MEMOIRE DE MASTER 2 RECHERCHE
DE L'UNIVERSITE MONTPELLIER II
Spécialité : Informatique CODA

par

Stéphanie FOUCART

Sujet :

# Modèle de composants pour le développement d'agents logiciels. MALICE

Soutenu le 1er Septembre 2006
Encadré par :

Mme Christelle URTADO
M. Sylvain VAUTTIER

# RÉSUMÉ

Pour faire face à la complexité et à l'aspect distribué grandissant des applications modernes, nous avons étudié la possibilité de combiner les avantages de deux paradigmes de programmation afin de pallier aux lacunes des modèles classiques : la programmation par composants et les systèmes multi-agents. Les systèmes multi-agents sont des sociétés d'entités actives organisées, capables de percevoir leur environnement et d'agir sur celui-ci ainsi que d'interagir (collaborer, négocier) avec les autres agents. La programmation à base de composants consiste en l'assemblage de briques logicielles réutilisables visant à développer des applications distribuées multi-tiers .

Nous avons prêté une attention particulière à certaines caractéristiques de ces modèles qui nous ont paru les plus intéressantes pour répondre aux besoins de l'informatique d'aujourd'hui, à savoir la réutilisation, l'adaptabilité du système, la distribution des composants, l'organisation sociale présente dans les SMA et l'autonomie des agents.

Notre participation consiste en la conception et l'implémentation au dessus de Fractal, d'un modèle de composants actifs MALICE qui présentent ces caractéristiques. Autrement dit, notre modèle propose de travailler sur des entités qui se définissent et se manipulent comme des composants mais qui se comportent comme des agents. Nous avons eu à l'esprit de proposer une plate-forme générique, indépendante de l'application à implémenter. L'intégration de ces paradigmes apportera selon nous de nombreux avantages en matière de performance, de fiabilité, de mobilité, de maintenance et de déploiement aux applications distribuées.

*Mots clés : composant, agent, SMA, Fractal*

# ABSTRACT

To face the complexity and the distributed aspect increasing in nowadays applications, we studied the possibility of combining the advantages of two programming paradigms as a solution to overcome the shortcomings of the classic models: component-based programming and multiagent systems. Multiagent systems are societies of organized and active entities, able to collect information from their environment and act on it as well as interact (collaborate, negotiate) with other agents. Component-based programming consists of the assembly of reusable software elements aiming at developing distributed applications (multi-tier architecture).

We paid particular attention to some characteristics of these models which appeared according to us as the most capable to meet the needs of the modern software engineering, basicaly reusability, adaptability, distribution, social organization found in the SMA and autonomy of the agents.

Our contribution consists of a model based on active components , MALICE, that is, entities which define themselves and can be handle as components but which behave as agents. Actually, we propose a generic platform, based on the component model FRACTAL, independent of the applications to be implemented. The integration of these paradigms will bring, according to us, advantages in terms of performance,  reliability, mobility, maintenance and deployment of distributed applications.

***KEY WORDS : COMPONENT, AGENT, MAS, FRACTAL***

# TABLE DES MATIÈRES

## Table des matières

Résumé.......... 2
Abstract.......... 3
Table des Matières.......... 4
Table des Figures, Schémas et Diagrammes.......... 6
Introduction.......... 7
Motivations et objectifs.......... 7
Démarche suivie.......... 8
PARTIE 1 : Etat de l'art.......... 9
I. Modèles de composants.......... 10
A. Evolution du modèle objet.......... 10
1) Définition .......... 10
2) Caractéristiques.......... 11
a) Architecture.......... 11
b) Gestion des aspects non fonctionnels.......... 11
c) Développement simple et rapide.......... 11
B. Différents modèles.......... 12
II. Modèles d'agents.......... 15
A. Généralités.......... 15
1) Définitions.......... 15
2) Propriétés.......... 15
3) Types d'agent.......... 16
4) Organisation.......... 16
B. Etat de l'art .......... 17
1) Domaines de recherches.......... 17
2) Plates-formes.......... 18
3) FIPA.......... 19
a) Caractéristiques fondamentales des agents .......... 20
b) Caractéristiques fondamentales de la plate-forme .......... 20
PARTIE 2 : Spécification de MALICE (Modèle d'agents logiciels implémenté grâce à des composants étendus).......... 22
III. Spécification du modèle.......... 23
A. Postulat.......... 23
B. Plate-forme.......... 23
1) Généralités.......... 23
2) Protocoles de communication.......... 25
3) Diffusion des messages.......... 25
C. Architecture d'un agent.......... 25
1) Généralités.......... 25
2) Schéma interne.......... 26
3) Architecture interne.......... 27
D. Organisation.......... 29
1) Organisation comportementale (interaction et coordination).......... 29
2) Organisation structurelle.......... 30

3) Propositions....32
E. Respect des normes FIPA....35
PARTIE 3 : Réalisations.... 36
IV. Implémentation....37
A. Fractal ....37
B. JMS....39
C. Implémentations.... 40
V. Expérimentation....43
A. Implementation.... 43
B. Déploiement.... 44
IV. Comparaison....45
Conclusion et Perspectives....46
Références bibliographiques .... 47
Annexe 1 : Spécification FIPA....50
Annexe 2 : Plate-forme....52
Annexe 3 : Agent....53
Annexe 4 : Configurations....54
1 Creating the component types....58
2 Creating the component templates....58
3 Instantiating and launching the application.... 60

# Table des Figures, Schémas et Diagrammes

Figure 1 : Plate-forme....22
Figure 2 : Bus de communications....22
Figure 3 : Réseau informatique....28

Schéma 1 : Schéma interne de l'agent....25
Schéma 2 : Architecture d'un agent....26
Schéma 3 : Composant Fractal (source : http://fractal.objectweb.org)....36
Schéma 4 : Présentation JMS (source : http://java.sun.com)....37
Schéma 5 : Queue JMS (source : http://java.sun.com)....38
Schéma 6 : Topic JMS (source : http://java.sun.com)....38

Diagramme 1 : Inscription sur la plate-forme....30
Diagramme 2 : Recherche simple de service....31
Diagramme 3 : Recherche complexe de service....31
Diagramme 4 : Notation....32

# INTRODUCTION

## Motivations et objectifs

Les systèmes multi-agents sont des sociétés d'entités actives, capables de percevoir leur environnement et d'agir sur celui-ci, dont les membres peuvent interagir. Ils ont initialement été développés pour répondre à des besoins dans le domaine de la résolution de problèmes (intelligence artificielle distribuée) ou la simulation (écosystèmes, systèmes sociaux, ...). Or avec l'essor de l'informatique, tant au niveau hardware - augmentation de la puissance de calcul, généralisation des réseaux - que logiciel - systèmes distribués, hétérogènes, augmentation du nombre et de la qualité des fonctionnalités proposées -, la complexité des applications devient telle que les techniques « classiques » ne suffisent plus. L'une des réponses aujourd'hui consiste en la distribution des tâches et l'amélioration des performances par la coordination ou la compétition des processus. Il nous a ainsi étonné que le paradigme agent ait jusqu'à présent été peu exploité en terme de génie logiciel.

Cependant la programmation à base de composants - briques logicielles réutilisables visant à développer des applications distribuées multi-tiers - peut se rapprocher par certains aspects des modèles à agents. Des plates-formes performantes basées sur certains modèles existent déjà et sont assez largement utilisées.

Bien qu'étant deux paradigmes distincts ils proposent tous deux des solutions pour le développement d'applications complexes. Nous avons prêté une attention particulière à certaines caractéristiques de ces modèles qui nous ont parus les plus intéressantes pour répondre aux besoins de l'informatique moderne, à savoir :

–*La réutilisation* : De nombreuses bibliothèques de composants métiers existent d'ores et déjà. Elles sont en général fiables et accélèrent le développement des applications.

–*L'adaptation*: Grâce au découplage possible des composants et a fortiori des agents entre les services qu'ils proposent et leur implémentation, les applications peuvent évoluer dans le temps de façon transparente. Les agents pouvant être mobiles, on peut également imaginer des systèmes dynamiques où les agents peuvent se déplacer – et ainsi modifier le comportement du système - en fonction de la charge (reconfiguration).

–*La distribution* : Agents et composants peuvent être déployés de façon distribuée et effectuer des traitements parallèles. De plus les agents agissent par nature de façon asynchrone, puisque leur mode de communication est traditionnellement l'envoi de messages abstraits et non l'invocation de méthodes, nous reviendrons sur ce point plus tard. Enfin les agents sont situés, c'est à dire qu'ils n'ont qu'une représentation locale de leur environnement, on quitte le modèle client/serveur classique, puisque l'information est répartie sur le réseau..

–*L'organisation* : Les systèmes multi-agents peuvent supporter différents modes d'interaction telle que la concurrence, la collaboration ou la négociation grâce à l'organisation de leur société

–*L'autonomie* : Les agents contrôlent leurs propres actions ainsi que leur état interne

Nous avons ainsi voulu concevoir un modèle de composants actifs qui présentent toutes ces caractéristiques, c'est-à-dire des entités qui se définissent et se manipulent comme des composants mais qui se comportent comme des agents. On peut également voir cela comme un modèle de composants destiné au développement d'agents. Durant ce projet, nous nous sommes donc attelés à

définir et implémenter un tel modèle. Il pourra servir de base au développement d'applications utilisant aujourd'hui déjà ce paradigme (simulation, IA) en proposant une plate forme générique et standardisée, mais pourra également servir dans toutes autres applications de génie logiciel où les agents trouvent leur place (principalement systèmes distribués ou embarqués, mais aussi applications de gestion, jeux...), en effet, nous avons eu à l'esprit de proposer une plate-forme générique, indépendante de l'application à implémenter.

L'intégration de ces paradigmes apportera selon nous de nombreux avantages en matière de performance, de fiabilité, de mobilité, de maintenance et de déploiement aux applications distribuées.

**Evolution des modèles de programmation**

De plus, la programmation par agents, s'inscrit dans la suite logique de l'évolution des modèles de programmation. Elle nous paraît être l'une des voies de la programmation du futur, c'est pourquoi elle requiert dès aujourd'hui des outils et un formalisme unifié.

## Démarche suivie

Pour la rédaction de ce document, nous avons eu une démarche en plusieurs temps. Elle a d'abord consisté en une étude bibliographique sur la notion de composants (concurrence, asynchronisme, message-oriented middleware) et de plates-formes à agents afin de cerner les hybridations possibles entre ces deux paradigmes de programmation.

Un deuxième temps a consisté à définir un modèle de composants permettant le développement et l'utilisation d'agents. Par le paradigme agent, nous avons conçu l'architecture de plus haut niveau. Pour ce qui est de la conception des mécanismes internes, nous sommes partis de l'existant, c'est à dire les modèles de composants pour arriver à un modèle permettant de créer des agents.

Une maquette du modèle a aussi été réalisée ainsi que des programmes de test de la plate-forme, afin de valider le modèle.

**Le plan de ce document est calqué sur cette démarche.**

# PARTIE 1 : ETAT DE L'ART

[BOO94] énonce les caractéristiques nécessaires d'un paradigme d'ingénierie logicielle permettant de construire des systèmes complexes :

- la décomposition - la capacité de diviser le problème en sous-problèmes plus simples et pouvant être traités d'une façon relativement autonome
- l'abstraction - la capacité de définir un modèle simplifié du système
- l'organisation - le procédé consistant à identifier et à gérer les rapports entre les divers sous-problèmes à résoudre. La possibilité de spécifier et d'établir des relations organisationnelles entre composants aide les concepteurs à faire face à la complexité.

Ces trois caractéristiques peuvent être retrouvées à la fois dans les modèles multi-agents et componentiels.

# I. Modèles de composants

## A. Evolution du modèle objet

Les langages à objets ont été une révolution pour le génie logiciel. Sans revenir sur le concept objet, nous voulons souligner différents aspects intéressants par rapport aux modèles antérieurs (notamment procéduraux) :.

-Réutilisation des objets (spécialisation, héritage)

-Conception des applications simplifiée en proposant une abstraction (regroupement code et données au sein des objets)

-Prise en compte l'environnement à l'exécution (pas seulement à la compilation).

Les modèles de composants peuvent être vu comme une évolution des langages à objets, améliorant les points cités précédemment. Grâce aux composants, on aspire à une granularité encore supérieure, la liaison se veut plus tardive (appel de méthode par envoi de message au composant proposant le service), la réutilisation se voit accrue grâce aux bibliothèques de composants.

### 1) Définition

Lorsque l'on parle de composants, il s'agit d'unités logicielles autonomes contenant du code applicatif, généralement regroupé en bibliothèque en fonction du domaine – ou métier – auquel il se rapporte (gestion, comptabilité...). Un composant est censé fournir un service bien précis. Les fonctionnalités qu'il encapsule doivent être en rapport et cohérentes entre elles.

Lors de l'édition de 1996 de la conférence européenne sur les programmes orientés objets (ECOOP) [COI96] la définition suivante a été retenue:

*Un composant logiciel est une unité de composition dotée d'interfaces spécifiées. Un composant logiciel peut être déployé indépendamment et sujet à une composition par une tierce entité .*

Un composant regroupe un certain nombre de fonctionnalités qui peuvent être appelées depuis l'extérieur. C'est pourquoi, pour pouvoir être utilisé, le composant doit fournir une interface, c'est-à-dire l'ensemble des signatures des fonctions accessibles depuis le programme ou composant client. Comme le composant n'est accessible que par ses interfaces, il n'est a priori pas possible de savoir

de quelle manière elles sont implémentées, il existe donc un découplage fort entre l'implémentation et l'appel grâce à cette indirection. L'implémentation du composant n'a d'influence que sur le comportement du composant : si les signatures de l'interface de 2 composants sont identiques, ils sont interchangeables.

Enfin, un composant doit être réutilisable, c'est à dire qu'il n'a pas pour vocation de servir uniquement au projet pour lequel il a été développé. Cet aspect n'est possible qu'à la condition qu'il possède un comportement suffisamment général. Trouver un compromis entre la spécialisation du composant pour optimiser son utilisation dans le cadre du projet actuel, et sa généralisation en vue de sa réutilisation.

## *2) Caractéristiques*

### a) Architecture

Les composants sont définis via des Interfaces requises ou fournies, ainsi les composants sont interchangeables s'ils ont la même interface. Dans la plupart de modèles de composants, un middleware de communication (type RMI) est utilisé pour pouvoir déployer les composants sur des machines distantes. La communication subit le plus souvent des indirections afin de permettre la distribution et une évolution facile de l'implémentation.

Au delà des composants, ces modèles proposent en général une plate-forme permettant aux composants de s'échanger leurs services, et proposant elle même des fonctions utilitaires que nous allons décrire.

### b) Gestion des aspects non fonctionnels

Les plates-formes appelées également conteneurs gèrent de nombreux aspects non fonctionnels, qui peuvent varier en fonction du modèle considéré. On peut citer pour exemple la sécurité, la communication avec la base de données, les utilisateurs, etc. Certains proposent aussi de gérer les files d'attente et les pools de composants pour améliorer les performances de l'application.

La gestion du déploiement est un cas particulier de ces aspects non fonctionnels. Ces outils permettent de mettre en place le système de façon distribuée, de faciliter la gestion et l'administration de l'application – modification de la charge sur les nœuds, redéploiement, changement de configuration - et enfin de faciliter la mise à jour des composants ou le changement de version. Cette fonctionnalité est devenue indispensable pour la gestion des applications modernes.

