UNIVERSITE PIERRE & MARIE CURIE
LA SCIENCE À PARIS

MASTER IAD

PROJET AAAAMA

Une application de commerce électronique utilisant le langage CLAIM

9 février 2006

Par:
ALEXANDRE BARGETON
BENJAMIN DEVÈZE

TABLE DES MATIÈRES

1 Introduction **1**

2 Application réalisée **2**

2.1 La classe Market **2**

2.2 La classe Client **3**

2.3 La classe Buyer **3**

2.4 Améliorations possibles **4**

3 De l'usage de CLAIM/SyMPA **5**

3.1 Commentaires liminaires **5**

3.2 De l'usage du langage CLAIM **5**

3.2.1 Avantages 5

3.2.2 Inconvénients et suggestions 6

3.3 De l'usage de la plateforme SYMPA **7**

3.3.1 Avantages 7

3.3.2 Inconvénients et suggestions 7

4 Conclusion **10**

Bibliographie **10**

Chapitre 1

Introduction

L'unité d'enseignement *Apprentissage Automatique, Agents Autonomes et MultiAgents* a pour vocation de nous donner les bases et une expérience concrète des méthodes et techniques d'apprentissage automatique et de modélisation multiagents. Dans ce cadre, nous avons pu mettre en pratique les principaux concepts du paradigme multiagent en déployant une application de commerce électronique utilisant le langage CLAIM et la plateforme associée : SyMPA.

Ayant déjà eu l'occasion de mettre en oeuvre des systèmes multiagents (SMA) dans le cadre de notre cursus, ce sujet nous touche particulièrement l'un et l'autre. C'est ainsi qu'Alexandre a élaboré une application de calcul distribué reposant sur la plateforme Grasshopper [MBB+]. Benjamin, quant à lui, a effectué un stage de 7 mois au sein de Thales durant lequel il a développé une plateforme complète de déploiement de SMA en Prolog. Il a également, dans ce contexte, eu l'occasion d'implémenter un grand nombre d'agents autonomes en langage Prolog. Ces expériences précédentes nous ont permis de mieux situer CLAIM/SyMPA dans le paysage des outils existants et d'en dégager les qualités et les défauts au regard des autres systèmes.

Le présent rapport traite dans une première partie des fonctionnalités de notre application et décrit ses mécanismes internes. La partie suivante relate le fruit de nos expériences avec CLAIM/SyMPA et aborde les avantages et les inconvénients de l'ensemble ainsi que nos suggestions d'améliorations. Nous dressons enfin, dans une dernière partie, un bilan général de notre utilisation de CLAIM/SyMPA.

CHAPITRE 2

APPLICATION RÉALISÉE

Sommaire

2.1 La classe Market 2
2.2 La classe Client 3
2.3 La classe Buyer 3
2.4 Améliorations possibles 4

L'objectif du projet était de programmer une application simple de commerce électronique en utilisant des agents intelligents et mobiles. Des agents mobiles (agents *Buyer*) créés par des clients (agents *Client*) migrent vers des marchés distribués (agents *Market*) afin d'acheter des produits.

2.1 La classe Market

Les agents appartenant à cette classe modélisent des marchés offrant des produits. Ils initialisent leur base de connaissances à partir d'un fichier qui présente la forme suivante :

Connaissances d'un agent *Market*

```
money(amount_of_money)
hasProduct(product1,price1,quantity1)
hasProduct(product2,price2,quantity2)
...
```

Les agents *Market* sont capables d'intéragir avec les agents *Buyer*. Pour ce faire, lorsqu'un agent *Buyer* leur demande si ils vendent un produit donné, ils répondent à cette requête[1] par un message indiquant son prix de vente. Notons que cette réponse engage l'agent *Market* à ce prix. Les agents *Market* sont à même d'accueillir en leur sein des agents *Buyer* désirant leur acheter une certaine quantité d'un produit. Ils leur vendent le produit demandé dans la capacité maximale de leur stock disponible. Cette opération s'achève bien sûr par l'actualisation de la quantité d'argent et de produits disponibles dans la base de connaissances. En sus de ces *capabilities* **réactives**, nous avons doté les agents *Market* d'une *capability* **proactive** permettant de réapprovisionner les marchandises en rupture de stock d'une quantité déterminée aléatoirement.

