractéristique intrinsèque. Nous proposons de réifier la capacité d'adaptation, en considérant l'adaptation dynamique au même titre que les autres caractéristiques non-fonctionnelles (mobilité...). Ainsi, il est possible de générer des architectures d'agent qui ne soient pas dynamiquement adaptables (donc avec des performances d'exécution légèrement supérieures et une meilleure sûreté de fonctionnement par exemple). Pour rendre un agent adaptable, il faudra intégrer des micro-composants qui gèrent les mécanismes d'adaptation

dynamique. En général, un seul composant d'adaptation ne suffira pas : il faudra prévoir un cycle de vie capable de prendre en compte les besoins et les demandes d'adaptation, de manière similaire à la proposition précédente.

Minimisation

Lors de l'étape de choix du modèle d'agent, il est possible que l'architecture sélectionnée ne soit pas optimale, c'est à dire que certains micro-composants non-fonctionnels aient pu être sélectionnés par le concepteur mais non utilisés en pratique. Cela peut se produire soit dans le cas d'un mauvais choix d'architecture, soit dans le cas où le concepteur est parti d'un type générique prédéfini d'agent qui s'avère un peu trop complet[7]. Par exemple, si le concepteur a réutilisé l'architecture d'acteur mobile pour implémenter un algorithme de calcul parallèle qui n'exploite pas la mobilité, la minimisation permet de suggérer le retrait du micro-composant de mobilité.

Pour cela, nous proposons de dériver du code fonctionnel de l'agent un type appelé *type réduit* (ou type d'implantation), qui correspond à un ensemble de types de composants effectivement utiles (ou encore aux mécanismes strictement nécessaires à l'exécution). A partir du type réduit ainsi identifié, il sera possible d'obtenir une architecture d'agent *ad hoc*, minimale en terme de nombre de composants.

Cette minimalité de l'architecture a des avantages en terme d'efficacité à l'exécution (d'autant plus si les agents se déplacent sur le réseau), mais aussi en termes de sûreté et de sécurité car la vérification et la validation des assemblages (dépendances...) sont facilitées. Par ailleurs, il n'est pas nécessaire d'expliciter les services de méta-niveau (mécanismes d'exécution) requis par le code fonctionnel puisqu'ils peuvent être directement déduits du code.

Dans le cas le plus particulier d'un agent qui pourrait changer de comportement fonctionnel (cas des acteurs), le type réduit pourrait être représenté par un diagramme états-transitions (cf. figure 5.4) : les transitions correspondraient aux changements de comportement et chaque état représenterait le type réduit pour un comportement.

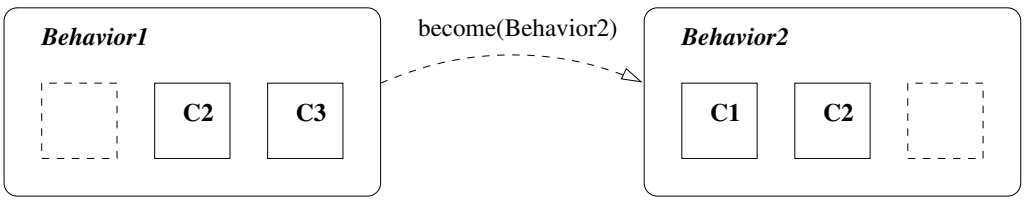


FIG. 5.4 – Type réduit d'agent, représenté par un diagramme état/transition

[7]Précisons que dans le cas inverse (pour un micro-composant manquant), l'erreur sera détectée lors de l'intégration du code fonctionnel, via une erreur explicite de compilation dans notre prototype.

5.2.3 Problèmes ouverts

Le problème de la « mutation » des agents, *i.e.* le changement dynamique de modèle, reste ouvert. Au delà d'un besoin qui reste à être démontré, nous pensons qu'il serait préférable de ne pas autoriser la mutation si on veut obtenir des systèmes fiables et vérifiables a priori (on pourrait cependant autoriser les mutations prévues lors de la conception).

L'adoption dynamique par un agent d'un micro-composant d'un type non prévu ne nous semble pas justifiée (un agent pourrait découvrir par lui-même la possibilité de « se bonifier » en changeant ses mécanismes opératoires mais, à notre sens, cela ne pourrait résulter que de capacités d'introspection beaucoup plus élevées que celles que nous envisageons aujourd'hui). En effet, pour utiliser à bon escient un composant découvert dynamiquement, il faudrait que l'agent puisse comprendre la sémantique d'utilisation de ce composant, puis de raisonner sur l'apport de ce composant dans son architecture avant de pouvoir l'y ajouter.

Néanmoins, il est envisageable qu'un agent puisse acquérir dynamiquement un comportement fonctionnel nouveau (dans le cas d'une adaptation fonctionnelle) qu'il aura, par exemple, reçu par message (évolution par changement de comportement dans le cas des acteurs de K. Hewitt et G. Agha). Pour que ce comportement soit effectivement exécutable, il faut que les micro-composants opératoires nécessaires soient disponibles au méta-niveau ; s'ils ne le sont pas, il faut reconfigurer le méta-niveau à partir de nouveaux micro-composants et redéfinir ainsi dynamiquement le modèle d'agent. On peut alors envisager l'intégration dynamique de micro-composants standards (création d'agents, communication, mobilité...).

5.3 Agent$^{\varphi}$

Le modèle que nous avons présenté a été entièrement réalisé sous la forme d'un prototype d'environnement de développement pour les architectures d'agents flexibles, appelé Agent$^{\varphi}$. Celui-ci se compose d'une bibliothèque de micro-composants non-fonctionnels prêts à l'emploi, d'un modeleur graphique permettant l'assemblage des micro-composants autour du contrôleur, d'un générateur d'architecture pour le langage Java (squelette de code) et d'un outil de minimisation de l'architecture basé sur le *framework* d'analyse de bytecode ASM.

L'architecture d'agent générée est exportée sous la forme d'une archive Java (Jar) autonome, qui peut être directement utilisée dans le *plugin* JavAct présenté dans le chapitre 3, et détaillé en annexe B.3. Ainsi, la conception et le développement sont facilités par l'utilisation d'outils graphiques intuitifs et simples d'emploi.

5.3.1 Bibliothèque de composants et de styles d'agents prédéfinis

Les micro-composants disponibles sont organisés de manière hiérarchiques, dans une bibliothèque (le système de fichier dans notre implémentation). A la racine on trouve pour chaque type de micro-composant[8] son interface Java ainsi que le descripteur (fichier de

[8]Dans nos propositions, un type de micro-composant est l'ensemble des services fournis, représenté par une interface Java standard.

méta-données) du type. Pour chaque type de micro-composant, un répertoire contient les différentes implémentations et chaque répertoire contient un ensemble de classes java complémentaires ainsi qu'un descripteur pour chaque implémentation. Ce descripteur est un fichier de méta-données similaire à celui de la figure 5.5 (l'attribut `Level` vaudra `Interface` par contre).

```
Name: MoveRMICt
Level: Base
Licence: LGPL
Provider: IRIT-SL
Description: Allows an agent to move itself on an other place via RMI/create
Requires: CreateCt.CreateRMICt, BecomeCtI, SendCtI
Provides: public void go(String place)
```

FIG. 5.5 – Méta-données d'un type d'agent

Par exemple, la figure 5.5 représente les méta-données du micro-composant de mobilité (`MoveRMICt`). Le mot-clé `Requires` indique les dépendances (services requis) de ce composant. Dans notre exemple, le micro-composant de mobilité requiert le micro-composant `CreateRMICt`, un micro-composant de type `BecomeCtI` et un micro-composant de type `SendCtI`. Le fait de spécifier un type de micro-composant dans la ligne des services requis permet de laisser le choix au concepteur d'architecture d'une implémentation particulière.

La discrimination du niveau du composant (base, méta-niveau ou interaction) permettra d'engendrer uniquement les méthodes qui doivent être accessibles pour chaque niveau (les primitives fournies par un micro-composant de niveau `Base` seront visibles au niveau de base, etc.)

5.3.2 Modélisation d'un agent

La construction d'une architecture d'agent se fait par assemblage de micro-composants issus de la bibliothèque, de manière graphique, conformément au style d'architecture décrit précédemment. Chaque ensemble de micro-composant ainsi constitué peut être sauvegardé sous la forme d'un modèle d'agent qui pourra être réutilisé (figure 5.6).

```
#Généré par AgentPhi Designer
#Sun Aug 13 17:47:37 CEST 2006
ComponentList=LifeCycleCt.ActorLifeCycleCt, MailBoxCt.MailBoxCt,
              ReceiveCt.ReceiveLocalCt
Name=Degraded-Local-Actor
Licence=GPL
Description=Simplest actor architecture, which can only react to incoming messages.
Provider=IRIT-SL
```

FIG. 5.6 – Méta-données associées à un modèle d'agent (acteur dégradé)

L'interface graphique (IHM) permet de charger un modèle d'agent prédéfini, ce qui est équivalent à l'intégration de son ensemble de micro-composants dans une architecture vierge. Dans la figure 5.7, l'IHM montre un modèle d'agent minimal, constitué de trois micro-composants. Il est possible d'obtenir le descripteur de chacun en cliquant sur leur représentation. Les traits en pointillés représentent les dépendances existantes entre les

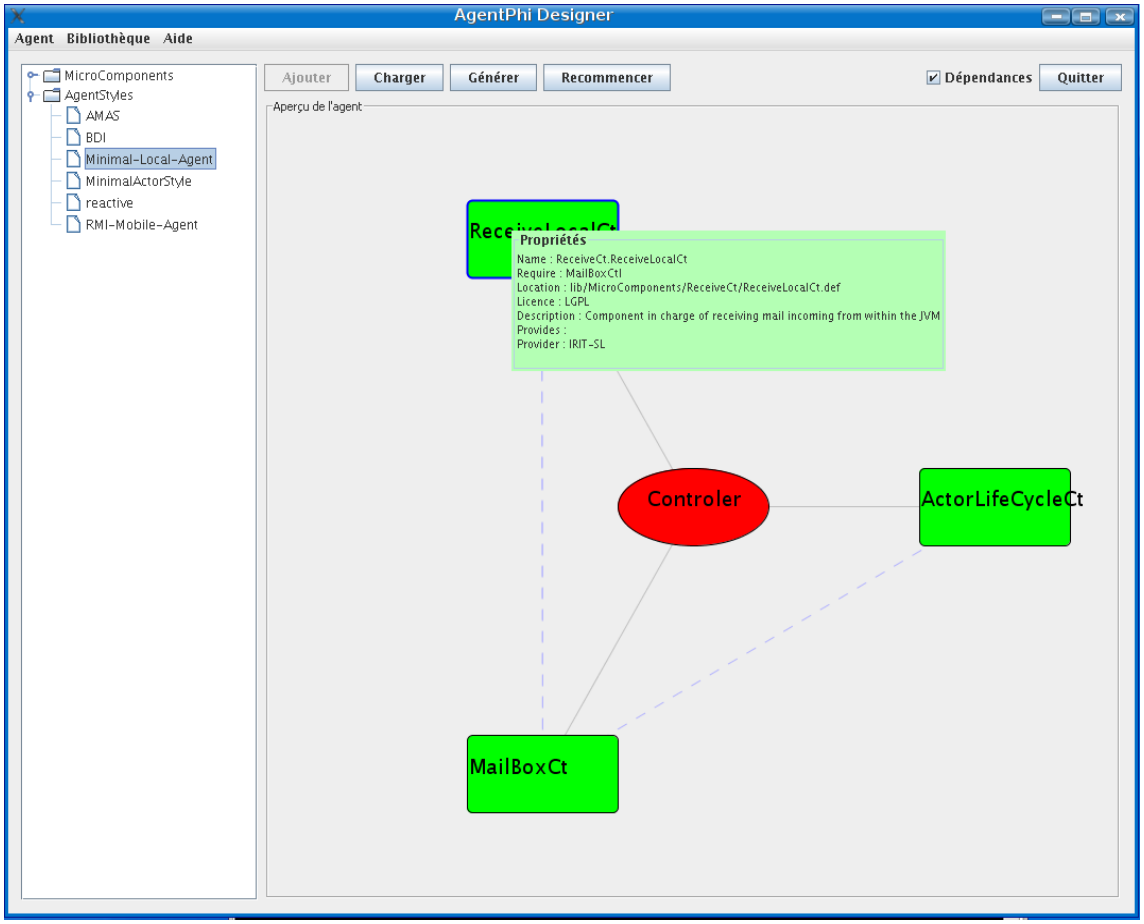


FIG. 5.7 – Affichage d'un modèle d'agent prédéfini

micro-composants (un symbole particulier permet de repérer immédiatement ceux dont les dépendances ne sont pas satisfaites) et les traits pleins la liaison avec le connecteur (`Controler`).

L'ajout et le retrait se font également de manière graphique (figure 5.8), en sélectionnant les micro-composants dans la bibliothèque dont l'arborescence est visible sur la partie gauche du prototype.

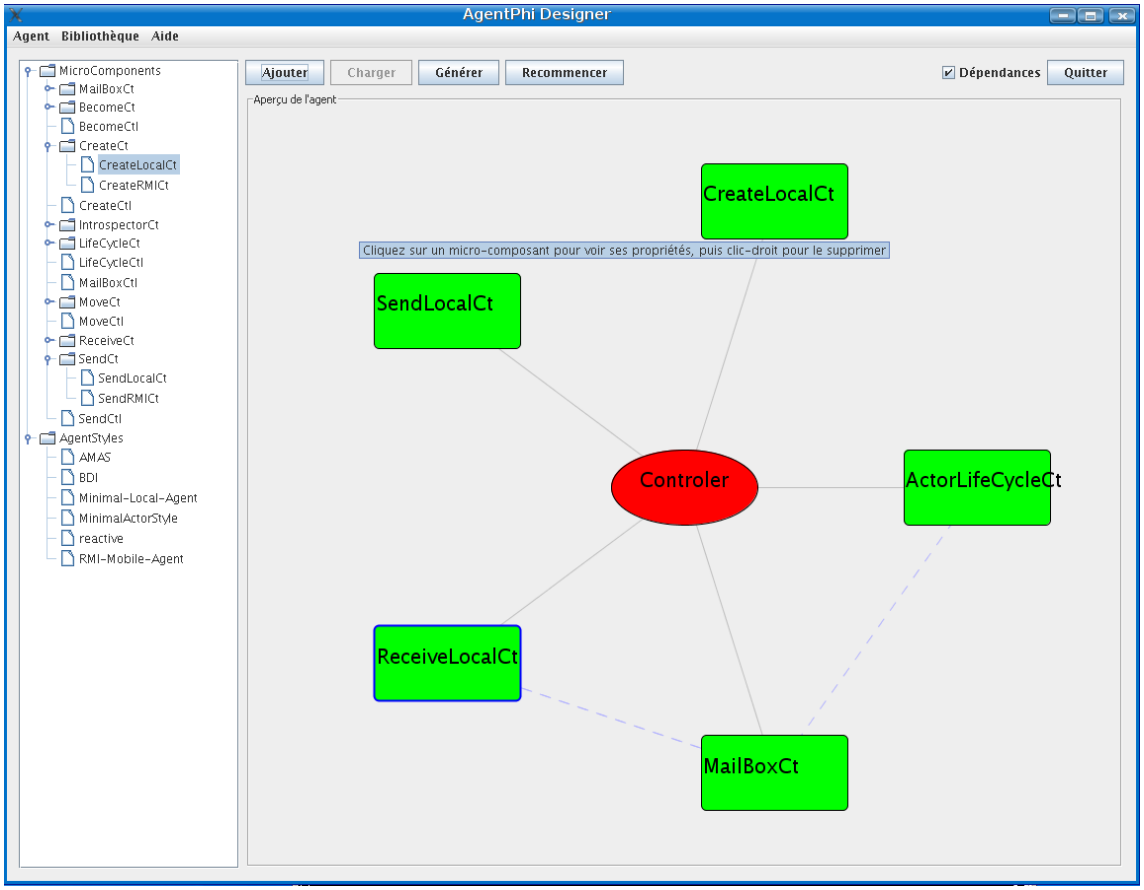


Fig. 5.8 – Ajout de micro-composants

5.3.3 Génération du squelette d'architecture

Un paquetage spécifique permet de regrouper les différents générateurs de squelettes (en prévision de l'export du modèle d'agent vers un ADL qui permettra des vérifications poussées de l'architecture). Dans notre prototype, seule la génération vers du code Java est implémentée, et nous décrivons dans cette partie quelques éléments de la génération.