### c) Développement simple et rapide

La programmation orientée composants consiste à utiliser une approche modulaire au niveau de l'architecture d'un projet informatique, ce qui permet d'assurer au logiciel une meilleure lisibilité et une meilleure maintenabilité. Les développeurs, au lieu de créer un exécutable monolithique, se servent de briques réutilisables (bibliothèques de composants)

La programmation par composants est particulièrement pertinent pour le travail en équipe et permet d'industrialiser la création de logiciels.

Les avantages à utiliser une approche orienté composant pour conduire un projet sont multiples :

–*Spécialisation* : L'équipe de développement peut-être divisée en sous-groupes fonctionnels, chacun

se spécialisant dans le développement d'un composant

–*Sous-traitance* : Le développement d'un composant peut-être externalisé, à condition d'en avoir réalisé les spécifications au préalable

–*Facilité de mise à jour* : La modification d'un composant ne nécessite pas la modification du projet complet

–*Facilité de livraison/déploiement* : Dans le cas d'une mise à jour, la livraison est facilitée, puisqu'il n'y a pas besoin de livrer de nouveau l'intégralité du projet, mais seulement le composant modifié

–*Productivité* : La réutilisabilité d'un composant permet un gain de productivité non négligeable car elle diminue le temps de développement, d'autant plus que le composant est réutilisé souvent

–*Abstraction* : La granularité des composants étant supérieure à celle des objets, le niveau d'abstraction augmente et la complexité diminue

–*Architecture n-tiers* : Ce type d'architecture facilite la conception des application et la séparation des différentes parties du code

–*Conception/modélisation* :Des outils de conception et de modélisation existent déjà pour la programmation orientée composants, à noter que la spécification d'UML 2.0 proposent des diagrammes de composition et de déploiement qui facilitent le développement. Il existe également des outils permettant de générer des composants directement à partir du modèle UML (Model Driven Architecture) - par exemple AndroMDA pour Maven génère des EJB.

De plus, un certains nombre d'implémentations proposent des outils de programmation visuelle comme les JavaBean ou Fractal.

## B. Différents modèles

Voici à présent une liste non exhaustive des principaux modèles de composants. Il s'agit de modèles possédant des implémentations. Ils se veulent assez polyvalents et adaptables, c'est à dire qu'ils ne sont pas conçus pour un type particulier d'application. Nous nous intéressons particulièrement à ceux ci car ils peuvent servir de base à notre projet.

–Le modèle ***JavaBean*** est le plus ancien des modèles à composants et a été développé par Sun. Les JavaBean sont persistants grâce à la sérialisation, communiquent grâce à des événements attrapés par les listeners introduit par java 1.1, on a accès aux attributs, méthodes et événements sans avoir le source et ils sont paramétrables. Leurs formats et méthodes sont normalisés dans le formalisme Java et peuvent se déployer dans la machine virtuelle standard. C'est le type de composants le plus adapté est le composant visuel. D'ailleurs, les composants des classes A.W.T. et Swing pour la création d'interfaces graphiques sont tous des JavaBeans.

–Les ***EJB*** (Enterprise Java Bean) bien qu'étant également développés par Sun n'ont pas grand chose à voir avec les JavaBeans. Leur objectif est faciliter l'intégration d'architecture multi-tiers client/serveur/traitement applicatif/base de données, ainsi que le déploiement et la maintenance d'applications. Ils nécessitent un serveur d'application pour se déployer. Il existe 3 types d'EJB, les EntityBean gérant la persistance et l'accès aux bases de données, les SessionBean contenant le code métier et le MessageDrivenBean qui autorisent la communication asynchrone. Il gère les pools d

composant. C'est l'un des modèles les plus complets du marché.

–***COM+*** est une version avancée de COM qui regroupe l'ensemble des services dédiées au applications distribuées de Windows, à savoir : COM, DCOM, MTS, MSMQ. En plus d'agréger ces services, COM+ apporte de nouvelles fonctionnalités comme les queued components, un mécanisme de gestion des évènements (publish/subscribe) ou la répartition de charge (load balancing). COM+ est apparu avec Windows 2000. C'est l'un des leader du marché avec les EJB.

–***CCM*** (Corba Component Model) est un modèle pour définir un composant dans une architecture distribuée CORBA. Il intègre des interfaces pour la configuration, la définition de la composition des composants et un modèle pour le déploiement. L'avantage de s'intégrer dans une architecture CORBA est d'être indépendant de la plate forme et du langage – il est même compatible EJB. Il possède une référence, 4 types de port - la facette (interface fournie), le réceptacle ( permet exploiter certains objets ), la source (elle diffuse un événement ) et le puit d'évènements( reçoit des événements après s 'être abonné ) - et d'attributs

–***SOFA*** et ***ACME*** sont des modèles de composants académiques, cependant la documentation sur ces systèmes est assez réduites.

–***Fractal*** est un modèle très puissant, permettant la composition hiérarchique. C'est une Plate forme assez minimaliste par rapport aux modèles commerciaux car l'accent est moins mis sur la performance que sur la cohérence du modèle. De plus, les sources des implémentations sont disponibles. Nous reviendrons plus en détail sur ce modèle dans la partie « Réalisation » car c'est elle que nous avons choisi comme base à notre implémentation.

De nombreux projets plus marginaux existent, mais nous ont paru moins intéressants, ils peuvent être classés en 3 familles :

–modèles idéaux, seul la spécification à été étudiée, aucune solution pour l'implémentation ou l'exploitation n'a été envisagé. Il s'agit de traités très généraux et difficilement exploitable concrètement. Il paraît évident qu'il serait intéressant d'avoir un modèle de composants efficaces, totalement réutilisables, auto adaptables, communiquant entre eux de facon transparente mais si aucune solution technique n'est proposée, on reste dans le domaine théorique. On préférera se rapprocher de traiter dans ce genre concernant les modèles d'agent.

–Modèles très spécifiques, utilisable dans un type particulier d'application ou dans un domaine spécifique

–Adaptation ou évolution des modèles précédemment cités visant à parfaire un point technique particulier

Quant aux implémentation des modèles présentés ci dessus, beaucoup se sont révélées inexploitables pour nous pour différentes raisons. La première est l'impossibilité d'accéder aux sources, ce qui est rédhibitoire dans notre cas. De plus dans la plupart des implémentations commerciales, le modèle est dégradé dans le sens où de nombreuses adaptations techniques ont été

faites pour améliorer les performances. Enfin, ces modèles sont souvent très complexes car les composants ne sont pas toujours génériques afin de faciliter le développement (ex :EJB).

# II. MODÈLES D'AGENTS

## *A. Généralités*

### ***1) Définitions***

Il n'existe pas de définition unifiée de ce qu'est un agent. Nous présentons ici les définitions les plus pertinentes et les plus répandues :

–*Un agent est un système informatique situé dans un environnement, capable d'actions flexibles et autonomes dans cet environnement, pour atteindre le but pour lequel il a été conçu.* [WOO95]

–*Un agent est une entité qui fonctionne continuellement et de manière autonome dans un environnement où d'autres processus se déroulent et d'autres agents existent.* [SHO93]

–*Un agent est plongé dans un environnement dont il a une perception partielle par des capteurs et sur lequel il peut agir via des effecteurs. Il possède donc une interface vers l'extérieur. Chaque agent est conçu pour résoudre une tâche particulière (but) participant à la résolution d'un objectif global du système. Un agent est autonome : il a le contrôle de son état interne et de son comportement. Il est capable d'adapter son comportement aux changements de l'environnement et de prendre des décisions qui permettront de satisfaire ses objectifs. .* [DUR87]

–*Un agent est une entité autonome, réelle ou abstraite, qui est capable d'agir sur elle-même et sur son environnement, qui, dans un univers multi-agents, peut communiquer avec d'autres agents, et dont le comportement est la conséquence de ses observations, de ses connaissances et des interactions avec les autres agents* [FER95]

On notera également que certains chercheurs considèrent que l'on peut définir un agent de façon isolée, mais ce point de vue est plutôt marginal. En effet, un agent n'a de sens que s'il est immergé dans une société afin de tirer profit de l'interaction avec d'autres agents. Par conséquent, nous considérerons la dimension sociale d'un agent comme une de ses propriétés essentielles. D'autres considèrent la mobilité comme faisant partie intégrante de la définition d'un agent. Mais nous considérerons que cet aspect dépend plus de l'application à réaliser que du modèle.

Nous nous sentons particulièrement proche de la définition donnée par [FER95] et c'est celle là que nous retiendrons dans le cadre de ce projet.

### ***2) Propriétés***

En partant de l'ouvrage [WOO95], et des définitions citées précédemment, un système multi-agents peut être caractérisé ainsi:

–chaque agent a des compétences et une représentation du monde limitées (partielles), de même les données et expertises sont distribués. Il doit donc pouvoir raisonner sur les connaissances et les capacités des autres dans le but d'une coopération effective.

–il n'y a pas de contrôle global du système multi-agents. L'agent est capable d'agir de façon autonome et contrôle ses propres actions ainsi que son état interne.

–le traitement et les communications sont asynchrones.

–l'agent est capable d'agir sur son environnement à partir de ses perceptions de ce même environnement, on peut dire que l'agent est situé.

Un agent est censé avoir un comportement proactif et opportuniste selon une grande part de la communauté scientifique. En effet, les systèmes multi-agents sont souvent associés à leur comportement intelligent qui résulte de l'activité coopérative. Or à notre sens, ces notions comportementales ne sont pas génériques et dépendent de l'application.

### *3) __Types d'agent__*

Pour la prise de décision, deux écoles s'opposent, celle du comportement réactif contre le délibératif. Dans certaines architectures hybrides les 2 coexistent. Il est apparu important de connaître les différents types d'agent pour concevoir l'architecture interne de notre proposition, afin qu'elle soit cohérente avec l'existant.

Les agents réactifs ne sont pas à proprement parler « intelligents ». Ils perçoivent des signaux de l'environnement et agissent en conséquence – stimuli/réaction. Ils n'ont pas une représentation symbolique de l'environnement. En général ils ne communiquent pas directement entre eux mais disséminent des marqueurs dans l'environnement. Un SMA constitué d'agents réactifs possède généralement un grand nombre d'agents et peuvent présenter un comportement global intelligent, par un phénomène d'émergence – comportement de groupe non explicable à partir des seuls comportements des individus. C'est le type de comportement que l'on peut observer dans la nature dans les sociétés d'insectes type fourmis ou termites. Le comportement réactif ne nous intéresse que peu, dans l'optique du génie logiciel on préfère l'éviter car l'émergence est difficile – voire impossible – à canaliser dans les applications complexes. Soit l'agent n'atteint jamais son but car il ne cesse de changer de stratégie, soit il entreprend des tâches irréalisables car l'environnement a changé.

Les agents cognitifs quand à eux possèdent des représentations internes complexes. Afin de les rendre rationnels – réfléchir avant d'agir -, des scientifiques ont ajouté la notion d'état mentaux. L'architecture BDI par exemple, « Belief-Desire-Intention », de [COH95] sert à rationaliser un agent intelligent. Les croyances d'un agent sont les informations que l'agent possède sur l'environnement et sur d'autres agents. Elles peuvent être incorrectes, incomplètes ou incertaines et peuvent changer au fur et à mesure que l'agent recueille de l'information. Les désirs représentent ses buts. Les intentions d'un agent sont les actions qu'il a décidé de faire pour satisfaire ses désirs. La notion d'utilité a également été introduite, elle traduit la satisfaction (atteinte du but).

A titre d'exemple voici différentes possibilités d'architecture pour la prise de décision :

- Architecture verticale (subsomption, découpage en tâches…)
- Architecture horizontale (tableau noir, découpage fonctionnel…)
- Architecture hybride (anticipation pour éviter les boucles infinies)

### *4) __Organisation__*

On devine que selon le comportement des agents de façon isolée, leur organisation sociale sera différente. On peut repérer plusieurs types d'organisations dans la nature, correspondant à des sociétés d'agents réactifs comme les bancs (vol d'oiseaux, poissons) ou les organisations à « effets de bords » (fourmis, termites).

Mais dans le cadre de notre projet, nous nous orientons plutôt vers des types d'organisation plus

évolués. On citera notamment les réseaux pair à pair et l'organisation de type AGR que nous allons développer. Dans ce modèle, chaque agent est une entité autonome et communicante qui joue un ou plusieurs rôles dans un ou plusieurs groupes. Chaque groupe permet un rapprochement de certains agents ayant des affinités de communication ou de fonctionnalité. Les communications entre agents sont autorisées uniquement au sein d'un même groupe. Les rôles des agents représentent leurs capacités à effectuer un traitement pour un groupe donné. Un agent peut avoir plusieurs rôles et appartenir à plusieurs groupes en même temps, ce qui offre des possibilités de communication entre groupes ou alors un agent mixte peut servir d'intermédiaire entre deux groupes distincts. Ce modèle est très répandu et permet de représenter énormément de configurations, de la plus simple à la plus complexe.

## *B. Etat de l'art*

### *1) Domaines de recherches*

On s'intéresse aux agents pour essentiellement 4 types de domaines :

–La simulation et l'intelligence artificielle

De nombreuses plates-formes existent puisqu'il s'agit du domaine historique des systèmes multi-agents, mais dans la plupart le comportement de l'agent est un comportement réflexe comme pour la modélisation des fourmis dans la plate-forme Manta [DRO93]. L'intérêt de ces plates-formes se situe plus dans les résultats des simulations que dans les plates-formes elles-mêmes, qui sont de ce fait très spécialisées.

–Résolution de problèmes

Comme dans le cas de l'intelligence artificielle, les plates-formes sont développée spécifiquement pour un type de problème. On est encore une fois loin de notre projet.

–Les systèmes distribués

On peut considérer deux sous familles :
D'une part les plates-formes fournissant une simple structure de communication. Leur unique utilité réside dans l'infrastructure de communication ou d'exécution *distribuée* facilitant ainsi le déploiement d'un système multi-agents. (exemple : Hive, Jini et JATLite [JEO00]).
D'autre part les plates formes d'agent mobiles. Elles sont en quelques sortes une extension des plates-formes de communications grâce à la possibilité de déplacer physiquement des agents d'ordinateur à ordinateur, via un réseau, au cours de l'exécution du système. (exemple : Grasshopper, Voyager). Le déploiement dans le monde réel est le point fort de ces plates-formes. Contrairement à notre proposition, ces plates-formes ne tirent souvent profit que du caractère situé des agents et délaissent leurs autres caractéristique (organisation sociale, autonomie).

–Le génie logiciel

Pour l'instant, ces plates-formes se concentrent particulièrement sur les aspects de conception et de spécification d'une solution, et n'abordent pas les aspects d'implémentation. Des outils de conception ont été étudié dans ce but, comme AUML (exemple de plates-formes : CommonKADS, DESIRE)
Cependant, il existe des plates-formes (telles que AgentBuilder, Jack, Madkit, MASK et Zeus) qui couvrent les quatre étapes de développement d'une application (Analyse, Conception, Développement, Déploiement). Toutefois, ces plates-formes se placent généralement sous un angle particulier des SMA et ne sont pas génériques.

Ces deux dernières catégories sont toutefois à rapprocher car le génie logiciel tend de plus en plus à se rapprocher des techniques de distribution et de décentralisation du cœur applicatif.

### *2) Plates-formes*

Nous nous sommes penchés sur les modèles à agents existant afin de situer notre projet par rapport aux travaux dans le domaine. Nous avons ciblés ici les plates-formes ayant des similarités avec le projet que nous voulions mener.