[1] Uniquement s'ils proposent ce produit

2.2 La classe Client

Les agents *Client* initialisent leur base de connaissances à partir d'un fichier contenant une liste de courses ayant la forme suivante :

Connaissances d'un agent *Client*

money(amount_of_money)
needProduct(product1,quantity1)
needProduct(product2,quantity2)
...

Nous considérons qu'un agent *Client* est prêt à payer le prix nécessaire pour acquérir l'ensemble des produits désirés et peut donc se mettre à découvert. Pour chaque produit de la liste, cet agent crée un agent *Buyer* mandaté pour obtenir la quantité requise du produit. C'est ainsi que plusieurs agents *Buyer* évoluent en parallèle. Pour éviter d'avoir à donner manuellement un nom à chaque nouvel agent *Buyer*, le nom est attribué automatiquement par l'agent *Client* par dérivation de son propre nom (nom1, nom2, ...). A tout moment, un agent *Buyer* peut revenir transmettre le fruit de son labeur et réclamer la somme d'argent dépensée. Le client met alors à jour ses connaissances en spécifiant qu'il a acquis certains produits et revoit donc à la baisse ses demandes. Lorsqu'une demande d'un produit est pleinement satisfaite, le client relève de ses fonctions (tue) l'agent mandaté pour ce produit.

Notons que nous avons sciemment décidé de ne pas utiliser les possibilités offertes par les *goals*/*effects*. Ce choix fut fait afin de ne pas alourdir l'exécution de notre application, mais il est clair que nous aurions pu donner comme but à l'agent *Client* de se fournir en chaque produit puis rajouter des *effects* permettant de remplir ces buts.

2.3 La classe Buyer

Lors de sa création l'agent *Buyer* ne connait pas les marchés existants. La programmation de la découverte des agents *Market* s'effectue en tirant profit de la communication *multicast* offerte par CLAIM permettant d'envoyer un message à tous les agents de la classe *Market*. A la réception des réponses des marchés l'acheteur enregistre dans sa base de connaissances le nom du marché ainsi que le prix de vente du produit désiré. Afin de remplir sa mission au moindre coût, l'acheteur migre d'un marché à l'autre, du moins cher au plus cher jusqu'à ce qu'il ait obtenu la quantité escomptée du produit ou bien que plus aucun marché ne puisse satisfaire à sa demande (rupture de stock). Ce mode de fonctionnement est illustré sur la figure 2.1.

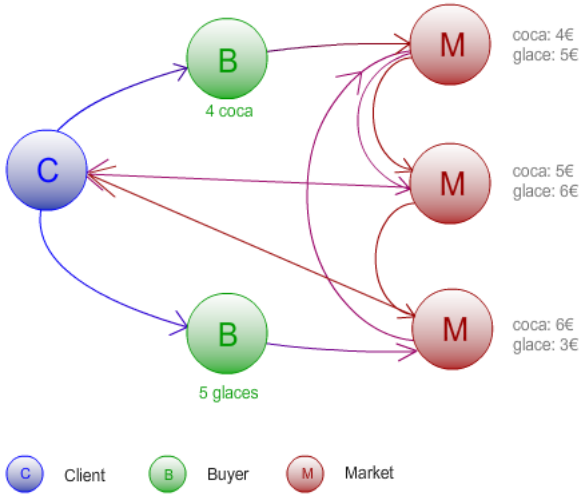

FIG. 2.1: *Stratégie d'achat des agents* Buyer

Après cette phase d'achats, il retourne auprès de son mandant afin de lui donner ce qu'il a d'ores et déjà acheté et de se faire payer. S'il n'a pu obtenir la quantité requise du produit, il repart achever sa mission auprès des marchés qui se sont réapprovisionnés.

2.4 Améliorations possibles

L'objectif du projet étant de programmer une application relativement simple, nous n'avons pas poussé plus loin les sophistications. Nous aurions toutefois pu rajouter les fonctionnalités suivantes :

- Gestion de départements spécialisés migrant d'un marché à l'autre en fonction de la clientèle.
- Restriction sur le nombre d'agents *Buyer* créés simultanément.
- Négociation sur le prix, vente aux enchères.
- Ajout d'un prix maximal que le client est prêt à payer pour un produit, gestion de la fluctuation des prix en fonction de l'offre et de la demande.
- Négociation d'une commission pour les agents *Buyer*.
- Concurrence de plusieurs agents *Buyer* pour la même tâche.
- Collaboration de plusieurs agents *Buyer* pour la même tâche.