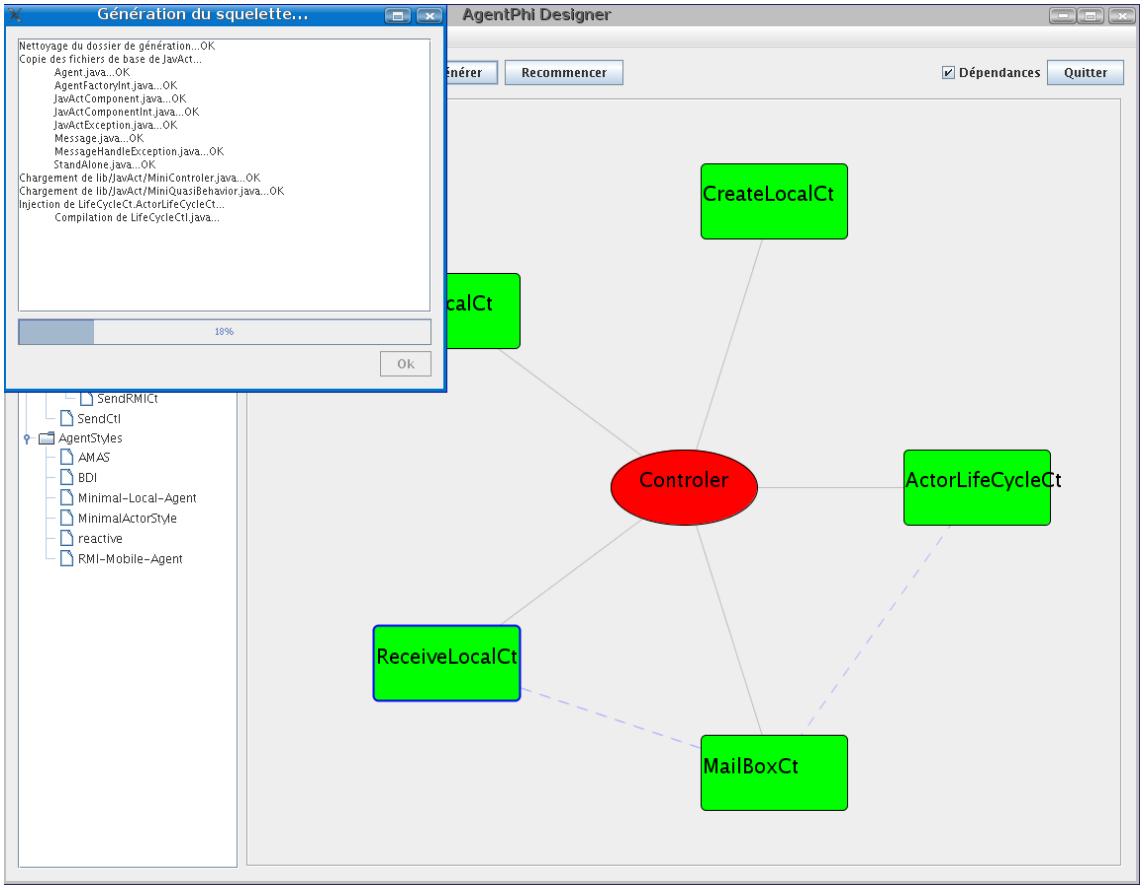


FIG. 5.9 – Démarrage de la génération du squelette

La génération se fait depuis l'interface graphique par un simple clic sur le bouton `Générer` (cf. figure 5.9). Une petite fenêtre permet de suivre la progression des différentes étapes (insertions de code, compilations...) et en cas d'erreur (généralement lors d'une compilation) de faciliter la résolution du problème.

La génération du squelette est basée sur un enrichissement progressif d'une architecture minimale. Celle-ci est composée d'un ensemble de classes Java, principalement des interfaces ou des classes abstraites. Deux fichiers particuliers (`MiniControler.java` et `MiniQuasiBehavior.java`) contiennent des repères (*tags*) qui permettent d'insérer du code source à des endroits spécifiques (cf. figure 5.10). Par exemple, les attributs permettant de connecter les micro-composants seront insérés au niveau du tag `%%ATTRIBUTES%%`. Ainsi, pour un micro-composant de boîte aux lettres (`MailBoxCt`, le code permettant d'associer le micro-composant au contrôleur sera une déclaration d'attribut : `private myMailBoxCt ;`

Ces fichiers seront renommés dans l'architecture pour former les classes `Controler`, le connecteur central de l'architecture d'agent, et `QuasiBehavior` qui joue le rôle d'une interface de services du méta-niveau pour les codes fonctionnels, de manière similaire à notre proposition d'architecture d'agent mobile adaptable.

```
import java.io.Serializable;
import java.util.Vector;
//%%IMPORTS%%

public final class Controler implements Serializable {

  private transient Agent myAgent = null;
  private QuasiBehavior myBehavior;
//%%ATTRIBUTES%%

  protected Controler(QuasiBehavior beh) {
    init(beh);
  }

//%%CONSTRUCTORS

  private void init(QuasiBehavior beh) {
    myBehavior=beh;
//%%CPTNEWC%%
//%%CPTSETCTRL%%
//%%INIT%%
    beh.setMyControler(this);
//%%POSTINIT%%
  }

  protected Agent getMyAgent() {
    return myAgent;
  }

  protected QuasiBehavior getMyBehavior() {
    return myBehavior;
  }

  protected void setMyBehavior(QuasiBehavior beh) {
    myBehavior=beh;
    if (myBehavior != null) myBehavior.setMyControler(this);
  }

//%%METHODS%%
}
```

FIG. 5.10 – Code de la classe `Controler` minimale (`MiniControler.java`)

Les codes nécessaires à la connexion et à la manipulation des micro-composants sont insérés sous forme de code source Java, via des méthodes d'insertion dédiées (figure 5.11).

```
//insertion des codes d'initialisation du contrôleur (interface et composants)
//ex. code généré : "private SendCtI mySendCt;\n"
insereCtrlCode("\tprivate "+nomInterface+" my"+nomAttribut+";\n","%%ATTRIBUTES%%");
```

FIG. 5.11 – Principe d'insertion de code

Pour chaque micro-composant de l'architecture, le générateur tente une compilation de l'architecture *en cours*, ce qui permet de détecter assez tôt des erreurs dans le code des micro-composants. Enfin, une phase de compilation globale permet d'engendrer le bytecode de l'ensemble des classes nécessaires au fonctionnement d'un agent Ces codes sont ensuite regroupés dans une archive Jar, qui servira de support au développement et à l'exécution des codes fonctionnels.

Cas particuliers

Le traitement générique effectué lors de la génération automatique du code de la classe `Controler` ne permet pas de prendre en compte tous les cas particuliers. Par exemple, dans le composant de gestion de la boîte aux lettres (`MailBoxCt`) que nous avons développé, les méthodes offertes au méta-niveau sont synchronisées pour verrouiller les accès concurrents à l'objet contenant les messages. Pour résoudre cette difficulté, nous avons laissé la possibilité au programmeur de micro-composant de spécifier des portions de code particulières pour contrôler la génération du `Controler`. Pour des raisons de commodité ces codes sont décrits dans les codes sources des micro-composants particuliers, dans des sections délimitées par les mots-clés tels `%%METHODS%%` ou `%%ATTRIBUTES%%` (figure 5.12).

```
//%%METHODS%%
  public synchronized Message getMessage() {
    while (myMailBoxCt.isEmpty())
      try { wait(); } catch (Exception e) {}
    return myMailBoxCt.getMessage();
  }

  public synchronized void putMessage(Message m) {
      myMailBoxCt.putMessage(m);
      notify();
  }
//%%END%%
```

FIG. 5.12 – Exemple de code spécifique : méthodes d'accès `synchronized` (extrait du composant `MailCt`)

Système d'accueil

Comme pour les agents mobiles adaptables, les agents engendrés à partir de ce système doivent s'exécuter dans un environnement spécifique, qui leur fournit un ensemble de ressources nécessaires à leur autonomie. Dans notre prototype, le système d'accueil est

une machine virtuelle Java qui héberge une fabrique d'agent (d'où le nom `AgentFactory`). La fabrique exporte des méthodes statiques pour la création d'agent et l'envoi de message depuis un programme non agent (particulièrement utile dans un programme de lancement d'une application agent). Une version spécialisée pour l'utilisation en réseau via le *middleware* RMI (`RMIAgentFactory`) exporte également des méthodes distantes pour la création d'agents à distance. Sur ce modèle, il est possible d'engendrer d'autres fabriques adaptées à des besoins spécifiques (par héritage par exemple).

5.3.4 Adaptation dynamique

Dans la section précédente, nous avons proposé d'intégrer la propriété d'adaptation dynamique via des micro-composants spécifiques. Notre prototype utilise un micro-composant capable d'introspection (d'où son nom : `Introspector`) qui permet de découvrir lors de l'exécution l'ensemble des micro-composants instanciés ainsi que les services qu'ils offrent, afin de manipuler ensuite l'architecture pour l'adapter. Par exemple, le cycle de vie d'un acteur adaptable appelle une méthode de remplacement de micro-composants après chaque traitement de message, la méthode `changeToNext` du micro-composant responsable de l'adaptation (code de la figure 5.13).

```
protected void changeToNext() {
  myControler.setMyBehavior(myControler.getMyNextBehavior());

  //changement des micro-comp.
  String[] cpts=myControler.getComponentsName();
  Object[] params=new Object[1];
  Object[] emptyparam=new Object[0];
  for (int i=0; i<cpts.length; i++)
    try {
      params[0]=myControler.invokeJavAct("getMyNext"+cpts[i],emptyparam);
      myControler.invokeJavAct("setMy"+cpts[i],params);
    } catch (Exception e) {}
}
```

FIG. 5.13 – Extrait du micro-composant d'adaptation

Le fonctionnement de ce service est le suivant : récupération de l'ensemble des micro-composants de l'architecture en utilisant les capacités du micro-composant `Introspector`, puis invocation (via l'API réflexive de JAVA) des méthodes qui permettent d'échanger ces micro-composants (récupération de l'instance du prochain micro-composant à utiliser, puis installation de ce micro-composant).

L'efficacité de ce composant est liée aux performances de l'API réflexive. Dans les implémentations actuelles (Java 1.5), les temps de réponses sont assez importants (mais les adaptations sont peu fréquentes). Une solution pour éviter la dégradation des performances consisterait en la génération du code correspondant à ces appels réflexifs. Cela est possible lors de la phase de génération, puisque tous les types de composants sont connus, donc les méthodes à appeler aussi.

5.3.5 Minimisation par rétro-ingéniérie

Pour calculer le type réel de l'agent, c'est à dire l'ensemble des micro-composants qu'il requiert lors de son exécution, nous proposons d'extraire les informations nécessaires par rétro-ingéniérie, en analysant le bytecode des codes fonctionnels. Cette étape ne peut pas être réalisée plus en amont, car il est nécessaire d'engendrer le squelette de l'agent avant de pouvoir écrire correctement les parties fonctionnelles. Pour cela, nous utilisons le *framework* ASM[9], qui permet de visiter le code compilé de classes Java. Le choix de cet environnement est lié d'une part à ses performances d'exécution (vitesse) et à la faible taille de son code (comparé à BCEL[10] ou SERP[11]), d'autre part c'est un projet *open source* du consortium ObjectWeb[12], ce qui est un gage de qualité et de pérénité.

Pour l'analyse statique du bytecode, ASM propose une approche similaire au patron de conception `Visitor`. Pour chaque élément d'une classe (signature, champs, constructeurs, méthodes...), le *framework* permet d'intervenir sur l'élément visité (figure 5.14). On peut ainsi détecter toutes les invocations de méthodes dans le code fonctionnel et regrouper les appels de méthodes dans la super classe `QuasiBehavior`, qui contient les primitives non-fonctionnelles des micro-composants du méta-niveau. On peut alors en déduire les micro-composants utiles à l'exécution de l'agent, ainsi que ceux qui ne le sont pas et proposer une modification du type d'agent pour obtenir un type réduit.

```
class TestClassVisitor implements ClassVisitor {

  public void visit(int access, String name, String superName, String[] interfaces,
                    String sourceFile) {
    System.out.println("Visiting class "+name+"...");
  }

  public CodeVisitor visitMethod(int access, String name, String desc,
                    String[] exceptions, Attribute attrs) {
    if (name.startsWith("<init>")) return null; //constructeur
    System.out.println("  -> Method : "+name);
    return new TestCodeVisitor();
  }

        ...

}
```

FIG. 5.14 – Extrait de la classe d'analyse qui implémente `ClassVisitor`

Dans le cas d'un agent qui change de comportement en cours d'exécution, cette étape doit être itérée pour chaque comportement, en constituant un graphe de types minimaux comme présenté en 5.2.2. Pour éviter les reconfigurations d'architectures en cours d'exécution d'un comportement (ce qui est plus efficace et plus fiable du point de vue de la validation de l'assemblage), et selon les arguments qui nous incitent à interdire les mutations d'agents (les changements dynamiques de types, cf. 5.2.3), le type réduit est constitué de l'union des types de micro-composants calculés sur le graphe.

[9] http://asm.objectweb.org
[10] http://jakarta.apache.org/bcel
[11] http://serp.sourceforge.net
[12] http://consortium.objectweb.org, créé en 1999 par Bull, France Télécom R&D et l'INRIA

5.4 Quelques exemples d'agents

Dans cette section, nous présentons quelques types d'agents qui peuvent être construits à partir de nos propositions. Nous montrons avant tout la démarche de conception, en justifiant le choix des micro-composants, au travers d'exemples concrets.

5.4.1 Agent-acteur mobile adaptable

Le modèle d'agent présenté au chapitre 3 a été validé à de nombreuses reprises (prototypes, enseignements et projets étudiants principalement), il est donc important de pouvoir engendrer une architecture d'agent mobile adaptable équivalente à cette proposition. Ainsi, nous reprenons les micro-composants définis pour le modèle d'acteur, en ajoutant la mobilité et les micro-composants d'adaptation. Le résultat est présenté dans la figure 5.15. L'architecture obtenue n'est pas strictement identique, puisque nous avons réifié l'adaptation (micro-composant `IntrospectorCt`). Toutefois, elle est équivalente au sens de la sémantique d'exécution. Sans ce micro-composant, on retrouve le modèle d'acteur mobile (non adaptable) initial.

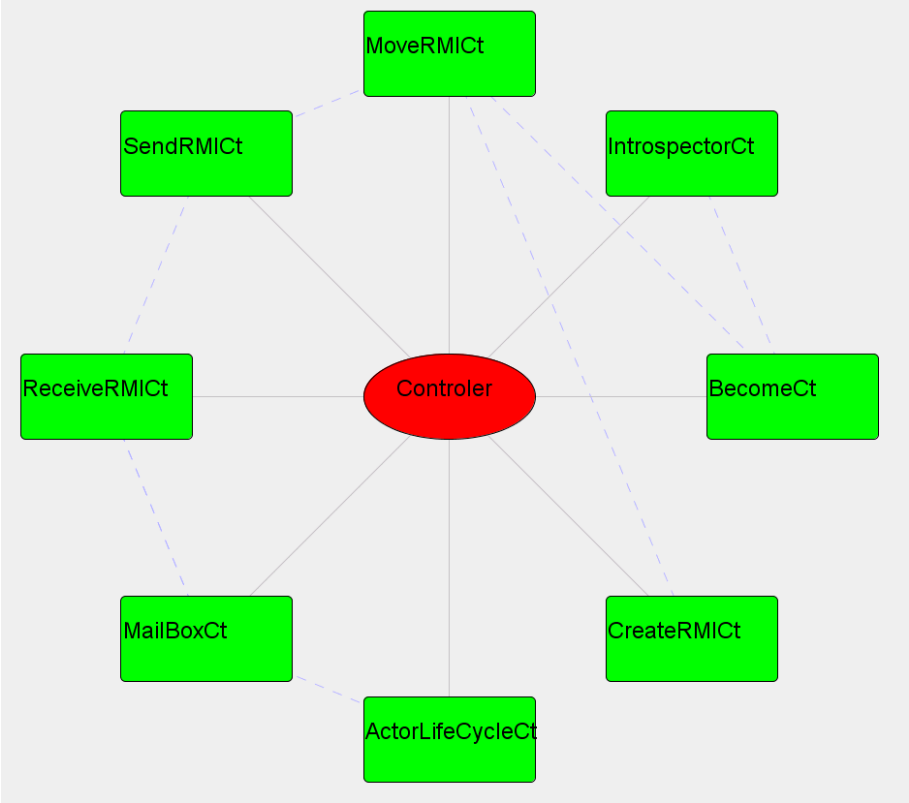


Fig. 5.15 – Exemple d'architecture d'agent mobile adaptable

L'architecture engendrée par notre prototype bénéficie des apports de sûreté discutés dans ce chapitre, en restant compatible avec la bibliothèque JavAct, sans aucun surcoût à l'exécution (cf. 5.6).

