IMPLEMENTATION DES E-MARKETS SOUS DES PLATEFORMES MULTI-AGENTS

Hamza Halioui, Maher Khemakhem

IMPLEMENTATION DES E-MARKETS SOUS DES PLATEFORMES MULTI-AGENTS

H. HALIOUI, M. KHEMAKHEM

MIRACL, Faculté des Sciences Economiques et de Gestion de Sfax, Tunisie

Résumé

Après la présentation des trois plateformes multi-agents *JADE*, *MADKIT et MAGIQUE* et leurs principaux packages, offerts par leurs *API,* nous avons prévu une première implémentation d'un marché électronique à base d'agents sous la plateforme *JADE.* Nous avons défini les trois agents principaux à savoir *Broker, Provider et Client* en se servant des classes offertes par *l'API JADE* comme *Agent, Behaviour, DFAgentDescription*. Dans la deuxième implémentation du marché électronique sous *MADKIT,* nous définissons, aussi, trois agents principaux : *Broker, Provider et Client* en se servant des classes offertes par l'*API MADKIT* comme *Agent, AgentAdress et ACLMessage*. Dans la troisième implémentation du marché électronique sous *MAGIQUE*, nous avons défini deux agents principaux *auctioneer et bidderOne* en se servant du package *fr.lifl.magique* qui offre des classes comme *Agent et Start*. Ainsi, nous avons défini quatre compétences à savoir *AuctioneerSkill, GraphAuctioneer, BidderSkill, GraphBidder* qui héritent de la classe *DefaultSkill* offerte par le package *fr.lifl.magique.skill.* Ces trois implémentations nous ont permis de détecter les similitudes, comme la simplicité d'exécution des agents sur ces trois plateformes et la possibilité de mise en œuvre des scénarios pour des marchés électroniques, et les spécificités de chacune des trois plateformes. Par exemple, les travaux en matière des services web [3] et les protocoles d'interaction [1] sont plus développés en JADE.

1. Introduction

Objet de longues dates de recherches en intelligence artificielle distribuée, les systèmes multi-agents forment un type intéressant de modélisation de société, et ont à ce titre des champs d'application larges. L'une de ces champs, où on peut prévoir une application importante des agents seraient les marchés électroniques, dans lesquels un utilisateur peut donner à ses agents la capacité d'achat, de vente ou bien de faire des recherches sur les marchés pour des services répondant à ses besoins. Ainsi, la conception d'un modèle qui définit les rôles et les relations entre les agents serait une première étape pour la réalisation de ces marchés.

Dans d'autre contexte, une plateforme des systèmes multi-agents est une infrastructure de logiciels utilisée comme environnement pour le déploiement et l'exécution d'un ensemble d'agents. Le développeur pourra alors créer des agents pour une plateforme et les employer sur tous les systèmes qui supportent cette plateforme sans changer le code. De plus, une plateforme devrait cacher au développeur les détails d'implémentation des protocoles de communication. Comme le choix d'une plateforme d'agents a une grande influence sur la conception et la mise en œuvre des agents, FIPA[14] a produit les normes qui concernent une plateforme d'agents doit être (ACL). Plusieurs plateformes multi-agents existent : les plateformes de simulation, de développement et d'exécution dont les plus connues *JADE[11], MADKIT[13] et MAGIQUE[12].*

Nous nous intéressons dans cet article à présenter les trois plateformes déjà cités et leurs principaux packages, décrits en code Java, et d'implémenter trois applications de marchés électroniques sur chacune des trois.

2. Présentation des plateformes

2.1. JADE

JADE, Java Agent DEvloppement framework est une plateforme multi-agents créée par le laboratoire TILAB et décrite par Belliffemine et al dans [11]. JADE permet le développement des systèmes multi-agents et d'applications conformes aux normes FIPA[14].

L'architecture du logiciel est basée sur la coexistence de plusieurs *Machine Virtuelles Java (JVM)* et la communication se fait par la méthode *RMI (Remote Method Invocation)* de Java entre machines virtuelles différentes. Chaque VM est un conteneur (container) d'agents qui fournit un environnement d'exécution complet pour l'exécution des agents et permet d'avoir plusieurs agents qui s'exécutent simultanément sur le même hôte.

JADE est alors composée de plusieurs conteneurs d'agents. Chaque conteneur d'agents est un environnement multi-agents d'exécution composé d'un thread d'exécution pour chaque agent et, en plus des threads crées à l'exécution par le système *RMI* pour envoyer des messages.