–***AgentBuilder*** est une suite intégrée d'outils permettant de construire des agents intelligents, il a été développé en JAVA par Reticular Systems Inc. L'élaboration du comportement des agents se fait à partir du modèle BDI et du langage AGENT-0. KQML est utilisé comme langage de communication entre les agents. AgentBuilder est composé d'une interface graphique et d'un langage orienté agent permettant de définir des croyances, des engagements et des actions. Il permet également de définir des ontologies et des protocoles de communications inter-agents. Un agent crée avec cet outil est typiquement un agent d'interface, chargé de faciliter la recherche d'information ou la réalisation de certaines tâches à la place de son utilisateur. Un tel agent sera capable de filtrer l'information, de négocier des services avec d'autres agents et dialoguer avec son utilisateur.

–***MACE*** (Gasser e.a., 1987) est le premier environnement de conception et d'expérimentation de différentes architectures d'agents dans divers domaines d'application. Dans MACE, un agent est un objet actif qui communique par envoi de messages. Les agents existent dans un environnement qui regroupe tous les autres agents et toutes les autres entités du système. Un agent peut effectuer trois types d'actions : changer son état interne, envoyer des messages aux autres agents et envoyer des requêtes au noyau MACE pour contrôler les événements internes. Chaque agent est doté d'un moteur qui représente la partie active de l'agent. Ce moteur détermine l'activité de l'agent et la façon dont les messages sont interprétés. MACE a été utilisé pour développer des simulations d'applications distribuées.

– ***ZEUS*** (Nwama e.a., 1999) est une plate-forme multi-agents conçue et réalisée par British Telecom (Agent Research Programme of BT Intelligent Research Laboratory) pour développer des applications collaboratives. ZEUS est écrit dans le langage Java et il est fondé sur les travaux de la FIPA. L'architecture des agents ZEUS est similaire à la majorité des agents collaboratifs. Elle regroupe principalement les composantes suivantes

- une boîte aux lettres et un gestionnaire de messages qui analyse les messages de la boîte aux lettres et les transmet aux composantes appropriées ;
- un moteur de coordination ;
- un planificateur qui planifie les tâches de l'agent en fonction des décisions du moteur de coordination, des ressources disponibles et des spécifications des tâches ;
- plusieurs bases de données représentant les plans connus par l'agent, les ressources et l'ontologie utilisée ;

- un contrôleur d'exécution qui gère l'horloge locale de l'agent et les tâches actives.

ZEUS met un fort accent sur la méthodologie de développement, fondée sur la notion de rôle.

–***MADKIT*** (Madkit, 2003) est une plate-forme développée par le Laboratoire d'Informatique, de Robotique et de Microélectronique de Montpellier (LIRMM) de l'Université Montpellier II. MADKIT est libre pour l'utilisation dans l'éducation. MADKIT est écrit en Java et est fondé sur le modèle organisationnel Alaadin.. Chaque agent a un rôle et peut appartenir à un groupe. Il y a un environnement de développement graphique qui permet facilement la construction des applications.

–***JADE*** (Java Agent Development Framework - Bellifemine, Poggi, Rimassa, 1999) est une plate-forme multi-agents développée en Java par CSELT (Groupe de recherche de Gruppo Telecom, Italie) qui a comme but la construction des systèmes multi-agents et la réalisation d'applications conformes à la norme FIPA (FIPA, 1997). JADE comprend deux composantes de base : une plate-forme agents compatible FIPA et un paquet logiciel pour le développement des agents Java.

–***MALEVA*** est sans doute le modèle qui a les motivations les plus proches de notre projet. Elle a été développé par M. Lhuillier dans le cadre de sa thèse à l'université Paris IV. Il propose un modèle à base de « composants » logiciels et propose un équivalent de composition hiérarchique pour implémenter les comportements des agents. Par contre ces « composants » ne sont pas ceux d'un modèle de composants déjà existant, donc pas de réutilisation. De plus il a été conçu pour le développement d'application de simulation et beaucoup de point comme l'organisation sociale et la communication sont très peu abordés.

Durant nos recherches, nous n'avons rencontré aucune plate-forme conçue au dessus d'un modèle de composants. La réutilisation des bibliothèques de composants que nous proposons manque donc aujourd'hui De plus il n'existe pas encore de plate-formes réellement génériques et orientées génie logiciel.

### *3) FIPA*

La FIPA (Foundation for Intelligent Physical Agents) est un organisme qui propose, regroupe et normalise les caractéristiques que doivent posséder les objets logiciels pour pouvoir se revendiquer d'être des agents. Leur principale motivation est d'établir des standards garantissant l'interopérabilité des systèmes. Ces recommandations n'étant pas antagonistes avec l'idée que nous avions de notre modèle, il a été décidé de les suivre. Nous présenterons ici les grandes lignes de ces recommandations, une version plus détaillée étant disponible en annexe.

Pour qu'une plate-forme multi-agents soit “FIPA compliant”, elle doit au minimum posséder des entités ou objets remplissant ces trois rôles principaux :

- Le Système de Gestion d'Agents (Agent Management System - AMS) : exerce le contrôle de supervision sur l'accès à et l'usage de la plate-forme ; il est responsable de l'authentification des agents résidents et du contrôle d'enregistrements.
- Le Canal de Communication entre Agents (Agent Communication Channel - ACC) : fournit les voies nécessaires aux interactions entre les agents ; il doit garantir un service fiable et précis

pour le routage des messages ; il doit être compatible avec le protocole IIOP pour l'interopérabilité entre les différentes plates-formes multi-agents.

•Le Facilitateur d'Annuaire (Directory Facilitator - DF) : un service de pages.

On peut remarquer qu'il n'y a aucune restriction sur la technologie utilisée pour l'implémentation de la plate-forme.

### a) Caractéristiques fondamentales des agents

La FIPA reste assez libre quand à la spécification des agents et de leur architecture interne. On a uniquement pour contrainte qu'ils soient autonomes, qu'ils aient un nom et une localisation et qu'ils soient capables d'interagir avec les autres par voie de messages. Les agents peuvent proposer des services, il est cependant spécifié que ces agents ne peuvent pas choisir arbitrairement de ne pas répondre à une requête sur un de leurs services, tout service proposé doit être acquitté en cas de sollicitation, on comprend en effet que si un agent requiert un service proposé par un autre agent pour fonctionner et accomplir sa tâche, la non réponse de ce dernier entraîne un blocage du système.

Par contre, il est possible de découpler au maximum l'agent des services qu'il peut rendre, notamment en passant par des interfaces. On se rapproche ainsi de l' « Ever Late Binding » proposé par [BRI04]. Les agents doivent également être capables d'introspection, de se présenter, d'intégrer un groupe d'agents et de proposer leurs services.

La FIPA encourage l'interopérabilité entre les plates-formes, c'est d'ailleurs l'une de ses missions que de définir des standards permettant d'éventuels transferts d'agents d'une plate-forme à l'autre, ou en tous cas de permettre à des agents dans différents environnements de communiquer entre eux en imposant un protocole de message détaillé.

### b) Caractéristiques fondamentales de la plate-forme

La spécification FIPA est très technique et sans interprétation possible, aussi nous ne rentrerons pas ici dans les détails car cela reviendrait à faire simplement une traduction du document. La FIPA introduit surtout un vocabulaire permettant d'unifier le jargon en la matière. Cependant nous allons présenter un résumé de ce qui doit absolument figuré dans une plate-forme multi-agents pour être « FIPA compliant ».

Elle doit comprendre pour répondre aux spécifications :

-un langage de communication inter agent (ACL) (ENCODING)

-un annuaire d'agents (NAMING)

-un système de transport de messages (TRANSPORT)

-un annuaire de services (NAMESPACE)

La plate-forme peut uniquement implémenter une couche transport de messages. Pour les autres services, ils peuvent être remplis par des agents ou des composants d'agent spécifiques. Ces dispositions seront à discuter au regard la faisabilité en fonction de nos choix techniques.

La réalisation concrète n'étant pas abordée dans les spécifications, le choix des solutions techniques est libre.

# PARTIE 2 : Spécification de MALICE (Modèle d'agents logiciels implémenté grâce à des composants étendus)

# III. Spécification du modèle

## A. Postulat

Nous plaçons nos travaux dans une optique génie logiciel, ainsi nous souhaitons proposer une plate forme de développement et de déploiement d'agents, générique, ne dépendant pas de l'application à concevoir, offrant ainsi une souplesse que la plupart des modèles déjà existants ne proposent pas. Ceci passe par la recherche de l'autonomie et du découplage maximum. Nous souhaitons également que notre modèle exploite la dynamique multi-agents, c'est à dire introduise une dimension organisationnelle, qui nous paraît être l'atout essentiel des SMA (notamment les notions de travail collaboratif, négociation).

Il ne faut pas non plus oublier la dimension componentielle que doit avoir le modèle. L'un des objectifs et de pouvoir réutiliser les bibliothèques de composants existantes. De plus, la notion de hiérarchisation des composants est essentielle, elle apporte une puissance et une facilité de développement qu'il serait dommage de ne pas retrouver.

Nous partons tout de même avec l'a priori de respecter les recommandations FIPA dans le but de toucher le plus grand nombre et de participer à l'effort de standardisation. Ces recommandations nous laissent cependant très libres et nous exposerons ci après les choix que nous souhaitons faire pour notre modèle.

## B. Plate-forme

### 1) Généralités

Commençons par définir les caractéristiques de notre plate forme. Nous optons pour une plate-forme minimaliste. Elle ne proposera en interne que ce qui est utile à la création d'agents et de composants, à leur assemblage et à leur déploiement et le middleware indispensable à la communication des agents déployés sur celle ci.

Ce sont des agents non fonctionnels qui gérerons tous les aspects de services indépendants de l'application à réaliser (sécurité, gestion des ressources, annuaire de nom et de service...). Ils auront une structure et un comportement équivalent aux agents applicatifs (appelés agent métier), mis à part le fait qu'ils seront obligés de répondre à toutes les requêtes. Même si ces agents ne font pas à proprement parler partie de la plate-forme, la plupart d'entres eux sont indispensables à son fonctionnement. Nous proposerons donc au dessus de la plate forme un pattern définissant le rôle et le nom de ces agents non fonctionnels

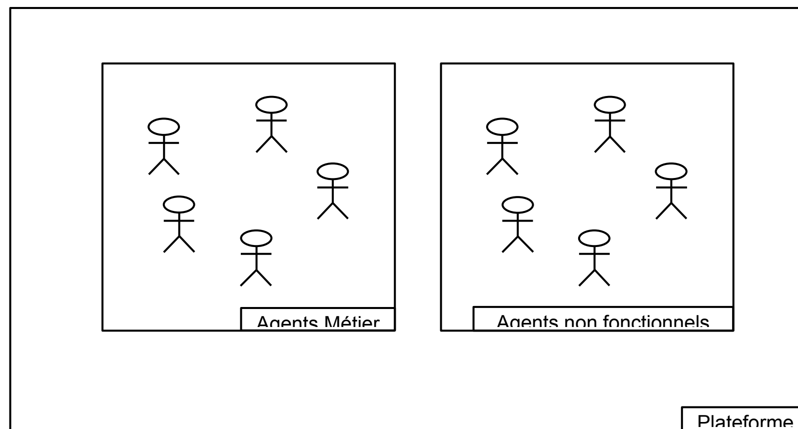



Figure 1 : Plate-forme

Nous faisons ce choix pour permettre au développeur de changer le comportement de son application au niveau non fonctionnel de façon plus souple que s'il était intégré à la plate forme. De plus, ceci apporte une certaine cohérence et uniformité au sein du modèle.

Ainsi, si les agents sont vus comme des entités autonomes analogue à l'homme, l'environnement lui peut être vu comme le monde environnant, avec ses objets (partagés entre les hommes), ses routes et ses moyens de communications. Au sein de la population, certains hommes représentent la loi et l'administration et sont en charge de la faire respecter.

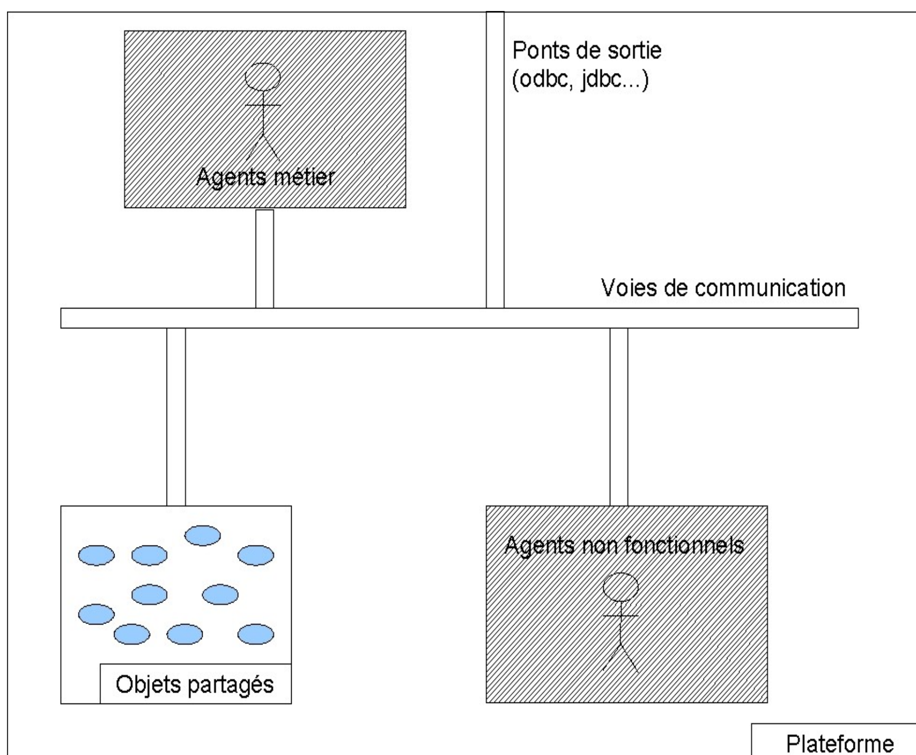



Figure 2 : Bus de communications

Sur ce schéma, les voies de communication sont représentées telles un bus général, en réalité, pourront coexister plusieurs types de communication, par exemple la communication entre agent et

les voies vers l'extérieur (driver) seront dissociées.

### 2) Protocoles de communication

La communication asynchrone entre agent par envoi de messages interprétés et non par invocation de méthodes fait intégralement partie de la définition du concept. Ainsi, nous aurons recours à un bus message ou MOM (Message-oriented Middleware) permettant de créer, d'adresser, d'acheminer et de détruire les messages. Ce point est essentiel si on considère que toute l'interaction entre agents et la synchronisation de l'aspect reparti des agents sera gérée par l'intermédiaire de messages.

Cependant, nous n'allons pas ici définir un n-ième protocole ou format propriétaire. Nous chercherons plutôt un MOM existant dont les spécifications sont les plus proches possible de la recommandation FIPA. Ce sera au designer de l'application qu'incombera la tâche de choisir l'ACL (Agent Communication Laguage) utilisé pour écrire le contenu des messages circulant sur la plate-forme (dialecte, syntaxe, vocabulaire...) et le protocole les encapsulant. Ainsi, il pourra rendre l'application qu'il développe interopérable avec ses précédentes réalisations ou tout autre système. De nombreuses propositions ont été faites à ce sujet (propriétaire ou non), par exemple KQML et ARCOL. Cependant, nous ne saurions que trop recommander l'usage de FIPA ACL qui oeuvre pour la standardisation et l'interopérabilité des systèmes.

### 3) Diffusion des messages

Au niveau du mode d'adressage des messages (point à point, broadcast ou multicast), plusieurs modes seront disponibles.

Le point à point (nommage explicite du destinataire) est un mode essentiel permettant de demander un service à un agent particulier(à noter que ceci présuppose que le destinataire existe et est connu mais ne garantit pas de comprendre le message). Il fera donc partie des options disponibles.