5.4.2 Agent minimal, exécution locale

Une des plus simples architectures est présentée dans la figure 5.16. Elle contient un cycle de vie d'acteur (non adaptable) et un micro-composant de réception de messages locaux (envoyés par un agent du même système d'accueil). Le rectangle jaune (contour en pointillés) signifie que l'architecture contient une indication de type de micro-composant. Dans cet exemple, cela signifie que le programmeur de l'agent devra instancier ce modèle avec un micro-composant de son choix mais du type `MailBoxCtI`.

Les agents créés à partir de cette architecture peuvent être de simples agents de trace, ou d'interaction avec l'utilisateur (affichage de messages).

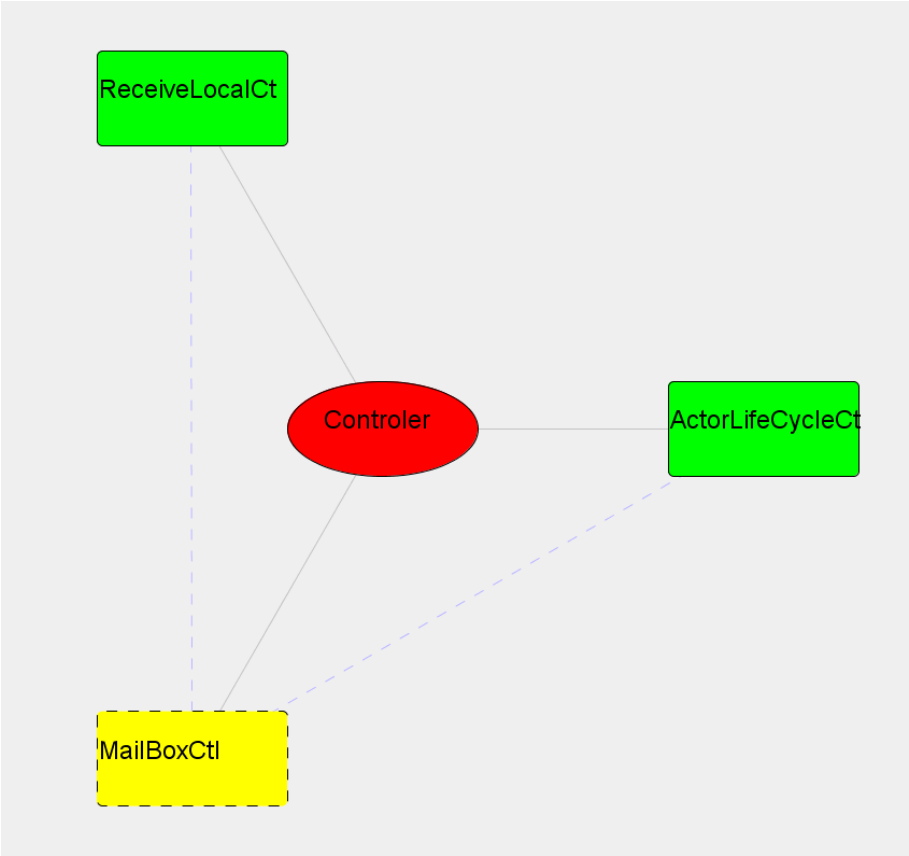


Fig. 5.16 – Exemple d'architecture d'agent minimal

5.4.3 Un agent embarqué

Dans la section 1.2.3, nous avons présenté un scénario de service pour les véhicules. Nous proposons ici une démarche pour construire une architecture d'agent plus adaptée à la mise en œuvre de cet exemple que le modèle utilisé dans la solution proposée au 4.2.4. Bien entendu, il s'agit d'une démarche de conception dans laquelle nous faisons plusieurs hypothèses, l'application grandeur nature n'existant pas réellement (mais l'architecture de l'agent a été réellement expérimentée).

Dans chaque véhicule, un agent peut recevoir des messages provenant d'autres agents, ainsi que depuis les points d'accès qui bordent les voies de circulation. Il est probable que dans un souci d'interopérabilité, ces bornes soient conçues pour dialoguer avec leurs utilisateurs via des protocoles usuels, par exemple basés sur des standards de fait type XML-RPC[13] ou de vrais standards type SOAP[14]. Pour des raisons d'efficacité et de performances, les agents communiqueront entre eux avec une technologie plus proche de leur implémentation, par exemple RMI pour des agents Java. Ainsi, deux micro-composants de réception distinct sont nécessaires pour implémenter ces agents. Les messages seront ensuite dispatchés dans la même boîte aux lettres, le micro-composant standard peut être utilisé à cet effet.

Nous avons proposé de permettre les mises à jour du code fonctionnel (initiées par un constructeur de véhicules par exemple), il faut donc intégrer un micro-composant de changement de comportement. De manière similaire, un constructeur pourrait proposer la mise à jour du code d'un composant non-fonctionnel, ce qui nécessite de mettre en place les micro-composants d'adaptation (composant d'adaptation dynamique et probablement cycle de vie adapté). Ce cas est différent du besoin plus classique d'adaptation dynamique à une variation du contexte d'exécution, vu qu'ici les agents ne sont pas prévus pour bouger et que leur environnement est probablement figé.

Enfin, pour émettre des messages à d'autres agents, il faut ajouter un micro-composant de communication compatible avec celui de réception, ici RMI.

En réutilisant les micro-composants de nos agents basés sur le modèle d'acteur, on obtient une architecture représentée dans la figure 5.17.

5.4.4 Eléments pour la mise en œuvre d'agents intelligents

Nous décrivons ici comment nous pourrions réaliser une architecture de type BDI avec notre proposition, puis comment l'étendre afin d'y ajouter, par exemple, des capacités de mobilité et d'adaptation dynamique. Les algorithmes et décompositions des fonctionnalités des agents BDI sont tirés en partie du livre *« An Introduction to MultiAgent Systems »* de Michael Wooldridge [Woo02].

Un cycle de vie d'agent BDI est donné dans la figure 5.18, qu'il est envisageable de traduire sous la forme d'un micro-composant spécifique, dans la catégorie des cycles de vie. Les opérations de révision des croyances, de processus de décision, de planification... peuvent également être codées sous la forme de micro-composants. Ce type de décomposition est proposé par de nombreuses architectures d'agents, soit sous forme d'objets, soit sous forme de composants pour augmenter la réutilisation de ces unités. Les autres intérêts à implémenter ces fonctionnalités sous la forme de micro-composants sont liés aux mécanismes de vérification des assemblages, lors de leurs réutilisations futures.

Ces micro-composants peuvent ensuite être assemblés selon notre style d'agent, auquel on peut ensuite ajouter suivant les besoins des micro-composants apportant des capacités supplémentaires telles que communication, mobilité... Pour permettre l'adaptation

[13] http://ws.apache.org/xmlrpc
[14] http://www.w3.org/TR/soap12-part1

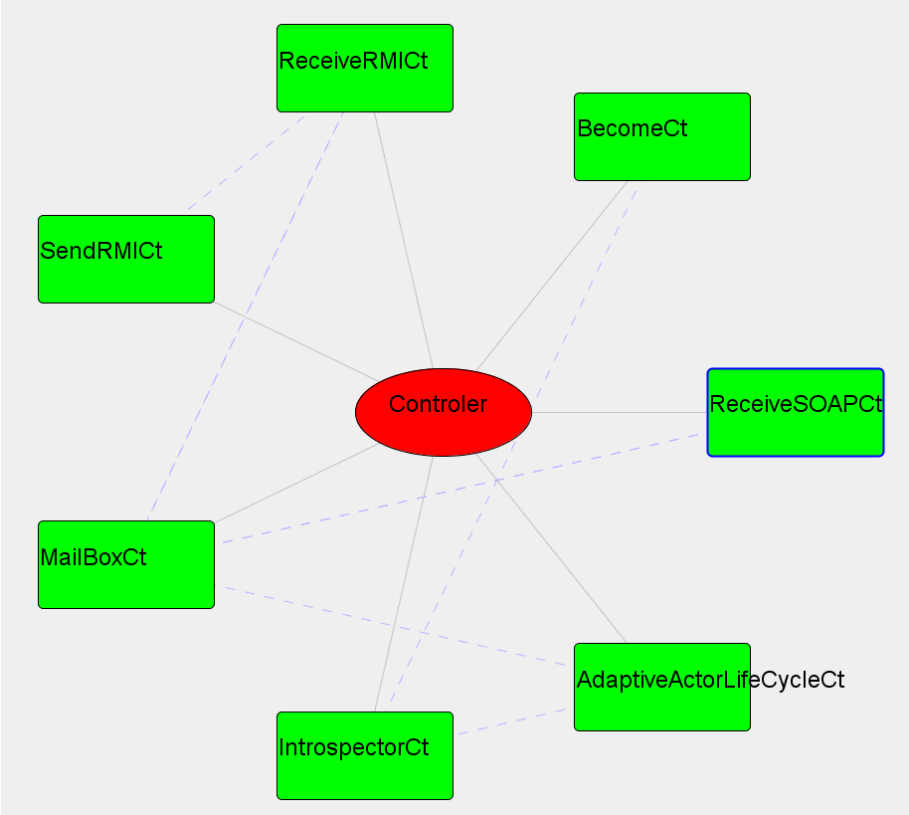


FIG. 5.17 – Exemple d'architecture d'agent de service pour les véhicules

```
Soient B0, D0 et I0 les croyances, désirs et intentions initiales de l'agent.

Algorithme de contrôle d'agent BDI

1       B = B0
2       D = D0
3       I = I0
4       répéter
            4.1     obtenir nouvelles perceptions p
            4.2     B = révision_croyances(B, p)
            4.3     I = processus_décision_options(D, I)
            4.4     D = processus_décision(B, D ,I)
            4.5     I = raisonnement_analyse(B, D, I)
            4.6     PE = planification(B, I)
            4.7     exécuter(PE)
        jusqu'à ce que l'agent soit arrêté
fin
```

FIG. 5.18 – Cycle de vie d'un agent BDI

dynamique comme nous l'avons décrite dans le chapitre 3, il est nécessaire de modifier l'algorithme de la figure 5.18 pour donner la main au composant d'évolution après l'exécution du plan (donc en 4.8), afin qu'il reconfigure l'architecture si besoin. Bien sûr, il faut également ajouter le micro-composant d'adaptation dynamique dans l'architecture de l'agent. Ainsi, il est possible de concevoir des architectures hybrides d'agents intelligents mobiles et dynamiquement adaptables, qui pourraient, de manière semblable à notre modèle d'agent

mobile adaptable, se révéler plus souples et plus sûres dans des environnements fortement répartis et instables.

La réutilisation des différents micro-composants est également justifiée par la diversité des cycles de vies d'agents BDI, comme par exemple dans les modèles d'*agent BDI avec obligation limitée* ou d'*agent BDI avec obligation ouverte* [Woo02].

5.5 Travaux connexes

Nous avons comparé nos travaux à certaines plateformes de développement agent utilisant des composants comme MAGIQUE, MAST, COMET, MALEVA ou MaDcAr. Certaines utilisent des techniques architecturales proches de nos propositions, toutefois l'objectif est assez différent. En effet, ces plateformes utilisent principalement les composants pour réaliser des activités fonctionnelles, c'est à dire pour implanter des mécanismes comportementaux. Dans MALEVA [Lhu98] par exemple, un agent est vu comme un ensemble de composants en interaction (implémentant des composantes d'agents réactifs), qui forment un composant composite. Les préoccupations telles la gestion de l'hétérogénéité ou l'adaptation à l'environnement ne sont pas prises en compte dans MALEVA.

Notre approche vise à permettre l'adaptation du fonctionnement des agents, c'est à dire de leurs activités non-fonctionnelles, en particulier pour leur permettre de s'adapter à leur environnement d'exécution. Notre approche se démarque également par son orientation génie logiciel, nous essayons de proposer des mécanismes simples et sûrs qui permettent d'augmenter la qualité et la réutilisabilité de l'architecture d'agent produite. De plus, nous proposons d'intégrer la mobilité d'agent comme un service non-fonctionnel quelconque, alors que la mobilité est souvent une caractéristique intrinsèque d'une plateforme (présente ou absente). Enfin, le niveau de granularité de l'adaptation (très fin) est une spécificité de nos travaux.

Nous présentons ci-dessous quelques travaux les plus proches de nos propositions.

5.5.1 MAGIQUE

Le projet MAGIQUE[15] (*m*ulti-*ag*ent hiérarch*ique*) [RMS01] propose de distinguer dans un agent sa *coquille* de ses compétences, c'est à dire ses mécanismes d'exécution de son savoir-faire. Une compétence peut aller de la capacité à réaliser un calcul élémentaire jusqu'à la résolution de gros problèmes de planification ou d'optimisation combinatoire. Ainsi, pour ses auteurs, un agent intelligent n'est qu'un cas particulier d'agent auquel on a donné des compétences faisant intervenir des techniques d'intelligence artificielle (réseaux de neurones, systèmes experts, logique temporelle, etc.).

Du point de vue de l'implémentation, une compétence peut être perçue comme un composant logiciel regroupant un ensemble cohérent de fonctionnalités. Les compétences peuvent donc être développées indépendamment de tout agent et donc réutilisées dans différents contextes. Une fois ces compétences créées, la construction d'un agent MAGIQUE se fait par un simple mécanisme d'enrichissement de l'agent à construire avec ces

[15] http://www2.lifl.fr/MAGIQUE

compétences. De plus, un agent MAGIQUE peut acquérir et invoquer dynamiquement de nouvelles compétences en cours d'exécution.

L'invocation de compétences se fait par le biais de primitives, par exemple `perform()` pour une invocation asynchrone sans retour de résultat, ou `asknow() pour un appel bloquant`. La compétence est désignée par un nom de méthode (chaîne de caractères), et on peut s'abstenir de préciser quel agent prendra en charge la compétence en laissant l'organisation hiérarchique d'agent trouver un destinataire. La figure 5.19 présente le code d'acquisition et d'invocation de compétences (`skill` et `ability`).

```
agent.perform("addASkill",new Object[]{"skill",...args...});
perform("ability",arg...);
```

FIG. 5.19 – Acquisition et invocation dynamique de compétences dans MAGIQUE

Notre approche est similaire à MAGIQUE sur le plan de la séparation des concepts entre *coquille* et compétences, elle est même complètement complémentaire dans le sens ou nous nous intéressons aux aspects non-fonctionnels (donc de la coquille de l'agent) et absolument pas aux aspects fonctionnels d'un agent. Par contre, la plateforme et le modèle ne bénéficient pas des propriétés de flexibilité que nous proposons pour exploiter les agents dans des environnements répartis instables.

Ensuite, contrairement à nos propositions dans lesquelles les soucis de cohérence et de sûreté sont des préoccupations majeures, il est possible d'écrire et de compiler des codes d'agents MAGIQUE qui ne fonctionneront pas. Soit par la faute d'une simple erreur typographique au niveau du code dans la désignation d'une compétence, soit par l'inexistence d'une compétence à moment donné (il n'y a pas de garantie sur la présence d'une compétence dans le système). Notre façon d'assembler les micro-composants, de vérifier les assemblages et notre interface entre les niveaux de programmation permet de répondre à ces problèmes.

Enfin, l'implémentation n'est pas vraiment adaptée à l'exécution dans un contexte de performances (cf. section suivante, le problème semble venir de l'implémentation qui abuse d'appels réflexifs coûteux) ni de répartition à grande échelle (centralisation de certaines compétences), les agents ne sont pas mobiles et possèdent des références statiques de la forme `shortname@host.domain.country:PORTNUMBER`.

5.5.2 De COMET à DIMA

Le *middleware* COMET[16] est un environnement Java qui permet de mettre en œuvre des applications réparties à base de composants et d'évènements. Son objectif est d'étendre le modèle client/serveur, en fournissant une solution qui se veut plus efficace et plus flexible dans les contextes de répartition et particulièrement de pair à pair. L'intergiciel est composé de deux parties : le noyau de COMET permet le fonctionnement des composants en mode réparti, et un langage dédié,appelé Scope, qui permet d'un part de décrire les composants, et d'autre part de réaliser des vérifications formelles de la structure.

[16] http://www.nongnu.org/comet

Les composants implémentent la logique métier de l'application et les invocations de services sont réalisées par des évènements asynchrones qu'écoutent les composants. Ainsi, il n'existe pas de références explicites entre les composants, ce qui permet de les modifier lors de l'exécution et de réaliser l'adaptation dynamique de l'application.

L'architecture de COMET est réflexive et présente trois niveaux distincts : le niveau de base contient les fonctionnalités statiques et prédéfinies des composants ; les fonctionnalités de contrôle sont réifiées au niveau méta-comportement pour offrir une vision uniforme au niveau de base ; le niveau méta-rôle représente la couche support des fonctionnalités adaptatives. Le niveau intermédiaire (méta-comportement) est similaire au méta-niveau de CodA (méta objets de contrôle tels réception, file de message, émission...).