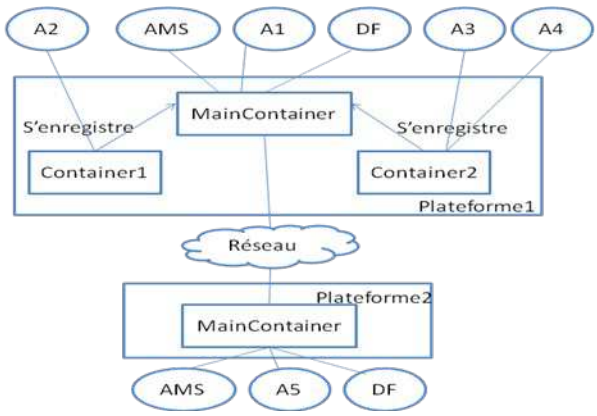

Figure 1 : Architecture des conteneurs JADE

La plateforme 1 de la figure 1 contient trois conteneurs à savoir *Main-Container, Container1 et Container2*. Le conteneur *Container1* possède un seul agent qui est A2 tandis que *Containe2* possède deux agents A3 et A4. Le conteneur spécial Main-Container joue le rôle d'un frontal de la plateforme : il présente la plateforme toute entière pour le monde extérieur et contient deux modules principaux nécessaires aux normes FIPA :

- *DF Director Faciltor* : fournit un service de pages jaunes à la plateforme
- et *AMS Agent Management System* : supervise l'enregistrement des agents, leur authentification, leur accès et l'utilisation du système.

Ces deux modules sont activés à chaque démarrage de la plateforme.

2.2. MADKIT

MadKit (Multi-Agents Developpement Kit) [13] est une plateforme de développement multi-agents développée en 1996 par M.Ferber au laboratoire LIRMM de l'université de Montpellier. Elle est écrite en Java et conçue selon le modèle d'organisation Alaadin AGR (Agent/Groupe/Role).

MadKit est construit autour du concept de ''micro-noyau'' et ''d'agentification des services''. Le Micro-noyau de MadKit est très petit, moins de 100Ko de code, car il ne gère que les organisations : groupes et rôles et les communications à l'intérieur de la plateforme.

Le mécanisme de distribution, les outils de développement et de surveillance des agents, les outils de visualisation et de présentation sont des outils supplémentaires qui viennent sous la forme d'agents que l'on peut ajouter au noyau de base.

On peut ainsi facilement développer à l'aide de MadKit et ensuite utiliser les agents ainsi construits dans des applications commerciales.

On reconnait à MadKit les qualités suivantes :

- Simplicité de mise en œuvre et de prise en main,
- Intégration très facile de la distribution au sein d'un réseau
- L'aspect pratique et efficace des concepts organisationnels pour créer différents types d'applications
- Hétérogénéité des applications et des types d'agents utilisables : on peut faire tourner sur MadKit aussi bien des applications utilisant des agents réactifs simples de type fourmis, plus de 50000 agents ayant un comportement simple ont tourné sur MadKit, que des applications disposant d'agents cognitifs sophistiqués.

2.3. MAGIQUE

MAGIQUE (Multi-Agents hiérarchIQUE) [12] est une plateforme de développement multi-agents développé par l'équipe SMAC du laboratoire LIFL de l'université de Lille. Elle propose à la fois un modèle organisationnel, basé sur une organisation hiérarchique par défaut, et un modèle d'agent, qui est basé sur une construction incrémentale des agents.

La structure hiérarchique permet de proposer un mécanisme de délégation de compétences entre les agents, facilitant ainsi le développement. Dans *MAGIQUE*, un agent est une entité possédant un certain nombre de compétences. Ces compétences permettent à l'agent de tenir un rôle dans une application multi-agents. Les compétences d'un agent peuvent évoluer dynamiquement (par échanges entre agents) au cours de l'existence de celui-ci, ce qui implique que les rôles qu'il peut jouer (et son statut) peuvent évoluer au sein du SMA.

Un agent est construit dynamiquement à partir d'un agent élémentaire vide, par enrichissement de ses compétences.

Du point de vue de l'implémentation, une compétence peut être perçue comme un composant logiciel regroupant un ensemble cohérent de fonctionnalités. Les compétences peuvent donc être développées indépendamment de tout agent et donc réutilisées dans différents contextes.

Du point de vue méthodologique, la construction d'applications avec *MAGIQUE* doit suivre la démarche suivante :

- Identifier les rôles impliqués dans l'application et les compétences qui correspondent à ce rôle
- Développer les compétences (ou réutiliser)
- Créer les agents et les répartir dans une structure hiérarchique
- Répartir les rôles (et donc les compétences) sur les différents agents.

3. Principaux packages :

3.1. Les packages de JADE

Le développeur d'agents doit étendre la classe *Agent* et implémenter les tâches spécifiques de l'agent par une ou plusieurs classes de comportements *Behaviour,* les instancier et les ajouter à l'agent. L'API JADE[11] offre plusieurs packages pour faciliter ces tâches dont les plus utilisés : *jade.core, jade.domain et jade.lang.acl*

jade.core est un package qui contient le micro noyau du système *JADE*. Il contient en particulier des classes comme *Agent, Runtime et ProfileImpl* et des sous packages comme *jade.core.behaviour*.

La classe *Agent* qui est une super classe commune pour tous les agents définis par l'utilisateur contient les méthodes nécessaire pour définir les tâches de base pour un agent comme :

- *setup(), takeDown()* : pour permettre la saisie des codes de démarrage et de suppression des agents en se servant des services d'enregistrement de l'agent *DF*. *setup()* inclut au moins un objet de comportement de type *Behaviour* afin que l'agent soit capable d'agir.
- *send(ACLmessage msg), receive(MessageTemplate msg)* : pour l'envoie et la réception des messages ACL vers et des autres agents.