Le broadcast (diffusion à tout le monde) présente un intérêt  limité. En effet, il  entraîne un encombrement des voies de communication et des boites aux lettres, une perte d'efficacité dues aux parasite de plus c'est un mode de communication totalement insécurisé. Nous ne le retiendrons donc pas

Le Multicast (diffusion à un groupe d'abonné) semble très approprié à la collaboration des SMA(sous réserve de désigner un gestionnaire qui fera la synthèse de la réponse (prise de décision) et gère aussi le retrait ou l'ajout d'agent à son groupe). De plus, il permet de faire du multicast si tous les agents du système s'inscrivent. Il fera donc également parie des types de diffusions utilisable sur notre plate-forme.

Nous proposerons enfin un dernier mode de diffusion, celui du forum. Les messages serons accessibles aux agents inscrits sur celui ci, cependant les messages ne leurs seront pas envoyés mais les agents pourront venir les consulter à leur guise.

## C. Architecture d'un agent

### 1) Généralités

Le concept d'agent est central. Qu'appelons nous agent ? Ce concept sera défini au fur est à mesure

selon différents aspects. Cependant il convient de définir immédiatement le concept de base. Un agent selon la définition FIPA est un module autonome pouvant proposer des services. Chaque agent étant autonome par essence, il semble difficile de concevoir un modèle uniquement basé sur des agents - au sens où chaque module de l'application, à toute granularité serait perçu comme un agent et en aurait toutes les propriétés, De même, la possibilité de grouper certains agents pour former un agent de granularité supérieure semble antagoniste avec cette définition, un groupe d'agents étant un SMA (Système multi agents) dans la communauté scientifique. La position que nous tenons est la suivante : le terme agent ne sera utilisé que pour les composants logiciels de plus haute granularité. Par contre, l'"intérieur" des agents pourra être vu comme un assemblage de composants de différentes granularité, chaque agent sera un composite répondant à certains patterns ou frameworks.

Cela permet d'adopter une approche incrémentale, processus de développement indispensable dans le génie logiciel moderne. Ceci permet en sus la réutilisation de composants par spécialisation alors que la technique standard appliquée jusqu'ici dans les modèles d'agents est d'appliquer des filtres (ex : capteurs), parfois difficile à mettre en œuvre et souvent limités.

Comme pour les modèles de composants, on retrouve la possibilité de faire des changements dynamiques de comportement (remplacement de composants éventuellement à chaud en vérifiant la compatibilité des interfaces). Cette architecture composite de l'agent laisse le concepteur libre et peut réellement adapter la structure interne au comportement souhaité et même le faire évoluer et s'adapter.

De plus, l'identification/introspection permise par la plupart des modèles de composant facilite l'inscription sur les annuaires de services, l'agent étant conscient des services qu'il peut ou veut mettre à la disposition de la société.

### *2) __Schéma interne__*

Pour respecter le concept d'agent, nous avons définir des patterns répondant à ses exigences. Nous définirons l'autonomie comportementale comme la capacité de pouvoir atteindre un but, indépendamment des autres agents du système.

Voyons d'abord un schéma simplifié de la décomposition fonctionnelle d'un agent. Il est plongé dans un environnement, duquel il se fait une représentation partielle (locale) par l'intermédiaire de ses capteurs. A l'aide de ces informations, il est en mesure de prendre des décisions et d'adapter son comportement en vue de réaliser une certaine tâche. Il modifie ensuite l'environnement par l'intermédiaire de ses effecteurs pour répondre à son dessein. On se place dans un cycle Perception – Délibération - Action

A noter que les autres agents du système sont simplement considérés comme des éléments de l'environnement.

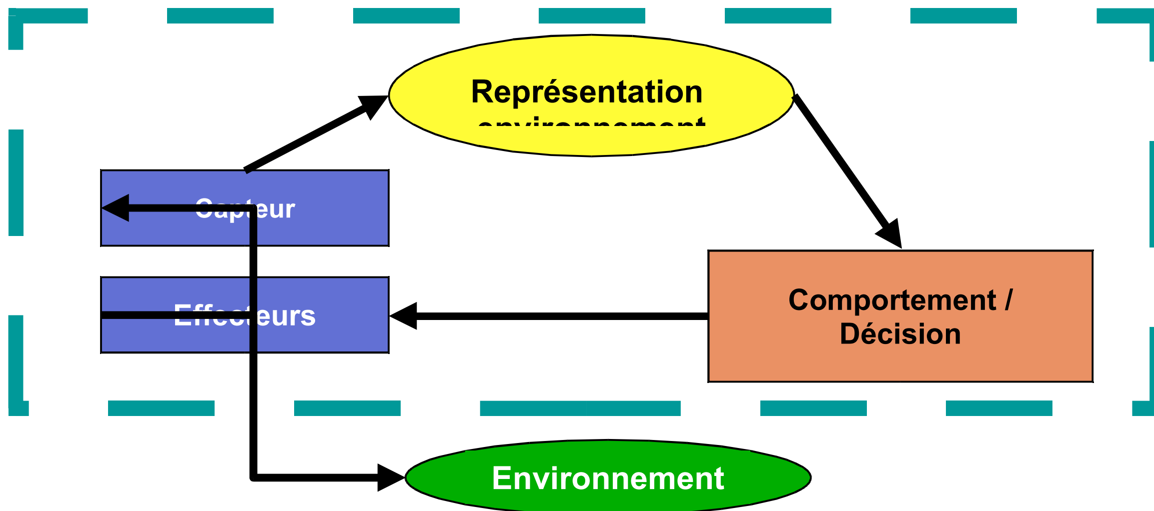



Schéma 1 : Schéma interne de l'agent

On remarque que les interfaces de l'agent, à savoir ses capteurs et effecteurs sont relativement indépendants du comportement interne, en tous cas découplées. Ceci autorise une grande souplesse mais a également pour contrainte de devoir fournir une documentation précise sur le comportement et les services présentés par l 'agent.

Dans la site, nous détaillerons ce modèle très général. Nous dissocierons le traitement des messages du comportement de base et ajouterons au modèle une boites aux lettres pour communiquer avec les autres agents du système.

On remarque qu'un agent doit avoir une certaine synchronisation. Sans pour autant l'imposer, nous recommandons que chaque agent soit déployé sur une seule machine plutôt que reparti sur le réseau afin d'améliorer la communication entre ses composants internes.

### *3) __Architecture interne__*

Notre modèle utilisera donc comme brique de base des composants. Parmi les intérêts nous pouvons citer : la réutilisation possible des bibliothèques de composants existant, la modularité de l'application, la rapidité de développement. De plus nous sommes intéressés par la propriété de certains modèles qui est de pouvoir faire de la composition hiérarchique et qui permet de voir un agent comme un assemblage de composants. Pour ce qui est de la communication entre composants internes à l'agent, nous prévoyons une communication synchrone à base d'appel de fonction, ceci améliorant les performances et la simplicité d'utilisation.

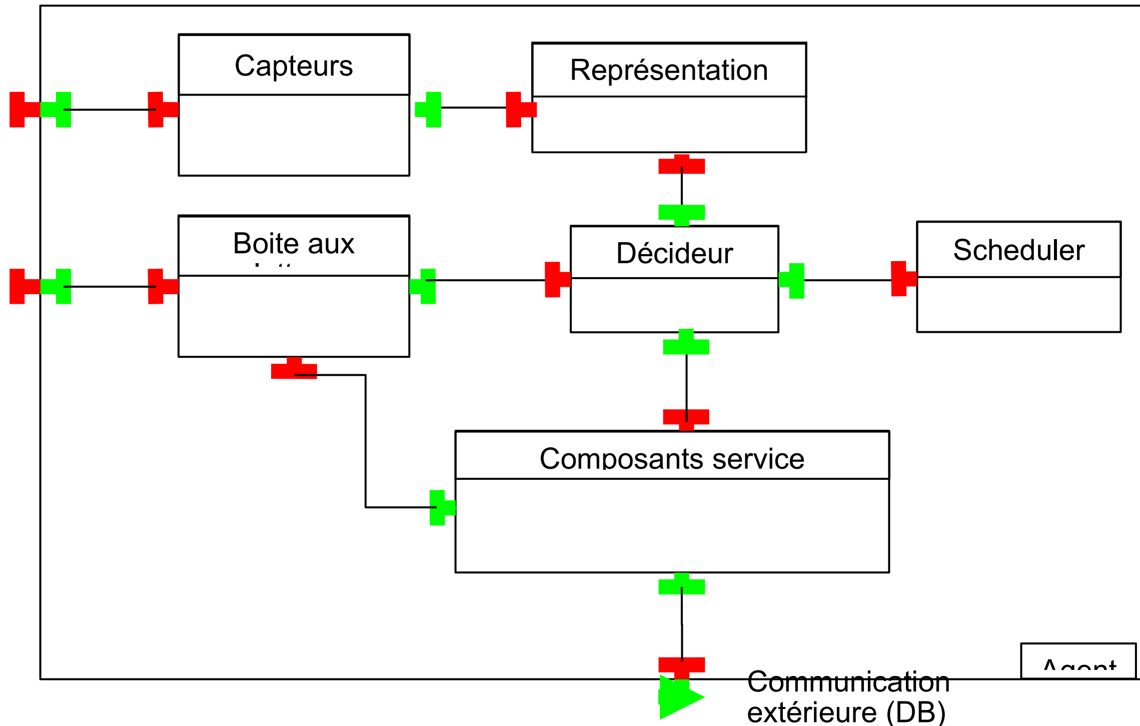



Schéma 2 : Architecture d'un agent

-La ***boite aux lettres*** sert à la fois à l'encodage et au décodage des messages. Elle gère la file d'attente des messages non traités. Plusieurs "adresses" peuvent correspondre physiquement à la même boite. Ainsi l'agent pourra présenter différentes interface selon qui le demande. Elle permet également de répondre ou d'envoyer des messages sollicitant un service. C'est le seul point d'accès de l'extérieur vers l'intérieur de l'agent. En rapprochant ce schéma du précédent, sur le plan fonctionnel, ce composant implémente les capteurs et les effecteurs permettant l'interaction avec les autres agents du systèmes.

-La ***représentation mentale*** peut être vue comme un cache à plus ou moins longue durée. On y stockera la représentation du monde qu'en a l'agent - c'est à dire locale en général). On pourra y trouver par exemple l'adresse des "agents amis" (agents dont les services sont souvent sollicités), la disponibilité des ressources, les membres de son groupe.

-Le composant ***décideur*** est le cerveau de l'agent, il choisit les actions à mener en fonction des messages qui lui parviennent et la connaissance qu'il a du monde. Il appellera les fonctions métiers appropriées pour répondre aux demandes. C'est en lui que résidera l'intelligence de l'agent.

-Le ***scheduler*** défini l'ordre de priorité des actions à mener.

-Les ***composants métiers*** sont des composants applicatifs standards, qui supportent la

composition fonctionnelle et hiérarchique. Ces composants peuvent demander de l'aide à d'autres agents en passant par la boite aux lettres et peuvent communiquer à l'extérieur grâce aux ponts de la plate forme, en l'occurrence une base de donnée ou une ressource partagée. Comme pour la boite aux lettres, si on met en rapport ce schéma avec le précédent, une grande partie des capteurs et des effecteurs est implémentée ici ; en somme,les interactions avec toutes les entités de l'environnement hormis les autres agents

Cette solution hybride permet de conserver le vocabulaire et les concepts familiers à la communauté agent sans pour autant introduire de nouveaux termes ou redéfinir des notions. Elle permet également de profiter des techniques fiables et efficaces du domaine des composants.

## *D. Organisation*

Il nous a paru intéressant d'inclure un aspect organisationnel entre les agents. C'est cet aspect qui donne toute la puissance aux systèmes multi-agents. C'est à notre sens ce qui différencie le plus fondamentalement un modèle de composants d'un modèle d'agent.

S'il était encore besoin de les préciser, nous allons ici rappeler les avantages des SMA par rapport à des agents isolés. D'abord ceux ci permettent d'accomplir des taches complexes, qu'un agent seul ne serait pas capable de réaliser grâce à la mise en commun des compétences et des données. Cela a pour conséquence d'optimiser le système: les agents peuvent être spécialisés et nous évitons les redondances non voulues. Au niveau des performances également, les SMA ont l'avantage de permettre une meilleure gestion des ressources pour peu que la coordination soit correctement implémentée et ont la possibilité d'accomplir des tâches en parallèles.

### *1) Organisation comportementale (interaction et coordination)*

La conception scientifique et technique commune est fondée sur l'idée qu'un système informatique est un bloc de sous-systèmes bien identifiés et figés. Or en passant sur le concept d'agent, nous devons également modifier notre perception et percevoir désormais un système comme une population d'agents autonomes en interaction.

L'interaction et la coordination sont les différences entre un agent seul et un groupe d'agents. Elle se traduit par des phases de compétition, de négociation et de collaboration. Or cette coordination implique une certaine organisation au sein du groupe - quel agent fait quoi, quand, comment, avec qui ? L'organisation elle même dépend de la perception que les agents ont du monde et de sa représentation. Les méthodes de prise décision influent également sur celle ci.

Dans le cadre du génie logiciel les organisations de type fourmilière nous paraissent peu indiqués : en effet, ces agents sont trop réactifs aux signaux émanant de l'environnement. Or nous souhaitons avoir un contrôle sur les phénomènes d'émergence au cours du passage du niveau micro (agent) au niveau macro (SMA) qui ne peuvent pas être déduit par l'observation des agents individuellement. Dans le cadre du génie logiciel, ce sera un point important à ne pas négliger, afin d'assurer la stabilité et la validité du comportement global du système.

Les organisations plus structurées nous semblent plus appropriées, mais elles comportent plus de contraintes. Il faut connaître les capacités individuelles de chaque agent de façon fiable et précise, afin de cerner toutes les possibilités du système. On doit également définir les rôles de chacun (domaine d'expertise privilégié), leur pouvoir décisionnel (hiérarchie) et dans une certaine mesure

leur niveau de compétence (permettre le choix entre différents agents). On peut également dans certaines situations définir un coordinateur, notamment pour les négociations afin de prendre une décision effective au nom d'un groupe. Cela implique que les agents devront avoir préférentiellement un comportement cognitif.

Pour les agents réactifs on peut se contenter de techniques de prise de décision dites « simples » comme la subsomption ou les réseaux de neurones, et donc de coordination « simples » car on peut décrire précisément la réaction à tenir en cas d'interaction avec d'autres agents, la scripter pour chaque agent. Pour les agents cognitifs la tâche est plus ardue. Nous devrons introduire la notion d'états mentaux, qui orientent le comportement. En effet, il faut ajouter des critères non matériels, à la différences des signaux reçus de l'environnement, appartenant à la représentation du monde, pour définir des objectifs collectifs pour coordonner les actions.

Le système le plus connu et reconnu est le principe du BDI (Belief – Desire - Intention) décrit dans l'état de l'art. Les croyances peuvent être définies selon la logique modale (voir sémantique de Kriepke, logiques temporelles).

La littérature sur le sujet est vaste et nécessiterait une étude à part entière sur la prise de décision et la gestion de la collaboration. Cependant, nous pensons que ce choix revient au concepteur des applications réalisées et non à celui de la plateforme, car différents modèles peuvent être intéressants selon l'application elle même et nous ne souhaitons pas réduire ce champ. Ces états mentaux et leur gestion seront implémentés au sein du composant représentant la partie cognitive de l'agent (Décideur). Par exemple, même si nous pensons que l'utilisation d'agents réactifs peut être dangereuse en génie logiciel, nous ne souhaitons pas fermer cette porte aux développeurs qui peuvent en avoir besoin dans un cas particulier.