COMET a été utilisé en particulier dans l'environnement DIMA[17] (Développement et Implémentation de Systèmes Multi-Agents), implémenté en SmallTalk [Gue98]. L'architecture d'agents DIMA propose de décomposer chaque agent en différents modules dont le but d'intégrer des paradigmes existants notamment des paradigmes d'intelligence artificielle. Ces modules représentent les différents comportements d'un agent telles que la perception (interaction agent - environnement), la communication (interaction agent - agent) et la délibération.

Notre approche se distingue de COMET par la granularité des composants et de l'adaptation (micro-composants non fonctionnels contre composants métiers dans COMET). Nous ne nous intéressons pas à la problématique de l'adaptation dynamique de protocoles ou de composants métiers mais à l'adaptation des mécanismes d'exécution qui peuvent les entourer. De manière identique, l'approche DIMA + COMET est à rapprocher de MAGIQUE, permettant la construction de systèmes multi-agents par assemblage de composants fonctionnels.

5.5.3 MAST

MAST[18] est un environnement pour le développement et le déploiement d'applications multi-agent. Il fournit des outils pour la conception d'agents par assemblage de composants génériques. MAST propose un modèle de composants spécifique [Ver04], avec pour objectif de permettre l'ajout et le retrait de composants sans perturber le fonctionnement de l'agent, ceci sans intervention extérieure (autonomie de l'agent).

Les composants sont assemblés sur un noyau d'agent, qui leur retransmet des évènements auxquels ils sont abonnés. Chaque composant spécifie des ensembles de rôles abstraits fournis et requis, permettant d'en faire un assemblage correct. La méthodologie pour le déploiement des agents est issue de celle des composants répartis : schémas de déploiement spécifié dans un fichier XML et outil de déploiement spécifique.

Nous rejoignons les préoccupations de flexibilité de MAST ainsi que certains principes architecturaux, mais notre approche diffère sur plusieurs points.

– Il est possible d'enlever ou d'ajouter n'importe quel composant à l'exécution dans MAST, sans perturber le fonctionnement des autres composants, mais il n'est pas

[17] http://www-poleia.lip6.fr/~guessoum/dima.html
[18] http://www.emse.fr/~vercouter/mast

possible de vérifier si les assemblages sont corrects. Par exemple, on peut enlever le composant de communication mais l'agent risque de fonctionner en étant isolé... Nous souhaitons empêcher ce type de faute à l'exécution.
- Les composants de MAST doivent être identifiables dans une méthodologie spécifiques aux SMA (approche *Voyelles*), nous ne souhaitons pas limiter nos composants à un tel ensemble, nous souhaitons par exemple pouvoir modifier le composant de cycle de vie, ou introduire de nouveaux composants qui n'ont rien de spécifique aux SMA (mobilité...).
- Dans MAST, tous les composants de l'agent sont au même niveau, alors que dans notre modèle les composants non-fonctionnels sont placés dans un méta-niveau pour permettre l'introspection et l'intercession sur la structure de l'agent afin de permettre l'adaptation dynamique de l'architecture de l'agent.
- Dans notre proposition, les composants sont des micro-composants, de très petite granularité (service unique), afin de faciliter les validations sur les assemblages.
- L'un des objectifs de notre modèle d'agent est de pouvoir simplifier le développement d'applications réparties. Notre solution permet aux agents de manipuler la répartition et la mobilité, au moyen d'abstractions simples disponibles au niveau de base. Ainsi, il n'est pas nécessaire de prévoir des outils et des schémas de déploiement des agents. De même, les opérations de communication sont réalisées dans des composants de méta-niveau et il suffit d'un appel de méthode au niveau de base pour envoyer un message. La même opération est plus compliquée et moins lisible dans MAST, malgré l'utilisation d'outils.

5.5.4 MaDcAr

MaDcAr [GBV06] est un modèle abstrait pour construire des moteurs d'assemblage dynamique et automatique d'applications à base de composants, appliqué à des agents construits à base de composants. Les ré-assemblages peuvent être initiés par le concepteur de l'application ou par une nécessité d'adaptation au contexte d'exécution. MadCar distingue le niveau applicatif (l'assemblage de composants) du niveau d'adaptation, ce dernier étant appelé *comportement d'adaptation*. La politique d'adaptation qui pilote ce comportement doit être implémentée par le concepteur de l'agent, donc prévue à l'avance.

Les auteurs distinguent deux formes de ré-assemblages : *avec changement de configuration* lorsque l'agent modifie le nombre de ses composants (ajout ou suppression) et *sans changement de configuration* lorsqu'un composant est remplacé par un autre (remplacement d'un composant de communication pour se conformer à un nouveau protocole de communication). Ils justifient le besoin de ré-assemblage avec changement dynamique de configuration dans un scénario ou un agent prend et perd à tour de rôle une capacité de sociabilité qui lui permet de maximiser ses gains (associés à l'exécution de différentes tâches).

MaDcAr s'inspire fortement de MAST, en essayant d'introduire une vision de plus haut niveau sur les changements de configurations. Un des objectifs est d'ailleurs d'ajouter à MaDcAr un formalisme d'expression des politiques d'assemblages, permettant de spécifier le comportement (d'adaptation) d'agents auto-adaptables. Tous les composants apparaissent au même niveau (fonctionnels et opératoires), et les objectifs d'adaptation semblent plus fonctionnels qu'opératoires.

Il existe une implémentation fonctionnelle, AutoFractal[19], qui est une spécialisation de MaDcAr appliquée aux composants Fractal.

5.6 Quelques performances

Bien que nos objectifs soient de nature génie logiciel, nous présentons dans cette section quelques performances mesurables de nos prototypes. Nous voulons simplement montrer que nos prototypes sont efficaces et présentent un faible encombrement.

Les tests ont été réalisés sur un PC portable (Dell Inspiron 510m), sous Linux (Kubuntu Dapper) avec un noyau 2.6.15-26-686 SMP PREEMPT. La machine possède 512Mo de RAM et tourne à 1,7GHz. Les exemples, réalisés en Java, ont été compilés et exécutés avec la version 1.5.0_06 fournie par Sun. Une seule machine virtuelle a été utilisée pour chaque exemple.

5.6.1 Taille de l'environnement d'exécution

L'environnement d'exécution est constitué de l'ensemble des classes nécessaires à l'exécution d'un agent, compressées dans une seule archive Jar afin que le déploiement de l'environnement soit minimal et simplifié. Nous avons engendré quelques architectures d'agents, et leurs volumes (occupation disque) sont donnés dans le tableau 5.1

JavAct 0.5.1	**Agent**$^{\varphi}$ (Agent mobile adaptable RMI)	**Agent**$^{\varphi}$ (Agent minimal)
63ko	19ko	9ko

TAB. 5.1 – Taille de l'environnement d'exécution (archive jar)

On voit immédiatement l'intérêt de générer une architecture adéquate, pour s'adapter au mieux aux contraintes de systèmes embarqués par exemple. Ces faibles tailles sont de bons arguments face à d'autres environnements agents (Aglets, ProActive, MadKit, MAGIQUE...) dont la taille de l'environnement d'exécution (hors interfaces graphiques etc.) dépasse souvent plusieurs centaines de ko. Par exemple 340ko pour MAGIQUE et 50ko pour le micro noyau d'un agent MadKit (auquel il faut ajouter quelques librairies qui font grimper la taille globale).

5.6.2 Empreinte mémoire

Nous avons créé une application JAVACT qui crée 2000 agents au comportement simple (lors de son activation, chaque agent affiche un message à l'écran). L'interprétation de la figure 5.20 est délicate, car pour être complet il faudrait tenir compte des spécificités de la machine virtuelle et du système lui-même. Toutefois, on constate que l'empreinte mémoire moyenne pour un agent est inférieure à 1ko (comprenant les données système : thread...) ce qui est suffisamment faible pour le déploiement sur des périphériques à capacité mémoire limitée. L'évolution de l'occupation mémoire est croissante, parfois brusquement diminuée (probablement par le passage du *garbage collector* de java).

[19] http ://adapt.asr.cnrs.fr/reunions/20061003/

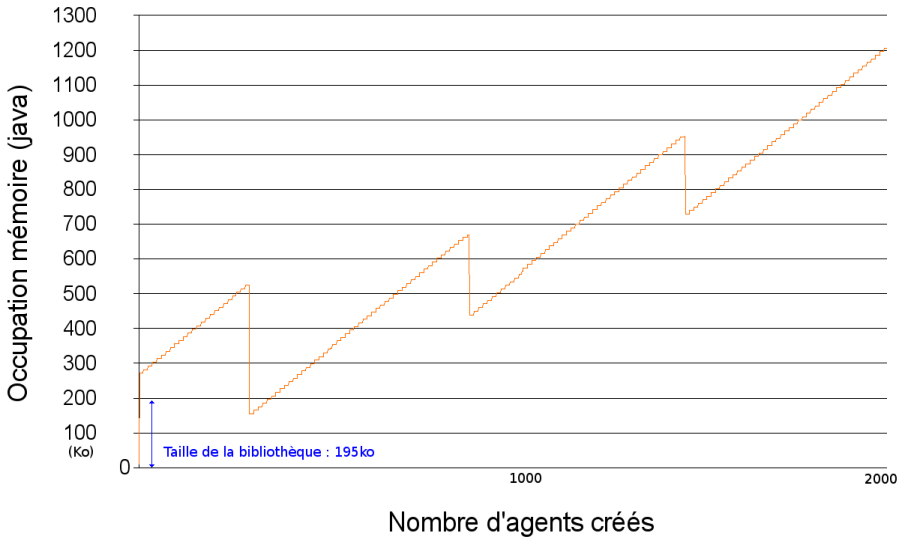


FIG. 5.20 – Empreinte mémoire d'une application JAVACT

5.6.3 Vitesse d'exécution

Nous avons réalisé une évaluation de performances brutes sur l'un des problèmes de référence (*toy problem*) de la plateforme MAGIQUE, à partir de l'exemple du calcul de la factorielle pour lequel l'ensemble des codes est proposé en téléchargement[20].

Le problème, un classique de la programmation concurrente adapté au modèle d'agent, consiste à créer une population d'agents composée de la façon suivante :

- un agent factorielle F1 qui sait comment on calcule une factorielle, mais qui ne sait pas multiplier,
- 3 agents multiplicateurs M1 M2 M3 qui savent effectuer des multiplications.

La figure 5.21 présente la totalité du code fonctionnel. Pour obtenir les meilleures performances nous avons construit un type d'agent spécifique : ses composants d'interaction sont locaux (la communication se fait directement par des appels de méthodes entre objets) et il ne possède pas de caractéristiques d'adaptation ni de mobilité. On obtient un type d'agent assez simpliste, construit à partir du modèle d'acteur dont on a repris le cycle de vie et les micro-composants de création et de communication.

[20] http://www2.lifl.fr/MAGIQUE/examples/factorial.html

```
class FactorialBeh extends QuasiBehavior implements Factorial{
  private Vector numberList;
  private Agent m1,m2,m3;
  private int roundrobin=1, runsDone=1, n;

  FactorialBeh(Agent m1, Agent m2, Agent m3) {
    this.m1=m1;     this.m2=m2;     this.m3=m3;
  }

  private Agent roundRobin() {
    switch (roundrobin) {       case 1: roundrobin++; return m1;
      case 2: roundrobin++; return m2; case 3: roundrobin=1; return m3; }
    return null;
  }

  public void compute(int n) {
    this.n=n;
    numberList=new Vector();
    BigInteger bigi=BigInteger.ZERO;
    //Construction de la liste des valeurs  multiplier
    for (int i=1; i<=n; i++) {
      bigi=bigi.add(BigInteger.ONE);
      numberList.add(bigi);
    }
    //Affectation des calculs
    for (int i=0; i<n; i+=2)
      send(new JAMmult((BigInteger)numberList.remove(0),
                       (BigInteger)numberList.remove(0)), roundRobin());
  }

  public void result(BigInteger parameter0) {
    System.out.println("Resultat␣intermdiaire␣:"+ parameter0);
    if (++runsDone==n) System.out.println("c'est␣fini␣!");
    if (numberList.size()>=1){
      JAMmult mess=new JAMmult((BigInteger)numberList.remove(0),parameter0);
      send(mess,roundRobin());
    } else numberList.add(parameter0);
  }
}

class MultiplierBeh extends QuasiBehavior implements Multiplier{
  private Agent Factorial;

  public void setFactorial(Agent Factorial) {this.Factorial=Factorial;}

  public void mult(BigInteger x, BigInteger y) {
    System.out.println(mySelf()+"␣multiplie␣"+x+"␣par␣"+y);
    send(new JAMresult(x.multiply(y)), Factorial);
  }
}

public class Skeleton1 {
  public static void main(String[] args) {
    int n=Integer.parseInt(args[0]);

    Agent m1=LocalAgentFactory.create(new MultiplierBeh());
    Agent m2=LocalAgentFactory.create(new MultiplierBeh());
    Agent m3=LocalAgentFactory.create(new MultiplierBeh());
    Agent fact=LocalAgentFactory.create(new FactorialBeh(m1,m2,m3));
    LocalAgentFactory.send(new JAMsetFactorial(fact),m1);
    LocalAgentFactory.send(new JAMsetFactorial(fact),m2);
    LocalAgentFactory.send(new JAMsetFactorial(fact),m3);
    LocalAgentFactory.send(new JAMcompute(n),fact);
  }
}
```

FIG. 5.21 – Implémentation du problème de la factorielle

Le tableau 5.2 montre les temps d'exécution pour le calcul de $n!$, pour différentes valeurs de n, mesurés par la commande `time` et arrondis à la seconde. Evidemment le programme java est optimal (le calcul se fait dans une boucle et le test est réalisé sur un seul processeur), mais le surcoût de nos différents prototypes n'est pas énorme, compte tenu du fait que le code n'a jamais été optimisé pour être efficace et que le surcoût peut être attribué pour la plus grande part à l'échange de messages. Dans le contexte visé par nos travaux (systèmes répartis à grande échelle), ces différences deviennent négligeables devant les latences et les temps de transferts liés aux réseaux.

$n!$	**Prog. Java**	**Magique** (local)	**JavAct** (RMI)	**Agent**$^{\varphi}$ (minimal)
100	0"	4"	<1"	<1"
500	<1"	8"	2"	1"
1000	<1"	15"	4"	2"
5000	1"	1'03"	8"	4"

TAB. 5.2 – Vitesse d'exécution (factorielle)

5.7 Conclusion

Dans ce chapitre, nous avons présenté un modèle d'agent flexible, qui étend le modèle d'agent adaptable du chapitre 3, en ajoutant un niveau de flexibilité dans l'architecture. Cette flexibilité permet d'engendrer des types d'agents différents, dynamiquement adaptables, avec des propriétés non fonctionnelles adaptées à leur environnement. De manière similaire au modèle d'agent mobile adaptable, les agents sont constitués d'un ensemble de micro-composants non-fonctionnels reliés à un connecteur. Les architectures engendrées sont plus performantes (taille minimale, meilleure vitesse d'exécution), et leur minimisation offre une meilleure sûreté (vérifications plus simples sur un ensemble de micro-composants minimal).

Nous avons proposé un patron de conception pour les applications réparties, basé sur un modèle d'agent adaptable. Il est tout à fait possible de réutiliser les architectures engendrées par les outils présentés dans ce chapitre, dans le cadre de ce patron de conception. Ainsi, le *framework* bénéficie de la minimisation, de la sûreté et de l'augmentation des performances des agents.

Au niveau de la méthodologie, nous proposons d'implémenter des caractéristiques de différents agents sous la forme de micro-composants intégrés dans le méta-niveau de notre architecture. Toutefois, l'insertion d'un composant au méta-niveau ne doit pas être systématique. En particulier, nous rejoignons les idées des architectures des SMA dans lesquelles la plupart des composants sont purement fonctionnels et placés au même niveau que le code fonctionnel. Un composant doit être placé au méta-niveau pour trois raisons principales : si l'on souhaite pouvoir l'adapter dynamiquement, s'il doit interagir avec d'autres composants de ce même niveau ou bien s'il doit être séparé des préoccupations liées au code fonctionnel (transparence).

Quelques problèmes sont ouverts, par exemple sur le besoin et la prise en compte des mutations des agents à l'exécution (changement dynamique de type), ou encore l'éventuelle

reconfiguration liée à l'acquisition dynamique d'un micro-composant fourni par un tiers (agent ou système d'accueil en particulier).

Enfin, en perspective à ce travail et pour améliorer la sûreté des architectures engendrées, nous prévoyons d'intégrer un générateur de code ADL, afin de réaliser des vérifications formelles de propriétés d'assemblage des micro-composants.