La classe *Runtime* permet de contrôler le système d'exécution JADE depuis une application externe. L'instance de cette classe est accessible à travers la méthode statique *instance()*. Elle possède la méthode *createAgentContainer(Profile p)* qui crée un nouveau agent conteneur pour la *JVM* actuelle, permettant l'accès à travers un objet proxy.

La classe *ProfileImpl* permet au noyau JADE de récupérer toutes les classes dépendantes de la configuration et le paramétrage de démarrage.

Le sous package *jade.core.behaviour* contient les classes nécessaires pour implémenter des comportements basiques pour les agents comme *Behaviour, CyclicBehaviour et CompositeBehaviour.*

Le deuxième package *jade.domain* contient des spécifications des agents FIPA et des ontologies. Il contient des classes comme *DFService* et des sous packages comme *jade. Domain.FIPAAgentManagement.*

La classe *DFService* contient les méthodes statiques pour la communication avec les services de DF qui inclut des spécifications FIPA.

Et le sous package *jade.domain.FIPAAgentManagement* contient la définition des ontologies pour les gestionnaires des agents FIPA y compris la classe *DFAgentDescription.*

Le troisième package *jade.lang.acl* contient un support pour le langage de communication des agents FIPA, ACL y compris des classes comme *ACLMessage et MessageTemplate.*

ACLMessage implémente un message ACL conformément aux spécifications ACL des messages pour FIPA 2000. Et *MessageTemplate* fait une correspondance pour les messages ACL entrants.

3.2. Les packages de MADKIT

Toutes les opérations de base permettant le déploiement des agents sous MadKit[13] sont incluses dans un package *madkit.kernel*. Il contient un ensemble de classes comme *AbstractAgent, AgentAdress et Message.* La classe de base d'un agent MadKit, *AbstractAgent,* offre les fonctionnalités pour la gestion du cycle de vie des agents, de l'organisation et de la communication. Pour gérer le cycle de vie des agents, la classe *Agent*, qui hérite de *AbstractAgent* des méthodes comme *activate()* pour l'enregistrement des agents et l'affectation initiale des rôles et des groupes et *live()* pour définir le comportement principal des agents.

Les agents sont aussi, dotés des primitives comme *createGroup()* permettant de créer des groupes *requestRole()* pour demander leurs rôles et *getAgentWithRole()* pour sélectionner les adresses des agents jouant le même rôle au sein du groupe.

Ainsi, les agents disposent des fonctions *sendMessage() et broadcastMessage()* pour l'envoi simple ou multiple des messages et *receive()* pour la réception.

La classe *AgentAdress* inclut un constructeur permettant d'attribuer des adresses aux agents sous la forme ''mka :a,b@k'' tels que ''a'' le nom de l'agent, ''b'' l'identificateur de l'agent, et ''k'' l'adresse du noyau.

La classe *Message* offre des méthodes comme *getSender() et getReceiver()* qui retournent les adresses des agents qui envoient ou reçoivent les messages.

Un autre package *madkit.messages* a été ajouté offrant la classe *ACLMessage* pour reconnaître les actes du langage *FIPA ACL* comme *REQUEST, MAKE-CONTRACT et ACCEPT-CONTRACT.*

3.3. Les packages de MAGIQUE

Les agents que les on peut créer, correspondants à un modèle d'agent construit incrémentalement à partir d'un agent atomique, ne possèdent que deux compétences : la communication et l'acquisition de compétences. Ainsi, l'API[12] offre une classe *AtomicAgent* correspondant à ce modèle. Elle offre les méthodes *connectTo() et send()* pour la communication et *addSkill(Skill s)* pour l'acquisition dynamique de compétences. Cependant pour pouvoir intervenir dans une organisation à la *MAGIQUE,* les agents doivent posséder un certain nombre de fonctionnalités de base. La classe *Agent,* obtenue par enrichissement de la classe *AtomicAgent*, permet de construire l'hiérarchie MAGIQUE en connectant

les agents à leur supérieur hiérarchique par la méthode *connectToBoss()*. Ce qui permet aux agents d'utiliser le mécanisme automatique de délégation de compétences. Il suffit de ne pas indiquer le destinataire lors d'une requête *perform()*.

Pour ajouter des compétences aux agents, nous utilisons le package *fr.lifl.magique.skill.* Ces compétences permettent à chaque agent de jouer un rôle dans l'organisation *MAGIQUE.*

4. Implémentation des e-market

4.1. E-market sous JADE

Comme tout scénario de transaction électronique, nous prévoyons deux agents principaux Acheteur *Client* et Vendeur *Provider*. L'agent *Client* demande la liste des articles du *Provider.* A sa réception du message, l'agent *Provider* envoie une liste. Ensuite, le *Client* transmet cette liste vers un troisième agent *Broker.* Celui-ci est aussi nécessaire dans ce scénario pour lancer le conteneur principal et démarrer les deux autres agents. (figure 2)