### *2) __Organisation structurelle__*

Essayons de nous représenter un SMA comme un réseau informatique avec toutes les analogies que cela comporte. En effet, un réseau permet aux ordinateurs qui le composent d'échanger de l'information, de travailler à distance et en collaboration (informatique répartie) et de partager de l'équipement, par exemple, une imprimante. On peut tout à fait substituer le terme agent au terme ordinateur.

Allons plus loin, un réseau est constitué d'ordinateurs, de fils, de routeurs, de hubs qui permettent le bon fonctionnement. Si « fils » est remplacé par « middleware de communication » (soit plate-forme), « routeur » par « coordinateur » et hub par « relayeur », les coordinateurs et relayeurs étant des agents spécialisés, l'analogie se poursuit.

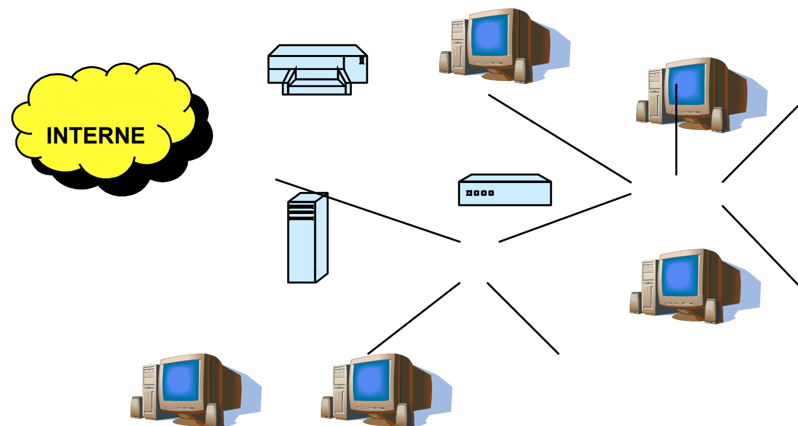



Figure 3 : Réseau informatique

Maintenant faisons le cheminement inverse. Quelles caractéristiques intéressantes les réseaux peuvent-ils nous inspirer ?

–*Nommage* : Un même ordinateur peut présenter plusieurs interfaces sur un réseau Ethernet représentées différentes adresses IP. Les ordinateurs se regroupent en réseaux et sous réseaux identifiés grâce à ces mêmes adresses et aux masques de sous réseau associés. On remarque qu'un ordinateur peut appartenir simultanément à plusieurs réseaux. Pour notre modèle, nous imaginons un système de nommage équivalent dont nous définirons les modalités exactes ultérieurement, de cette façon les identifiants porterons un sens et pourront être localisés et organisés spatialement de façon efficace.

–*Communication* : Les ordinateurs communiquent entres eux par envoi de messages, le support de transmission n'est pas un élément déterminant. Le plus souvent, la communication asynchrone est buffurisée pour réduire le trafic. Nous reprendrons également cette idée.

–*Adressage* : Les paquets transmis sont acheminés à travers le réseau au(x) destinataire(s) spécifié(s) dans l'entête du message grâce aux tables de routages présentes sur chaque machine. On note que certaines adresses IP sont réservées, pour le broadcast, pour désigner un réseau entier, etc. Cette caractéristique nous intéresse aussi, on peut également imaginer une adresse réservée pour les coordinateurs lors de travaux collaboratifs.

–*Recherche d'agent* : D'autres éléments peuvent être intéressants, à commencer par les DNS. Il est courant qu'un nom soit donné aux machines, il est donc nécessaire d'avoir un service capable de faire la conversion du nom en adresse IP et inversement. Pour se faire, quand le cas se présente, la machine voulant faire une conversion interroge le DNS (Domain Name Service). Nous serons obligés de recourir à ce genre de service pour trouver la localisation des agents sur le réseau.

–*Configuration* : Autre atout, le protocole DHCP (Dynamic Host Configuration Protocol). Ce protocole permet à un ordinateur se connectant à un réseau d'obtenir dynamiquement sa configuration selon le context. Le but principal est la simplification de l'administration d'un réseau. Cela est idéal dans notre cas, pour le déploiement ou le retrait d'agents du système. En assouplissant

cette configuration, nous évitons les risques de panne.

–*Interconnexion* : Un autre intérêt repris cette fois est de pouvoir décentraliser les annuaires de services ou de nom en créant des annuaires locaux,. Les requêtes se propageraient en cas de non réponse sur l'annuaire local suivant, défini dans une table de routage. Cette idée permet de mieux répartir la charge sur les nœuds, évite une partie du trafic et réduit l'encombrement qui lui est associé.

### *3) Propositions*

Au vue de ce qui vient d'être énoncé et sachant que les réseaux informatiques sont aujourd'hui robuste et fiables, nous avons décidé de transposer un certain nombre de concepts aux sociétés d'agents dans les termes que nous allons définir ici.

Nous souhaitons associer nos agents en groupes et étant donné que nos agents proposent des services, il nous paraît judicieux de construire ces groupes de telle sorte à minimiser le nombre d'interfaces requises par les agents du groupe non implémentées par d'autres au sein du même groupe. Il s'agit de former des groupes fonctionnels en fonction des rôles de chacun. On retrouve ici la structure AGR décrite dans l'état de l'art.

Comme énoncé précédemment, certains agents seront nécessaires au bon fonctionnement du système et nous allons vous les exposer ici.

-Agent proposant un service de nommage

Cet agent correspond à l'AMS de la FIPA (Agent Management Service) il gère l'arrivée et le départ d'agents sur la plate-forme, il attribue un adresse fixe et unique à chaque agent et connaît sa localisation, c'est à dire son adresse. Cette adresse comme l'adresse IP dépendra du groupe et de la localisation de l'agent

-Agent proposant un annuaire de service

Interlocuteur privilégié des agents, il sert de pages jaunes, à l'arrivée d'un agent sur la plate-forme, il est averti par le service de nommage et inscrit l'agent selon les services qu'il propose. Tout changement dans le comportement des agents lui sera signifié par l'agent lui même.

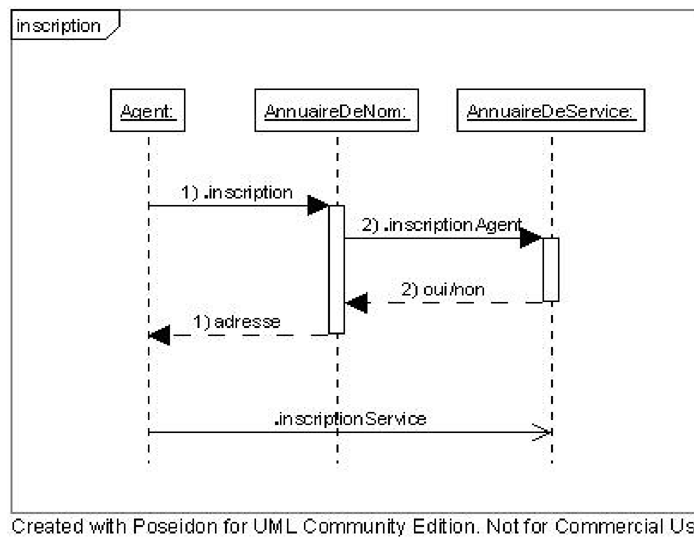



Diagramme 1 : Inscription sur la plate-forme

Pour trouver un service, les agents s'adresse à lui à la première occurrence. Ensuite, la localisation des agents "amis" (dont les services sont souvent utilisés) sera stockée en cache dans la représentation interne de l'agent.

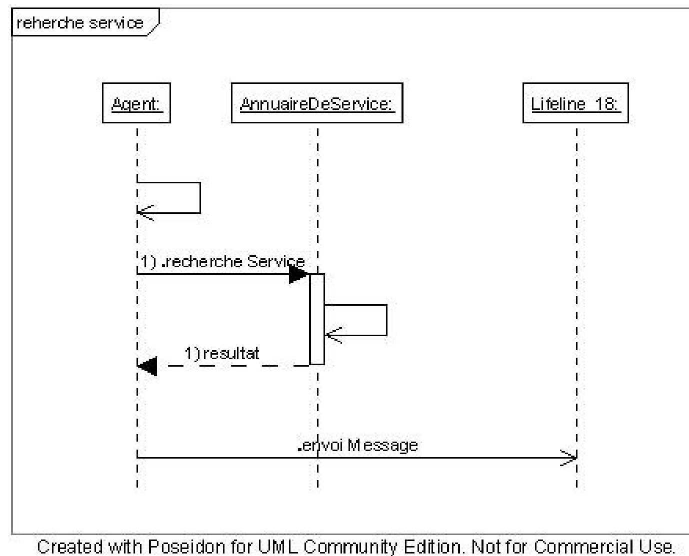



Diagramme 2 : Recherche simple de service

On peut imaginer également qu'il n'y ait pas qu'un annuaire de service, mais plusieurs, ayant chacun une vue locale, en coopération, si bien qu'ils pourront s'interroger les uns les autres.

Nous avons introduit un système de notation, permettant de juger de la qualité de service d'un agent. Elle pourra se déclinée sous différents critères comme le temps de réponse ou la précision des résultats.

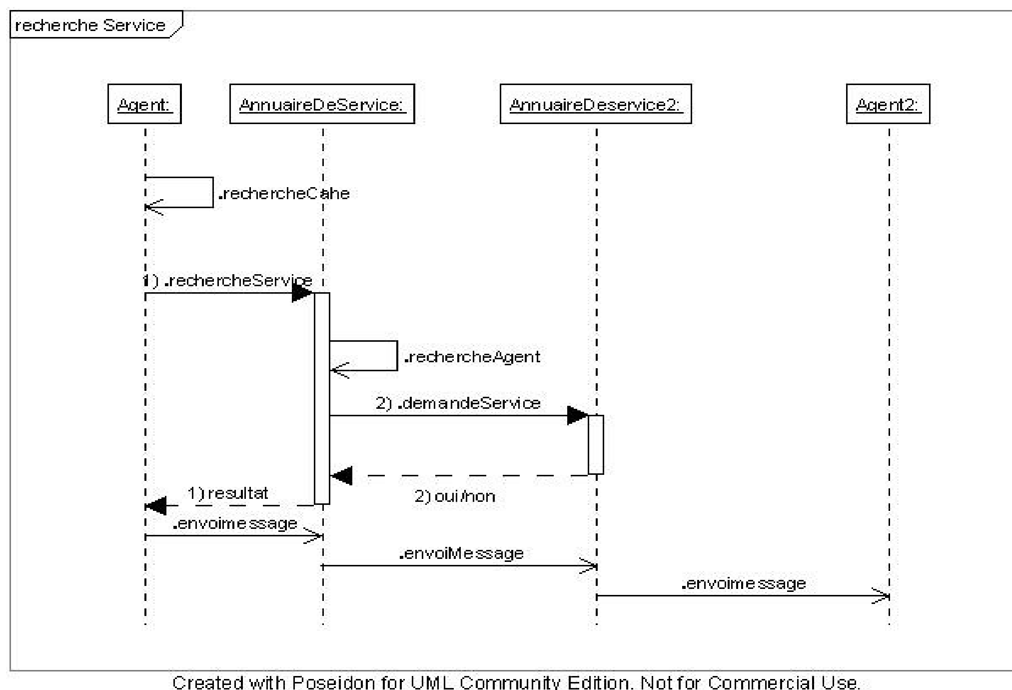



Diagramme 3 : Recherche complexe de service

Une autre fonctionnalité que nous souhaitons proposer est la possibilité de noter les agents. Quand un agent A demande un service à l'agent B, A peut évaluer B selon différents critères comme le temps de réponse ou la validité du résultat. A gardera cette note dans sa représentation mentale et la communiquera à l'annuaire de service qui fera un moyenne avec la note qu'il avait déjà concernant B. L'intérêt de cette notation est de pouvoir reconfigurer le système dynamiquement. Si un agent ne remplit pas son rôle, l'annuaire de services ne communiquera plus son adresse et pourra par exemple le remplacer par le clone d'un agent C fournissant les mêmes services mais de façon plus efficace ou alors le signaler à l'administrateur.

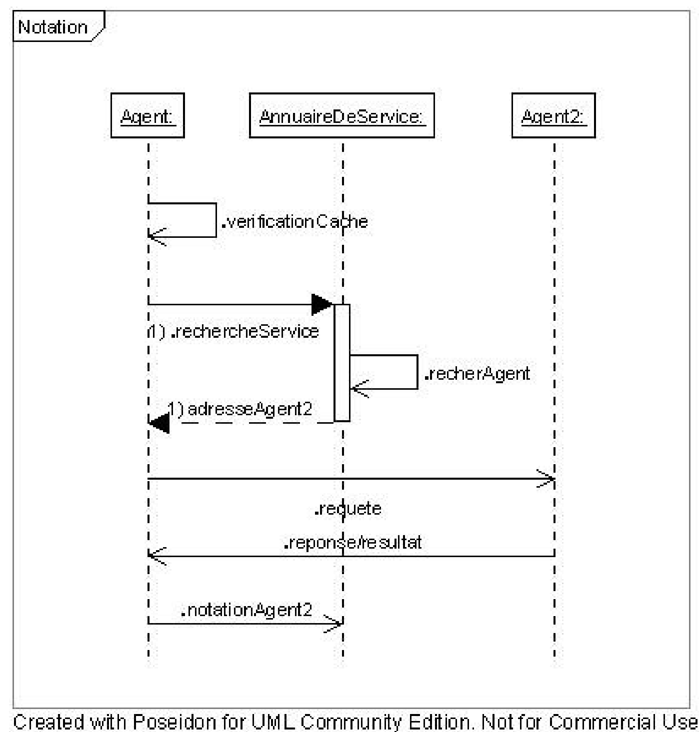



Diagramme 4 : Notation

## *E. Respect des normes FIPA*

Ces normes sont assez souples, cependant elles soulèvent différents problèmes. Les agents ne peuvent pas échanger des objets ou des références sur des objets. Cette contrainte est très forte et oblige l'architecte d'une application à modifier ses méthodes de conception. Nous pensons à contourner cette contrainte en offrant la possibilité de créer des objets partagés, manipulables par les agents comme des ressources partagées au même titre que de la mémoire ou une base de données. Il faudra se préoccuper dans ce cas de notions de sécurité plus avancée, notamment en ce qui concerne les verrous, les risques d'inter blocages et le garbage collector.

Une autre question soulevée est celle de la recherche de service. Contrairement au modèle objet classique, où l'objet client connaît explicitement le nom de la méthode à appeler au sein d'un autre objet, le découplage et l'abstraction voulue dans les SMA empêche ce mécanisme. Nous pouvons soit opter pour la solution présente dans la plupart des modèles de composants, à savoir interface requise/interface fournie, mais cela exige d'avoir connaissance des nomenclatures (signature des méthodes par exemple) pour la description et le nommage des services. Cela pose certains problèmes lorsque les agents sont fabriqués à partir de bibliothèques de composants. Dans ce cas on peut laisser le soin au concepteur d'unifier les noms des services de son application ou éventuellement développer des composants d'interfaçage et du code glue (comme proposé durant LMO 2006). Une possibilité plus satisfaisante serait d'ajouter des méta-informations sous forme de tags dans le code permettant de construire une ontologie sur les services. Cette possibilité apporte plus de flexibilité et permet la substitution d'agent proposant des services approchant.

# PARTIE 3 : RÉALISATIONS

# IV. Implémentation

## A. Fractal

Nous avons choisi de nous appuyer sur un modèle de composants existant pour réaliser l'implémentation de notre proposition. En effet, les mécanismes élémentaires de compositions sont déjà présents dans ces modèles et nous ne voyons pas d'intérêt à redéfinir un modèle de composants supplémentaire, alors qu'il en existe qui peuvent répondre à nos besoins. De plus cette réutilisation de la plate forme permettra ensuite aux développeurs d'application de réutiliser leurs bibliothèques de composants déjà implémentées. Au cours de nos recherches, nous avons découvert Fractal développé par les laboratoires de INRIA et il a retenu notre attention notamment sur les points suivants.