Résumé des contributions :

- Nous proposons un modèle d'agent flexible, qui permet d'engendrer différents types d'agents dynamiquement adaptables, par assemblage de micro-composants non-fonctionnels réutilisables, qui peuvent bénéficier via des micro-composants adéquats de propriétés de mobilité ou d'adaptation dynamique au contexte d'exécution [LA06].
- Ce modèle est conçu pour supporter des analyses et des vérifications à partir d'une transformation de la description d'une architecture d'agent dans un langage de description d'architecture (ADL) et d'outils dédiés aux vérifications.
- Nous proposons, sur la base de ce modèle d'agent flexible, une implémentation contenant un environnement de modélisation graphique d'architecture d'agent, une bibliothèque de micro-composants non-fonctionnels et d'architectures prédéfinies, un générateur de squelette d'architecture et un outil de minimisation des architectures prenant en compte les besoins fonctionnels.

Chapitre 6

Conclusion

Pour contribuer à la simplification du développement d'applications réparties flexibles, nous avons exploité un ensemble de technologies qui s'articule autour des composants logiciels, du modèle pair à pair et des agents mobiles, pour proposer des architectures logicielles capables de supporter l'adaptation dynamique. Nous reprenons dans cette conclusion les principales contributions et nous discutons quelques perspectives à nos travaux.

La prise en compte des besoins de flexibilité dans les applications réparties ouvertes à grande échelle augmente la complexité du développement, du déploiement et de la maintenance. Afin de contribuer à la maîtrise de cette complexité, nous proposons des modèles d'architectures logicielles basés sur les technologies agent logiciel, composant logiciel et modèle pair à pair dont nous exploitons la complémentarité. Ces architectures permettent l'adaptation dynamique des aspects non fonctionnels des applications à leur contexte d'exécution ainsi que la personnalisation (l'adaptation fonctionnelle est supportée au niveau des agents, en partie grâce à la mobilité logicielle -mobilité d'agent et de code-).

Les solutions s'appuient sur une décomposition des agents en composants de grain fin et respectent les principes d'autonomie et de séparation des préoccupations et des niveaux. Par ailleurs, elles semblent compatibles avec la prise en compte des problèmes de sécurité. Les expériences menées se sont montrées encourageantes sur le plan de la simplicité d'utilisation et de la réutilisation.

6.1 Synthèse de la contribution

6.1.1 Au niveau micro (chapitre 3)

Nous proposons un modèle d'agent logiciel mobile basé sur le modèle de programmation par acteur. Son architecture dynamiquement reconfigurable est constituée d'un

ensemble de micro-composants qui implémentent des services non-fonctionnels. L'organisation des micro-composants en étoile autour d'un connecteur central permet d'adapter dynamiquement ces services en fonction du contexte d'exécution [LA04a].

Nous avons conçu une implémentation complète de ce modèle, sous la forme d'un prototype appelé JavAct$^{\delta}$ [LA04a], ainsi qu'une implémentation simplifiée (sans analyseur et sans sonde dans le système d'accueil), JavAct 0.5 [AHL+04]. Cette version et le *plugin Eclipse* qui permettent d'accélérer et de simplifier le développement sont diffusés sous la forme de logiciels libres[1] (licence LGPL).

6.1.2 Au niveau macro (chapitre 4)

L'analyse d'un certain nombre d'exemples applicatifs a mis en évidence une architecture logicielle générique à base d'agents mobiles adaptables et de composants logiciels, que nous avons abstrait sous forme d'un patron de conception [LA05].

L'utilisation d'agents mobiles simplifie la phase de localisation des ressources dans les systèmes P2P, et en particulier dans les systèmes P2P purs. Nous proposons également d'utiliser les agents mobiles pour personnaliser la phase d'exploitation des ressources, par déploiement d'un comportement adéquat fourni par le client [LAP04].

Plus généralement, nous proposons d'employer des agents mobiles adaptables comme support de déploiement pour les comportements (composants fonctionnels) : d'une part la mobilité d'agent permet un transport adaptatif des composants fonctionnels sur le réseau, d'autre part le méta-niveau joue le rôle de conteneur pour le composant fonctionnel, en lui fournissant les services nécessaires à son exécution et permettant sa configuration (choix des micro-composants) ainsi que sa reconfiguration dynamique [ALPre]. Aussi, on peut considérer que le modèle d'agent mobile adaptable avec ses capacités de déploiement constitue un modèle de composant pour les applications réparties à grande échelle.

En pratique, le patron de conception a été mis en œuvre sous la forme d'un *framework* codé en Java à partir duquel nous avons dérivé différents prototypes dont une application pour la mutualisation de ressources en P2P.

6.1.3 Au niveau micro (chapitre 5)

L'utilisation de micro-composants (granularité fine à l'échelle d'un service unique) au méta-niveau, l'organisation en étoile autour d'un connecteur et l'interface entre les niveaux constituent un style d'architecture compatible avec la proposition du chapitre 3.

Conformément à ce style d'architecture, nous proposons un modèle d'agent flexible qui permet d'engendrer différents modèles d'agents dynamiquement adaptables, par assemblage de micro-composants non-fonctionnels réutilisables. Ainsi les agents peuvent bénéficier, *via* un ensemble de micro-composants adéquats, de propriétés de mobilité ou d'adaptation dynamique au contexte d'exécution [LA06]. Ce modèle est conçu pour supporter

[1] http://www.irit.fr/recherches/ISPR/IAM/JavAct_fr.html

des analyses et des vérifications à partir d'une transformation de la description d'une architecture d'agent dans un langage de description d'architecture (ADL) et d'outils dédiés aux vérifications.

Nous proposons, sur la base de ce modèle d'agent flexible, une implémentation contenant un environnement de modélisation graphique d'architecture d'agent, une bibliothèque de micro-composants non-fonctionnels et d'architectures prédéfinies, un générateur de squelette d'architecture et un outil de minimisation des architectures prenant en compte les besoins fonctionnels.

6.2 Problèmes ouverts et perspectives

6.2.1 Flexibilité et sémantique

Pour compléter nos travaux et dépasser le stade du prototype, il faudrait intégrer à l'environnement de modélisation des agents, présenté dans le dernier chapitre, des outils (ADL et outillage spécifique) permettant de décrire des architectures dynamiques et de vérifier de manière plus poussée les assemblages de micro-composants. L'utilisation dans les méta-données des micro-composants de contraintes (*via* des langages de contraintes comme OCL[2] par exemple) pourrait également renforcer la sûreté des assemblages.

La construction d'architectures d'agents par assemblage de micro-composants offre un bon niveau de réutilisation et de simplification pour le concepteur d'une application agent. Toutefois, si le modèle d'agent mobile adaptable que nous avons présenté au chapitre 3 possède une sémantique claire (basée sur celle de l'*acteur* d'Agha), cela n'est pas évident pour un agent créé de manière *ad hoc*. La multiplication et la diversité des architectures pourrait être à l'origine d'une perte de sémantique, qui devrait donc être palliée à d'autres niveaux. L'introduction d'un niveau sémantique dans les méta-données des micro-composants pourrait constituer une solution, mais le problème reste ouvert.

Dans nos propositions, nous avons défini certaines limites sur ce qui pouvait être adapté et ce qui devait resté figé au niveau architectural. Nous ne permettons pas à un agent de changer de modèle en cours d'exécution (un agent BDI ne deviendra pas un acteur mobile), ni d'acquérir dynamiquement un micro-composant de type inconnu lors de l'étape de conception (puisque s'il n'est pas connu, l'agent ne saura pas comment l'utiliser correctement). Ces choix pourraient être remis en question si des besoins étaient identifiés, nos limitations étant surtout présentes pour simplifier les étapes de vérifications des architectures.

6.2.2 Expérimentations

Les questions ouvertes en terme d'adaptation statique (jusqu'où peut-on aller dans la construction de modèles d'agents ?) et dynamique (jusqu'où peut-on aller en terme de réorganisation dynamique ?) nous conduisent à envisager de nouvelles expérimentations.

[2] http://www.omg.org/docs/ptc/03-10-14.pdf

Nous allons évaluer ces propositions et ces idées dans le cadre du projet inter-équipe *Neo-Computing* de l'Institut de Recherche en Informatique de Toulouse (IRIT). Ce projet porte sur la conception de systèmes mobiles communiquant par le biais d'un environnement en vue de satisfaire au mieux les besoins dynamiques de leurs utilisateurs. Le projet s'intéresse à des problématiques porteuses que l'on trouve actuellement en *autonomic computing*, *ambient intelligence*, *informatique pervasive*, *informatique ubiquitaire*, *domotique*.

La place des micro-composants dans l'architecture est un problème ouvert d'ingéniérie : ils peuvent être intégrés au niveau méta pour bénéficier des apports en terme de vérification d'assemblage et d'adaptation sans que cela constitue une règle impérative. On peut espérer trouver des éléments de réponse lors des expériences concrètes d'utilisation de nos prototypes.

6.2.3 Déploiement et sécurité

Nous avons proposé un patron de conception pour les applications réparties instables, dans lequel le déploiement de composants métiers est réalisé par des agents. Les composants déployés ne dépendent pas d'un modèle particulier, et il reste à évaluer l'apport de cette méthode de déploiement à des composants industriels (EJB, CCM...). De manière similaire, dans nos architectures d'agents nous utilisons des micro-composants sans faire de supposition sur le modèle de composant. Une prochaine étape consiste à évaluer l'intérêt de modèles de composants industriels tels Fractal[3].

Du point de vue de la sécurité, on peut espérer que la construction d'architectures d'agent par assemblage de micro-composants « sur étagère » certifiés (fournis par des tiers de confiance) permettrait de renforcer les propriétés de sécurité de l'architecture elle-même. Il reste à valider la pertinence d'une composition de propriétés de sécurité ainsi qu'un modèle de composant adéquat.

6.2.4 Ingéniérie des modèles

L'apport de l'approche IDM (Ingénierie Des Modèles) dans la démarche de modélisation de l'adaptation telle qu'elle est abordée dans nos travaux fait l'objet d'une thèse en cours dans notre équipe de recherche. L'objectif est de développer un outil d'aide au développement d'applications réparties adaptables, au travers de plusieurs niveaux de modélisations. Cet outil intégrerait un langage de modélisation dédié (DSML) aux micro-composants. La sémantique de ce langage serait donnée par un *mapping* avec les micro-composants d'agents existants. Le modèle d'agent obtenu par assemblage de micro-composant pourrait alors être utilisé au sein d'un outil de modélisation d'applications réparties adaptables.

[3] http://fractal.objectweb.org

Annexe A

Sécurité pour les agents mobiles

La sécurité est une préoccupation de premier plan dans les systèmes répartis. Dans cette annexe, nous faisons le point sur les problèmes de sécurité qui entourent l'utilisation d'agents mobiles et sur les solutions qui peuvent être apportées.

A.1 Problématique

L'utilisation d'agents mobiles dans un contexte industriel est souvent freinée par des questions de sécurité. Nous essayons dans cette partie de faire le point dans ce domaine, en commençant par montrer comment des solutions classiques issues des systèmes répartis puis des technologies venant du *code mobile* permettent de répondre à certains besoins de sécurité des agents mobiles.

A.1.1 Localisation des problèmes de sécurité

Afin de rester dans le cadre le plus général possible, nous ne nous focalisons pas sur un modèle ou une implémentation particulière d'agent mobile. En effet, pour considérer l'ensemble des problèmes de sécurité potentiels, un modèle simple suffit. Ainsi au niveau le plus abstrait, un agent mobile est un programme autonome, se déplaçant sur le réseau avec son code et ses données. Pour cela, il s'exécute sur une plateforme d'accueil lui fournissant les ressources nécessaires à son exécution. De plus, ses déplacements et ses communications sont véhiculés par un réseau.

Nous pouvons alors identifier directement trois entités contenant des informations, qui peuvent donc être attaquées :

– l'agent mobile
– la plateforme d'accueil
– les informations transmises par le réseau

En complément, il n'y a pas que l'information qui peut être la cible d'attaques, mais aussi la disponibilité des entités. Le *déni de service* consiste à saturer une ou plusieurs ressources (processeur, mémoire) afin d'empêcher un système de rendre un service. Il est possible d'attaquer via un déni de service n'importe quelle entité active d'un système, ici les agents, les plateformes ou encore le réseau.

La plupart des auteurs dans le domaine de la sécurité des agents mobiles classent les attaques selon quatre catégories : agents contre plateforme, plateforme contre agents, agents contre agents et attaques au niveau réseau [Tsc99, JK00, AB04]. Nous pensons qu'il est plus réaliste de ne pas considérer l'origine de l'attaque mais seulement l'entité attaquée. En effet, on ne sait pas toujours quelle entité émet un message ou délivre un service...

A.1.2 Attaques possibles

Contre le réseau

Dans cette catégorie, nous pensons à toutes les attaques classiques des systèmes répartis, comme par exemple l'écoute, le rejeu, l'altération ou l'interception des communications, l'usurpation d'identité et les déni de services. Elles peuvent d'une part toucher les communications entre agents, d'autre part porter atteinte aux agents eux-même lors de leurs déplacements sur le réseau, le code des agents et leurs données étant sérialisés et transmis à la volée.

Ces attaques dépendent le plus souvent de failles existant dans les protocoles de communication, donc du système d'exploitation sous-jacent. Des organismes comme le CERT[1] recensent les vulnérabilités et publient des bulletins d'information pour permettre la prévention ou la correction de ces failles. N'étant pas du niveau du *middleware*, nous ne nous intéressons plus à ces problèmes de sécurité du réseau.

Contre une plateforme

Le système d'accueil est un serveur dont le rôle consiste à exécuter du code transmis par un client, en lui fournissant des ressources et un environnement d'exécution adapté. Les problèmes de sécurité peuvent alors être les suivants :
– accès à des ressources interdites (lecture, altération des fichiers du serveur...)
– installation ou exécution de codes néfastes (virus, troyens...)
– usurpation (d'identité ou de droits pour des services fournis à certains agents...)
– déni de service (simple boucle infinie pour saturer un processeur, ou utilisation abusive d'un service de la plateforme, par exemple création infinie d'agents...). En fonction de ses droits d'accès, une entité peut aller jusqu'à terminer l'exécution de la plateforme (un simple *System.exit*() en java).

[1] http://www.cert.org

Contre un agent

L'intégrité des agents amenés à se déplacer de site en site est difficile à assurer dans le cas général, puisque la gestion des sites est assurée par des tiers ayant tous les droits sur l'exécution des systèmes d'accueil et donc des agents s'y exécutant. La plupart des attaques suivantes seraient probablement initiées par la plateforme, mais sont également envisageables depuis n'importe quelle autre entité de plus bas niveau, d'un composant du système d'exploitation par exemple.

- altération de l'agent (code/données, par exemple d'un agent cherchant le prix minimal d'un produit...)
- altération d'un service (au niveau de la sémantique d'un service, ou résultat faux pour induire un agent en erreur...)
- usurpation d'identité (dans le but de faire croire à un agent qu'il est sur une plateforme donnée pour lui extorquer de l'information...)
- vol d'information de la part d'une entité locale à l'agent (ou encore monitoring de l'activité...)
- déni de service (limitation de l'autonomie de l'agent, limitation en ressources voire terminaison de l'exécution...)

A.2 Eléments de solution

A.2.1 Propriétés de sécurité

Les propriétés suivantes, requises pour assurer la sécurité dans les systèmes répartis classiques [Sch01], sont également nécessaires au bon fonctionnement des agents mobiles.

- Confidentialité : seul le destinataire de l'information peut en lire le contenu.
- Authentification : l'identité des différents acteurs doit pouvoir être vérifiée.
- Intégrité : l'information ne doit pas pouvoir être corrompue sans que cela ne se voit.
- Non-répudiation : lorsqu'une information est transmise entre deux acteurs, l'émetteur ne doit pas pouvoir nier avoir envoyé l'information, et le destinataire l'avoir reçue. Chaque partie possède ainsi la preuve de l'existence de la transaction.
- Contrôle d'accès : l'accès à certaines ressources (informations ou services) doit être restreint à des acteurs autorisés.
- Disponibilité : l'accès à une ressource ou un service doit être toujours possible.
- Anonymat : le minimum d'information doit être dévoilé pour réaliser une transaction. Ainsi pour effectuer un paiement électronique, il n'est pas nécessaire de divulguer le numéro de sécurité sociale du client humain.