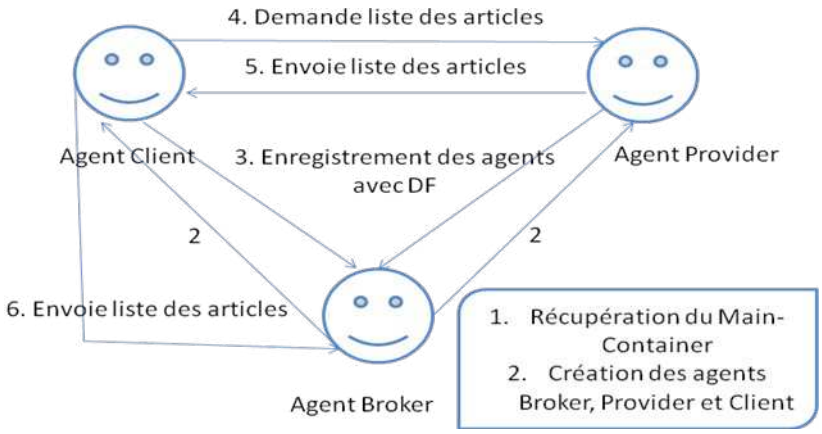

Figure 2 : Implémentation d'un e-market sous JADE

Nous commençons cette implémentation par la récupération du *Main-Container de JADE*, puis le lancement des trois agents *Borker, Provider et Client*

```
Runtime rt= Runtime.instance() ;
ProfileImpl p= new ProfileImpl(false) ;
AgentContainer container=rt.createAgentContainer(p) ;
AgentController Agent=null ;
Agent=container.createNewAgent(''Broker'', ''market.Broker'', null) ;
Agent.start() ;
Agent=container.createNewAgent(''Provider'', ''market.Provider'', null) ;
Agent.start() ;
Agent=container.createNewAgent(''Client'', ''market.Client'', null) ;
Agent.start() ;
```

Puis, l'agent *Client* envoie un message, au niveau de sa fonction *setup(),* (flèche 4) vers l'agent *Provider* demandant la liste des articles à travers ces commandes :

```
ACLMessage msg= new ACLMessage(ACLMessage.INFORM) ;
msg.setContent(''demande la liste des articles'') ;
msg.addReceiver(new AID(''Provider'', AID.ISLOCALNAME)) ;
send(msg) ;
```

La méthode *action()* incluse dans *setup()* de l'agent *Provider* contient le code lui permettant de recevoir le message du *Client* et d'envoyer un objet contenant des articles ''ART1'', ''ART2'' (flèche 5).

```
ACLMessage msg= receive(MessageTemplate.MatchPerformative(ACLMessage.INFORM)) ;
if(msg.getContent()=''demande la liste des articles'') ;
ACLMessage reply=msg.createReply() ;
Object[] obj={''ART1'', ''ART2''} ;
reply.setContentObject(obj) ;
myAgent .send(reply) ;
```

Ainsi, ayant reçu la liste des articles de l'agent *Provider,* l'agent *Client* transmet la liste de type objet vers le troisième agent *Broker* au niveau de sa fonction *action()*, aussi incluse dans *setup()* (flèche 6).

```
ACLMessage msg= receive(MessageTemplate.MatchPerformative(ACLMessage.INFORM)) ;
obj=(Object[])msg.getContentObject() ;
ACLMessage msg1= new ACLMessage(ACLMessage.INFORM) ;
msg1.setContent(obj) ;
msg1.addReceiver(new AID (''Broker'', AID.ISLOCALNAME)) ;
send (msg1) ;
```

4.2. E-market sous MADKIT

Nous disposons dans cette implémentation de trois types d'agents : *Broker,* Acheteur *Client* et Vendeur *Provider* comme indiqué dans la figure 3 :

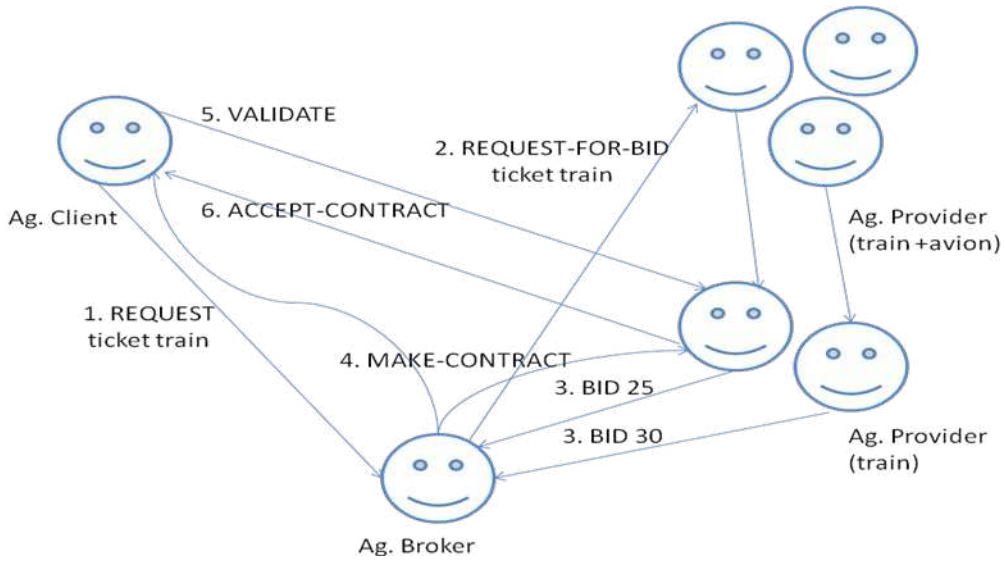

Figure 3 : Implémentation d'un e-market sous MADKIT

Nous commençons cette implémentation par la création d'un groupe *''travel-clients''* dans chacun des agents *Client et Broker* au niveau de leurs fonctions *activate():*

createGroup(true,''travel'', ''travel-clients'', null, null) ;