CHAPITRE 3

DE L'USAGE DE CLAIM/SYMPA

Sommaire

3.1 Commentaires liminaires 5
3.2 De l'usage du langage CLAIM 5
3.2.1 Avantages 5
3.2.2 Inconvénients et suggestions 6
3.3 De l'usage de la plateforme SYMPA 7
3.3.1 Avantages 7
3.3.2 Inconvénients et suggestions 7

3.1 Commentaires liminaires

Avant de rentrer dans le vif du sujet, il convient de faire quelques commentaires. Notons tout d'abord que nous avons testé la plateforme sous linux et que donc certains problèmes mentionnés ne sont peut-être pas valables sur d'autres systèmes.

Nous sommes pleinement conscients du fait que CLAIM/SyMPA ne sont encore que des prototypes et que, de ce fait, certaines de nos suggestions sont prévues mais non encore implémentées. Nous avons toutefois pris le parti de mentionner tous nos commentaires en nous positionnant comme un utilisateur final de l'outil.

Bien qu'ayant lu la thèse d'Alexandru Suna [Sun05] ainsi que l'article [SFS04], relatifs à CLAIM/SyMPA, certains éléments ont pu nous échapper et de ce fait il se peut que certaines de nos doléances soient liées à une compréhension imparfaite de notre part, ce dont nous nous excusons.

Forts de l'expérience acquise durant le développement de l'application de commerce électronique, il est maintenant temps de faire part de nos impressions concernant tout d'abord le langage CLAIM puis la plateforme SyMPA. Nous procédrons dans les deux cas en dégageant les avantages puis les inconvénients qui nous ont marqué.

3.2 De l'usage du langage CLAIM

3.2.1 Avantages

Il nous semble que le premier avantage du langage CLAIM est de placer le programmeur d'agent à un niveau d'abstraction plus élevé et mieux adapté que le langage Java. Cela permet de réduire le fossé existant entre conception d'agent et implémentation qui,

dans bien des systèmes repose sur le langage Java et force à adapter le paradigme agent au paradigme objet. La découpe d'un agent en *knowledge*, *goals*, *capabilities*, *processes* et les mécanismes internes du système semblent bien adaptés. Ils permettent de spécifier aisément le comportement d'un agent capable de planifier ses actions.

A cet égard on peut dire que le langage CLAIM permet de développer des agents de manière déclarative. La lisibilité du code est bonne. Le langage permet de développer rapidement et de manière efficace des agents.

Nous avons également apprécié l'effort de formalisation du langage permettant la démonstration de certaines de ses propriétés voir à terme une vérification automatique des programmes d'agent.

Le langage CLAIM offre la possibilité de décrire extrêmement simplement des comportements réactifs aussi bien que proactifs.

Un autre point fort de CLAIM est qu'il autorise intrinsèquement une mobilité forte des agents, ce qui est fort appréciable lorsque l'on sait que la plupart des outils existants n'offrent qu'une mobilité faible.

La routine *send* permet d'aisément mettre en place une communication point à point mais surtout offre des facilités très appréciables pour la communication *multicast* et le *broadcast*.

Le fait de pouvoir définir des classes d'agents réutilisables à loisir constitue également un avantage notable.

3.2.2 Inconvénients et suggestions

Il nous semble cependant, qu'en l'état, le langage est beaucoup trop limité et manque de flexibilité. Durant le développement de notre application nous avons cherché à restreindre au maximum l'utilisation de code Java. Toutefois, compte tenu des limitations du langage nous avons été contraint d'y recourir beaucoup plus que nous le voulions. Nous pensons que le langage Java, ou tout autre langage impératif, ne devrait être utilisé qu'en dernier recours pour déployer des agents implémentant des algorithmes naturellement impératifs ou pour déployer des éléments d'interface graphique. Le langage CLAIM devrait dans cette optique être doté d'un minimum d'opérateurs arithmétiques et de capacités à effectuer des opérations simples sur les variables. Dans le même ordre d'idée, il serait souhaitable de pouvoir accéder directement aux champs d'un agent depuis le langage CLAIM. Prenons le problème d'un agent souhaitant connaître son propre nom. Pour le résoudre nous avons testé deux alternatives : soit l'appel à une méthode Java qui renvoie l'attribut *name* de l'agent, soit en n'utilisant que CLAIM l'agent s'envoie un message à lui même et le champs *sender* le renseigne sur sa propre identité. Cela nous semble excessivement lourd pour une opération aussi simple. Notre doléance principal est donc que le langage est encore trop esclave du Java.