Les agents communiquant par messages, ce sont en partie ces conteneurs d'informations que l'on cherche à protéger (confidentialité, authentification des agents, intégrité et non-répudiation). Mais, les codes et les données des agents se déplaçant sur différentes machines, il est nécessaire de porter une attention identique à leur protection et à celle des plateformes. Ainsi, il semble judicieux de considérer d'une part l'intégrité et l'authenticité du code d'un comportement d'agent reçu sur une plateforme, d'autre part de limiter ses droits d'accès en fonction de l'identité (vérifiée) du fournisseur avant de l'exécuter. De

manière réciproque, les agents une fois activés ont intérêt à vérifier l'authenticité de la plateforme et à mettre en place des mécanismes de protection de leurs données.

Nous ne considérons ici que des solutions logicielles, capables de répondre aux propriétés de sécurité énoncées plus haut. La plupart sont des moyens de prévention limitant les capacités d'action de l'attaquant, mais certaines ne sont que des techniques de détection.

A.2.2 Solutions génériques

Dans le cadre de la protection d'informations, les quatre premières propriétés peuvent être garanties par des techniques cryptographiques [Sch01]. Il existe deux types principaux d'algorithmes de chiffrement souvent utilisées conjointement, en fonction des besoins de l'utilisateur :

- Chiffrement à clé secrète (symétrique) : L'émetteur et le destinataire utilisent une même clé pour chiffrer et déchiffrer le message. Par exemple DES, AES, RC5... Ces algorithmes sont souvent utilisés pour transmettre des quantités importantes de données en respectant la confidentialité et l'intégrité, car ils sont moins coûteux en ressources que ceux de la catégorie suivante.
- Chiffrement à clé publique (asymétrique) : Un message chiffré avec la clé publique (connue de tous) d'un utilisateur A ne peut être déchiffré qu'avec la clé privée de A (et connue de lui-seul). Par exemple RSA, DSA... Ils sont principalement utilisés pour déterminer une clé secrète à l'initiation d'une communication, ou encore pour générer des signatures et des certificats numériques afin de permettre l'authentification et la non-répudiation.

Ces algorithmes permettent de mettre en place des solutions complètes pour la transmissions d'informations sur le réseau comme par exemple les PKI (Public Key Infrastructure). La mise en oeuvre de ces techniques est souvent facilitée par l'existence de composants pré-existants. Par exemple en java il est possible d'utiliser l'extension cryptographique JCE[2].

A un autre niveau, il est possible de verrouiller un système en installant un contrôle d'accès. Il en existe différents modèles [SdCdV01], exprimant les autorisations en fonction des utilisateurs, de leur rôle ou des ressources... En java, il est possible de choisir un gestionnaire de sécurité en fonction du contexte d'utilisation (classe $SecurityManager$), associé à un fichier de configuration de permissions[3]. Celui-ci sert à déterminer quelles sont les fonctionnalités autorisées pour les codes s'exécutant sur la machine virtuelle, avec un grain assez fin.

La disponibilité peut être assurée par l'utilisation d'une architecture conçue en tenant compte de cette préoccupation, via des mécanismes comme la redondance, la supervision et l'adaptation dynamique de l'architecture au contexte d'exécution. La redondance permet d'une part d'absorber des flux importants de requêtes, d'autre part de délivrer un service continu même si une entité vient à tomber en panne. Dans ce cas, le rôle d'une entité de supervision peut être le redémarrage ou la réparation du service en erreur, voire une re-configuration de l'architecture pour optimiser le service.

[2] http://java.sun.com/j2se/1.4.2/docs/guide/security/jce/JCERefGuide.html
[3] http://java.sun.com/j2se/1.4.2/docs/guide/security/permissions.html

Mécanismes de confiance

La confiance[4] est une propriété fondamentale dans les relations sociales, nécessaire lorsqu'on ne sait pas obtenir (ou qu'on ne veut pas) l'ensemble des éléments de preuve dictant une conduite. Dans le domaine de l'informatique, un cadre juridique récent définit ce que devrait être la confiance numérique[5].

Par exemple, un internaute lambda qui se connecte de manière sécurisée sur le site de sa banque ne cherche pas à vérifier l'authenticité du serveur qui l'accueille, tant qu'il ne reçoit pas de message d'erreur de son navigateur. Pourtant, même vérifiée par le logiciel, une telle preuve n'est pas évidente. En effet, le navigateur accepte le certificat de la banque car il est probablement signé par une autorité de certification, et il fait peut être confiance au système d'exploitation pour lui indiquer que ce le certificat de ce dernier est valide. Ce qui ne repose sur aucune preuve, mais uniquement de fortes présomptions, basées sur des certificats et des signatures de données.

L'utilisation de code signé est l'une des premières applications de la confiance, pour la protection de l'agent comme de l'hôte. L'intégrité et l'authentification de l'entité sont vérifiables via la signature. Ensuite, il est possible pour un client d'exécuter un de ses agents sur une plateforme en qui il a confiance et qui lui assurerait (contractuellement) certaines propriétés de sécurité, en ayant simplement vérifié la validité d'un certificat. Et inversement pour l'hôte qui pourrait faire confiance à un agent. Bien sûr, cette propriété n'est pas suffisante et dans un système ouvert à grande échelle elle peut sembler difficile à poser comme hypothèse de conception. Toutefois, étant simple et peu coûteuse à mettre en place, elle ne doit pas être ignorée, éventuellement en complément des techniques qui suivent.

A.2.3 Solutions spécifiques code mobile

Les principales différences entre du *code mobile* et un *agent mobile* tiennent dans l'autonomie et la proactivité de ce dernier, qui se déplace selon sa propre initiative sur des systèmes différents, aux politiques et exigences de sécurité potentiellement distinctes. Ainsi, les contraintes et les possibilités de protection offertes aux agents mobiles sont-elles plus importantes [LMR00]. Mais la plupart des techniques de protection des hôtes ou des données pensées initialement pour le code mobile s'appliquent également aux agents mobiles.

Protection de l'hôte

Le *Sandboxing* ou *bac à sable* consiste à exécuter un code non-sûr dans un environnement restreint. Certaines fonctionnalités de bas niveau sont inhibées ou possèdent une sémantique altérée pour rendre un service sans altération possible du système. Le mécanisme de SandBoxing le plus connu est celui des applets Java (JDK 1.0) dans les navigateurs Web. Cette technique est le plus souvent employée conjointement à des mécanismes de contrôle d'accès et de permissions configurables (cf. A.2.2).

[4] http://fr.wikipedia.org/wiki/Confiance
[5] Loi du 21 juin 2004 pour la confiance dans l'économie numérique.

L'*isolation* du code impose à ce dernier l'utilisation exclusive d'un petit composant d'interface avec le système support d'exécution. Ce composant peut être validé par des méthodes formelles et ne fournir qu'une API minimale d'accès aux services de la plate-forme.

Les *codes auto-validants* encapsulent des preuves de certaines propriétés, réalisées par le fournisseur du code. Elles peuvent être assez rapidement vérifiées, avant l'exécution. Cette approche a de nombreux désavantages, les preuves dépendant notamment du matériel et du système d'exploitation utilisés et étant particulièrement difficiles à fournir.

Des *vérifications statiques* de code peuvent être réalisées avant son instanciation. Ces méthodes permettent de garantir certaines propriétés de sécurité avec un coût souvent important à l'exécution. Le *ClassLoader* de Java vérifie par exemple le *bytecode* de chaque classe avant de pouvoir l'instancier pour détecter les débordements de pile. De la même manière, il est possible de déterminer l'ensemble des permissions nécessaires à l'exécution du code, afin de les confronter à celles autorisées par le mécanisme de contrôle d'accès.

Protection du code et des données

L'*obscurcissement* consiste à transformer le code ou les données d'une entité tout en préservant leur sémantique à l'exécution, de façon à les rendre difficilement analysables, et compréhensibles par un observateur extérieur. La durée de validité de la protection est courte, elle est liée au degré du brouillage, mais difficile à évaluer.

Le *chiffrement* des données permet d'obtenir l'anonymat souhaité pour certaines informations. Bien sûr, comme le code doit s'exécuter à moment donné, il n'est pas possible de le rendre confidentiel même pour la machine hôte.

Le *calcul par fonction cachée* essaie de contourner la limitation précédente, en ne dévoilant qu'une fonction intermédiaire fabriquée à la demande, servant de masque à la fonction réelle qui est ainsi cachée à l'hôte. Pour faire calculer $f(x)$ sans dévoiler f, un client peut chiffrer f qui devient $C(f)$ et fournir à l'hôte le programme $P(E(f))$ qui implémente cette nouvelle fonction. Il récupère ensuite chaque résultat sous la forme $P(E(f))(x)$ qu'il peut décrypter pour obtenir $f(x)$.

A.2.4 Solutions spécifiques agents mobiles

Protection de l'agent

L'*encapsulation partielle des résultats* consiste à envoyer au client les résultats de chaque exécution sur chaque machine de l'itinéraire de l'agent et de conserver dans ses données une partie de ces résultats. Au retour de l'agent, le client peut vérifier si les résultats reçus sont compatibles avec les enregistrements de l'agent et ainsi détecter une modification des données de l'agent par un hôte malveillant.

Le *rejeu* d'un agent auprès d'un hôte permet de vérifier la consistance des services de ce dernier. Un système fournissant des réponses différentes pour une requête identique peut parfois être la marque d'un hôte malveillant.

L'utilisation de mécanismes de *traces* internes aux agents permet de vérifier *a posteriori* leur bonne exécution. Par exemple leur itinéraire (avec les signatures des plateformes), ou les réponses des différents services pour chaque machine visitée. On peut ainsi détecter un hôte malveillant.

La *génération de clé dépendant de l'environnement* permet à un agent de réaliser certaines actions prédéfinies, en fonction de son environnement d'exécution. Par exemple, le résultat de la recherche de la signature d'un hôte peut servir à générer une clé qui débloque un morceau de code chiffré de l'agent. La condition environnementale doit être cachée par une technique cryptographique. Les portions de code ainsi cachées ne peuvent être dévoilées à un observateur tant que la condition n'est pas vérifiée.

A.3 Conclusion

S'il est vrai que les systèmes à base d'agents mobiles sont attaquables au même titre que tout système réparti, ceux-ci semblent plus facilement exposés aux abus et aux mauvaises utilisations dont les conséquences peuvent se révéler néfastes. Cet état de fait est un frein important à l'utilisation des technologies agents mobiles dans des projets industriels. Mais même si la conception d'une solution générique de sécurité pour les agents mobiles reste un problème ouvert, nous pensons que l'ensemble des éléments présentés et combinés suivant les besoins spécifiques à une application semblent donner suffisamment de pistes pour répondre au problème initial.

Annexe B

Réalisations

Cette annexe présente quelques prototypes d'applications développées durant cette thèse, afin d'illustrer nos propositions par des exemples concrets.

B.1 Présentation de l'API de JavAct

Cette section présente brièvement les primitives offertes par la bibliothèque JavAct pour la programmation de comportements d'agents. Toutes ces méthodes sont accessibles par héritage (de la classe `QuasiBehavior`) et sont déclarées `final`, il n'est donc pas possible de les redéfinir. Pour plus de détails (notamment sur les types) vous pouvez vous référer à la documentation et au tutorial disponibles sur http://www.irit.fr/recherches/ISPR/IAM/JavAct_fr.html.

B.1.1 Création d'un acteur

Ces primitives permettent de créer un acteur dont on récupère la référence, ou `null` si la création à échoué. Dans ce cas, une `Runtime` exception `CreateException` est levée.

`Actor create(QuasiBehavior b)`
Création d'acteur sur la place locale à partir du comportement `b`.

`Actor create(String p, QuasiBehavior b)`
Même chose, mais création sur une place `p`.

`Actor create( String p, QuasiBehavior b, MailBoxCtI box, BecomeCtI bec, CreateCtI crt, LifeCycleCtI lif, MoveCtI mve, SendCtI snd)`
Même chose, mais en donnant l'ensemble des micro-composants qui régissent le fonctionnement de l'acteur.

B.1.2 Envoi de message

Ces primitives permettent d'envoyer un message à un acteur. Si l'émission échoue, une `Runtime` exception `SendException` est levée.

`void send(Message m, Actor a)`
Envoie le message `m` à l'acteur de référence `a` de manière asynchrone.

`void send(MessageWithReply m, Actor a)`
Envoie le message `m` à l'acteur de référence `a`. On *peut* ensuite attendre le résultat avec `m.getReply()`. En cas d'erreur (ré-émission du même message par exemple), une `Runtime` exception `JSMSendException` est levée. En cas de problème côté récepteur, une `Runtime` exception `JavActException` est levée (côté récepteur inaccessible à l'émetteur).

B.1.3 Changement de comportement

Ces primitives permettent de changer le comportement (état + méthodes) d'un acteur après la fin du traitement du message courant.

`void becomeAny(QuasiBehavior b)`
Depuis la version 0.5.0 de JavAct, les primitives typées de changement de comportement sont générées automatiquement d'après la spécification des comportements d'acteurs. Néanmoins, il reste possible mais déconseillé de faire des changements de comportements non typés (et donc potentiellement moins sûrs) avec la primitive `becomeAny()`.

`void suicide()`
Permet de terminer l'exécution de l'acteur, qui ne répondra plus à aucun message et ne traitera pas les éventuels messages en attente.

B.1.4 Déplacement de l'acteur

Ces primitives permettent de déplacer un acteur vers une place accessible du réseau. Pour des raisons de cohérence, la mobilité n'est effectuée que lors du changement de comportement (soit à la fin du traitement du message courant). L'appel de ces primitives retourne sans erreur liée à la mobilité et le code qui suit leur exécution est toujours exécuté. Par contre, si une erreur se produit durant le déplacement (place inaccessible, problème de réseau...) une `Runtime` exception `GoException` est levée.

`void go(String p)`
Déplacement de l'acteur sur une place `p`, sans gérer les erreurs.

`void go(String p, HookInterface h)`
Déplacement de l'acteur sur une place `p`. Si une `GoException` est levée, la bibliothèque redonne la main à la méthode `resume(GoException e)` de l'objet `h`. Cet objet peut être l'instance d'une classe interne du comportement, afin de pouvoir accéder à toutes les information nécessaires au recouvrement de l'erreur.

B.1.5 Localisation et auto-référence de l'acteur

```
String myPlace()
```
Retourne l'adresse de la place sur laquelle l'acteur s'exécute, sous la forme d'une chaîne de caractères (`nom:port`).

```
Actor ego()
```
Retourne l'auto-référence de l'acteur.

B.1.6 Adaptation des micro-composants

JavAct permet l'adaptation dynamique par remplacement de ses micro-composants. Cette adaptation n'est effectivement réalisée que lors du changement de comportement.

```
void with(MailBoxCtI box)
void with(BecomeCtI bec)
void with(CreateCtI crt)
void with(MoveCtI mve)
void with(SendCtI snd)
void with(LifeCycleCtI lif)
```
Remplacement du composant boîte aux lettres, changement de comportement, création d'acteur, mobilité, envoi de messages et gestion du cycle de vie.

B.2 Encadrement d'étudiants

Dans le cadre de nos travaux sur les architectures d'agents et les modèles de programmation pour les systèmes répartis instables et afin de vérifier certaines de nos propositions, nous avons encadré plusieurs projets étudiants. Je tiens à remercier encore ici mon directeur de thèse, Jean-Paul Arcangeli, pour la confiance qu'il m'a accordée en m'autorisant à superviser ces projets. La majorité des étudiants étaient du niveau M1 (ex-maîtrise) en informatique et effectuaient, en petits groupes, un travail de TER (Travaux d'Etudes et de Recherche). Le TER est un module universitaire dans lequel les étudiants mettent en pratique leurs connaissances et découvrent les différents aspects d'un projet logiciel significatif.

En 2003, nous avons proposé un premier TER[1], pour vérifier l'adéquation du modèle d'agent mobile adaptable au contexte du P2P pur. Intitulé *Développement d'un outil de recherche et de téléchargement de fichiers répartis à base de sources multiples* il proposait de concevoir et d'implémenter un prototype de logiciel P2P (similaire aux clients P2P tels eDonkey ou Kazaa), dans lequel les agents mobiles adaptaient dynamiquement les différentes phases d'exécution (recherche, téléchargement...).

En 2004, nous nous sommes intéressés aux problèmes de sécurité de notre modèle d'agent et de la bibliothèque JavAct. Ce TER[2] nous a permis de vérifier la viabilité

[1] http://noman.flabelline.com/ter-2003.html
[2] http ://noman.flabelline.com/ter-2004.html

des solutions classiques de protection cryptographique pour les agents mobiles. En plus d'intégrer un micro-composant de communication crypté robuste, les étudiants ont proposé un protocole sécurisé pour la transmission de comportements d'agents, qu'il est possible d'étendre à la transmission de micro-composants.