Et affecter les rôles *''Broker'' et ''Client''* pour ces deux agents :

requestRole(''travel'', ''travel-clients'', ''broker'', null) ;
requestRole(''travel'', ''travel-clients'', ''client'', null) ;

Pour l'agent *Client,* la récupération de l'adresse de l'agent *Broker* se fait comme suit :

broker=getAgentWithRole(''travel'', ''travel-clients'', ''broker'') ;

Puis, il envoie sa requête à *Broker* pour chercher un billet de voyage soit par train ou par avion (flèche 1).

sendMessage(broker, new ACLMessage(''REQUEST'', ticket)) ;

La fonction *handleMessage(ACLMessage m)* incluse dans la fonction *live()* du *Broker* permet la récupération du message du *Client* et la sélection du type, du contenu et de l'adresse de l'émetteur.

m.getAct().equalsIgnoreCase(''REQUEST'') ;
String ticket=m.getContent().toString() ;
AgentAdress client=m.getSender() ;

Nous procédons par la suite, à la création d'un deuxième groupe *''travel-providers''* dans chacun des agents *Broker et Provider a*u niveau de leurs fonctions *activate() :*

createGroup(true,''travel'', ''travel-providers'', null, null) ;

Et affecter les rôles *''Borker'' et ''competence-Provider''* à ces deux types d'agents :

requestRole(''travel'', ''travel-providers'', ''broker'', null) ;
requestRole(''travel'', ''travel-providers'', competence+''-provider'', null) ;

L'agent *Broker* sélectionne premièrement les offreurs du ticket demandé par le *Client* dans un tableau d'adresses des agents :

AgentAdress[] bidders=get AgentWithRole(''travel'', ''travel-providers'',ticket+ ''-provider'') ;
Cpt=bidders.length ;
BidAnswer[] answers=new BidAnswer[cpt] ;

Ensuite, il envoie un message multiple demandant aux *Providers* d'envoyer leurs offres (flèche 2) :

ACLMessage req=new ACLMessage(''REQUEST-FOR-BID'', ticket) ;
broadcastMessage(''travel,''travel-providers'', ticket+ ''-provider'', req) ;

Ainsi, suite à leurs réceptions de la demande d'offre de l'agent *Broker,* les agents *Providers* envoient leurs prix au niveau de la fonction *handleMessage(ACLMessage m) (flèche 3):*

m.getAct().equalsIgnoreCase(''REQUEST-FOR-BID'') ;
sendMessage(m.getSender(), new ACLMessage(''BID'', String.valueOf(prix))) ;

Suite à ce message, l'agent *Broker* prévoit la méthode *receiveBid(ACLMessage m)* pour sauvegarder les offres dans un tableau à deux dimensions *answers* et sélectionner le plus bas prix dans une autre fonction *bestContrat()*.

```
Cpt-- ;
answers[cpt]=new BidAnswer (m.getSender(), new Integer (m.getContent().toString())).intValue()) ;
```

La fonction *bestContrat()* de l'agent *Broker* consiste à parcourir le tableau *answers* et sélectionner le plus bas prix pour un *Provider*.

```
AgentAdress best=answers[i].getBidder() ;
Int bestOffer=answers[i].getValue() ;
```

Puis, l'agent *Broker* envoie un message *MAKE-CONTRACT* pour chacun du *Provider* sélectionné et du *Client* (flèche 4).

```
sendMessage(best, new ACLMessage (''MAKE-CONTRACT'',''contract'')) ;
sendMessage(client, new ACLMessage (''MAKE-CONTRACT'',''contract'')) ;
```

A leurs réceptions du message *MAKE-CONTRACT*, les agents *Provider et Client* créent un nouveau groupe *''contract''* et prennent les rôles respectivement du service et du client.

```
String group=m.getContent().toString() ;
createGroup(true, ''travel'', group, null,null) ;
requestRole(''travel'', group, ''service'', null) ;
requestRole(''travel'', group, ''client'', null) ;
```

L'agent *Client* récupère l'adresse du *Provider* affecté à ce nouveau groupe et lui envoie un message *VALIDATE* (flèche 5).

```
provider=getAgentWithRole(''travel'', group, ''service'') ;
sendMessage(provider, new ACLMessage(''VALIDATE'') ;
```

Ainsi, l'agent *Provider* lui répond par un message *ACCEPT-CONTRACT* (flèche 6).

```
sendMessage(m.getSender(), new ACLMessage(''ACCEPT-CONTRACT'') ;
```