Globalement la grammaire est trop peu permissive. Pourquoi autoriser des espaces après le $=$ d'une définition de champs mais pas avant ?

La syntaxe à utiliser pour les préconditions des *capabilities* est trop lourde et manque cruellement de l'utilisation d'opérateurs.

Il est peu pratique d'être contraint d'utiliser la boucle *forAllKnowledge* lorsque l'on veut accéder uniquement au premier élément de la base de connaissances qui correspond à un motif. Pourquoi ne pas donner l'accès aux variables des préconditions d'une *capability* dans le *do* ?

CLAIM devrait également offrir la possibilité d'écrire des commentaires. Bien que la lisibilité du langage soit bonne c'est une fonctionnalité toujours appréciable et ce, d'autant plus lorsque l'on travaille sur le code d'autres programmeurs.

Il nous semble un peu lourd de devoir spécifier tous les champs d'un agent alors qu'un grand nombre sont à *null*, ne serait-il pas intéressant de mettre ces champs à *null* automatiquement lorsqu'ils sont omis ?

Certains choix de la grammaire du langage semblent étranges, pourquoi, par exemple, séparer les *cap_action* par '.' alors que ce séparateur n'est jamais utilisé et qu'il apparaît comme peu lisible à l'usage [1] ?

Lorsque l'on recherche les agents *Market* susceptibles de nous vendre un produit nous avons posé un temps d'attente afin de leur laisser le temps de nous répondre et de traiter ces messages. Le choix de ce temps est arbitraire et ne nous permet pas d'être certain que tous les agents ont répondu. Nous aurions pu décider de déclencher l'achat à la réception d'un nombre donné de réponse. Ces deux stratégies semblent peu satisfaisantes et le langage pourrait peut être nous donner la possibilité de pallier ce problème.

Le langage ne semble enfin pas très bien adapté pour mettre en place des comportements avancés reposant sur une successions de plusieurs messages.

3.3 De l'usage de la plateforme SYMPA

3.3.1 Avantages

La notion de hiérarchie d'agents, le fait de pouvoir migrer dans un autre agent et de reconfigurer dynamiquement la structure d'un SMA nous est apparu très pertinent et d'un grand intérêt pratique.

Il est très appréciable que la plateforme prennent en compte les aspects liés à la sécurité et cela de manière complètement paramétrable selon la confiance que l'on porte à un agent donné.

Fournir un environnement de développement d'agent est également un bon élément.

Enfin le caractère distribué et la conformité aux standards OMG MASIF sont à noter.

3.3.2 Inconvénients et suggestions

Relativement à la conception de la plateforme

La plateforme est limitée dans le sens où elle ne supporte que des agents CLAIM.

Le serveur central nous semble constituer une trop forte centralisation et constitue un goulot d'étranglement et un potentiel point faible.

Relativement à l'implémentation de la plateforme

Notons tout d'abord que les scripts de lancement sous unix ne sont pas directement utilisables puisqu'ils contiennent des \r qu'il faut supprimer à la main. Notons que nous avons subi un problème similaire avec la fonction *getKnowledge* fournie.

L'exécution semble très lente sur une machine récente. L'ensemble est très gourmand en *threads*. L'utilisation du processeur et de la mémoire semble conséquente, notamment lorsque l'on utilise des capacités proactives. La migation en réseau local s'avère également excessivement lente et l'interface *freeze* durant la migration.

Nous avons subi de nombreux plantages de tout l'environnement graphique ainsi qu'une consommation excessive du processeur.

Un grand nombre de bugs ont fortement terni notre expérience de la plateforme et nous ont souvent contraints à contourner les problèmes d'une manière moins satisfaisante voir à nous limiter dans nos choix d'implémentations. Citons en quelques un. Il n'est, par exemple, pas possible pour un agent de se tuer lui même, si cela est normal le message n'est pas explicite puisque SyMPA nous dit que l'agent que l'on souhaite tuer n'existe pas.