En 2005, nous avons encadré une équipe d'étudiants de 2ème année de l'ENSEEIHT (Ecole Nationale Supérieure d'Electrotechnique, d'Electronique, d'Informatique, d'Hydraulique et des Télécommunications) qui a repris le prototype de logiciel P2P pour l'adapter selon notre proposition de patron de conception (chapitre 4). Le prototype qui en est issu, permet par exemple d'assembler graphiquement des composants-comportements d'agents, afin d'adapter le fonctionnement des agents depuis la phase de localisation jusqu'à la phase d'exploitation des ressources (figure B.1)).

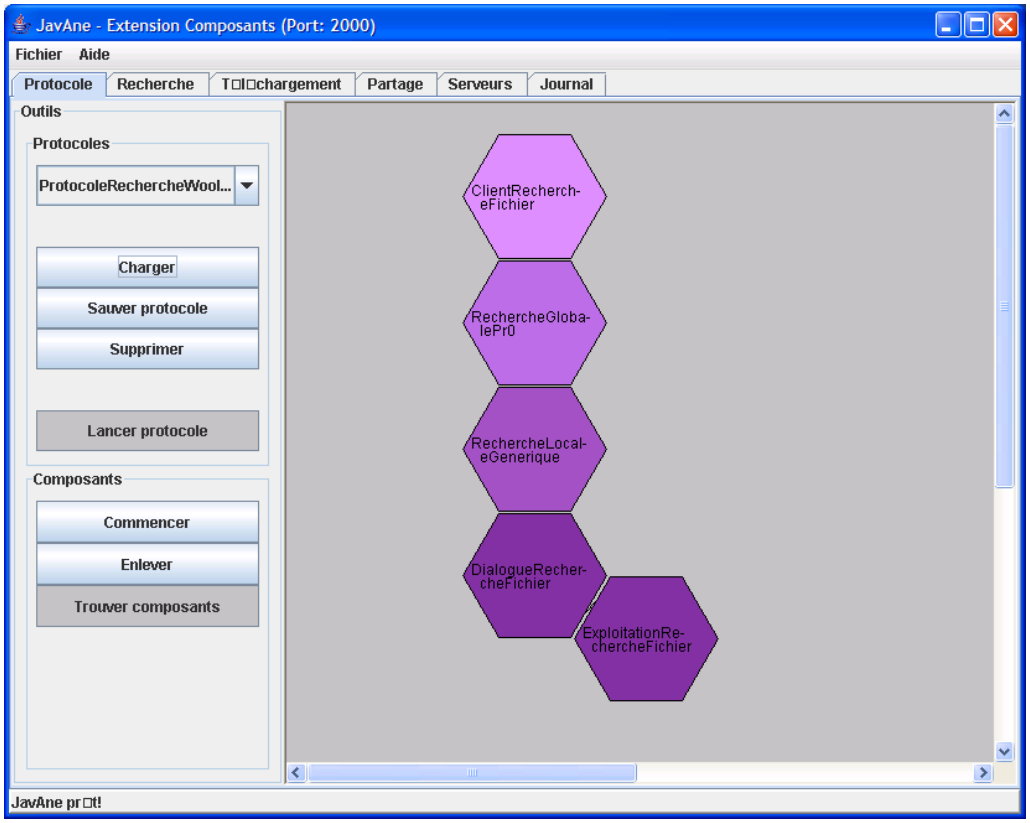


Fig. B.1 – Application du patron de conception au prototype de P2P

En 2006, notre TER[3] a porté sur la réalisation d'un prototype de calcul distribué. Nous avons proposé de concevoir et d'implémenter un prototype de logiciel utilisant les agents mobiles pour déployer des données et un moteur de rendu sur des sites répartis, en pair à pair, d'une part pour utiliser un plus grand nombre de processeurs, d'autre part pour superviser la génération de l'animation.

[3] http://noman.flabelline.com/ter-2006.html

B.3 Plugin JavAct pour Eclipse

Pour obtenir un maximum de simplicité dans la phase de conception et d'implémentation de programmes basés sur JavAct, nous avons proposé dans le cadre d'un travail d'étudiants de 3ème année de l'ENSEEIHT, de réaliser un plugin pour l'environnement de développement *open source* Eclipse[4]. Eclipse est un environnement dit *intégré*, possédant des fonctionnalités pour l'édition et l'exécution de code java (coloration syntaxique, auto-complétion, modifications performantes d'éléments du code source, debugger graphique...) qu'il est possible d'étendre par de très nombreux plugins.

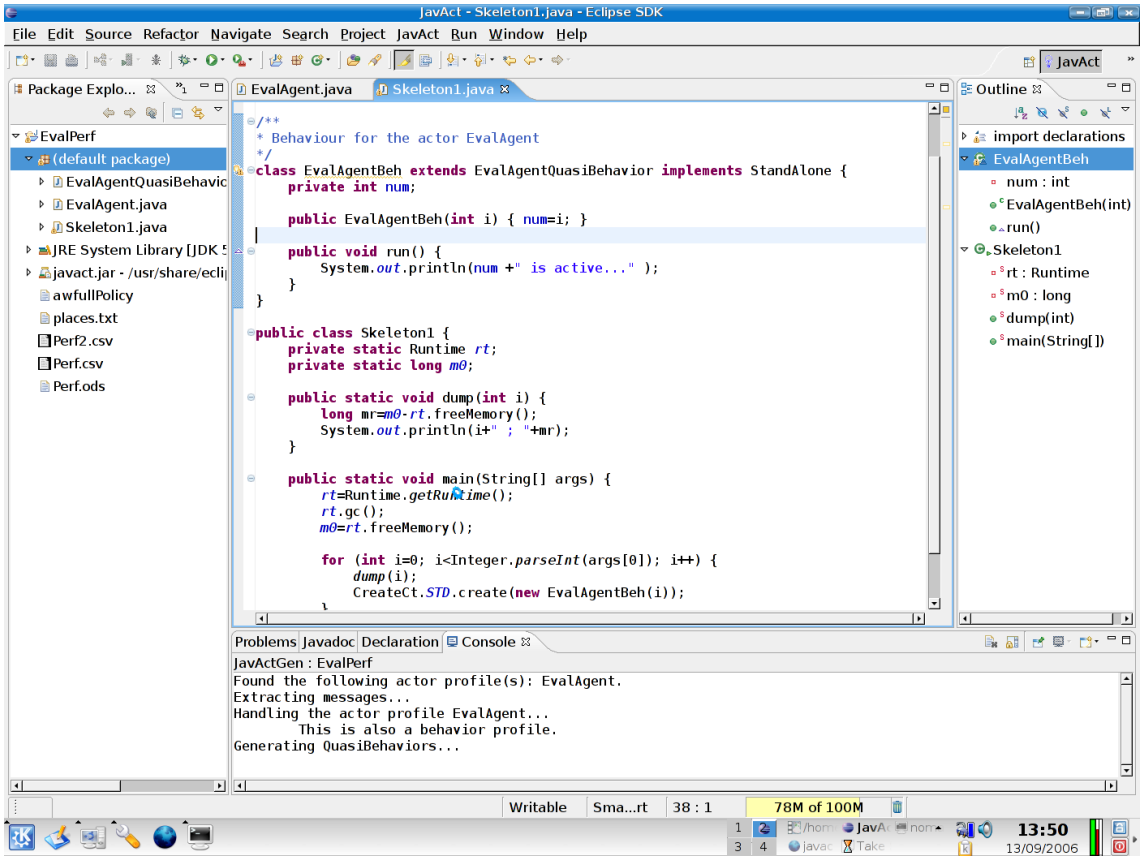


FIG. B.2 – Ecriture du code fonctionnel après génération des classes intermédiaires

Le plugin réalisé[5] permet de développer une application JavAct sans se préoccuper des difficultés inhérentes aux phases de compilation, de génération des classes intermédiaires, etc. De nouveaux boutons dans la perspective JavAct permettent, d'un simple clic, de remplacer les lignes de commande à rallonge, propices aux erreurs, lors de l'exécution d'applications en mode réparti. De plus, la perspective possède un mode permettant de tracer l'exécution des agents (envois de messages, déplacements, créations...).

[4] http://eclipse.org
[5] Le manuel et le plugin sont sur http://www.irit.fr/recherches/ISPR/IAM/JavAct_fr.html.

B.4 Messagerie instantanée mobile

Notre équipe intervient dans certaines formations proposées aux élèves de niveau M2 (ex DESS et DEA universitaires et 3ème année école d'ingénieurs), en particulier autour des technologies de la répartition. J'ai eu le plaisir et la chance de reprendre plusieurs travaux pratiques, dans lesquels j'ai pu mettre en place de nouveaux sujets. Par exemple, en deuxième séance de TP (donc après quelques heures de cours sur le modèle de programmation par agent et un TP introductif à la plateforme) je propose la réalisation d'une application de messagerie instantanée (type MSN/GAIM/Jabber[6]) qui a la particularité d'être mobile. La justification proposée part de l'exemple suivant : en fin de journée, un utilisateur de cette messagerie souhaite continuer une discussion commencée sur son poste de travail fixe, pour la poursuivre sur un support mobile (palm et connexion wifi par exemple) tout en rentrant chez lui, puis encore une fois changer de support pour finir sur son ordinateur fixe personnel (figure B.3).

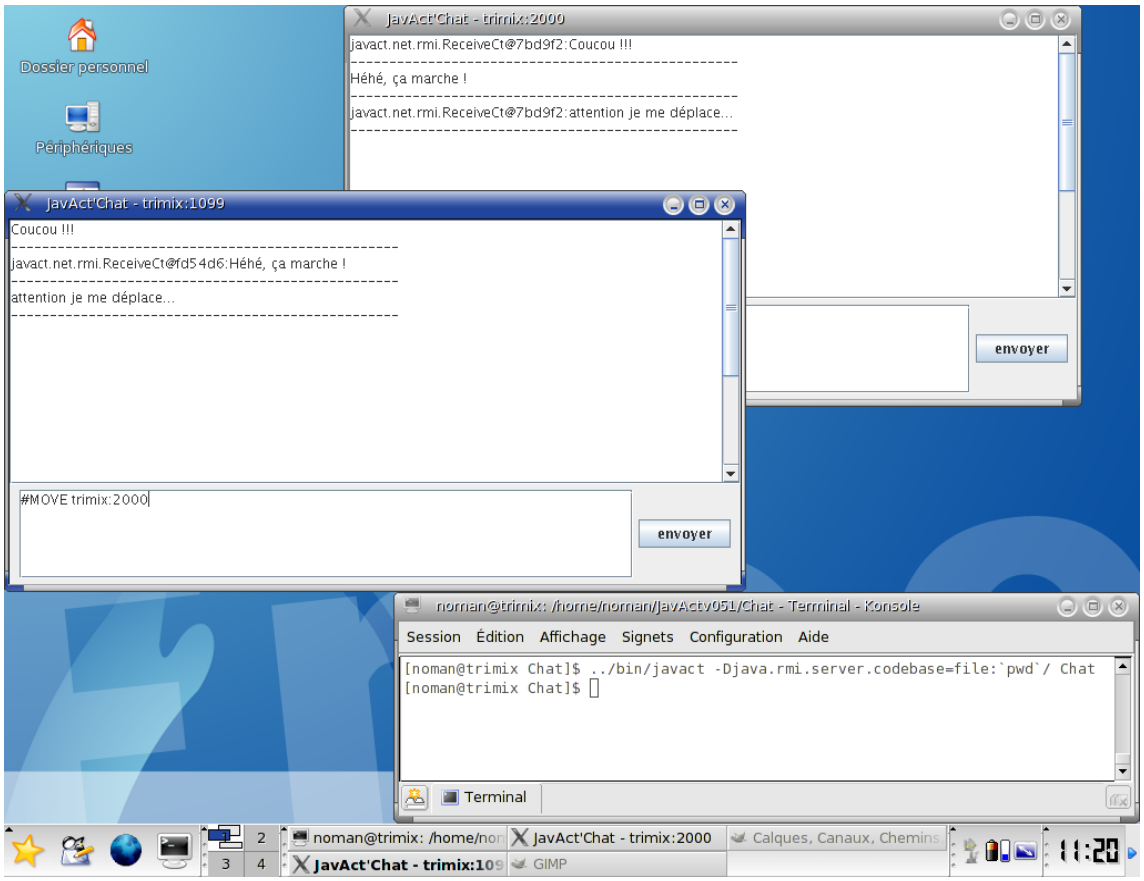


Fig. B.3 – Messagerie mobile en action

Pour continuer de façon transparente une conversation sur les logiciels de messagerie standards, un serveur doit enregistrer toutes les discussions et les retransmettre à chaque nouvelle connexion d'un utilisateur, ce qui nécessite entre autres une duplication

[6] http://gaim.sourceforge.net - http://www.jabberfr.org

des conversations et des capacités de stockage et de transmissions éventuellement importantes (en particulier dans le cas de flux plus complexes mêlant audio, texte et vidéo...). Nous leur proposons d'employer la capacité de mobilité des agents pour supporter le déplacement transparent d'une application, sans avoir recours à un serveur tiers, ainsi que leur capacité de communication asynchrone pour l'échange de données sur le réseau.

En deux heures et en leur mettant à disposition une classe chargée de l'interface homme-machine (la classe `IHM`), les étudiants réalisent la conception, l'implémentation et les tests de leur prototype. Une solution est proposée dans la figure B.4 pour montrer la simplicité du code. Les interactions entre l'IHM et l'agent se font par invocation de méthode classique dans le sens agent→IHM, et par envoi de message dans le sens IHM→agent. Cette technique de déploiement et de mobilité d'applications gérée par des agents peut être appliquée à de nombreux autres exemples.

```
class ChatAgentBeh extends ChatAgentQuasiBehavior implements ChatAgent, StandAlone {
  IHM monIHM;
  Actor corresp;

  ChatAgentBeh() {
    monIHM=new IHM();
  }

  public void run() {
    monIHM.show(ego(),myPlace());
  }

  public void setCorresp(Actor c) {
    corresp=c;
  }

  public void printText(String msg) {
    monIHM.print(msg);
  }

  public void sendText(String msg) {
    if (msg.indexOf("#QUIT")==0) { suicide(); monIHM.dispose(); }
    else if (msg.indexOf("#MOVE")==0) {
      monIHM.dispose();
      go(msg.substring(6));
    }
    else {
      monIHM.print(msg);
      send(new JAMprintText(ego()+":"+msg),corresp);
    }
  }
}
```

Fig. B.4 – Comportement de l'agent mobile conteneur

Bibliographie

[AB04] M. Alfalayleh et L. Brankovic : An overview of security issues and techniques in mobile agents. *In 8th conference on Communications and Multimedia Security (CMS'2004)*, 2004.

[ABM+00] J.-P. Arcangeli, L. Bray, A. Marcoux, C. Maurel et F. Migeon : Réflexivité pour la mobilité dans les systèmes d'acteurs. *In 4ème École d'Informatique des SYstèmes PArallèles et Répartis (ISPAR'2000)*, 2000.

[Agh86] G. Agha : *Actors : a model of concurrent computation in distributed systems.* M.I.T. Press, Cambridge, Ma., 1986.

[Agh02] G. Agha : Adaptive middleware (introduction). *Communications of the ACM*, 45(6), 2002.

[AHL+04] J.-P. Arcangeli, V. Hennebert, S. Leriche, F. Migeon et M. Pantel : JavAct 0.5.0 : principes, installation, utilisation et développement d'applications. Rapport technique IRIT/2004-5-R, IRIT, 2004.

[ALP03] J.-P. Arcangeli, S. Leriche et M. Pantel : JavAne : partage et recherche d'information à base d'agents mobiles adaptables . Rapport de recherche IRIT/2003-25-R, IRIT, Université Paul Sabatier, Toulouse, décembre 2003.

[ALP04] J.-P. Arcangeli, S. Leriche et M. Pantel : Development of Flexible Peer-to-Peer Information Systems using Adaptable Mobile Agents. *In 1st Int. Workshop on Grid and Peer-to-Peer Computing Impacts on Large Scale Heterogeneous Distributed Database Systems (GLOBE'04), Zaragoza, Spain*, pages 549–553. IEEE Computing Society, 30 août-3 septembre 2004.

[ALPre] J.-P. Arcangeli, S. Leriche et M. Pantel : Un framework à composants et agents pour les applications réparties à grande échelle. *Numéro spécial de la revue L'Objet - Agents et Composants*, à paraître.

[AMM01] J.-P. Arcangeli, C. Maurel et F. Migeon : An API for high-level software engineering of distributed and mobile applications . *In 8th IEEE Workshop on Future Trends of Distributed Computing Systems, Bologna (It.)*, pages 155–161, Los Alamitos, Ca., U.S.A., 2001. IEEE-CS Press.

[AP03] P. Amestoy et M. Pantel : Grid-tlse : A web expertise site for sparse linear algebra. *In Sparse days and Grid computing at St Girons workshop*, 2003.