4.3. E-market sous MAGIQUE

Dans cet exemple, nous lançons un agent vendeur *auctioneer*, initiateur d'enchères et un agent acheteur *bidderOne*. Nous disposons dans cette implémentation de quatre compétences principales *AuctionnerSkill, GraphAuctioneer, BidderSkill et BidderGraph* qui héritent de la classe *DefaultSkill* ,offerte par le package *fr.lifl.magique.skill,* et incluent les principales fonctions dans cette implémentation. (Voir figure 4)

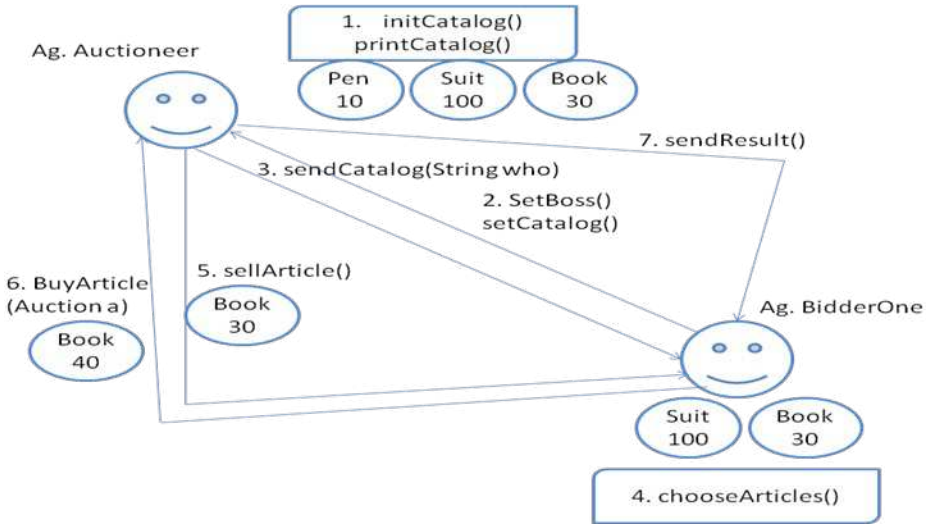

Figure 4 : Implémentation d'un e-market sous MAGIQUE

Dans la classe de démarrage *TestAuction,* nous procédons, tout d'abord, à la création de notre premier agent *auctioneer*, puis de l'enrichir par deux compétences *AuctionnerSkill et GraphAuctioneer*.

```
public class TestAuction extends AbstractMagiqueMain
{… Agent auctioneer=createAgent(''auctioneer'') ;
auctioneer.addSkill(''AuctioneerSkill'', new Object[] {auctioneer}) ;
auctioneer.addSkill(''GraphAuctioneer'', new Object[] {auctioneer}) ;
auctioneer.perform(''initCatalog'') ;
auctioneer.start() ;….}
```

Lors de son lancement, l'agent *Auctioneer* appelle, par des requêtes *perform(),* les méthodes *initCatalog(),* pour l'initialisation de ses articles, et *printCatalaog(),* pour l'affichage, incluses respectivement dans ses compétences AuctionSkill et GraphAuctioneer.

```
public class AuctioneerSkill extends DefaultSkill
{… public void initCatalog()
{catalog.add(new Article(''book'', 30)) ;…
private String myAgent ;

public void action()
{perform(myAgent, ''printCatalog'', catalog) ;}…
                }
```

De même que l'*auctioneer*, nous procédons, tout d'abord, à la création de notre deuxième agent *bidderOne* dans la classe de démarrage *TestAuction,* l'enrichir par les deux compétences *BidderSkill et GraphBidder.*

```
public class TestAuction extends AbstractMagiqueMain
{… Agent bidderOne=createAgent(''bidderOne'') ;
bidderOne.addSkill('' bidderSkill'', new Object[] { bidderOne, auctioneer.getName() }) ;
bidderOne.addSkill(''Graph bidder'', new Object[] { bidderOne, auctioneer.getName()}) ;
bidderOne.start() ;  ….}
```

La méthode *connect()* de la classe et compétence *GraphBidder* utilise la méthode *connectTo(myBossLong)* de la classe *AtomicAgent* pour connecter directement le *bidder* à son *auctioneer.* Ce genre de connexion permet aux agents d'exploiter leurs méthodes respectives grâce à la méthode *perform(),* comme indiqué dans la commande ci-dessous *: perform(myBoss, ''sendCatalog'', myAgent).*

```
public class GraphBidder extends DefaultSkill
{... public void connect(int i, String strat, String ip)
{...
myBossLong+=''@''+ip+'' :4444'' ;
connectTo(myBossLong) ;
myBoss=myBossLong ;

perform(myAgent, ''setBoss'', myBoss) ;
perform(myBoss, ''sendCatalog'', myAgent) ;
...}
}
```