Fractal est un modèle de composants permettant de gérer les assemblages complexes et dynamiques de composants. Une application est représentée sous forme d'un composant racine, défini par l'assemblage de sous composants, qui peuvent être être eux même définis récursivement par un assemblage de composants, jusqu'à atteindre le composant primitif. Fractal permet ainsi de gérer les architectures les plus complexes sous la forme structurée d'une hiérarchie de composition. Ce modèle de composition hiérarchique le distingue de nombreux autres modèles où les composants sont seulement juxtaposés. En utilisant comme base le modèle Fractal, le concepteur peut faire à la fois de la composition fonctionnelle c'est à dire connecter deux composants via leurs Interfaces ou structurelle car on peut créer un nouveau composant à partir de composants existant). Cette première caractéristique est celle qui nous intéresse le plus car nous avons défini un agent comme un assemblage de composants.

De plus, le contenu applicatif d'un composant (implémentation) est encapsulé dans une membrane qui décrit les Interfaces du composant qui regroupent l'ensemble des signature des méthodes accessibles. Ces interfaces peuvent aussi bien être applicatives que de contrôle. Elle se décline sous forme d'interfaces requises (client) et d'interfaces fournies (serveur) et sont dirigées soit vers l'intérieur, soit vers l'extérieur du composant. Ce composant est donc vu comme une boite noire accessible uniquement par ses interfaces, ce qui assure un découplage total du contenant et du contenu. Cette deuxième caractéristique a également retenue notre attention. En effet, il est préférable que les agents ne puissent pas accéder directement au contenu des autres entités. C'est un des points essentiels de l'autonomie.

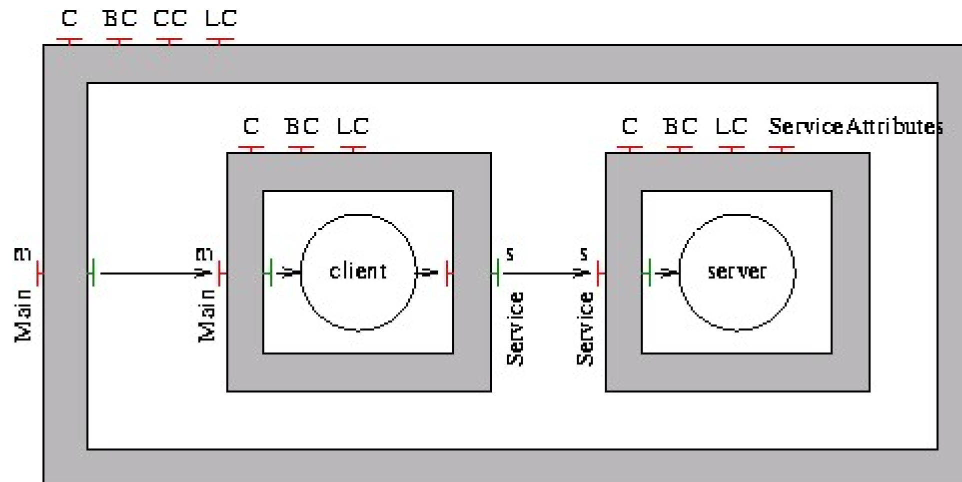



Schéma 3 : Composant Fractal (source : http://fractal.objectweb.org)

Sur ce schéma représentant un composant Fractal, les interfaces serveur sont en rouge, les clientes sont en vert. Les interfaces C, BC et LC sont les interfaces de contrôle du composant. LC gère l'activation, C l'introspection, CC la gestion du contenu et BC les connections. On peut créer d'autres interfaces de contrôle comme ServiceAttributes sur l'exemple qui sert à accéder aux attributs du composant.

En outre, Fractal est un modèle de composants et supporte plusieurs implémentations, ce qui dans notre cas permet de garder l'esprit de généralisation et de généricité. Ainsi les agents développés sur notre plate-forme pourront être hétérogènes mais rester interopérables à condition d'utiliser les mêmes format et protocole pour l'envoi de message.

Le modèle n'est cependant pas asynchrone par spécification, contrairement à d'autres modèles comme J2EE pour ne citer que lui, or cet aspect est une nécessité dans notre projet. Nous justifierons ci après notre choix.

Parmi les implémentations disponibles :

- ***Julia*** est l'implémentation complète de référence de Fractal. Ecrite en Java, elle est open source et maintenue par le consortium ObjectWeb
- ***AOKell*** est une implémentation similaire à Julia, mais basée sur AspectJ.
- ***FracTalk*** est une implémentation SmallTalk .
- ***FractNet*** est une implémentation .Net
- ***Plasma*** est une implémentation C++ , destinée aux applications multimédia.
- ***Proactive*** est une implémentation  distribuée et asynchrone en Java pour le Grid Computing.
- ***Think*** est une implémentation C destinée au développement d'OS

Nous avons préféré Julia aux autres modèles, même ceux implémentant l'asynchronisme tel ProActive. Ce manque peut cependant se révéler être un avantage, puisque ainsi nous pourrons contrôler dans notre implémentation ces mécanismes essentiels à la communication de nos agents.

Nous avons préféré nous tourner vers les MOM qui permettent de rendre Fractal asynchrone. De surcroît, Julia est la plus générique et documentée des implémentations. Elle n'est pas orientée vers un type d'application en particulier et est assez peu contraignante.

Elle possède aussi une des conditions requises: les sources sont disponibles. Bien que cet argument soit purement technique, il a eu une importance décisive dans nos recherches car nous avons dû écarter sans autre forme de procès toutes les implémentations de Fractal et des autres modèles que nous avons étudiés dont le code n'était pas accessible, car cela induisait le fait que nous aurions des difficultés à implémenter notre propre modale.

### *B. JMS*

Les Middlewares Orientés Message (MOM) fournissent des mécanismes de communication permettant à des composants de s'échanger des messages de manière asynchrone. Il nous seraient utiles pour rendre Fractal asynchrone. La notion de messages dans un MOM est la même que la notre, à savoir des messages dont la sémantique doit être interprétée par le composant destinataire.

Interagir avec un composant ne requiert que de connaître l'adresse du destinataire et le protocole à utiliser. Aucune autre information n'est nécessaire, ce qui, appliqué aux agents, permet de préserver l'autonomie et le découplage. Quand un composant envoi un message décrivant de manière déclarative sa requête, il est possible que le destinataire ne puisse pas la satisfaire voire même ne pas la comprendre. Ils se distinguent ainsi des middleware standards (Corba, RMI, ...) qui pratiquent l'invocation de méthodes, même si c'est de manière distante, au travers d'une représentation locale du composant. Les MOM sont d'ailleurs fréquemment utilisés pour réaliser l'intégration d'application hétérogènes dans les réseaux d'entreprise (EAI), on peut de la même façon garantir l'interopérabilité de notre plate-forme.

L'ensemble de ces caractéristiques motive notre choix d'adopter un MOM comme mécanisme de communication entre agents. JMS (Java Message Service) est une spécification qui définit une API standardisant l'utilisation des MOM.

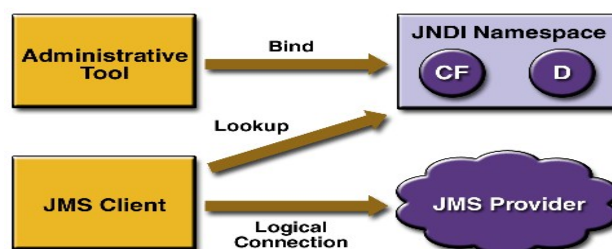



Schéma 4 : Présentation JMS (source : http://java.sun.com)

Les clients JMS s'inscrive sur le serveur JNDI (Java Naming and Directory Interface) et obtiennent une clé permettant de les identifier. Dans notre cas, chaque agent est un client. Ils peuvent ensuite envoyer et recevoir des message selon deux techniques.

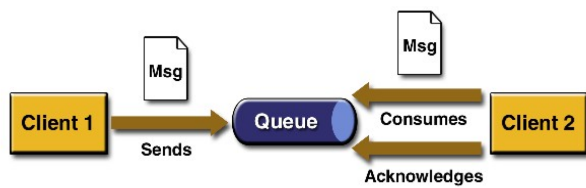



Schéma 5 : Queue JMS (source : http://java.sun.com)

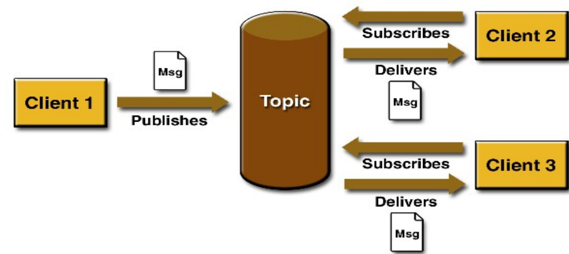



Schéma 6 : Topic JMS (source : http://java.sun.com)

Dans les deux cas, un client produit un message et l'inclue à une liste de messages accessible par le(s) destinataire(s). C'est la consommation des différente : dans le cas du point à point, il y a un seul destinataire, le message est détruit après lecture, pour le cas du publish/suscribe, un message peut être lu par tous les clients inscrits sur le Topic; c'est une forme de liste de diffusion.

Nous avons eu à choisir entre ***OpenJMS*** et ***Joram*** qui sont deux implémentations libres de JMS. Nous avons fini par opter pour Joram, bien que OpenJMS nous ait paru un peu mieux documenté, ce que l'on peut justifier par plusieurs arguments :

- OpenJMS implémente JMS 1.0.2, alors que Joram implémente JMS 1.1
- Joram fonctionner de façon distribuée et permet le clustering de Topics
- Joram semble être un projet plus activement maintenu

## *C. Implémentations*

Nous allons maintenant nous intéresser plus précisément à l'aspect technique de l'implémentation. Pour cela, il faut d'abord décrire comment se passe la réalisation d'une application avec Fractal.

Après avoir bien sûr réalisé la conception de l'application, nous allons décrire des interfaces java représentant les Interfaces des composants[1]. Puis les implémenter dans différentes classes.

Nous allons ensuite créer la fonction d'entrée du composant.

–On récupère le contexte initial et on crée un bootstrap.

–On fabrique les Type de composant à partir d'une liste d'Interfaces[1] par l'intermédiaire d'une Factory, objet permettant de créer de façon générique d'autres objets, ici des Types.

–Ensuite on instancie ces composants également par l'intermédiaire d'une factory qui prend en paramètre un contenu (ensemble de composant déjà instanciés) et un type de composant. Tous les composants implémentent l'interface Component et sont donc tous manipulables de la même façon.

–On instancie les agents de façon générique grâce à une factory. Pour l'instant, il n'existe qu'un Type d'agent. On crée aussi les Agents AnnuaireDeService.

–Enfin, on réalise l'assemblage et l'encapsulation des composants au sein des agents.

[1] Dans la suite, " interface" désignera le type ja et "Interface " les Interface des composants

L'application est prête à être lancée.

De notre point de vue, un agent est seulement un composant particulier. Tous nos agents seront donc de type java Agent, héritant de Component.

```
package org.objectweb.fractal.api;
public interface Component {
  Type getFcType ();
  Object[] getFcInterfaces ();
  Object getFcInterface (String interfaceName) throws NoSuchInterfaceException;
}

package org.objectweb.fractal.malice.api;
import org.objectweb.fractal.api.Component;
public interface Agent extends Component {
  Groupe[] getAgentGroupe();
  Groupe getAgentGroupe (String groupeName) throws NoSuchGroupeException;
}
```

On remarque qu'un agent fait obligatoirement partie d'un groupe. Nous avions plusieurs possibilités techniques pour stipuler le fait qu'un agent appartienne à un Groupe. Nous pouvions choisir par exemple de créer un composant de plus haut niveau représentant un SMA et contenant tous les agents de cette société. Nous avons préféré introduire un champs Groupes dans l'implémentation de l'agent et y accéder par l'intermédiaire d'une interface de contrôle GroupeControler. Ce choix n'a pas été simple, ce qui nous à fait pencher pour la solution retenue est que nous estimons que la gestion de la reconfiguration dynamique des groupes sera plus simple de cette façon.

Nous avons donc développé un Type prédéfini BasicAgentType qui ne possède qu'une Interface applicative BasicAgentInterface permettant uniquement de recevoir et d'envoyer des messages, en effet, on ne peut pas permette à des agents de manipuler le contenu ou d'invoquer des méthodes internes d'autres agents directement, sinon l'autonomie est perdue. Cette Interface est la même que celle du composant Boite aux lettres, elles seront liées l'une à l'autre.

```
package org.objectweb.fractal.malice.type;
public class BasicAgentInterfaceType extends
             org.objectweb.fractal.julia.type.BasicInterfaceType {

      public BasicAgentInterfaceType(String nom) {
             super(nom, "org.objectweb.fractal.malice.util.MailBoxInterface",
             false , false, false);}
}
```

```
package org.objectweb.fractal.malice.type;
public class BasicAgentType extends
                org.objectweb.fractal.julia.type.BasicComponentType
public BasicAgentType (String nom)
    throws InstantiationException{
       super( new InterfaceType[]{new BasicAgentInterfaceType(nom)});}
}
```

Il nous a fallu également implémenter des Factory afin d'instancier des agents génériques. Comme montré dans le Modèle, un agent possède plusieurs composants internes selon le pattern prédéfini. Nous préférons en effet ne pas instancier les agents directement, ainsi l'implémentation de l'agent est découplée du corps de l'agent. Le code est plus maintenable et on peut changer d'implémentation plus facilement.

Penchons nous plus particulièrement sur la boite aux lettre. C'est ce composant qui rend Fractal asynchrone.

- On commence par récupérer le contexte JNDI (serveur de nom) pour trouver le serveur JMS
- On ouvre une Connection et une Session
- On crée un message et on l'envoi

L'agent producteur de messages demande au MOM de déposer un message dans une boite aux lettres. Le MOM rend la main à l'expéditeur dès qu'il peut garantir que le message sera délivré . Le destinataire se connecte au MOM pour relever ses messages. Ainsi la production et la consommation des messages sont totalement découplés.

Dans la réalité de l'implémentation, nous avons fait une entorse au modèle. Notre annuaire de nommage n'est pas un agent, mais le service JNDI – Java Naming and Directory Interface. En effet les adresses utilisées par JMS pour diffuser les messages sont les enregistrement des composants implémentant une Queue ou un Topic dans la base JNDI. Il ne nous a pas paru nécessaire de compliquer l'implémentation en introduisant un agent responsable du serveur de nom. C'est un agent AnnuaireDeService qui gérera les entrée du JNDI. Ainsi, un agent se présentant sur la plate-forme se signalera auprès de cet Agent.

Nous avons dû ajouter deux interfaces de contrôle supplémentaires ServiceFournisController et ServicesRequisController. C'est par ces interfaces que l'AnnuaireDeService prendra connaissance des services que peut rendre un agent et de ceux qui lui sont nécessaire. Ce sont ces contrôleur qui permettent l'inscription effective de l'agent.

Le reste du code se trouvant en annexe, nous ne nous étendrons pas plus sur l'implémentation

puisqu'elle ne sert que de validation à notre modèle. Les difficultés rencontrées étaient plus d'ordre technique que conceptuelle. Nous passerons ensuite à l'implémentation de prototypes d'agents basiques permettant la création d'une application simple permettant le test de la Plate forme.