Autre exemple, ce code ne compile pas, bien qu'apparamment conforme à la grammaire :

[1] il se confond aisément avec le point utilisé pour les appels à Java par exemple

```
defineAgentClass Test1(){
authority=null ;
...
capabilities={
test{
message=null ;
condition=null ;
do{forAllKnowledge(P( ?p, ?q,0))
{ ?qte=Java(A.r(20)).Java(A.w(2)).changeNumKnowledge(P( ?p, ?q, ?s),3,+, ?qte)}}
effects=null ;
}
}
processes=null ;
agents=null ;
}
```

Tandis que le même code en enlevant *Java(A.w(2))* compile parfaitement.

```
defineAgentClass Test1(){
authority=null ;
...
capabilities={
test{
message=null ;
condition=null ;
do{forAllKnowledge(P( ?p, ?q,0))
{ ?qte=Java(A.r(20)).changeNumKnowledge(P( ?p, ?q, ?s),3,+, ?qte)}}
effects=null ;
}
}
processes=null ;
agents=null ;
}
```

Dans un autre registre, il aurait sans doute été souhaitable d'implémenter les *knowledges* sous forme de tables de hachage plutôt que sous forme de tableaux. Les clés construites sur la base du nom de la proposition et de son arité permettraient ainsi d'accélérer les opérations de recherche dans la base de connaissances et éviterait de boucler sur tous les éléments.

Enfin, puisqu'en l'état le langage CLAIM repose fortement sur le langage Java, il serait bon de documenter le code[2] avec javadoc pour que le programmeur d'agent accède rapidement aux fonctionnalités dont il peut disposer.

Relativement au parser

Il serait souhaitable d'avoir une visualisation précise de la localisation et de la nature des erreurs de parsing, éventuellement dans l'interface textuelle présentant le code de l'agent.

Il faudrait également ajouter une reprise sur erreur afin que tout ne s'arrête pas à la première erreur. Les définitions de tous les agents syntaxiquement valides d'un même fichier doivent pouvoir être prises en compte.

Un système de coloration syntaxique dans l'éditeur d'agent serait également bienvenu.

[2] Notamment la classe Agent

Relativement à l'interface

- Le code Java des agents pourraient être compilé automatiquement et le code compilé placé où il convient.
- L'affichage du code de l'agent pourrait automatiquement être mis à jour à chaque *run*.
- Il faudrait améliorer l'affichage des fenêtres des agents, la zone de *log* est trop petite et n'est pas redimensionnée avec la fenêtre.
- Il serait intéressant de pouvoir scripter le lancement d'un SMA afin d'éviter d'avoir à tout refaire manuellement à chaque lancement.
- Il arrive que la liste des classes d'agents disponibles présente des redondances perturbantes.
- Il serait bien d'ajouter un bouton d'ajout d'agent dans la barre d'outil ainsi qu'un bouton permettant de stopper tous les agents en cours d'exécution.
- L'utilisateur devrait pouvoir supprimer, directement depuis l'interface, des classes d'agents enregistrées sans avoir à manuellement effacer un fichier de configuration.
- Il serait utile que les fenêtre de *knowledge* se mettent à jour à la volée.
- Les fenêtres dépendants d'un agent (*knowledge*...) devraient se fermer avec l'agent.
- Il faudrait donner à l'utilisateur humain l'opportunité d'intéragir en cours d'exécution avec les agents. Il serait par exemple bon de pouvoir lui permettre d'envoyer des messages aux agents.
- Donner automatiquement des noms aux agents qui n'en ont pas permettrait de gagner du temps.
- Il pourrait être intéressant d'utiliser l'interface du *centralSystem* pour des tâches plus avancées : migration d'agents, migration d'un système d'agents, suppression des agents, fermeture d'un système d'agents.

Chapitre 4

Conclusion

En guise de conclusion, nous pouvons donc dire que l'ensemble semble prometteur mais manque encore de maturité pour être vraiment utilisable en pratique. Un effort important devra être consenti pour que le système puisse passer à l'échelle et supporter des applications industrielles avec de nombreux agents. Nous avons toutefois apprécier les qualités du couple CLAIM/SyMPA et avons pu développer notre application dans des conditions satisfaisantes.

BIBLIOGRAPHIE

[MBB+] Thomas Magedanz, Christoph Baumer, Christoph Baumer, Markus Breugst, and Sang Choy. Grasshopper - a universal agent platform based on omg masif and fipa standards.

[SFS04] Alexandru Suna and Amal El Fallah-Seghrouchni. A mobile agents platform : architecture, mobility and security elements. In *Proc. of the Second International Workshop on Programming Multi-Agent Systems'04*, 2004.

[Sun05] Alexandru Suna. *CLAIM et SyMPA : Un environnement pour la programmation d'agents intelligents et mobiles*. PhD thesis, Université Paris 6 - Pierre et Marie Curie, 2005.