L'agent *bidderOne* demande à l'*auctioneer* de lui transmettre le catalogue par la méthode *setCatalog(Catalog c)* incluse dans sa compétence *BidderSkill (flèche 2).* A sa réception, l'agent *Auctioneer* lui transmet son catalogue d'articles *remaining,* par la fonction *sendCatalog(),* qui est incluse dans sa compétence *AuctioneerSkill*, tout en lui ajoutant à son tableau d'acheteurs *myBidders*. (flèche 3)

```
public void sendCatalog(String who)
{perform(who, ''setCatalog'', remaining) ;
                                        ...

myBidders.add(who) ;}
```

Ensuite, l'agent *bidderOne* choisit aléatoirement deux articles parmi les trois proposés par l'*auctioneer* en se servant de sa méthode *chooseArticles(Catalog c).(4)*

```
public void setCatalog(Catalog c)
{catalog =c ;
perform(myAgent, ''printCatalog'', catalog) ;
wanted=chooseArticles(c) ;}

public Catalog chooseArticles(Catalog c)
{ Catalog chosen= new Catalog() ;
int numberWanted=2 ;

for(int i=0 ; i<numberWanted ; i++)
{int choice=(int)(Math.random()*(double)c.size()) ;
chosen.addElement(c.elementAt(choice)) ;}
...
return chosen ;}
```

Après, l'agent *auctioneer* choisit, *dans sa fonction sellArticle(Integer i) (flèche 5),* un article parmi son catalogue, crée une instance de l'objet *Auction*, et initialise l'acheteur avec ''none'. Il envoie un appel aux enchères à tous les acheteurs *bidders* via la fonction *sendArticleDescription(lastAuction).*

```
public void sellArticle(Integer i)
        {Article     toSell=((Article)catalog.elementAt(i.intValue())) ;
if (remaining.contains((Article)catalog.elementAt(i.intValue())))
{lastAuction= new Auction(''none'', toSell) ;
perform(myAgent,''addMessage'', ''I'm selling the folowing article :'') ;
perform(myAgent, ''addMessage'', toSell.toString()) ;
sendArticleDescription(lastAuction) ;}
```

La fonction *buyArticle(Auction a)* (flèche 6) permet au *bidder* de soumettre son offre. L'acheteur *bidderOne* invoque sa compétence *bid* par la primitive *askNow,* définie par *MAGIQUE* pour une réponse immédiate. Il demande aussi à son manager *myBoss* d'actualiser l'enchère en cours via la commande *perform(myBoss, ''setLastAuction'', currentArticle).*

```
public void buyArticle(Auction a){

Auction currentArticle= new Auction(a.getBidder(), a.getArticle()) ;
lastAuction=currentArticle ;      ...
if(a.getBidder().compareTo(myAgent) !=0)
{ int enchere=((Integer)askNow(''bid'', currentArticle, new Integer(currentMoney))).intValue() ;
...
if(enchere>0)
{currentArticle.getArticle().improvePrice(enchere) ;
currentArticle.setBidder(myAgent) ;
perform(myBoss, ''setLastAuction'', currentArticle) ;

}
```

Enfin, la méthode *sendResult()*de l'*auctioneer* (flèche 9) permet de retirer l'article gagné de l'ensemble du catalogue *remaining*, d'informer tous les agents acheteurs *bidders* du résultat et d'ajouter l'objet d'enchère terminée *eoa* à un vecteur *sold*

```
private void sendResult()
        {remaining.remove(lastAuction.getArticle()) ;
Auction eoa=lastAuction ;
perform(''addMessage'', ''the auction for this article are over'') ;
perform(myAgent, ''addMessage'', ''the result is :'') ;
perform(myAgent, ''addMessage'', eoa.toString()) ;
...

sold.add(eoa) ;
perform(myAgent, ''updateResult'', eoa.getArticle()) ;
perform(myAgent, ''updateOwners'', eoa.getBidder() ) ;

}
```

5. Conclusion

Après la présentation des trois plateformes multi-agents *JADE*, *MADKIT et MAGIQUE* et leurs principaux packages, offerts par leurs *API,* (voir l'annexe) nous avons prévu une première implémentation d'un marché électronique à base d'agent sous la plateforme *JADE*. Nous avons défini

les trois agents principaux à savoir *Broker, Provider et Client* en se servant des classes offertes par *l'API JADE* comme *Agent, Behaviour, DFAgentDescription*.

La classe *Agent* offre les méthodes pour démarrer et arrêter les agents ainsi pour l'envoi et la réception des messages ACL. La classe *Behaviour* permet de définir les comportements des agents par la méthode *action().*

Dans la deuxième implémentation du marché électronique sous *MADKIT,* nous définissons, aussi, trois agents principaux : *Broker, Provider et Client* en se servant des classes offertes par l'*API MADKIT* comme *Agent, AgentAdress et ACLMessage*.

La classe *Agent,* qui hérite de *AbstractAgent*, offre des méthodes comme *activate()* pour l'enregistrement des agents et l'affectation initiale des rôles et des groupes et *live()* pour définir le comportement principal des agents. Les agents sont aussi, dotés des primitives comme *createGroup()* permettant de créer des groupes*, requestRole()* pour demander leurs rôles et *getAgentWithRole()* pour sélectionner les adresses des agents jouant un rôle donné au sein du groupe.