# V. EXPÉRIMENTATION

Afin de valider notre modèle, nous avons réalisé une application très simple de type Helloworld. Pour montrer comment développer et déployer une application avec malice, nous proposons ce tutorial.

Dans cet exemple, nous avons réalisé une application très simple. Sur la plate-forme seront déployés deux agents applicatifs et un annuaire de service. L'un des agents applicatifs « serveur » propose un service d'affichage, l'autre « client » le requiert. Ils forment un unique groupe.

## *A. Implementation*

Cette phase consiste à implémenter les composants applicatifs internes aux agents. Nous créerons donc les Interfaces et nous les implémenterons.

Les interfaces à implémenter sont `Service` et `Main et sont` programmées comme une interface java standard car Fractal n'impose pas de contraintes sur elles.

```
public interface Service {
 void print (String msg);
}

public interface Main {
  void main (String[] args);
}
```

La classe implémentant le composant client, ClientImpl doit implémenter l'interface Main pour lancer une requête et l'interface BindingController afin d'être reliée avec le composant boite aux lettres de l'agent.

```
public class ClientImpl implements Main, BindingController{
  private MailBox mn;
  public void main (final String[] args) {
    mb.send("requete", "print", "hello world", (String)args[0]);}
 public String[] listFc () {
```

```
    return new String[] { "mb" };
  }

  public Object lookupFc (final String cItf) {
    if (cItf.equals("mb")) {
      return mb;
    }
    return null;
  }

  public void bindFc (final String cItf, final Object sItf) {
   if (cItf.equals("mb")) {
     mb = (MailBox)sItf;
    }
  }

  public void unbindFc (final String cItf) {
    if (cItf.equals("mb")) {
      mb = null;
    } } } }
```

La classe implémentant le composant serveur, `ServerImpl`, doit implémenter l'interface Service et est identique à ClientImpl en ce qui concerne le BindingController

```
public class ServerImpl implements Service, BindingController {
      private MailBox mb;
      public void print (final String msg) {
        System.out.println(msg);
        }
      ...
      }
```

Pour ce qui est de la syntaxe des messages, nous avons fait simple pour cet exemple. On doit d'abord indiquer le type de message (ici requête), le service demandé en l'occurrence print et les paramètres d'entrée au service, soit la String « HelloWorld » dans notre exemple. On passe également en paramètre l'adresse du destinataire. A partir de ces éléments, la Boite aux lettres génère un message et l'envoi au destinataire.

### *B. Déploiement*

Pour déployer notre application, nous avons à créer un point d'entrée en utilisant MALICE afin d'instancier nos composants et nos agents. La procédure est la même que pour créer des composants Fractal standard. Un tutorial est fourni en annexe.

Notons tout de même qu'il faut modifier le fichier de configuration afin d'y faire apparaître nos alias que les Factory utiliserons.

### *C. Démonstration*

Nous avons également réalisé une application de démonstration plus complexe, mettant en avant l'aspect collaboratif des SMA. Il s'agit d'une application simulant la gestion de la sécurité d'un entrepot. Les agents applicatifs sont dotés d'une panoplie de capteurs (température, fumée, détecteur de présence...) et mettent en place la procédure appropriée en cas de déclaraion d'un feu.

Une démonstration en sera faite au cours de la soutenance.

# IV. Comparaison

Au vu de notre travail, il nous a semblé judicieux d'ajouter un comparatif entre ce que propose notre plate-forme et les autres modèles utilisés aujourd'hui dans le développement d'applications. Nous nous plaçons sur le même terrain que des plates-formes génériques utilisées dans l'industrie.

| | ***Interfaces synchrones*** | ***Communication par messages*** | ***Composition*** | ***Agent*** |
|---|---|---|---|---|
| EJB | oui | oui | non | non |
| CCM | oui | oui | non | non |
| ProActive | non | non | oui | non |
| ScalAgent | non | oui | oui | oui |
| MALICE | oui | oui | oui | oui |

Notre modèle est donc tout à fait à même sur ces points de rivaliser avec les plus utilisés des modèles à composants. Toutefois, l'implémentation de MALICE est vraisemblablement plus gourmande en ressource que les plates-formes commerciales. En effet, nous nous sommes focalisés sur le modèle et non sur les performances.

# Conclusion et Perspectives

Nous avons durant ce projet réussit à définir un nouveau modèle pour l'implémentation d'agents à base de composants, dont les spécifications sont centrées sur l'autonomie des agents et la généricité de la plate-forme. Nous sommes parvenus à une base permettant de manipuler les concepts inhérent aux agents tout en respectant l'aspect componentiel.De manière factuelle, ceci se traduit par la réalisation des modèles ainsi que d'une implémentation de la plateforme et d'applications d'exemple.

Notre contribution et donc double, à la fois conceptuelle, puisque nous proposons un ensemble de patterns, mais également technique, grâce↨aux implémentations validant notre modèle.

Au niveau du modèle, il nous semble pour l'heure assez robuste, mais il nécessite d'être mis à l'épreuve sur plus de types d'application. Et c'est par l'implémentation et son amélioration que nous comptons valider et faire éventuellement évoluer le modèle.

Dans un premier temps, il sera nécessaire de réaliser des application utilisant notre implémentation de plus grande envergure et aux caractéristiques et configurations variées afin de tester toutes les options de notre système et de les améliorer. Par exemple, nous voulons ajouter la possibilité de faire du load balancing ou permettre au système de gérer des pools d'agents pour certaines tâches critiques. De même nous avons à mettre en pratique de façon plus rigoureuse la gestion des objets partagés. Nous pensons également à définir d'autres patterns pour la structure interne des agents ainsi qu'un set étendu de composants génériques pouvant les implémenter.

Enfin, nous souhaiterions travailler sur un outil d’aide au développement inspiré du MOA, basé sur AUML, afin de proposé un outil plus complet.

# Références bibliographiques

## *Bibliographie*

[BIG01] - Bigus, Bigus. *Constructing Intelligent Agents Using Java, Second Edition*, John Willey & Sons Inc, 2001

[BOO94] - Booch G. : *Object-Oriented Analysis and Design with Applications*. Addison-Wesley Object Technology, 1994

[BOO98] – Booch G., Rumbaugh, Jacobson. *The Unified Modelling Language User Guide*, Addison-Wesley, MA, 1998.

[BRI01] - Briot, J.P. , Demazeau Y. *Principes et architectures des systèmes multi-agents*, Hermès, Lavoisier, 2001.

[BRI04] - Briot, J.P.. *Agents et composants : une dualité à explorer, JMAC'04, 2004*.

[BOI01] - Boissier, *Systèmes Multi-Agents, Cours SMA-DEA-CCSA,* SMA/ENS Mines Saint-Etienne, 2001

[BRU03] - Bruneton R.., Coupaye T., Stéfani J.-B., *The Fractal Component Model, Technical report,* the ObjectWeb Consortium, 2003

[CHE03a] - Cherfour D., André F., *Auto-adaptation de composants ACEEL coopérants*, CFSE'3, Conférence Française sur les Systèmes d'Exploitation, France, 2003

[CHE03b] - Cherfour D., André F., *Développement d'applications en environnements mobiles à l'aide du modèle de composant adaptatif ACEEL*, LMO 2003, Hermès, 2003.

[CHE93] - Chevrier, V. *Etude et mise en oeuvre du paradigme multi-agents : De ATOME à GTMAS*. Thèse de doctorat, Université Henri Poincaré, Nancy I, 1993.

[COH95] - Cohen P., Levesque H., *Communicative Actions for Artificial Agents,* First International Conference on Multi-Agent Systems, San Francisco, Cambridge, AAAI Press, 1995.

[COI96] - Cointe P., *Lecture Note in computer Science*, ECCOP'96 - Object-Oriented Programming, 10th European Conference, Autriche, 1996.

[DAV05] - David P.-C., *Développement de composants Fractal adaptatifs : un langage dédié à l'aspect d'adaptation*, Thèse de doctorat, Université de Nantes / École des Mines de Nantes, 2005.

[DRO93 ] Drogoul A., Corbara B., Fresneau D. & Lalande S. *Simulating the Sociogenesis Process in Ant Colonies with MANTA, Practice of Autonomous Systems II*, MIT Press, Cambridge, 1993

[DUR87] - Durfee E-H, Lesser V., Corkill D., *Coherent Cooperation among Communicating Problem Solvers,* IEEE Transactions of Computers, 1987.

[FER95] - Ferber, J.. *Les systèmes multi-agents: Vers une intelligence collective*. InterEditions, 1995.

[FER99] - Ferber J., Multi-Agent Systems. An Introduction to Distributed Artificial Intelligence. Addison-Wesley, 1999.

[JEO00] - Jeon H., Petrie C., Cutkosky M. *JATLite: A Java Agent Infrastructure with Message Routing, Stanford Center for Design Research (CDR),* 2000

[SHO93] - Shoham Y., *Agent oriented programming*, Artificial Intelligence, 1993

[WOO95] – Wooldridge M., Jennings N., *Intelligent Agents : Theory and Practice*, Knowledge Engineering Review, 1995.

[WOO00] – Wooldridge M., *Reasoning about Rational Agents*, The MIT Press, 2000.

[WOO02] – Wooldridge M., *An Introduction to Multiagent Systems*. John Wiley and Sons, 2002.

## *Webliographie*

OMG. The Object Management Group, http://www.omg.org/

AUML. The Agent Unified Modelling language, http://www.auml.org/

FIPA: Foundation for Intelligent Physical Agents. Specifications. 1997. http://www.fipa.org

Plate-forme JADE: Java Agent Development Framework, 2000. http://jade.cselt.it/

Plate-forme ZEUS http://www.labs.bt.com/projects/agents/zeus

Plate-forme MADKIT, 2003 http://www.madkit.org

Une présentation des plates-formes et environnements pour la construction des systèmes multi-agents http://epfl.ch/~iagents/Slides/lecture2_v1.1.ppt

Liste plates-formes multi-agents http://www.agentlink.org/resources/agent-software.php

Liste plates-formes multi-agents http://www.agentbuilder.com/AgentTools/

**KQML** http://www.cs.umbc.edu/kqml/

Autres liens

http://www.msci.memphis.edu/~franklin/AgentProg.html

http://cui.unige.ch/ScDep/Cours/IA/ia00-01

http://www.agentlink.org/resources/agent-software.php

http://turing.cs.pub.ro/auf2/html/chapters/chapter2/sommaire.html

http://www.magma.ca/~mrw/agents/

# Annexe 1 : Spécification FIPA

Les agents doivent pouvoir accéder à un annuaire de services et un annuaire d'agents, pour communiquer avec le reste de la plate-forme et recourir le cas échéant aux services d'un autre agent. Les agents s'inscrivent auprès de ces annuaires lors de leur déploiement. On doit pouvoir retrouver et contacter un agent à partir des données présentes dans l'annuaire (nom de l'agent, localisation, protocole de communication en cas de cœxistence de plusieurs protocoles sur la plate-forme…). Chaque agent doit donc posséder un identifiant unique permettant de dialoguer avec lui. Ces deux annuaire ont à peu près les mêmes caractéristiques, ils permettent une certaines autonomie, en effet des agents peuvent communiquer sans se connaître directement. De même ces annuaires en conjonction avec la communication par envoi de message impliquent un découplage quasi-total, ce qui offre la possibilité de remplacer des composants à chaud de façon transparente, il suffit dans l'annuaire de services de modifier le nom de l'agent fournissant le service et sa localisation. De plus on ne s'expose pas au problème de référence fantôme pouvant apparaître dans certains systèmes comme RMI (pas de référence pointant directement sur un objet distant mais communication par messages).

Ces services peuvent très bien être rendus par des agents spécifiques et non inclus directement à la plate-forme (ce qui permet de choisir différentes politiques selon l'application à réaliser). Une piste également sous entendue est d'utiliser un classement par ontologie des services, afin de pouvoir proposer des alternatives proches en cas de défaillance d'un agent, seul à offrir le service désiré.

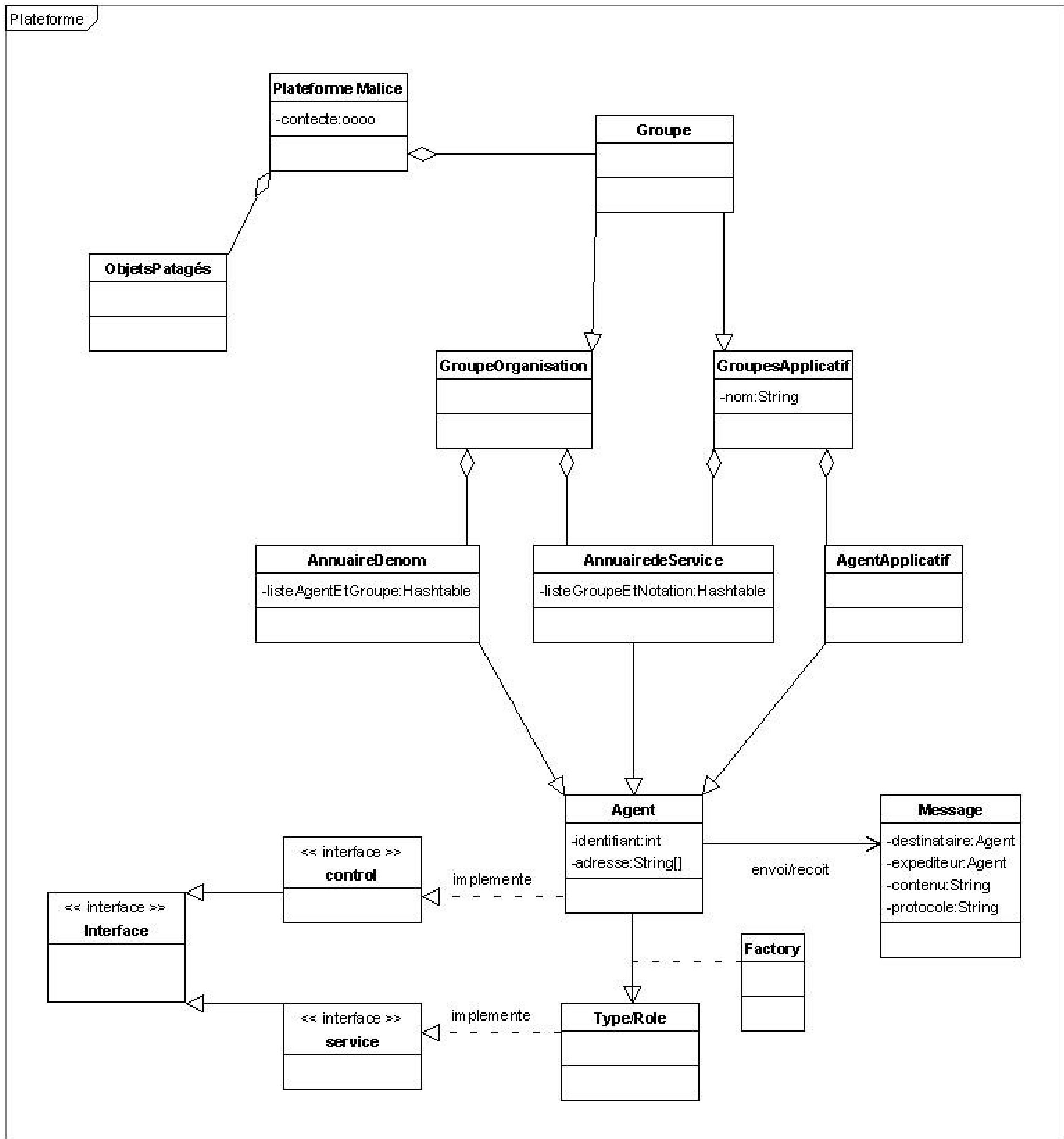
Les agents communiquent par envoi de messages qui constituent des actes de langage traduits en ACL (Agent Communication Language). La plus grande partie des recommandations concerne ce point, en effet, le but de la FIPA est de créer une certaine interopérabilité entre les systèmes répondant à ses normes, le point essentiel est donc la forme de la communication. Il convient donc de prévoir un système d'encodage/décodage des messages. A ce titre, plus que le corps du message, c'est l'entête qui permet d'assurer l'interopérabilité (même s'il peut nécessiter une remise en forme ou une traduction, chaque champ requis par la FIPA doit avoir un équivalent sur tous les systèmes respectant les principes FIPA). Eventuellement plusieurs types de messages peuvent être défini et circuler entre agents. Ces fonctions peuvent être remplies par un (ou plusieurs) composants inclus dans chaque agent.