[BBC+06] L. Baduel, F. Baude, D. Caromel, A. Contes, F. Huet, M. Morel et R. Quilici : *Grid Computing : Software Environments and Tools*, chapitre Programming, Deploying, Composing, for the Grid. Springer-Verlag, January 2006.

[BC01] F. Boyer et O. Charra : Utilisation de la réflexivité dans les plateformes adaptables pour applications réparties. *In Revue électronique sur les Réseaux et l'Informatique Répartie*, 2001. ISSN 1262-3261.

[BCHV02] F. Baude, D. Caromel, F. Huet et J. Vayssière : Objets actifs mobiles et communicants. *Revue des sciences et technologies de l'information, série Technique et science informatiques (RSTI-TSI)*, 21(6):823–849, 2002.

[BCTR04] F. Bellefemine, G. Caire, T. Trucco et G. Rimassa : Jade programmer's guide (jade3.2). Rapport technique, TILab S.p.A, 2004.

[Ber96] P. A. Bernstein : Middleware : A model for distributed services. *In Communications of the ACM*, pages 86–97, 1996.

[BGM+99] B. Brewington, R. Gray, K. Moizumi, D. Kotz, G. Cybenko et D. Rus : Mobile agents in distributed information retrieval. *In Intelligent Information Agents*. Springer-Verlag, 1999.

[BI02] G. Bernard et L. Ismail : Apport des agents mobiles à l'exécution répartie. *Revue des sciences et technologies de l'information, série Techniques et science informatiques*, 21(6):771–796, 2002.

[BL01] N. Bouraqadi et T. Ledoux : Le point sur la programmation par aspects. *In Technique et science informatique*, volume 20, 2001.

[BMM02] O. Babaoglu, H. Meling et A. Montresor : Anthill : A framework for the development of agent-based peer-to-peer systems. Rapport technique UBLCS-2001-09, Dept. of Computer Science, Univ. of Bologna, Italy, 2002.

[Boi02] P. Boinot : *Une approche déclarative de la flexibilité du logiciel*. Thèse de doctorat, Université de Rennes, 2002.

[Bra03] L. Bray : *Une plateforme réflexive ouverte pour la gestion d'applications concurrentes réparties à base d'acteurs*. Thèse de doctorat, IRIT - université Toulouse III, 2003.

[Bri04] J.-P. Briot : Agents et composants : une dualité à explorer. transparents de présentation aux Journées Multi-Agents et Composants, JMAC 2004, novembre 2004. http://csl.ensm-douai.fr/MAAC/uploads/3/agents-comp-jmac'04.pdf.

[CA02] D. Chefrour et F. André : Aceel : modèle de composants auto-adaptatifs - application aux environnements mobiles. *In Journées Systèmes à composants adaptables et extensibles*, 2002.

[Car93] D. Caromel : Toward a method of object-oriented concurrent programming. *Communications of the ACM*, 36(9):90–102, 1993.

[Car99] L. Cardelli : Abstractions for mobile computation. *In Secure Internet Programming*, pages 51–94, 1999.

[CDJL02] S. Cazalens, E. Desmontils, C. Jacquin et P. Lamarre : De la gestion locale à la recherche distribuée dans des sources d'informations et de connaissances. *Revue des sciences et technologies de l'information, série L'Objet*, 8(4):47–69, 2002.

[CFH+98] A. Carzaniga, A. Fuggetta, R. S. Hall, A. van der Hoek, D. Heimbigner et A. L. Wolf : A characterization framework for software deployment technologies. Rapport technique CU-CS-857-98, Dept. of Computer Science, University of Colorado, April 1998.

[CGGG03] D. Capera, J.-P. George, M.-P. Gleizes et P. Glize : The amas theory for complex problem solving based on self-organizing cooperative agents. *In Twelfth International Workshop on Enabling Technologies : Infrastructure for Collaborative Enterprises*, 2003.

[CHK94] D. Chess, C. Harrison et A. Kershenbaum : Mobile agents : Are they a good idea ? Rapport technique, IBM Research Division, New York, 1994.

[CSWH00] I. Clarke, O. Sandberg, B. Wiley et T. Hong : Freenet : A distributed anonymous information storage and retrieval system. *In Proc. of the ICSI Workshop on Design Issues in Anonymity and Unobservability.* International Computer Science Institute, 2000.

[DD97] J. Dale et D. C. DeRoure : A mobile agent architecture for distributed information management. *In VIM'97*, 1997.

[DEC04] *DECOR'2004, Actes de la 1re Conférence Francophone sur le Déploiement et la (Re)Configuration de Logiciels*, octobre 2004. ISBN-2-7261-1276-5.

[Déj03] T. Déjean : Développement d'applications multi-agents adaptatifs avec l'api javact. Rapport technique, Institut de Recherche en Informatique de Toulouse, 2003.

[DS03] R. Douence et M. Südholt : Un modèle et un outil pour la programmation par aspects événementiels. *In LMO'2003*, 2003.

[dSeSEK03] F. da Silva e Silva, M. Endler et F. Kon : Developing adaptive distributed applications : a framework overview and experimental results. *In International symposium on Distributed Objects and Application, Catania, Sicile*, 2003.

[Dun01] C. R. Dunne : Using mobile agents for network resource discovery in peer-to-peer networks. *In SIGecom Exchanges*, 2001.

[Fer95] J. Ferber : *Les systèmes multi-agents - Vers une intelligence collective.* InterEditions, 1995.

[FG99] Y. Fan et S. Gauch : Adaptive agents for information gathering from multiple sources. *In AAAI Symposium on Agents in Cyberspace*, pages 40–46, 1999.

[FK98] I. Foster et C. Kesselman : *The Grid : Blueprint for a New Computing Infrastructure.* Morgan Kaufmann, San Francisco, California, 1998.

[FPV98] A. Fuggetta, G. Picco et G. Vigna : Understanding code mobility. *IEEE Transactions on Software Engineering*, 24(5):342–361, 1998.

[GBV06] G. Grondin, N. Bouraqadi et L. Vercouter : Assemblage automatique de composants pour la construction d'agents avec madcar. *In 2ème journée Multi-Agent et Composant (JMAC'06)*, 2006.

[GF00] O. Gutknecht et J. Ferber : Madkit : a generic multi-agent platform. *In Autonomous Agents (AGENTS 2000)*, 2000.

[GHJV94] E. Gamma, R. Helm, R. Johnson et J. Vlissides : *Design Patterns - Elements of Reusable Object-Oriented Software*. Addison Wesley Professional Computing Series, 1994.

[GHJV95] E. Gamma, R. Helm, R. Johnson et J. Vlissides : *Design patterns : elements of reusable object-oriented software*. Addison-Wesley Professional, 1995.

[Gue98] Z. Guessoum : Dima : Une plate-forme multi-agents en smalltalk. *revue L'Objet*, 3(4):393–410, 1998.

[Hew77] C. Hewitt : Viewing control structures as patterns of passing messages. *Artificial Intelligence*, 8(3):323–364, 1977.

[HF00] J. Hammer et J. Fiedler : Using mobile crawlers to search the web efficiently. *Int. Journal of Computer and Information Science*, 1(1):36–58, 2000.

[HHP04] A. Hurault, V. Hennebert et M. Pantel : Répartition et mobilité en JavAct : une approche dérivée d'un modèle formel. *Langages et Modèles à Objets LMO'2004, Revue des sciences et technologies de l'information, série L'Objet*, 10(2-3):47–60, 2004.

[HHvdHW97] R. Hall, D. Heimbigner, A. van der Hoek et A. Wolf : The software dock : A distributed, agent-based software deployment system. Rapport technique CU-CS-832-97, Department of Computer Science, University of Colorado, 1997.

[HKFM04] S. Handurukand, A.-M. Kermarrec, F. L. Fessant et L. Massoulié : Exploiting semantic clustering in the edonkey p2p network. *In SIGOPS European Workshop*, pages 109–114, 2004.

[Hoa78] C. A. R. Hoare : Communicating sequential processes. *Commun. ACM*, 21(8):666–677, 1978.

[JA04] M.-W. Jang et G. Agha : On efficient communication and service agent discovery in multi-agent systems. *In 3rd International Workshop on Software Engineering for Large-Scale Multi-Agent Systems (SELMAS 2004)*, pages 27–33, 2004.

[JK00] W. Jansen et T. Karygiannis : Nist special publication 800-19 - mobile agent security. Rapport technique, National Institute of Standards and Technology, 2000.

[JSM99] A. Joshi, M. Singh et M. Ma : Special section on multiagent systems on the net and agents in e-commerce. *Communications of the ACM*, 42(3), 1999.

[KDB01] H. Kosch, M. Doller et L. Boszormenyi : Content-based indexing and retrieval supported by mobile agent technology. *In Second International Workshop on Multimedia Databases and Image Communication*, pages 152–166, 2001.

[KdRB91] G. Kickzales, J. des Rivières et D. Bobrow : *The Art of the Metaobject Protocol*. M.I.T. Press, Cambridge, Ma., 1991.

[KLM+97] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. V. Lopes, J. Loingtier et J. Irwin : Aspect-oriented programming. *In ECOOP 1997 - LNCS 1241*, 1997.

[LA04a] S. Leriche et J.-P. Arcangeli : Une architecture pour les agents mobiles adaptables. *In Actes des Journées Composants JC'2004*, 2004.

[LA04b] S. Leriche et J.-P. Arcangeli : Déploiement et adaptation de systèmes p2p : une approche à base d'agents mobiles et de composants. *In Actes des Journées Multi-Agents et Composants*, 2004.

[LA05] S. Leriche et J.-P. Arcangeli : Apports mutuels des technologies P2P, agents mobiles et composants logiciels. *In Actes du workshop OCM-SI'05*, 2005.

[LA06] S. Leriche et J.-P. Arcangeli : Vers un modèle d'agent flexible. *In Actes des Journées Multi-Agent et Composant JMAC'2006*, 2006.

[LAP04] S. Leriche, J.-P. Arcangeli et M. Pantel : Agents mobiles adaptables pour les systèmes d'information pair à pair hétérogènes et répartis . *In Nouvelles Technologies de la Répartition, NOTERE 2004 , Saidia (Ma.)*, pages 29–43. CIISE - ENST - Univ. Mohammed 1er, 27-30 juin 2004.

[Ler03] S. Leriche : Techniques adaptatives pour la mobilité d'agents. Rapport de stage DEA, IRIT, Université Paul Sabatier, Toulouse, juin 2003.

[Lhu98] M. Lhuillier : *Une approche à base de composants logiciels pour la conception d'agents. Principes et mise en œuvre à travers la plate-forme Maleva.* Thèse de doctorat, Université Paris 6, 1998.

[LM98] D. B. Lange et O. Mitsuru : *Programming and Deploying Java Mobile Agents Aglets.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1998.

[LMR00] S. Loureiro, R. Molva et Y. Roudier : Mobile code security. *In ISYPAR 2000 (4ème Ecole d'Informatique des Systèmes Parallèles et Répartis)*, 2000.

[LPP04] S. Lacour, C. Pérez et T. Priol : Deploying corba components on a computational grid : General principles and early experiments using the globus toolkit. *In Component Deployment : Second International Working Conference (CD 2004), LNCS 3083*, pages 35–49. Springer-Verlag, 2004.

[Mae87] P. Maes : Concepts and experiments in computational reflection. *In OOPSLA '87 : Conference proceedings on Object-oriented programming systems, languages and applications*, pages 147–155. ACM Press, NY, USA, 1987.

[Mat01] F. Mattern : The vision and technical foundations of ubiquitous computing. *Upgrade*, 2(5):2–6, octobre 2001.

[McA95] J. McAffer : Meta-level programming with coda. *In Proceedings of the European Conference on Object-Oriented Computing (ECOOP), LNCS 952*, pages 190–214. Springer-Verlag, 1995.

[Med96] N. Medvidovic : Adls and dynamic architecture changes. *In Proceedings of the Second International Software Architecture Workshop (ISAW-2)*, pages 24–27, 1996.

[Mig99] F. Migeon : *Etude et implantation de mécanismes réflexifs dans un langage concurrent.* Thèse de doctorat, Université Paul Sabatier, Toulouse III, 1999.

[MKL+02] D. Milojicic, V. Kalogeraki, R. Lukose, K. Nagaraja, J. Pruyne, B. Richard, S. Rollins et Z. Xu : Peer-to-peer computing. Rapport technique HPL-2002-57, HP Laboratories Palo Alto, 2002.

[MMKA04] T. Mine, D. Matsuno, A. Kogo et M. Amamiya : Agent community based peer-to-peer information retrieval. *In Workshop CIA-2004 on Cooperative Information Agents*, 2004.

[MMMS98] A. Marcoux, C. Maurel, F. Migeon et P. Sallé : Generic operational decomposition for concurrent systems : semantics and reflection. *Parallel and Distributed Computing Practices*, 1(4):49–64, 1998.

[MP01] R. Marvie et M.-C. Pellegrini : Modèles de composants, un état de l'art. *RSTI-Série L'Objet*, 8(3):61–89, 2001.

[MP02] R. Marvie et M.-C. Pellegrini : Modèles de composants, un état de l'art. *Revue des sciences et technologies de l'information, série L'Objet*, 8(3):61–89, 2002.

[Ous05] M. Oussalah : *Ingenierie des Composants : Concepts, techniques et outils.* Vuibert Informatique, 2005. Ouvrage collectif.

[PK02] B.-H. Park et H. Kargupta : Distributed data mining : Algorithms, systems, and applications. *In Data Mining Handbook*, pages 341–358. IEA, 2002.

[PSDF01] R. Pawlak, L. Seinturier, L. Duchien et G. Florin : Jac : A flexible framework for aop in java. *In REFLECTION'01*, 2001.

[PSP00] S. Papastavrou, G. Samaras et E. Pitoura : Mobile Agents for World Wide Web Distributed Database Access. *IEEE Transactions on Knowledge and Data Engineering*, 12(5):802–820, 2000.

[RFH+01] S. Ratnasamy, P. Francis, M. Handley, R. Karp et S. Shenker : A scalable content-addressable network. *In Proceedings of the ACM SIGCOMM '01 Conference*, pages 161–172, 2001.

[RG95a] A. S. Rao et M. P. Georgeff : BDI-agents : from theory to practice. *In Proceedings of the First Intl. Conference on Multiagent Systems*, San Francisco, 1995.

[RG95b] A. S. Rao et M. P. Georgeff : Bdi agents : From theory to practice. Rapport technique 56, Australian Artificial Intelligence Institute, Melbourne, Australia, 1995.

[RHC04] R. R. Rao, S. Hewavitharana et W. Choo : Security issues in mobile agents and their solutions. Rapport technique, National University of Singapore, 2004. http://www.comp.nus.edu.sg/~cs4274/termpapers/0304-I/group5/paper.pdf.

[RMS01] J. Routier, P. Mathieu et Y. Secq : Dynamic skills learning : A support to agent evolution. *In AISB'O1, Symposium on Adaptive Agents and Multi-agent Systems*, 2001.

[Rou05] S. Rougemaille : MdlAct : un langage de modélisation acteur. Rapport de stage DEA, IRIT, Université Paul Sabatier, Toulouse, juin 2005.

[Sat04] I. Satoh : Self-deployment of distributed applications. *In FIDJI*, pages 48–57, 2004.

[Sch01] B. Schneier : *Cryptographie appliquée, 2e édition.* Vuibert, Paris, 2001. ISBN : 2-7117-8676-5.

[SdCdV01] P. Samarati et S. de Capitani di Vimercati : Access control : Policies, models, and mechanisms. *In FOSAD'2000 - LNCS 2171*, pages 137–196, 2001.

[Smi82] B. Smith : *Reflection and Semantics in a Procedural Programming Language.* Thèse de doctorat, MIT, Boston, MA, US, January 1982.

[SMK+01] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek et H. Balakrishnan : Chord : A scalable peer-to-peer lookup service for internet applications. *In Proceedings of the ACM SIGCOMM '01 Conference*, pages 149–160, 2001.

[Szy02] C. Szyperski : *Component Software - Beyond Object-Oriented Programming.* Addison-Wesley / ACM Press, 2002.

[TCA01] P. Thati, P.-H. Chang et G. Agha : Crawlets : Agents for high performance web search engines. *In Mobile Agents 2001, LNCS 2240*, pages 119–134. Springer-Verlag, 2001.

[Tsc99] C. Tschudin : Mobile agent security. *In Intelligent Information Agents.* Springer-Verlag, 1999.

[Ver04] L. Vercouter : Mast : Un modèle de composant pour la conception de sma. *In Actes des Journées Systèmes Multi-Agents et Composants*, 2004.

[Woo02] M. Wooldridge : *Introduction to MultiAgent Systems.* John Wiley and Sons, 2002.