La classe *ACLMessage* permet de reconnaitre les actes du langage *FIPA ACL* comme *REQUEST, MAKE-CONTRACT et ACCEPT-CONTRACT.*

Dans la troisième implémentation du marché électronique sous *MAGIQUE*, nous avons définis deux agents principaux *auctioneer et bidderOne* en se servant du package *fr.lifl.magique* qui offre des classes comme *Agent et Start*. La classe *Agent* permet l'ajout des compétences aux agents grâce aux méthodes *addSkill() ou perform().* La classe *Start* offre la méthode *go()* pour démarrer la main classe avec la plateforme *MAGIQUE.*

Ainsi, nous avons défini quatre compétences à savoir *AuctioneerSkill, GraphAuctioneer, BidderSkill, GraphBidder* qui héritent de la classe *DefaultSkill* offerte par le package *fr.lifl.magique.skill.*

Ces trois implémentations nous ont permis de détecter les similitudes, comme la simplicité d'exécution des agents sur ces trois plateformes et la possibilité de mise en œuvre des scénarios pour des marchés électroniques, et les spécificités de chacune des trois plateformes. Par exemple, les travaux en matière des services web[3] et les protocoles d'interaction[1] sont plus développés en JADE.

6. Annexe

Implémentations	Classes utilisées	Méthodes et rôles
JADE - 3 agents et 2 conteneurs : Broker dans Main-Container, Client et Provider dans Container1	*Agent*	- *setup(), takeDown()* : code de démarrage et suppression des agents
		- *send(ACLMessage m), receive(MessageTemplate m)* : envoi et réception des messages ACL
	Runtime	- *createAgentContainer(Profile p) :* créer un agent conteneur
	AgentContainer	- *createNewAgent()* : créer un agent
	Behaviour	- *action()* : définir les comportements des agents
MADKIT - 5 agents : Broker, Client et 3 Providers.	*AbstractAgent*	- *launchAgent()* :lancer un nouveau agent dans la plateforme
		- *activate():* définir un 1er comportement de l'agent
		- *createGroup(), requestRole()* : créer un groupe et

-3 groupes : travel-provider, travel-clients et travel		affecter un rôle
		- *getAgentWithRole()* : retourne l'adresse de l'agent jouant ce rôle dans le groupe
		- *sendMessage(AgentAdress a, Message m), receive(Message)* :envoi et réception des messages
	Agent	- *live()* : définir un 2ème comportement de l'agent
	Message	- *getSender() , getReceiver() :* retournent les adresses des agents émetteur ou récepteur du message
	ACLMessage	- Reconnaître les performatives FIPA ACL
MAGIQUE - 2 agents : auctioneer et bidder - 4 compétences : AuctioneerSkill, GraphAuctioneer, BidderSkill, GraphBidder	*Agent*	- *addSkill(), perform(), askNow() :* Ajout des compétences
	AtomicAgent	- *start()* : démarrer l'activité décrite dans la partie action()
		- *connectTo()* :connecter un agent à un autre
	DefaultSkill	- *action()* : créer une compétence proactive

Tableau des principales classes et méthodes offertes par les API lors des implémentations

Bibliographies:

[1]H. Halioui, Intégration des protocoles de négociation dans la méthode ForMAAD. Mémoire de mastère de la Fcaulté des Sciences Economiques et de Gestion de Sfax, 2010

[2]K. Liu, E-Commerce oriented automated negotiation based on FIPA interaction protocol specification, China, 2007

[3]T. Jarraya, Réutilisation des protocoles d'interaction et démarche orientée modèles pour le développement multi-agents. Thèse de doctorat de l'Université de Reims Champagne Ardenne, 2006

[4]T. Melliti, Interoperabilité des services web complexes. Application aux systèmes multi-agent. Thèse de doctorat de l'université Paris IX Dauphine, 2004

[5]M.H. Verrons, GeNCA: un modèle général de négociation de contrats entre agents. Thèse de doctorat de l'Université de Lille, 2004

[6]N. Jennings A classification scheme for negotiation in electronic commerce, EPSRC, 2000

[7]M. Ganzha, JADE based multi-agent e-commerce environnement: initial implémentation. Symposium SYNASC Romainia, 2004

[8]O. Hioual, Vers une architecture à base d'agents pour la composition sémantique et dynamique des services web dans le contexte d'EbXML, Conférence MOSIM, Hammamet, 2010

[9]M. Lyell. On software agents and web services: usage and design concepts and issues. Workshop on web services and agent-based engineering, Melbourne, Australia, 2003

[10]M. P. Papazoglou, Service Oriented Computing: concepts, characteristics and directions 2003

[11]Jade Java Agent Developpement Framework http://jade.tilab.com/

[12]Magique Multi Agent hiérachique http://lifl.fr/SMAC/projects/magique/

[13]Madkit Multi Agent Devlopment Kit http://madkit.org/

[14]Foundation for Intelligent Physical Agents, 2002

[15]L. Zeghache, Agents mobiles et web services pour l'e-commerce. Centre de Recherche en Informatique, Alger

[16]N.Jennings, Automated Negotiation : Prospects, Methods and Challenges, 2001

[17]M He, On agent Mediated Electronic Commerce, IEEE, 2003