Pour acheminer ces messages, il faut également une couche de transport, définissant les adresses, réalisant les accès, distribuant les messages à travers le réseau. Elle peut être vu comme un agent spécifique (ou plusieurs), analogue à un (groupe de) facteur(s) ou alors être intégrée à la plate-forme. Il pourra permettre plusieurs modes de diffusion, tels que le point à point, le broadcast, le topic ou le multicast.

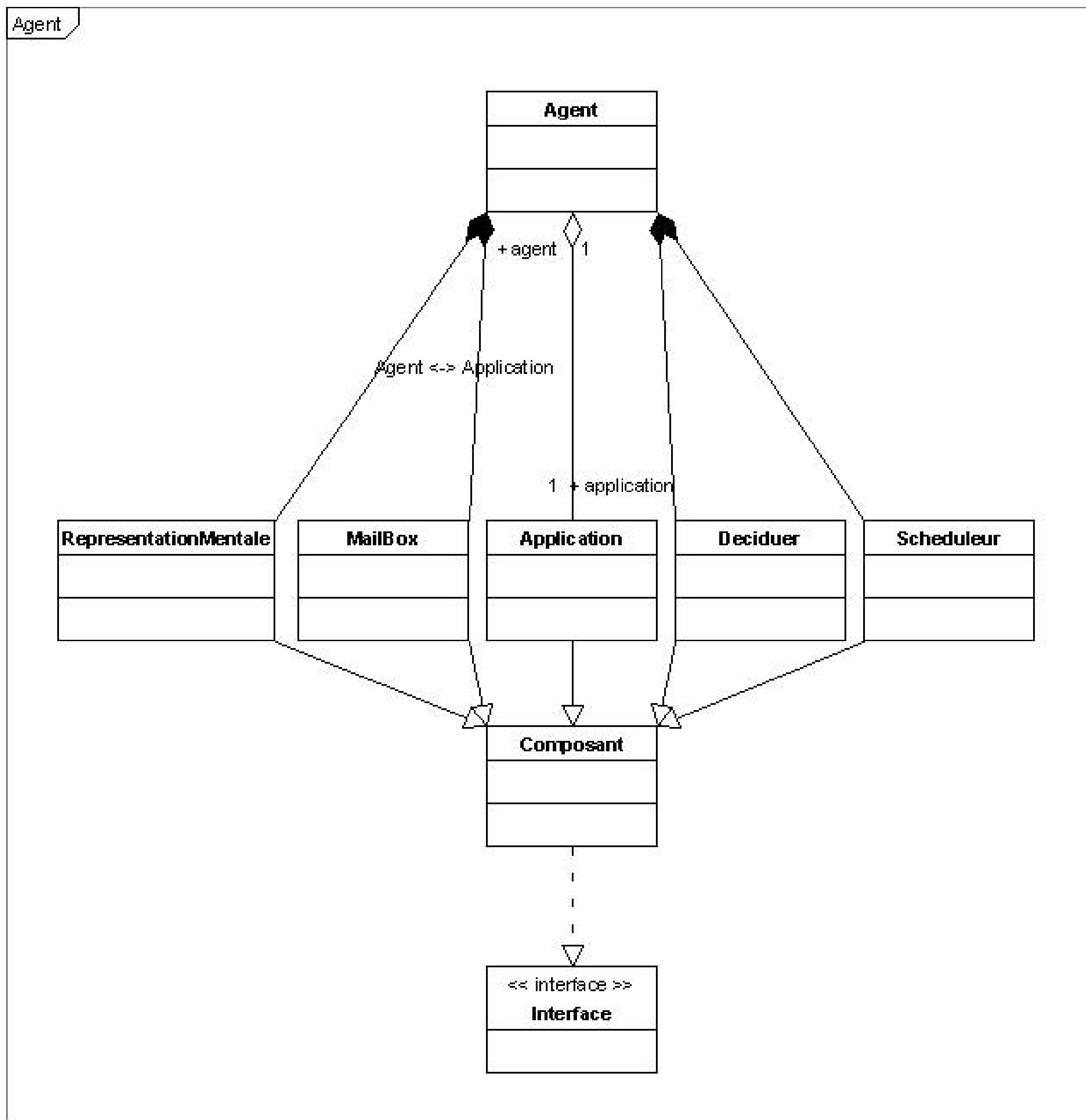
Le standard spécifie aussi le Langage de Communication d'Agents (Agent Communication Language - ACL) - La communication des agents est basée sur l'envoi de messages. Le langage FIPA ACL est le langage standard des messages et impose le codage, la sémantique et la pragmatique des messages. La norme n’impose pas de mécanisme spécifique pour le transport interne de messages. Plutôt, puisque les agents différents pourraient s'exécuter sur des plates-formes différentes et utiliser technologies différentes d'interconnexion, FIPA spécifie que les messages transportés entre les plates-formes devraient être codés sous forme textuelle. On suppose que l'agent est en mesure de transmettre cette forme textuelle. La norme FIPA préconise des formes communes pour les conversations entre agents par la spécification de protocoles d'interaction, qui incluent des protocoles simples de type requête-réponse, mais aussi des protocoles plus spécifiques.

Voici la liste des élément devant faire partie de la plate-forme selon la FIPA.

# Annexe 2 : PLATE-FORME



# Annexe 3 : Agent



# Annexe 4 : Configurations

```
# agent interface
(agent-itf
  (agent org.objectweb.fractal.malice.api.Agent)
)

# AgentTypeFactory interface
(agent-type-factory-itf
  (agent-type-factory org.objectweb.fractal.api.type.TypeFactory)
)

# GenericFactory interface
(agent-generic-factory-itf
  (agent-generic-factory org.objectweb.fractal.api.factory.GenericFactory)
)

# agent implementation
(agent-impl
  ((org.objectweb.fractal.julia.asm.MixinClassGenerator
    AgentImpl
    org.objectweb.fractal.julia.BasicControllerMixin
    org.objectweb.fractal.malice.api.BasicAgentMixin
    # to check type related constraints, and for collection interfaces support:
    org.objectweb.fractal.julia.TypeComponentMixin
  ))
)



# AgentTypeFactory implementation
(agent-type-factory-impl
  ((org.objectweb.fractal.julia.asm.MixinClassGenerator
    TypeFactoryImpl
    org.objectweb.fractal.julia.BasicControllerMixin
    org.objectweb.fractal.julia.type.BasicTypeFactoryMixin
  ))
)
```

```
# AgentGenericFactory implementation
(agent-generic-factory-impl
  ((org.objectweb.fractal.julia.asm.MixinClassGenerator
    GenericFactoryImpl
    org.objectweb.fractal.julia.BasicControllerMixin
    org.objectweb.fractal.julia.BasicInitializableMixin
    org.objectweb.fractal.julia.loader.UseLoaderMixin
    org.objectweb.fractal.julia.type.UseTypeFactoryMixin
    org.objectweb.fractal.julia.factory.BasicGenericFactoryMixin
    # to check the component content descriptor with the Java Reflection API:
    org.objectweb.fractal.julia.factory.CheckGenericFactoryMixin
  ))
)

# BindingController implementation (for composite components)
(composite-binding-controller-impl
  ((org.objectweb.fractal.julia.asm.MixinClassGenerator
    CompositeBindingControllerImpl
    org.objectweb.fractal.julia.BasicControllerMixin
    org.objectweb.fractal.julia.control.binding.BasicBindingControllerMixin
    # to initialize the BasicBindingControllerMixin from the component's type:
    org.objectweb.fractal.julia.control.binding.TypeBasicBindingMixin
    # to check some basic pre conditions (interface not already bound, ...)
    org.objectweb.fractal.julia.UseComponentMixin
    org.objectweb.fractal.julia.control.binding.CheckBindingMixin
    # to check type related constraints for bindings:
    org.objectweb.fractal.julia.control.binding.TypeBindingMixin
    # to check content related constraints for bindings:
    org.objectweb.fractal.julia.control.content.UseSuperControllerMixin
    org.objectweb.fractal.julia.control.binding.ContentBindingMixin
    # to check lifecycle related constraints for bindings:
    org.objectweb.fractal.julia.control.lifecycle.UseLifeCycleControllerMixin
    org.objectweb.fractal.julia.control.binding.LifeCycleBindingMixin
    # to manage the getFcItfImpl links of the Interface objects:
    # choose one of ComponentBindingMixin and OptimizedCompositeBindingMixin
    # (the last one creates and updates shortcuts links when possible)
    org.objectweb.fractal.julia.control.content.UseContentControllerMixin
    # org.objectweb.fractal.julia.control.binding.CompositeBindingMixin
    org.objectweb.fractal.julia.control.binding.OptimizedCompositeBindingMixin
  ))
)
```

```
(bootstrap
  (
    'interface-class-generator
    (
      'component-itf
      'type-factory-itf
      'generic-factory-itf
      (loader org.objectweb.fractal.julia.loader.Loader)
    )
    (
      'component-impl
      'type-factory-impl
      'generic-factory-impl
      # choose one of the following classes:
      # the first one loads all classes from the classpath
      # the second one can generate missing classes on the fly, dynamically
      # org.objectweb.fractal.julia.loader.BasicLoader
      org.objectweb.fractal.julia.loader.DynamicLoader
    )
    (
      # no interceptors
    )
    org.objectweb.fractal.julia.asm.MergeClassGenerator
    none
  )
)

(composite
  (
    'interface-class-generator
    (
      'component-itf
      'binding-controller-itf
      'content-controller-itf
      'super-controller-itf
      # only if super-controller-itf does not designate the Julia interface:
      # 'julia-super-controller-itf
      'lifecycle-controller-itf
      # only if lifecycle-controller-itf does not designate the Julia interface:
```

```
    # 'julia-lifecycle-controller-itf
    'name-controller-itf
  )
  (
    'component-impl
    'composite-binding-controller-impl
    'content-controller-impl
    'super-controller-impl
    'composite-lifecycle-controller-impl
    'name-controller-impl
  )
  (
    # no interceptors
  )
  org.objectweb.fractal.julia.asm.MergeClassGenerator
  'optimizationLevel
 )
)
```

# Annexe 4 : Tutoriel Fractal

http://fractal.objectweb.org/tutorials/fractal/index.html

## 1 Creating the component types

We begin by creating objects that represent the types of the components of the application. In order to do this, we must first get a bootstrap component. The standard way to do this is the following one (this method creates an instance of the class specified in the `fractal.provider` system property, and uses this instance to get the bootstrap component):

```
Component boot = Fractal.getBootstrapComponent();
```

We then get the TypeFactory interface provided by this bootstrap component:

```
TypeFactory tf = (TypeFactory)boot.getFcInterface("type-factory");
```

We can then create the type of the composite component, which only provides a `Main` server interface named "m":

```
// type of the root component
ComponentType rType = tf.createFcType(new InterfaceType[] {
  tf.createFcItfType("m", "Main", false, false, false)
});
```

The type of the client and server components are created in a similar way:

```
// type of the client component
ComponentType cType = tf.createFcType(new InterfaceType[] {
  tf.createFcItfType("m", "Main", false, false, false),
  tf.createFcItfType("s", "Service", true, false, false)
});
// type of the server component
ComponentType sType = tf.createFcType(new InterfaceType[] {
  tf.createFcItfType("s", "Service", false, false, false),
  tf.createFcItfType("attribute-controller", "ServiceAttributes", false,
false, false)
});
```

## 2 Creating the component templates

We could now create the components directly, but we will use intermediate template components here, in order to illustrate how they can be used. We must therefore create template components corresponding to the components of the application. In order to do this, we first get the GenericFactory interface provided by the bootstrap component:

```
GenericFatory cf = (GenericFatory)boot.getFcInterface("generic-factory");
```

We then create a composite template component to instantiate the composite component. Here the "compositeTemplate" argument is supposed to describe, in the Fractal implementation that is actually used, a component with the Factory, BindingController and ContentController interfaces. Likewise, the "composite" argument is supposed to describe a component with the

LifeCycleController, BindingController and ContentController interfaces.

```
// template to create the root component
Component rTmpl = cf.newFcInstance(
  rType, "compositeTemplate", new Object[] {"composite", null});
```

We then create a template to instantiate client component instances. Here the "primitiveTemplate" argument is supposed to describe, in the Fractal implementation that is actually used, a component with the Factory and BindingController interfaces. Likewise, the "primitive" argument is supposed to describe a component with the LifeCycleController and BindingController interfaces. The "ClientImpl" argument is of course the name of the client component class.

```
// template to create the client component
Component cTmpl = cf.newFcInstance(
  cType, "primitiveTemplate", new Object[] {"primitive", "ClientImpl"});
```

We then create a template to instantiate server component instances. Here the "parametricPrimitiveTemplate" argument is supposed to describe, in the Fractal implementation that is actually used, a component with the Factory, BindingController and AttributeController interfaces. The "ServerImpl" argument is of course the name of the server component class.

```
// template to create the server component
Component sTmpl = cf.newFcInstance(
  sType, "parametricPrimitiveTemplate", new Object[]
{"parametricPrimitive", "ServerImpl"});
```

We can then configure the attributes of the server component template, which has the same attribute controller interface as the server component. The template attributes will be automatically copied into all the components created by the template.

```
ServiceAttributes sa = (ServiceAttributes)sTmpl.getFcInterface("attribute-
controller");
sa.setHeader("-> ");
sa.setCount(1);
```

At this stage we have the following architecture:

We can then either instantiate each template one by one, put the resulting primitive components inside the composite component, connect all these components, and finally start them. But we can also put the primitive templates inside the composite template, connect these templates together, and then instantiate the whole application by just instantiating the composite template component. This is what we do here.

We begin by putting the primitive template components inside the composite one:

```
ContentController cc = (ContentController)rTmpl.getFcInterface("content-
controller");
cc.addFcSubComponent(cTmpl);
cc.addFcSubComponent(sTmpl);
```

We then bind the internal client interface "m" of the composite template to the server interface "m" of the client template:

```
((BindingController)rTmpl.getFcInterface(
 "binding-controller")).bindFc("m", cTmpl.getFcInterface("m"));
```

Finally, we bind the client interface "s" of the client template to the server interface "s" of the server template:

```
((BindingController)cTmpl.getFcInterface(
  "binding-controller")).bindFc("s", sTmpl.getFcInterface("s"));
```

The final architecture is the following:

## 3 Instantiating and launching the application

Now that the template components have been created and bound to each other, the application components can be instantiated and bound to each other automatically, by just calling the `newFcInstance` method on the root composite template component:

```
Component rComp = ((Factory)rTmpl.getFcInterface(
  "factory")).newFcInstance();
```

The result is shown in the figure below:

Likewise, all the application components can be started automatically by just calling the `startFc` method on the root application component:

```
((LifeCycleController)rComp.getFcInterface(
  "lifecycle-controller")).startFc();
```

We can finally (!) launch the application:

```
((Main)rComp.getFcInterface("m")).main(null);
```